CS 111 Notes

Below are my notes from CS 111 during the Winter 2015 quarter at UCLA. I hope that current CS 111 students find them useful. I also highly recommend the OSTEP book, a free book on operating systems that's easy and clear in its explanations.

Open a pull request if you find any errors, and I'll gladly correct them. To read the notes offline, download this page's Markdown version and open it in Sublime Text, with the MarkdownEditing package installed.

Happy studying!

- VDSO

    - Processes
        - Process Creation
        - Process Replacement
        - Process Destruction
- Discussion #2: 16 January 2015
    - Hints on Lab 1b
        - execlp
        - fork
        - dup2
        - pipe
        - if, until, and while commands
    - WeensyOS 1
- Lecture #5: 19 January 2015
    - Orthogonality
        - Access to files (and devices)
            - Device Types:
        - Access to Processes
    - Modeling OS resources in a program
        - Redirection
    - Piping
        - Piping Issues
- Discussion #3: 23 January 2015
    - Lab 1c
        - clock_gettime()
        - Execution time clock choice
        - clock_getres()
        - getrusage()
        - Additional
        - Where to start
- Lecture #6: 26 January 2015
    - Orthogonality and Trouble
        - I/O on an Unlinked File
        - Unlinked Flash Drive
        - Zombie Processes
    - Named Temporary File with Race Condition
    - Example with gzip
    - Signals
        - signal() system call
        - Signal Masks
        - Critical Sections
        - Side Effects of Signal Handlers
        - Reasons for Signals and a List of Them
        - Asynchronous Signal-Safe Functions
    - Threads
        - Expensive Resources
        - Cheap Resources
- Lecture #7: 28 January 2015
    - Threads
    - Scheduling Threads

        - Scheduling Goals
        - Scheduling Policies
            - Real-time Scheduling
            - Hard Real-Time Scheduling
            - Soft Real-Time Scheduling
        - Scheduling Mechanisms
            - Cooperative Multitasking
                - yield()
                - Busy Waiting
                - Polling
                - Blocking
            - Pre-emptive scheduling
                - Timer Interrupt
            - Done or not done
        - Scheduling Metrics
        - First-Come, First-Serve
        - Shortest Job First
- Discussion #4: 30 January 2015
    - Problem 1
    - Problem 2
    - Problem 3
    - Problem 4
        - Part A
        - Part B
        - Part C
    - Problem 5
    - Problem 6
    - Problem 7
    - Problem 8
- Lecture #8: 2 February 2015
    - Round Robin Scheduling
    - Priority Scheduling
    - Synchronization
        - The Issue
        - Ways to Prevent Race Conditions
            - Sequence Coordination
            - Isolation
            - Atomicity
                - Kinds of Atomic Actions
            - Cheating Atoms
        - Example: ATM withdrawal/deposity
        - Critical Sections
        - Pipes in Threads
            - Lock Grain
- Discussion #5: 6 February 2015
    - Kernel Module
    - Code Layout
    - Spinlock
        - wait_event_interruptible
    - printk
- Lecture #9: 9 February 2015

        - How to implement critical sections
            - Back to lock/unlock
            - Special atomic x86 instructions
                - lock incl
                - xchgl
                - compare_and_swap
            - Pipe Example
                - Blocking Mutex
            - From TBP Tutoring with Mickey
            - Summary of Possible Problems
        - Detour: semaphore
        - Pipe with Blocking Mutex
        - Condition Variables
        - Read-write lock
- Lecture #10: 11 February 2015
    - Hardware Lock Elision
    - Deadlock
        - Conditions for Deadlock
            - Mutual Exclusion
            - Circular Wait
            - No pre-emption of locks
            - Hold and wait
        - Solving Deadlock
            - Detecting Deadlocks Dynamically
            - Redesign so that deadlocks cannot occur
    - Priority Inversion
    - Livelock
    - Event-driven programming
    - File Systems Performance
        - GPFS
            - Stripe Like Crazy
            - Distributed Metadata
            - Efficient Directory Indexing
            - Distributed Locking
            - Partition Awareness
    - I/O Performance
    - Scheduling and File Systems
- Discussion #6: 13 February 2015
    - Building the Disk
    - Superblock
    - Free block bitmap
    - allocate_block() and free_block()
    - Inode
        - Accessing Data from Inode
    - Data for Directory
    - Hardlink and Symbolic Link
- Lecture #11: 18 February 2015
    - Disk Scheduling
        - Shortest Seek Time First
        - First Come First Serve
        - SSTF + FCFS

        - Elevator Algorithm
            - Circular Elevator
    - Anticipatory Scheduling
    - Concurrent I/O
    - File System Design
    - FAT File System
    - UNIX File System
    - Berkeley Fast File System (FFS)
- Discussion #6: 20 February 2015
    - Scheduling
    - Synchronization
    - Midterm 1 Post-Mortem
        - Problem 1
        - Problem 2
        - Problem 3
        - Problem 4
        - Problem 5
- Lecture #12: 23 February 2015
    - Inodes and Indirect Blocks
    - Wedding Cake Abstraction of File Systems
        - chdir()
        - chroot()
        - chrooted jail
    - Symbolic Links
        - Symlink Implmenetation
        - lstat()
        - readlink()
        - unlink()
    - Special Files
        - mknod()
        - mkfifo
    - Dealing with Low-Level Failures
        - shred
- Lecture #13: 25 February 2015
    - Invariants for File Systems
        - Consequences of Violating Invariants
    - Performance Issues
        - Out-of-order Writes
            - fsync()
            - fdatasync()
            - sync()
            - rename()
        - fsck
    - Interaction Between Scheduling and Robustness
    - Robustness Terminology and Theory
        - Atomicity
            - Golden Rule of Atomicity
            - Lampson-Sturgis Assumptions
        - Commit Records
        - Journaling
- Discussion: 27 February 2015

        - Permissions
- Lecture #16: 9 March 2015
    - Media Faults
        - Flash drive problems
        - RAID
            - Hardware RAID vs Software RAID
            - RAID Robustness
    - RPC Failure Modes
        - HTTP
        - X Window System
        - Network File System (NFS)
            - Stateless Server
- Lecture #17: 11 March 2015
    - Security
    - Main Forms of Attacks
    - General Goals
        - Security Testing
    - Threat Modeling and Classification
    - General OS Security Functions
        - Authentication
            - Attacking Passwords
            - External and Internal Authentication
        - Integrity
        - Authorization
            - Methods of Access
            - Access Control Lists (ACLs)
            - Capabilities
        - Auditing
        - Correctness
    - Trusted Computing Base (Kerckhoff's Design Principle)
- Discussion: 13 March 2015
    - Problem 2
    - Problem 3
    - Problem 4
    - Problem 5
    - Problem 6
    - Problem 7
    - Problem 8
    - Problem 9

Lecture #1: 5 January 2014

Magical Grep

Eggert showing some output:

```
$ ls -l big
-rw-rw-r-- 1 eggert faculty 9223372036854775000 Oct 6 11:41 big
$
```

permissions, 1 hard link, ..., ..., size

~10^19 bytes in size

Look for char 'x' in it.

```
$ grep x big
$
```

No matches. Then he timed it:

```
$ time grep x big
real 0m009s
```

Just 9 milliseconds. Whaaat? That's waaay too fast: 10^19 bytes / 10^{-2} seconds => 10^21 bytes/second => 8 * 10^21 bits/second => 8 Zb/second (zetabits/second)

Eggert is encouraging us to read xkcd What If #31, haha, which asks what is the entire bandwidth of the Internet? The current estimate is 167 Tb/second (Terabits/second). Then, the comic asks how we can move data faster?

Here, at UCLA, we have that problem too. Solution: Use the Sneakernet. Move the hard disks physically. So if we use all of the US freight industry (UPS, USPS, FedEx, etc.) to move a bunch of MicroSD cards, then we can move 0.5 Zb/s. How was grep 16 times faster? It cheated.

The file wasn't actually that big; we pretended to make it that big, enough to fool all of the standard Unix utilities. Grep was smarter: "in all of the places you cheated, there are no x's".

Say we did:

```
$ echo x >> big
```

Then:

```
$ grep x big
Binary file 'big' matches
```

We're going to learn and understand how things work, especially operating systems. It is essential.

"This is the pep talk here, where we try to encourage more people to watch the course. Maybe I shouldn't have done that." Haha. xD

Course Administration

What we should know (listed for the transfers):

CS 32: data structures & algorithms CS 33: assembly code, caches, ALUs, etc. CS 35L: Make, GDB, threads, etc.

Why did we get 35L? 111 was too hard, so they made the first 20% of it into 35L.

Eggert also recommends:

CS 131 (programming languages): how languages work CS 151B (computer systems architecture): CS 33 on steroids CS 118 (networking)

The faculty had a big "arm-wrestling contest", and Eggert lost, so 111 is actually a pre-requisite for them. Problem: They require each other for understanding. We get to learn on the fly.

Faculty fought hard to make it a 5-unit course:

4 hours lecture 2 hours lab 9 hours outside study

This is what is reported to the dean. YES, Eggert will have a 5-10 minute break in the middle

Really though: 3.5 hours lecture 1.7 hours lab 20 hours outside study

His recommendation: No procrastination.

Course organization: 17 lectures He thinks it should be a semester-length with 30 lectures.

1/9 of grade: 1 midterm, 100 minutes, open notes (esp. assignment printouts), during lecture midway through quarter

2/9 of grade: 1 final, 180 minutes, open notes also

1/3 of grade: 4 labs: done with partners 1. Write a shell (with performance features). It's different from the others. It's designed to make everyone code a lot of C, and get really good at debugging it too. It comes in 3 parts, which we will turn in separately. We're expected to be able to break up the other assignments on our own. 2. Kernel hacking, like write a driver 3. Write a file system 4. Distributed. "The labs are based on real Linix. No toy operating systems. Go take classes at Berekley."

2/15 of grade: 2 minilabs: done solo 1. Scheduling 2. Virtual memory

1/12 of grade: Design problem (with partner) 1. Pick a design problem from one of the labs, with consultation from TA 2. Write up a report, etc.

1/20 of grade: Scribe notes (in groups up to 4) Use HTML 5 or 4.01, *validated* or else points lost. He wants scribe notes that will "past the test of time". Points loss: confusing, ugly, out-of-order; make Eggert look like a genius, like it's the Feyman Lectures Go to validator.w3c.org Sign-up during break in class. Due 1 week after lecture.

1/15 of grade: A 2-to-3 page "research" paper, Really a review paper on some topic related to operating systems. We'll read papers and articles on them, and then write a report. We can suggest one of our own too.

Lateness penalty: 1% penalty for (0, 1) days late 2% penalty for (1, 2) days late 4% penalty for (2, 3) days late, etc.

Drop-dead date: 13 March, the last day of instruction: no assignments accepted after this day.

"I've been in meetings with lawyers trying to defend the students, and I've beaten the lawyers every time."

"I have been known to seed the Internet with wrong answers."

Email a TA or Eggert if unsure after something.

No formal curve. His finals are North Campus'y, combining many ideas from multiple places, so the problems tend to be hit-or-miss. He shoots for a mean of 50%. Often, it's above or below that. A typical GPA for 111 is a 3.0, meaning a ~(B-).

Textbook (which is apparently a controversial choice): *Principles of Computer System Design*. This is MIT's computer system design course. The authors are stars of OS design. The grad students write the book for them. This book has everything, and doesn't really dilineate into discrete chapters.

Eggert will not limit himself to what is in the book, though he will try to keep them related. Even the assignment-lectures link is loose. The discussion sections are actually very important, as the TAs have been informed of all the gritty details.

Operating Systems

Articles related to OS':

"Red Star 3.0: North Korea launches its own OS" 2014-12-31 Sky News (UK)

"Crouton for Chromebooks: Run Ubuntu in a browser tab" 2014-12-28 lilputing (Doesn't work with a diff. GNU/Linux distro. though; find out why!)

"Samsung says new (2015) TVs will run Tizen [formerly LiMo]" CNET 2015-01-05

Quotes on OS':

Marina Weisbard, leader of Geman Pirate Party (they like piracy): "We don't offer a ready-made programme, but an entire operating system." -- The Economist 2013-01-15, p. 19

Operating systems are connected to politics, undreamed of before.

Definition: System

System as defined by the Oxford English Dictionary (1928, coedited by Tolkien):

I. An organized or connected group of objects II. A set of principles, etc., a scheme, method From Greek, see "systima" in Greek: organized whole, government, constitution. The roots mean "set up with".

This is what "system" used to mean, but the words defining it have changed in meaning with time. Even so, the notion of a "system" is ancient. Marina is appealing to the ancient meaning of "system".

The book's definition, from §1.A.2:

"A set of interconnected components that has a specified behavior observed at the interface with its environment."

Operating system from Bill Gates' personal encyclopedia, Encarta (2007):

"Master control program in a computer"

Here, it's who's in control that matters.

From the American Heritage dictionary (the most conservative dictionary, in its 4th edition from 2000):

"Software designed to control the hardware of a specific data-processing system in order to allow users and application programs to make use of it."

This is a very traditional definition of OS. Note how it assumes OS' are hardware-specific. That part is obsolete, but users and apps still matter.

From Wikipedia (page id: 640779030, 2015-01-03):

"Software that manages computer hardware and software resources and provides common services for computer programs"

Note how "control" has become "manage" and the appearance of "resources". Resource-management will be of great importance. As for "common services", it comes back to how we implement that system's interface, e.g. any program can ask the OS to translate UI text to Spanish for it. We want to avoid duplication. The overall goal here is *clarity*, as few as lines of code as possible.

"How can we make our systems obvious and clear?"

Problems in System Design

Common Systems Problems:

1. Incommensurate scaling

Well known in physical engineering designs. Example: a 12-foot tall human would need a a modified skeleton. Even

stretching in all 3 directions would not work. We'd be 8 times as heavy and 2 times as tall, but our bone strength grows as the square of their cross-section.

We are hitting *diseconomies of scale*, where things get more expensive per unit. Example: an 8-port gigabit router is $100, but a 48-port gigabit router is thousands of dollars. It's a *star network* where the number of possible combinations of connections grows as the square.

Contrast with: *economies of scale*, as with the pin factory from Adam Smith's *Wealth of Nations*. It's more efficient to have someone specialize, invest capital for eqipment, and make many pins rather than everyone else produce their own. We need to look for *both* of these phenomena into account when we scale our systems. Otherwise, they cause breakage or waste (e.g., too many pins).

1. Emergent properties

These are properties that only become appparent *qualitatively*. "Things just happen" as the system gets bigger, and we didn't predict them. We're looking for things that look like no big deal or maybe don't even exist or are noticeable at a small scale, but break things at a large scale.

20 years ago, we got the latest and fastest Internet connections. The goal: Do homework faster. Instead, they pirated songs, as with Napster. UCLA got a huge amount of cease-and-desist letters. At one point, more than 80% of the network traffic was pirated materials.

Another example, the Tacoma Narrows bridge: They forgot about resonant frequencies, in this case from the wind. Every civil engineer gets an earful of this now.

1. Propagation of effects

Operating systems are digital, not analog. The smallest thing can make an operating system crash. We *think* we have some well-designed OS with boundaries between components. Effects may still cross boundaries and wreck things.

Example: A file system takes requests like a:\b\c and gives you the file:

```
cat a:\b\c
```

Somewhere else, we have an encoding for Japanese characters, which cannot fit in one byte (more than 256 chars).

ASCII has the top, first bit be 0. Japanese will have the top bit be 1, and have two bytes. This gives us 2^15 combinations, and so we can handle Japanese now.

Say we had:

```
cp a:\b\c d:[Japanese]
```

Giving him a message like:

```
"cannot create directory [Japanese]"
```

What happened? The top bit of the second byte had a 0, and the rest of it looked like a backslash. This happened with an early version of Windows. To fix this, Microsoft had to break down the boundary between the encoding and the file system and make things more complicated.

// "Shift JIS"? I can't read the encoding name

1. Trade-offs

We face all the previous problems, and we start trying to solve them, only to find that that we make other problems worse.

Example: sorting algorithms. Bubble sort is faster (than heap sort or mergesort for small-enough datasets, but doesn't scale the way they do.

1. COMPLEXITY

// Eggert actually wrote it in all-caps

This is the problem special to CS.

Consider Moore's Law: Complexity doubles every 18-24 months. (for the cheapest design point; it doesn't hold for more expensive CPUs, say). Originally, the law Gordon Moore formulated referred to the number of transistors on a chip. Eggert said Moore's Law is petering out, but he said that 10 years ago too.

Our computers are getting more complicated, but we designers are not.

Consider Kryder's Law: Disk drive capacity is also growing exponentially. Eggert's first home computer had a 1 GB hard drive in 1992 for $1,000.

We need to build systems that work, and that we *understand* despite their immense complexity. The biggest problem we're going to face is how to manage these enormously complicated objects, and make them work--well.

Lecture #2: 7 January 2015

(Very) Paranoid Word Counter

// Eggert's still trying for a larger room. // The chances of getting a larger room are higher now though

"Advisor Guides Obama into the Google Age" -- NY Times

Megan Smith is the CTO of the US. She was originally a senior executive at Google. When we she got the CTO job, she was issued a BlackBerry and some clunky Dell laptop. There's a culture clash there. Why is the US Gov't using such ancient tech?

Suppose we have very important secrets. They're useless if we can only sit on them. So, we need a way of processing this secret data for paranoid users.

"There's a lot of work being done in this area, which I cannot tell you about."

We're gonna use a toy app: wc (word count). Our words are seeecret, ullllltra secret, so we can't use any old wc, as there might be a backdoor in the wc.

"You gonna trust a bunch of South Africans to count your words?" (Canonical, the creators of Ubuntu, are based in South Africa).

Not only that, do we trust the shell? Do we trust the OS or the Linux kernel?

"There's Russian contributors to the kernel; we're not gonna trust it."

We're gonna write our own wc from scratch without a shell or a kernel. NSA's router firmware, for example, is this paranoid. We'll run on commodity hardware:

White box (built themselves) desktop: CPU: Intel Core i3-4160 (we're assuming we trust Intel) @ 3.6 GHz with a 3 MiB L3 cache and Intel HD integrated graphics RAM: 4 GiB dual-channel DDR3 SDRAM @ 1600 MHz HD: 1 TB hard drive, SATA, 7200 rpm (Are we assuming trust of the HD too?) Disk controller (lets us abstract the disk hardware as sectors)

Our UI: the power button. We turn it on, and a little bit later, a number appears on the screen.

"What did the customer forget to tell you?" Any questions on the spec?

Additional requirements:

- Run fast, relative to power-on
- File size from 0 to 1 TB
- Definition of a word: [A-Za-z]* (ASCII letters)

One of the most important questions: "What are possible future expansions?" We need to plan for future work and design for it *now*.

Possible later features:

- Count lines
- Edit the document

Boot Sequence

We're assuming a x86 processor in 32-bit mode.

Examining just the first 1 MiB of the physical RAM:

1. CPU powers on and sets the instruction pointer (ip) to 0xFFFF0 (that's 2^20 - 16, almost the very end of the 1st MiB of memory) And it begins executing.

Problem: We need to get the instructions there. RAM is voltile. Its contents are lost when power is lost.

Note: Even if we power off the computer, the SDRAM still has traces of its previous contents, some of which is readable. What we could do (like the olden days) is have a front panel with toggles for input. But that's difficult to use.

Solution: We use ROM (read-only memory) that survives power outage. Nowadays, the most popular form is EEPROM, whose contents are constant. Phoenix Software (from Phoenix, Arizona) specializes in this boot ROM software.

Now, we could put the wc in this ROM, and then we could put a jmp instruction after the boot program pointing to the wc. This prevents tampering, but it's expensive to commission a custom boot ROM.

Instead, we'll have the EEPROM-hosted software go to our hard drive and put our VPWC on the regular hard drive instead.

Say we put our WC at the start of the disk, and it occupies 40 KB. The EEPROM boot software will copy the wc to RAM and then execute it. However, our EEPROM is not yet generic, as it has to know that our wc is 40KB long, and it has to know where to place it in RAM. We need something more generic.

Solution: We can always add another level of indirection. Our EEPROM will always load some very small program. We'll assume that we always load the first sector (512 bytes) of the hard drisk into some specified location (0x7C00) in RAM, and then the CPU jumps to it. Our 512-byte program will then load the real wc into memory and then run it.

We're assuming that our disk can be viewed by software as a long list of sectors, so we don't have to worry about that.

This is *chain-loading*: a program loads a program that loads another program.

Let's see how this looks in the shell:

Some shell script with permissions 755:

```
#!/bin/bash

foo=bar
```

```
exec sed "s/foo/$foo/g"
```

The exec command means "discard the shell, and replace it with the program referred to by this command". It's nice to have this at the end, rather than have the shell spawn another process.

Back to the VPWC. We have reliability issues. We need a safety mechanism, a convention understood by the EEPROM's BIOS.

Let's talk about the **Master Boot Record (MBR)** layout for x86, 32-bit. This layout is ancient, but universally supported.

Let's look at that Sector 0 of 512 bytes on disk:

[x86 code | partition table | sig.]

The last 2 bytes are a signature: 0xAA55 == 0x55 0xAA (little Endian) If a disk does not have this signature, we'll assume it's not setup for our boot process.

Before them are 64 bytes, divided into four 16-byte sections that tell us about the layout of the disk, the **partition table**. For each partition, they'll tell us:

- The start sector
- Number of sectors
- Flags (bootable, etc.)

The preceding 446 bytes is x86 (machine) code that we jump right into and begin executing. If we can squeeze something in that can find our wc program and load it, then we're good.

In practice:

The BIOS (Basic Input Output System) in the EEPROM (electrically erasable programmable ROM) does: (When we select the bootup device, that's the BIOS.)

1. Hardware self-tests
2. Scans for devices via the hardware controller bus
3. Looks to see if a storage device is bootable by looking at the first sector
4. It reads and jumps into the code if so

Now, the MBR takes over and it:

1. It looks for a **volume boot record (VBR)** specific to a partition VBR is to partition as MBR is to the whole disk.
2. IT reads and loads the kernel

The kernel takes over and executes programs.

Our VPWC will do BIOS -> MBR -> wc

How hard disks work

Since we'll be reading from the HD, we need to learn how they work:

We have magnetic platters in a stack, with a reading head spindle stack that's very close to but not touching the drive. Our HDs are sealed, to keep dust out, but it's not a vacuum, so that our reading heads can float just above the platters.

7200 rpm -> 120 Hz Time to transit all around: 1/120 seconds -> 8.333 ms/rotation

"If you have perfect pitch, you'll be able to tell that you got what you paid for."

Suppose we want to read a random sector, we have to move the heads from the track they're on. There's a **seek time** to

move from one track to another. A typical delay is ~10 ms. Then once there, we have to wait for the disk to rotate till the disk comes around with the right sector. There's an average rotational latency, which is half of our worst-case: ~4.1666 ms So our total random read time latency is ~15 ms, regardless of the amount of data wanted.

How to read from disk

We could have our CPU handle all the disk-reading, etc., but that's impractical. Instead, we have a CPU attached to a bus which has DRAM and a hard disk attached indirectly via a controller. The controller has a hard-wired program that handles talking to the disk, and it has its own cache.

The CPU will send commands to the disk controller. We use special x86 instructions to specify our I/O wishes to the disk controller. Example:

```
inb 0x1F7   // in-byte
```

This gets a byte of data from the device with address 0x1F7, with the bus handling communication between the disk controller and the CPU.

```
outb 0x1F7  // out-byte
```

This puts a byte of data to the device. This was designed for when buses were 8 bits wide.

To read a sector from disk:

1. Look at location 0x1F7, the status register. Read it and see if the controller is ready. Do so in a loop.
2. Store into 0x1F2 the *number of sectors* (1-255 sectors) to be read
3. Store the *sector offset* into 0x1F3 through 0x1F6. The sector offset is a 32-bit number to tell the disk controller which group of sectors to read.
4. Write to 0x1F7 a bit pattern that means READ. The status register doubles as a command-sending register. The controller now becomes busy and waits on the disk, etc.
5. Read 0x1F7 and wait until the data has been read and the controller is ready Then, we get the results from the disk controller's cache and copy it into RAM

Finally, code!

```
// This thing's actually not called; it's jumped straight to
// via Bootloader
void main(void)
{
    // 40 KiB / 512-byte sectors 80 sectors
    for (int i = 0; i < 80; i++)
    {
        // read_ide_sector(the sector number + offset to avoid overwriting MBR, memory address in RAM)
        read_ide_sector(i + 100, 0x2000 + (i * 512));
    }

    // Jump to the first instruction
    goto *0x2000;
}

// Implementation using our previous pseudocode algorithm
void read_ide_sector(int s, int a)
{
```

```
    // The special x86 instruction, modeled as a function
    // We're checking the status register
    // We want to check the top two bits; if they're 01,
    // the disk controller is ready

    wait_for_ready();

    // Tell the controller that we want 1 sector
    outb(0x1F2, 1);

    // Tell the controller where the sector is, 8 bits at a time
    for (int i = 0; i < 4; i++) {
        outb(0x1F3 + i, (s >> (8 * i)) & 0xFF);
    }

    // Now, we tell the controller to read; 0x20 is the READ command
    outb(0x1F7, 0x20);

    // We wait for the disk controller to become ready again,
    // but we don't want to repeat code, so we'll define a subroutine
    wait_for_ready();

    // Once the disk controller is ready again, the data has been
    // read and is in the controller's cache. We need the insl direction
    // insl is implemented via asm() also; it's like inb
    // Copy 128 bytes at a time to the RAM address a via the 0x1F0 register,
    // which I assumes refers to the controller's cache
    insl(0x1F0, a, 128);

    // Data flows from the controller to the CPU and then to the RAM
}

// a for addresses
inline char inb(int a)
{
    // Inline assembly
    asm("inb ...");
}

void wait_for_ready (void)
{
    while ( (inb(0x1F7) & 0xC0) != 0x40 )
        continue; // Clearer than just the empty statement, and basically a NOP
}
```

"The only thing that's left to do is to write the program!

Here's the word counter:

```
// Also jumped straight to, so main's void return type is unimportant
void main(void)
{
    // 1 TB > 2^31; so use a 64-bit int
    long long int nwords = 0;

    // In case we are in the middle of a word at the end/start of a sector
    bool inword = false;

    int len;
    int s = 50000;
    do {
        char buf[513];
        buf[512] = 0;
```

```
        len = strlen(buf);

        // File is layed out sequentially in disk
        read_ide_sector(s++, (int)buf);

        // Count number of words in sector, as copied to buf
        // Let it know the length of the buffer, and whether it's
        // in the middle of a word
        nwords += cws(buf, len, &inword);
        }
    } while (len == 512);

    display_ans(nwords);
}
```

But wait! We're assuming the read succeeds always. That's not always true, e.g. the read request was for past the end of the disk. Also, we haven't accounted for if a file has a nullbyte in the middle of it.

We have yet to write cws() or display_ans()!

```
int cws(char* buf, int bufsize, bool* inword)
{
    // Number of words
    int w = 0;
    for (int i = 0; i < bufsize; i++) {
        // We can count the start OR the end of the words
        // See if we have a letter, and we're not already in a word

        // isalpha() won't be defined if the char is negative,
        // as chars are signed in C
        bool alpha = isalpha((unsigned char)bug[i]);

        // Single ampersand works too
        w += alpha & !*inword;
        *inword = isalpha(buf[i]);
    }

    return w;
}
```

To display the answer, we need yet another device. We talk to it differently, treating it like memory with a location of 0xB8000 with 2 bytes for location for our 80 x 25 pixel window.

[displayByte1 displayByte 2 | ...]

Part of the stored info is flags for font formatting, etc.

```
// Place cursor into middle of screen
char* screen = 0xB8000 + 200;
do {
    screen[0] = (nwords % 10) + '0';
    screen[1] = 7; // Gray on black; some control char
    screen -= 2;
} while ((nwords /= 10) != 0)
```

Discussion #1: 9 January 2015

Lab 1a Problem Description

Diyu Zhou is our TA for this section. The other TA for 111 (we have but two) is at 4 pm.

// The section is so full... // We need another TA for a better student/TA ratio!

1a should have input like this:

```
sort < a | cat b - | tr A-Z a-z > c
```

Becomes:

```
  sort < a \
|
  cata b - \
|
  tr A-Z a-z > c
```

A simple command is a sequence of words divided by spaces or tabs:

```
cat b -
```

Commands can be simple commands or subshells ending with:

```
cmd < word
cmd > word
cmd < word > word
```

Example:

```
cat < t.txt > out.txt
```

A pipeline has one or more commands separated by `|`:

```
cmd1 | cmd2 < t.txt > out.txt
```

A complete command has one or more pipelines separated by semicolon(s) or newline.

```
cmd1 | cmd2 > out.txt ; cmd3 | cmd4
cmd5 | cmd6 | cmd7
```

Compound commands are the if-then-else-fi, while-do-done, and until-do-done blocks:

```
if A
then
    B
fi

if A
then
    B
else
    C
fi
```

```
while A
do
    B
done

until A
do
    B
done
```

Where A, B, and C are complete command (possibly pipelined, delimited by newlines or semicolons). Note that A, B, C can also be compound commands, i.e. we can nest conditionals and loops.

Subshells has complete commands wrapped in parantheses:

```
(A)
```

Where A is a complete command (could be multiple commands with pipes and semicolons, etc.)

And of course, ignore comments starting with `#` till the next newline. It doesn't look like we need to support the backslash escape for multiline commments

Skeleton Code for 1a

General algorithm:

1. Read the command from the shell script.
2. Parse the command to check the syntax of the command.
3. Store the validated command to an internal representation The array of 3-ary trees.
4. Print the stored commands in a standard format (with the -t option, as already handled

test-t-bad.sh and test-t-ok.sh are for Part 1a only, I think, with the error and okay cases, I think. We are allowed to add our own .c and .h files, but we must update the Makefile to include them.

alloc.c

static functions are restricted to the file in which they are declared/defined.

memory_exhausted() will tell us if memory is exhausted.

check_nonnull() is for when we're checking the return value of malloc() and realloc() function for null (e.g. failure on allocate).

The (size ? size : 1) arg has malloc()/realloc() allocate the size if size > or allocate 1 if size is 0 (so that we get at least 1 byte). Also, size is of type size_t, so it won't be negative.

checked_malloc() and checked_realloc() are juste the wrappers for malloc() and realloc() with these clean checks for memory exhaustion failures with them.

checked_grow_alloc() has:

```
size_t max = -1;
if (*size == max) {
    memory_exhausted(0);
}
*size = *size < (max / 2) ? 2 * (*size) : max;
return checked_realloc(ptr, *size);
```

size_t is an unsigned tye, so -1 is all 1's in the binary representation.

We're checking to see if the size is equal to the max (and we can't assign a value larger than what size_t supports without a compiler error, so we don't need to worry about that).

Fucking Eggert. Translated code:

```
if (*size < (max/2)) {
    *size *= 2;
} else {
    *size = max;
}
```

The thing here is how much to allocate. Either we ask for less than half of the max, and so we allocate twice that, or we asked for more than half of max, and so we allocate max.

alloc.h

We need `<stddef.h>` for size_t.

It just declares the previously defined functions.

main.c

usage() is a static function that gives the user an error message reminding them how to use the program.

get_next_byte() gets the next byte from a stream; it's a wrapper for getc()

The usage of our shell looks like this:

```
profsh -p logFile.txt -t shellScript.sh
```

read_command.c

This is where we'll do most of our work. How we implement the command_stream is up to us.

How about we process one line at a time? We'll have auxiliary functions specific to each command type (they may need helpers too). We'll look for their tell-tale signs (e.g. a pipe character), and then call their respective functions. These functions can call each other as needed. We're breaking the commands apart. Interestingly, the pipe command is associative:

```
A       B|C|D
A|B     C|D
A|B|C   D
```

should all give us the same result.

Each command_t will store information about the command it holds. We just need to parse the scripts well, so that we can have all this info. For the input/output streams in the command_t, they'll point to the input from a previous command, and afterward, the output that will be passed to the next command.

test-t-bad and test-t-ok

test-t-bad.sh has a bunch of syntax errors: our shell should give us an error for each and every single one of of them. It will tell us if our shell succeeded rather than gave an error.

We need to add our own test cases here.

test-t-ok.sh has a bunch of cases with correct syntax. It adds a bunch of test cases into a test.sh file. It will also create a test.exp file that has the expected output from profsh -t, and then compare that with the output found in test.out generated from test.sh.

To add our own test cases, add our test case before the first EOF, and then the expected output before the second EOF.

print_command.c

This just handles the standard output for each command type. The indent argument indicates how many spaces to print before the command. The indent + 2 is because we have moved to the next indent level.

The %* in

```
printf("%*s %s \n")
```

is a formatting specifier for printf.

Something like:

```
%5s
```

Means a width of 5. If we pass in a 3-char string, we get 2 spaces before the 3 chars to make them fit.

Here, we're passing in indent (the star lets us pass in something), instead of specifying 5. This lets us implement some of the indenting.

The printing is recursive. It goes through all the commands in each command's 3-ary tree recursively. This is a good idea for Part 1b.

Discussion with Taqi

"Your life this quarter may be miserable."

"No weekends."

Our goal with Lab 1 is to build a shell that reduces the *terminal time* (the time from input to desired output). To start, we should identify the problems with the original shell.

test.sh: the input shell script for parsing

test.out: the actual output after parsing

test.exp: the expected output after parsing

test.err: difference between expected output and actual output

"Do not give up."

// Email the TAs if we fix everything, even // when late. For example, if 1a was screwed up // at first, submit an updated solution with // the fixed 1a code.

"The assignment is (50%) done once you understand it."

// The TA wants us to struggle like hell. // Everyone's asking him how to get started // (and is probably still confused on the code) // and he's being real darn cryptic about it.

"Use a data structure."

He's saying that our parsing is different. We have to look ahead at how the code behaves. We might have to read a whole line more, etc. to make a decision about something.

// If any grading screws up, it's probably // a problem with the grader's difference in // environment

Lecture #3: 12 January 2015

// Eggert is no longer optimistic for getting a larger room

Improvements to VPWC

Let's look at making some changes to our program:

- Performance improvement

Double buffering

Consider the timeline of our current implementation: It reads, asking the disk controller to send data back, and waits until the data is available, and the processing time is very litte. But notice the delays:

read: --- --- ---

process: - -

We want to overlap I/O and processing: we want to process the first sector while reading the second sector, etc. We read all the time and process the previous sector while we read the current one.

read: --- --- ---

process: --- ---

This technique is called **double buffering**. We'll need two buffers at any time: the buffer being read into with the current sector, and the buffer holding the data being processed. They alternate between being read into and processed. i

Consider a case where processing takes a long time, as with decryption:

read_encrypted: --- ---

process/decrypt: -------- ---------

But again, we can overlap. Double buffering works best when reading and processing takes about the same amount of time, giving us a ~double performance improvement.

read_encrypted: --- --- --- process/decrypt: ---------- ----------

Increase buffer size

We can also improve our performance by reading more than one sector at once. Rather than waiting for a sector to come back around with each rotation, we shrink the latency associated with reading.

DMA: Direct Memory Access

So far, we've been doing PIO (programming I/O), with the CPU talking to RAM and the disk controller.

```
CPU
|
-------------
```

```
|          |
RAM        Disk controller
```

The bytes are going through the CPU, and so they cross the bus twice as part of the `insl` instruction.

Instead, we can have the disk controller communicate directly with the RAM. We send more complicated instructions: where to read it (from hard disk), but also where to place it (in RAM) as it's read. Then, later, the CPU can ask the controller when it's done, and then it can process the contents of RAM.

Problems with Growing Our Program

We can think of other improvements, but the point is that we want to make changes to our program.

Consider the parts that read data from disk:

- BIOS (in EEPROM) has its own read_ide_sector()
- Bootloader (in MBR) has its own read_ide_sector()
- VSWC (on disk somewhere) has its own read_ide_sector() for reading in words and counting them

In practice, updating the BIOS copy doesn't matter too much, but ideally, we'd like just one read_ide_sector(), so that any improvements we make translate to all of them.

1. One solution: Make the BIOS store a copy of read_ide_sector at some address 0x1ffc800 by convention. In the olden days, the BIOS was treated as subroutine library so that our programs could fit into memory. We still have some informal industry standardization as to what functions BIOS have.

Limitation: We're stuck with the BIOS as a basis for writing applications. It's not enough for general purpose OS', for example: new devices, like 5K displays, etc., are not supported. It'd be way too much of a pain to try to make changes to the BIOS.

We need to free our applications from the BIOS, such that we can forget about what's in there.

1. Another solution: place our suped-upp read_ide_sector() in the MBR, and our VPWC will call it. We've freed ourselves from the BIOS.

Problem: the MBR is puny, at just 512 bytes.

We're running into a fundamental problem: bugginess increases with complexity.

It's very difficult, however, with our current setup to catch bugs and recover from them, i.e. **fault tolerance and recovery**.

Finally, we're gonna have a hell of a time (not a good one) trying to run several apps simultaneously. For example, what if we have two programs that both need read_ide_sector()? They'll collide with each other: data may go to the wrong program, keep the CPU eternally busy from a bad deadlock cycle, etc.

Summary of problems of using BIOS as a basis for writing apps:

1. Not enough for new devices
2. Too much of a pain to change
3. Too much of a pain to recover from faults.
4. Too much of a pain to run several apps simultaneously

Modularity and Abstraction

Two design approaches come to the rescue:

Modularity breaks solutions down into smaller, manageable pieces. We get entaglement between many things otherwise.

Consider the effect of modularity on finding bugs. Often, we spend more time fixing bugs than we do writing new code.

Assume N lines of code (SLOC). Assume we have K modules, and that the number of bugs is proportional to N (in reality, the relationship is worse than linear)., and that the time to find and fix a bug is also proportional to N (looking in bigger and bigger modules).

In the case of no modules (K=1), the debug time is proportional to number of bugs * time to fix a bug: N * N = N^2

But suppose we have K modules, with bugs evenly distributed between the modules. Then, the debug time is (N/K)(N/K) * K modules = N^2/K.

Problematic assumptions: The bugs are not localized to one module, more lines of code are needed to glue together modules, we may need more time to find a bug with the (inherently more complicated, modularized code).

Abstraction is modularity with "nice" boundaries that let us see our problem in a simple picture. This is what makes modularity powerful.

"Good" vs. "bad" modularity: some metrics

1. Performance

Modularity should help us make speed improvements that ripple through the whole system. Unfortunately though, modularity boundaries normally inherently hurt performance, so there's a trade-off. Why? Because there will be runtime cost of linking together modules (e.g. call & return), but there will also be secrets. Different parts will know different things, and we won't be able to optimize as much as we could.

Negative impacts of modularity on performance:

- Boundary code itself
- Secrets

1. Robustness

How well are errors/failures isolated to the failing module? We're controlling propagation of effects here.

1. Lack of assumptions, a.k.a flexbility or neutrality

We want our modules to be general, such that we can hook our modules together easily. They should avoid making assumptions about the rest of the application. Otherwise, we'll be restricted by the constraints they place on us.

1. Simplicity

The modularity should be:

- Easy to learn
- Easy to use

We may be quite willing to give up performance and robustness for these.

Improving read_ide_sector()

Previously: read a sector `s` into a memory address `a`:

```
void read_ide_sector(int s, int a);
```

This is good performance-wise, so let's improve our modularity boundary. For example, supporting multiple disks (that may not all be IDE):

```
void read_sector(int diskno, int sector, int addr);
```

We've improved flexibility, but what about robustness? We can return the status (e.g. succeeded or failed)>

```
int read_sector(int diskno, int sector, int addr);
```

To read in multiple sectors, we can do:

```
int read_sector(int diskno, int sector, int addr, int numSectors);
```

As we're limited at the hardware level for how many sectors we can read: we can have the return int be the number of sectors read, or be negative if error.

Another problem: we're assuming that the sectors are 512 bytes in size. To handle this, we can just specify a byte_offset from the start of disk, e.g. start reading here.

```
int read_bytes(int diskno, int byteoffset, int addr, int nbytes);
```

Compare this to the POSIX read() system call:

```
// I despite single-char variable names

ssize_t read(int fd, void* a, size_t nbytes);
```

We can now read into any buffer pointed by a, which accounts for the problem that on many 64-bit systems, `sizeof(int) != sizeof(void*)`

size_t is 8 bytes wide and unsigned, useful for sizes.

ssize_t is a signed type for holding return values, including negatives if there's a failure.

fd is a file descriptor, an abstraction that can talk about many things. Notice also that we no longer have a byte_offset, so we're less general. We actually a separate system call, p_read, that has an `off_t o` argument too.

Why is read() more popular than p_read()? Because it's easier to use!

"I'm not getting enough scribe notes volunteers. And if continues, I'll begin volunteering you."

How to implement modularity

1. None (just like "Failures? Forget about them!")

This would mean global variables and spaghetti code in one massive main routine.

- Fast
- Hard as hell to understand, debug, or change

1. Function call modularity

We break up our program into pieces called functions, and we'll use them by "calling"s them.

In memory we have a read-only part called `text`. Next to that, we'll have a read/write portion with variables we intend to modify called `data`

Putting them next to each other let us easily copy instructions and data from disk into memory.

Then, we'll have a read-write area, initially zeros, `bss` (block-save storage?), that doesn't have to be stored on disk. Instead, we just store on disk how much bss we want.

Down below, we have the stack, with an unused area in between.

```
[ text ]
[ data ]
[ bss  ]
[ heap ]

[ unused ]

[ stack ]
```

The stack pointer keeps track of where we are in the stack.

How does the `new` operator work? There's a `break`, and a system call `sbrk()` that works like this: `sbrk(1024)`, which adds 1024 to the break pointer, etc.

We won't discuss the heap until CS 131, but we will review CS 33's discussion of the stack.

This is how we implement function call modularity.

Recursive function example

This is both caller and callee, with recursion as well!

```
int fact(int n) {
    return n ? n * fact(n - 1) : 1;
}
```

What happens if n < 0? Infinite loop that grows the stack until it collides with the brk, overwrites our program, and then hits the read-only portion.

Computing the factorial this way is stupid. Use a loop instead! Even 20! will approach overflow though. In fact, GCC (at its default optimization level), will change this to the iterative version:

```
gcc -S -O2 fact.c
```

To get the unoptimized version with the recursion:

```
gcc -S -O0 fact.c
```

This gets us:

```
fact:   pushl %ebp
        movl  $1, %eax        // Default return value of 1
        movl  %esp, %ebp
        subl  $8, %esp
        movl  %ebx, -4(%ebp)
```

```
        movl  8(%ebp), %ebx   // The argument n
                              // placed here by convention
        testl %ebx, %ebx
        jne   .L5

.L1:    movl  -4(%ebp), %ebx  // Restoring the registers
        movl  %ebp, %esp
        popl  %ebp
        ret
```

If %ebx is 0 (%ebx was not non-zero), we'll return one.

```
.L5     leal  -1(%ebx), %eax  // Set %eax = %ebx - 1
                              // We're treating ints like addresses
        movl  %eax, (%esp)
        call  fact            // We'll eventually get the answer in %eax
        imull %ebx, %eax      // The actual multiplication
        jmp   .L1
```

Use `si` to step through source code instructions in GDB.

There are plenty of inefficiencies here. Jumps and conditional jumps especially suck. CPUs are constantly making predictions for those, and they won't always guess right.

Say we had an extra instruction, a bug:

```
 fact:  pushl %ebp
        movl  $1, %eax        // Default return value of 1
        movl  %esp, %ebp
        subl  $8, %esp
        movl  %esp, 100(%esp) // <-- BUG
        movl  %ebx, -4(%ebp)
        movl  8(%ebp), %ebx   // The argument n
                              // placed here by convention
        testl %ebx, %ebx
        jne   .L5
```

This bug means we'll be modifying some caller's caller's values.

"We are messing with the caller's brain, and the caller won't know." The callee can do whatever it likes with the caller."

The converse is true too:

```
movl  $0, %esp
jump  fact
```

We're gonna get a negative stack pointer in the fact frame. Bugs here are not isolated to the modules; callers and callees can mess with each other.

Say we have a bug just within the callee:

```
.L2:
.L5: jmp .L2
```

"Haha, I'm not gonna return!"

Soft vs. Hard Modularity

In **soft modularity**, the components agree to conventions that are not enforced anything/anyone. Civilization goes to hell the minute anyone messes up.

We want **hard modularity** that guarantees that failures are isolated to specific modules.

Two ways (of many) to achieve the latter:

1. Multiple machines with message-passing (client/server model)

Say we're suspicious of a piece of code. We can place it on a completely different machine, and have it send back results to us. We'll need sendmsg() and receivemsg() primitives. If the code crashes or goes into an infinite loop, we'll simply move on. The biggest downside? IT'S SLOOOOW.

"It takes *milliseconds*! We don't really have the time for that. We want something faster."

1. Virtualization

We have but one real machine, but we pretend to have other, virtual machines that are heavily isolated from the real machine. The VM sends special virtual instructions that requests the real machine perform real actions for it. As the real machine is in charge of executing the instructions, the virtual machine contains any disasters happening in it.

Lecture #4: 14 January 2015

// START LESLIE'S NOTES // Reorganized by me

Hard modularity

Two main models for achieving this:

- Client-server

Here, one computer acts as client, and one as the server. The communication link (network) between them acts as the modularity enforcer.

Virtualization

The simplest approach:

1. Write a simulator of x86' architecture

It'll have a subset of real machine memory, and a subset of real machine instructions, plus new instructions just for the virtual environment.

e.g.: inb() becomes read_system()

An example of a simulator: QEMU

Biggest con: It's very slow (usually by a factor of 10)

1. Hardware assistance for fast simulators of x86 on x86 or x86-64

We'll have a subset of the instruction set, and partition them into 2 categories:

- Dangerous instructions - PRIVILEGED
    - Instructions that access the outside world
    - These will fail

- Safe instructions - UNPRIVILEGED

Hardware has 2 modes 1. PRIVILEGED: can run anything 2. UNPRIVILEGED: can run only unprivileged instructions

To make new instructions:

- By convention, if an application runs a certain unprivileged instruction (INT: an interrupt), we will treat it as meaning they want to run a privileged instruction.

We execute a **protected transfer of control**:

```
-> Will trap. Trap vector contains pointer to code
    to execute when trap occurs (in kernel).
    Will switch to privileged mode when trap occurs
-> Kernel code is trusted code
-> INT:
    pushes ss (stack system)
    pushes esp (stack pointer)
    pushes eflags (eg. what mode)
    pushes cs (code segment)
    pushes eip (instruction pointer)
    pushes error code
    eip leap [0x80]

Linux convention:
    - syscall# = %eax
    - args1... = %ebx, %ecx, %edx, %esi, %edi, %edp

CALL -> RET
CALL -> RETI (return from interrupt)
(pops everything and returns to application)
```

// END LESLIE'S NOTES

SYSENTER & SYSEXIT

Recent versions of the Linux kernel include the

SYSENTER and SYSEXIT system calls.

SYSENTER sets cs, eip, ss, esp to values in machine-specific privileges register.

SYSEXIT does the opposite.

We switch to privileged mode. It acts like a trap but doesn't push everything onto the stack; instead, it's all in registers, so it's faster. This is one way of speeding up system calls.

VDSO

But even this isn't fast enough.

Let's look a at simple example:

```
pid_t getpid(void);
```

In traditional Linux, it's implemented as

```
pid_t getpid(void)
{
```

```
    asm("movl %49, %eax int 0x80")
}
```

This is an interruption, which is rather slow.

A simpler way of handling this is just placing the process ID in the user memory, and then retrieving it later:

```
return __process_id__;
```

Where:

```
int __process_id__
```

Was declared at the top level.

Another idea: VDSO: virtual dynamically linked shared object

The kernel dynamically links in these easy, efficient implementations of system calls, rather than the traditional trap system.

An example:

```
ldd /bin/sh
```

This runs the dynamic linker on a program, but does not run the program. It figures out which shared libraries will be needed by the program. The output will tell us things like:

```
linux--vdso.so.1 => (-0x7744000)
```

This tells us that this implementation will have us jumping into the kernel and then returning, (call + return), faster than trapping or sysenter/sysexit.

We can see:

Applications --|--\ ..... | \ C library -----| \ ..... | \ System call | VDSO's ..... | | Kernel code | | ..... V V [privileged | non-privileged] Kernel ..... Hardware

The kernel code walls off the privileged instructions, and can also execute the non-privileged instructions. Applications cannot directly execute the privileged instructions. It provides an abstract level, the system calls, which is used by the C library, which are in turn used by applications.

This is the traditional layers.

We can also build kernels atop kernels, e.g. a higher-level kernel, etc. This is generalizing.

There is hardware support for this multiple layers scheme.

x86 has 4 levels of privilege. Thus, it has 2 privilege bits, with possible settings: 00, 01, 10, 11

There are some operating systems that work this way.

1. Apps
2. Daemons (OS administration)
3. File systems (cannot tamper with memory management)
4. Memory management
5. Hardware

This is better, hard modularity. Sometimes, this is drawn as a ring diagram, with concentric circles.

Linux could have been designed this way, but it's not.

It has Levels 0, 3, and nothing else. It's just one big kernel. Why this choice? It's faster! We'd have to pass 3 different protection domains otherwise. It's not just CPU overhead, it's intellectual overhead too.

There was a spirited disucssion between Andrew S. Tannenbaum and Linus Torvald. Tannenbaum argued that a monolithic kernel was a mistake; he favored a microkernel approach.

All of these are mechanisms for virtualization.

Processes

A process is a program in execution on an isolated domain. It's running, but not in control of the whole machine. It's a high-level construction that we can manipulate via system calls.

Simplest:

```
pid_t getpid(void)
```

The pid_t might be implemented like this:

```
/usr/include/sys/types.h
typedef int pid_t;
```

Process Creation

There's a system call named:

```
pid_t fork(void);
```

This is the traditional call for creating a process. It "clones" the current process, and returns an indication of how well that went.

It returns 0 in child (they're not exactly clones) in its %eax register and it returns the child PID in parent.

It's up to the application to figure out what to do with the child process.

In the event of failure, it returns -1.

Say we had:

```
int main(void)
{
    for (int i = 0; i < 3; i++) {
        fork();
    }

    return 0;
}
```

We're gonna end up with 8 processes (2^3). Each child forks, so we get grandchildren.

1st fork: parent + 1st child

Each one executes a fork() again, so we have:

2nd fork: (parent + 2nd child) + (1st child + 1st grandchild)

And now each executes yet another fork():

3rd fork: (parent + 3rd child) + (1st child + 2nd grandchild) + (2nd child + 3rd grandchild) + (1st grandchild + 1st great-grandchild)

This is called a **fork bomb**. Because of 111 students trying this fork bomb on the SEASnet servers, the sys admins added a limit of like 30 processes. Previously, it filled up all the process table slots.

We could also have exponential growth:

```
while (fork() != -1)
    continue;
```

This is like a virus. As some processes die, they make room for other processes to be made. The infection will refuse to subside.

How is the child different from the parent?

1. The return value of fork()
2. Different PIDs
3. getppid() (get parent process ID) will return a different result
4. They have different set of file descriptors that start off the same, but are in different pieces of memory They'll have different tables pointing to the same things. The child closing a file descriptor does not affect one which the parent points to.
5. Accumulated execution times. The child begins with 0 (it doesn't inherit the parent's)
6. File locks. If the parent locks a file, the child does NOT inherit.
7. Pending signals. Say the Control-C signal reaches the parent just as the child is made; the child will keep going.
8. Etc.

Process Replacement

This is a rather boring way of creating processes that is so much like bacterial multiplication.

However, the next system call we'll look at doesn't create processes; it replaces them. It destroys the currently running program and replaces it.

"Someone's in a house and kaboom! They're dead. Someone else moves in."

This call is named:

```
int execvp(char const* file, char* const* argv);
```

Pointer to constant char [actually an array of chars] Pointer to constant pointer to char [actually an array of pointers; Walk through until we hit a nullptr.

The second arg is seen as:

```
char* const argv[]
```

Why not a const char for second one? Complicated historical reasons.

This is a brain transplant. `file` will have the replacement process (I believe this is because it's supposed to point to the filename to run). argv will hold the argument list (this is in fact how the commandline argument list is implemented; argv to argv!).

execvp always returns -1 and it always sets errno (global error variable). Why? Because if it ever returns, it didn't blow away the existing program, so it failed. There may have been an nonexistent file, I/O error, etc. If it doesn't return, that's proper, because we just replaced the current process, as intended.

The parent will notice if its child is replaced. All the process metadata remains the same. If a program is the first to run, its getppid() result is 0 or 1, usually, and OS-dependent.

Process Destruction

One approach:

```
void exit(int);
```

This cleans out I/O buffers, for example, and then exits the program such that it is no longer running. This is a problem from the OS-level point of view.

Instead:

```
void _exit(int);
```

It just exits. It doesn't execute any more instructions The argument is the exit status. 0 means success; any other number means failure, and should be 0-255 (don't use -1, as that's not portable).

This doesn't actually destroy the process object. It still occupies memory and what-not. To actually destroy a process, we'll need:

```
pid_t waitpid(pid_t pid, int* status, int flags);
```

It waits for the process PID to finish, fills in the STATUS, and returns the PID of the process it terminated. Usually, a parent calls this on a child. When waitpid() succeeds, the child process is actually gone. FLAGS just specifies things like how patient it should be with the child.

Thus a process can be in multiple states:

1. Runnable
2. Exited

"Exited" means that _exit() has been called, but it's waiting around for someone to call waitpid() on it.

There's a command called `date` under Unix:

```
Wed Jan 14 13:34:03 PST 2015
```

And say it's here via `which date`:

```
/usr/bin/date
```

We give it the -u option, which means use Universal Time Coordinated, which 8 hours ahead of PT.

Say we're lazy and we want to write a function that uses `date`. How then we do invoke date? execvp!

```
// Return whether we could print the date
bool printdate(void)
{
    // But we're gonna lose this program if we call
    // just execvp; this process' brains will be replaced

    // Solution: Have a child do it!
    pid_t p = fork()

    if (p < 0) {
        return false;
    }

    if (p == 0) {
        // Array of pointers to chars; the first arg is the name
        // of the program, which is argv[0], its name
        // We can lie to our children about their names!
        execvp("/usr/bin/date", (char*[]) {"date", "-u", 0})

        exit(127);
    }

    // If we're here, we must be in the parent, as
    // we either failed or the fork() was successful
    // So we can destroy the process
    int status;
    if (waitpid(p, &status, 0) < 0) {
        return false;
    }

    // We want to know that the program actually
    // exited and that its status was 0
    return (WIFEXITED(status) && WEXITSTATUS(status) == 0);

    // These last two are from wait.h (standard library)
}
```

We can lie to our children about their names!.

All of this is rather complicated. What if we had just one primitive (rather than fork() and execvp())? What if we could do both at once?

The problem with fork() is that the memory is copied too. That's a lot of expensive copying and memory usage. So there is a system call in Linux that does both fork() and exec(). That's:

```
// Spawn a process
int posix_spawnp(pid_t* restrict pid,
                 char const* restrict file
                 posixspawn_actions_t* file_ads
                 posixspawn_attr_t* restrict attr_t attpp
                 argv,
                 envp);
```

`restrict` is a keyword in C that means the compiler can assume that a pointer points to something that NOBODY else points to. No aliasing games allowed. Otherwise, the kernel can do whatever it wants to the program.

The third argument lets us do things ahead of the spawning, like close(0) (close stdin). With posix_spawnp, we're listing out what we want done before the child begins running.

This is fork, everything to be done before a exec, and then the exec. This is more efficient, but far more complec. envp is the environment, modeled as an array of strings.

Discussion #2: 16 January 2015

Hints on Lab 1b

We're implementing a standard execution model for the shell.

This is when our bash really becomes bash.

execlp

System called used to implement simple command

```
int execlp(const char* file, const char* arg, ...)
```

file: command to execute (can be relative or absolute; can just "ls" rather than "/bin/ls")

arg and subsequent arguments: arguments passed into file

- First argument should be filename
- Last argument should be NULL

Example: ls -a -l

```
execlp("ls", "ls", "-a", "-l", (char*)NULL)
```

The first is the path to the file to execute; the second is the name to give to the program.

Example where they're different (as with Hombrew's g-prepends), as with a different location outside of standard $PATH:

```
tar -c -v -f tarball.tar.gz dir/
execlp("/local/bin/gtar", "tar" "-c" "-v" -"f" "tarball.tar.gz", "dir/", (char*)NULL)
```

See linux.die.net/man/3/exec for the man page

All the exec functions are similar except for differences in the arguments. See the rest at the man page.

fork

execlp() replaces the calling process image with a new process image.

Example:

```
int main()
{
    execlp("ls", "ls", "-a", "-l", (char*)NULL);
```

```
    return 0;
}
```

`return 0` is actually not executed; as we're replacing this process with a different one altogether.

Use fork() to generate another process to execute execlp(); have the child be replaced, not the parent.

Return 0 for child process, PID for parent process, and negative values for errors. See Minilab 1.

See the man page: linux.die.net/man/2/fork

Example fork usage:

```
int main(void)
{
    pid_t pid = fork();

    if (pid == 0)
    {
        printf("Child process!\n");
    }
    else if (pid > 0)
    {
        printf("Parent process!\n");
    }
    else // Fork failed
    {
        printf("\nFork failed, quitting.\n");
        return -1;
    }

    return 0;
}
```

The child and parent processes will have different PIDs.

fork() traps into the OS, which copioes current process state to generate a clone of of the process.

The return value i ssassigned ifferently

Sequence command: ls -a -l; top;

```
fork() Child: execlp("ls", "ls", "-a", "-l", (char*)NULL);
parent: wait() or waitpid()

for() Child: execlp("top", "top", (char*)NULL);
parent: waiti() or waitpid()
```

The parent waits on the child to finish running. We fork a child, which does the execlp() for "ls", and the parent waits in the meanwhile.

Then, the parent does a second fork, and that child calls execlp() for "top", and again, the parent waits until it's done.

dup2

```
int dup2(int oldfd, int newfd);
```

This can be used for input/output redirection

After succesfully executing dup2, the old and new file descriptors can be used interchangeably.

Example: grep good < test.in

```
int fd1 = open("test.in", O_RDWR); // SEE documentation!
dup2(fd1, 0); // The 0 means the file descriptor of stdin
```

If we add a scanf here, scanf will read from test.in

```
execlp("grep good", "grep", "good", (char*)NULL);
```

0: stdin 1: stdout 2: stderrs

We'll have to figure out the redirection ourselves.

pipe

```
int pipe(int pipefd[2]);
```

pipefd[0]: file descriptor of the read end pipefd[1]: file descriptor of the write end

Combine this with dup2() tio implement pipe commands.

Example: cat test.in | grep "keyword"

Pipe(), then fork(). Child executes while redirecting stdout to write end Parent: fork() if commands remaining, and redirect to read end of pipe

See the man page:

```
linux.die.net/man/2/pipe
```

This example:

The parent writes the string contained into the program's commandline argument to the pipe, and the child reads this string a byte at a time from the pipe and echoes it on stdout.

```
int pipfd[2];
pid_t cpid;
char buf;

if (argc != 2)
{
    fprintf(stderr, "Useage: ...");
    exit(EXIT_FAILURE):
}

if (pipe(pipfd) == -1)
{}
    perror("pipe");
    exit(1);
}
cpid = fork();
if (cpid == 0)
```

```
{
    // Close unused write end; parent closing
    // file does not affect child
    close(pipefd[1]);
    while (read(pipefd[0], &buf, 1) > 0)
        write(STDOUT_FILENO, &buf, 1);

    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);
    _exit(0);
}
else
{
    // Close unused read end
    // Parent writes argv[1] to pipe
    close(pipefd[0]);
    write(pipefd[1], argv[1]), strlen(argv[1]));

    // Reader will see EOF
    close(pipefd[1]);

    // Wait for child
    wait(NULL);

    exit(0);
}
```

Execution order of child and parent not guaranteed?

TRICKY: cat < file | sort

if, until, and while commands

Counterintuitive: 0 means success in bash

Where to start: execute_command() in execute-command.c

Recommended to use recursive calls similar to command_indented_print()

WeensyOS 1

We'll basdically implement sys_fork(), sys_wait(), sys_exit(), and answer several questions.

Files overview:

- bootstart.S, mpos-boot.c: boot loader
- lib.c: basic C memory functions (memcpy, memmove, memset, strlen, strnlen), and printf equivalent with console_printf (use this for debugging)
- mkbootdisk.c: Code to make the iimmage, not part of the OS code
- mpos-app.c and mpos-app2.c: user-level apps (we'll implement syscalls for them)
- mpos-int.S: low-level interrupt-handling code
- mpos-kern.c: kernel code to implement system calls
- mpos-loader.c : ELF loader
- mpos-x86.c: x86-dependent code

The amount of code is much less in comparison to Lab 1a.

Make: mkbootdisk.c -> mkbootdisk

```
bootstart.S, mpos-boot.c : bootloader

Setup initial system state and then load kernel_image

rest -> kernel_image

Generate mpos.img that contains bootloader and kernel_image

After BIOS initializes, it loads the first 512-byte sector
fo the hard disk into physical addresses 0x7C00-0x7DFF

mkbootdisk puts the bootloader at the first 512-yte sector
of the hard disk, and control has been passed from BIOS to
bootloader

void run(process_t* proc);
```

We pass the pointer to the process we want to run.

NOTE: `process_t` has the `registers_t` struct as a member; we'll need that for forking, etc.

Register state is already saved, but memory state is not. That's what we'll need to implement.

In a fork(), copy the stack, and point the child's instruction pointer to point into it.

The volatile int checker checks for stack corruption. Note that the child and parent have different checkers.

Lecture #5: 19 January 2015

Orthogonality

Eggert's car in the shop today. One issue he had was hot air blasting him in the face each time he turned the wheel leftward. This was a failure of orthogonality.

In math, we have our orthogonal standard axes. Along any one of the axes, the choice of coordinates along the other axes is irrelevant.

OS' are far more complicated than cars.

We need to be able to support **separation of concerns** in an immensely complicated system, where we *need* to able to understand one piece without having to understand the thousands of other pieces.

Here are our first few major "axes" in an OS:

Access to files (and devices)

The UNIX model has a small set of primitives for these:

```
open()
close()
read()
write()
```

We can further split up our axes into sub-axes for additional orthogonality.

Access to devices is sometimes made a different axis, with primitives like connect(), disconnect(), send(), and receive().

In Linux, however, devices are abstracted like files. We'll use the same primitives, with the same behavior, regardless of whether it's a file or device.

Example:

```
open("/dev/sd/05", O_RDONLY) // Reading a risk
```

We're opening a device, a disk (numbered 5 on this system), and we're seeing its (read-only) data at a very low level.

Example:

```
open("/dev/tty/07", O_RDONLY) // Reading a serial port
```

We can read off of everything from USB drives to networks to keyboards to mice. They're *all* modeled as files, and all the system calls for files work on devices too. This leads to simplicity.

Example: copy data from disk to disk

```
cp /dev/sd/01 /dev/sd/02
```

`cp` is implemented with standard libraries and system calls. This is an orthogonal design that allows for generic system calls.

Problems: We could easily copy and overwrite the boot drive, e.g.

An illustrating example:

```
cp /dev/tty/03 transcript
```

Copying bytes from a serial device into a regular file.

Problem: This process never finishes! We're just waiting for the device to send more bytes. There is no EOF; EOF is a concept that makes no sense for devices.

We're seeing the differences between regular files and devices.

Device Types:

1. Network/serial

- The data is generated spontaneously
- A stream of data (we can't go back and look unless stored)
- The stream is infinite (in principle)

1. Storage/disk

- Request/response paradigm We ask the storage device for data
- Random access to data
- The amount of storage is finite

We want our Linux system calls to work despite the differences between the underlying devices. In the POSIX interface, we do this by having two different system-calls:

```
read(fd, buf, size)
```

This allows for stream-reading and storage access. The size parameter is what allows a finite amount.

BUT, read() can hang indefinitely while waiting for the device. For robust, portable programs, we need to account for this.

In addition, have a another syscall:

```
// seektype: Seek from start, end or current position
lseek(fd, offset, seektype)
```

With a data stream, lseek() fails. It's meant for random access on storage devices, not seeking on a stream-oriented device. We'd get a -1 error and errno is set otherwise to indicate "not seekable".

We now have a set of syscalls that work on devices, and a superset that works on seekable devices and files. We've lost some orthogonality, however.

Access to Processes

Here's another major axis, with primitives like:

```
fork()
waitpid()
_exit()
```

Guiding principles: generality and orthogonality, but sometimes exceptions are needed.

We need to pick an OS organization. Thus far, we've chosen organization via virtualization.

One way to see this. Let's write down as it's presented in architecture.

- CPU
  - ALU
  - Registers
- Primary memory (RAM)
- Devices (I/O)
- Time

We need a virtual machine that runs and executes instructions over time. Each process a virtual machine. Typically, they'll get th real registers and ALU, but only for a time slice. During the rest of the time, they're frozen while waiting for machine to do other things.

We model processes with a **process descriptor** that's held in RAM, with a copy of the registers when it's not runnning (this is junk data while the process is running, as the real values are held in the real registers; we offload when the process is not running).

For RAM, we could do something similar as with registers, but that's silly and memory-intensive, where we're copying memory to memory.

Hand-waving: **virtual memory**. The process descriptor holds information about where in memory its data can be found. Some OS' don't use virtual memory.

Note that RAM is accessed often, but devices are accessed relatively rarely. We're virtualize differently, by adding a layer of indirection via system calls. Using syscalls would be too much overhead for RAM, by comparison.

Now, our process descriptor also holds file descriptors, numbered 0 to 1023 (max is adjustable; 1023 is just common). We can see these an array of pointers to memory for specific files.

When we fork(), we get a copy of the process descriptors, with a copy of the registers, of the pointers to memory (separately

But their file descriptors are not affected by each other. One can close a descriptor talking about the same file without another process being affected.

We have file descriptor entries lying in between processes and file descriptions.

We have a file descriptor table that points to file descriptors. The processes have different internal offsets stored in the table. This lets us have different processes oepn the same files.

The file descriptions hold the inode info, last accessed, etc. This sits on disk. File descriptors are associated with processes' access to files.

The memory info is (in principle) copied for each process, but in practice we can often cheat and avoid copying. Hand-wave; we'll learn more later.

When we open(), fork(), etc., we get numbers: PIDs and file descriptors.

Modeling OS resources in a program

Some ways to handle this;

1. The classic way to do it is with a small integer.

This is a *handle* that acts as an opqaue identifier ("opaque" meaning that we need context, like "It's 39 and it's a process ID" or "It's 39 and it's a file descriptor for process 32") that needs context to interpret. The operating system supplies the context. This means opportunity for mistakes, e.g. type safety. Consider:

```
typedef long pid_t;
typedef int fd_t;

pid_t p;
fd_t f;

f = p
```

In C, we have an implicit cast, and no warnings unless -Wall and -Wextra are on.

We could have done:

```
typedef struct { long l; } pid_t;
```

But then we can't do:

```
pid_t p = fork();
if (p < 0)
    error();
```

And when we overload the operator, we introduce the original problem we were trying to structure away.

1. Use a pointer to an object that represents a resource

An example syntax:

```
process* fork();
```

Advantage: We catch more stupid mistakes, thanks to type checking. This is what we'd probably do if we were told to write an operating system at the end of CS 32.

Why is (1) preferred? For one thing, it allows indirection and kernel-kept secrets, e.g., we don't know about the process descriptor table. We have to interact with system calls.

(2) is preferred for performance, and where robustness matters less (as pointers directly into memory don't let the kernel regulate things). WeensyOS is an example of (2). It totally lacks memory protection.

What about forged handles? What if a rogue process tried:

```
char c = random()
waitpid(c, ...)
```

Simple: The OS is always checking context, and will reject invalid handles. Something like:

```
close(1000) // Close file descriptor 1000
```

We'll get a return value < 0, and `errno == EBADF`.

With approach (2), we can still check context, but it's harder.

Redirection

`cat` reads from stdin and writes to stdout.

```
$ cat < a > b
```

How does the shell implement this?

We could open `a`, then `b`:

```
// Should check for failure
int a_fd = open("a", O_RDONLY);
int b_fd = open("b", O_WRDONLY, ...);
```

shell's file descriptor table:

Array of file descriptors:

```
12 -> a's data
15 -> b's data
```

Then we do:

```
execlp("/bin/cat", (char*)[]){"cat", 0});
```

Problem 1: We blew away the shell. Use child labor!

```
pid_t p = fork();
if (p == 0)
    execlp(...)
```

```
int status;
waitpid(p, &status, 0);
```

Problem: cat reads from stdin, but we're reading from a.

Instead, we'll first do:

```
close(0);
close(1);
```

Then, when we open a and b, we get stdin and stdout.

PROBLEM: We closed stdin and stdout in the shell. The next time use stdin and stdout in the shell, we'll trash b. Instead, we have the child fiddle about:

```
pid_t p = fork();
if (p == 0)
{
    close(0);
    close(1);

    // Should check for failure
    // open() always opens the lowest numbers
    // This is a cheating way of doing it
    int a_fd = open("a", O_RDONLY);
    int b_fd = open("b", O_WRDONLY, ...);

    execlp(...)
}

int status;

waitpid(p, &status, 0);
```

Piping

Say we had:

```
du | sort -n
```

We could implement this as:

```
du > tmp
sort -n < tmp
rm tmp
```

Some shells implement this. Problem: This wastes a lot of memory and disk space. Irony: We're running `du` to check disk space.

This is time-inefficient too. We're copying to/from disk, which means lots of latency. We'd like to leave all of this in RAM and avoid copying it about.

Let's be clever. Treat part of our main memory as a sort of file system. There's a special place on SEASnet that's not really on disk, and is very fast.

Suppose our temp file goes there. We still have a slowdown because we are forced to serialize our work. We want to parallelize and have sort begin its work even as du continues to output information.

We want to parallelize orthogonally, meaning that read() and write() calls should work normally; no special flags or mechanisms required.

This gets us to the idea of **pipes**, where:

- We use the file API to read/write it as a stream (lseek() will fail)

They are **bounded buffers used for inter-process communication**.

Process descriptor -> file descriptor -> file description

The file description says "I'm a pipe" and we have read/write pointers that wrap-around. It too can hang:

[ ]
^ ^ | | read write

read() when empty -> hang write() when full -> hang

Pipes are all held in the kernel; system calls let processes use them.

To model all this, we have another system call:

```
int pipe(int fd[2]);
```

This creates a pipe, and puts into:

```
fd[0]: the read end
fd[1]: the write end
```

Note that we'll need to create fd[] and pass it in.

To destroy a pipe, we simply use close() on its file descriptors.

Now, let's reimplement the above:

```
// This code is not necessarily in the right order

pipe();

// We could use the close(0), and open() trick
// here too, but people dislike closing stdin, stdout
dup2(fd[0], 0);
dup2(fd[1], 1);

fork();
fork();

execlp("du");
execlp("sort", "-n");
```

But to get the child reading from fd[0] and fd[1], and not 0 (stdin) and stdout (1), we'll need to do:

We're going to have a parent and 2 children. We need to think about this and make sure that all our processes have the right file descriptors.

This is the **Dancing Pipes** problem. We know the system calls we need to use, and now we have to orchestrate things.

What about a longer pipe?

```
du | sort -n | less
```

Recursion!

```
(du | sort -n ) | less

A | less
```

We simply keep grouping recursively.

One way to do the process tree (to allow recursion):

```
        sh
    sh'     less
du    sort
```

Or, we could do:

```
sh
    du
        sort
            less
```

This is walking through the pipeline and forking as we go. Problem: The spec for Lab 1b specifies that the exit status for a pipeline is the exit status of the last command. We can't find out the exit status of the last command this way.

So maybe we reverse the order:

```
sh
    less
        sort
            du
```

Piping Issues

What can go wrong?

1. read(), but the pipe is empty, and the writer never writes -> hang indefinitely (very common mistake)

Example:

```
int fd[2];
pipe(fd);
char c;             // Nobody is writing to the pipe
read(fd[0], &c, 1); // This is part of how we'll do 1b
```

1. read(), but the pipe is empty, and there are no writers (they all closed the pipe, exited, etc.)

In this case, read() returns 0, showing no more bytes to be read.

The problem: (1) and (2) look very similar. Example where we intended (2) but introduced (1):

```
parents calls pipe()
fork();
fork();
dup2();
dup2();
close();
```

The parent must remember to close the pipes, as otherwise the *shell itself* will count as a writer to the pipe. Every pipe should have only the needed read/write ends open, and every other end closed. We can have multiple readers/writers. We just have to be careful.

Example:

```
(cat a & cat b) > f
```

In this case, we get intermingled data in f.

But we could also do:

```
(cat a & cat b) | sort
```

Here, it's still intermingled, but now in parallel.

Strange example:

```
(cat a & sed s/x/y b) | (sort & wc)
```

We have cat and sed writing to a pipe, and sort & wc reading from it.

1. write() but pipe is full, and the reader never reads -> hang indefinitely

Again, be careful to avoid this.

1. write(), but the pipe is full, and there are no readers

We could have write() return 0, but many program retry a write() in that case. Instead, we could return -1 and set errno.

But in a case like this:

```
while(something())
    printf(%d\n", something_else());
```

Notice how we don't check printf()'s return value. The internal write() in printf() will fail, but the while() ignores it. To prevent this, we'll have an asynchronous interrupt, `SIGPIPE` kills the program when no readers remain. The kernel handles this.

Discussion #3: 23 January 2015

Lab 1c

Use:

```
clock_gettime()
getusage()
proc()
getrusage()
```

-p is the profiling option, and we spit out profiling information there:

```
time when command finished // clock_gettime()
                           // time since 1970-01-01 00:00
execution time  // Decide which clock to use
user CPU time   // time used for non-system call processes
system CPU time // getusage() and proc(); time used for system calls
```

We do this for all commands executed by the shell

Record start time, execute command, then record the end time. Find the difference, and that's the real execution time. That's in seconds.

end - start

sys + user != real

Because there are multiple processes. We add up all the time used up by each. The user and sys times don't include

Say we had ls and sleep:

```
80% on ls
20% on sleep
```

Assume a fixed ratio, then when we execute for 5 seconds, we then exit. 4 seconds were actually used for ls, and 1 second was actually used for sleep.

So the user time reported for sleep would be 1 second. LOOK THIS UP

Think man-hours vs real hours.

clock_gettime()

```
int clock_gettime(clockid_t clik_id, struct timespec* tp)
```

We identify which clock we want:

```
CLOCK_REALTIME // For seconds; this is the clk_id
```

timespec has:

```
struct timespec
{
    time_T tv_sec; // seconds
    long tv_nsec; // nanoseconds
}
```

So make a struct, pass in the REALTIME ID arg, and we'll get the time in the timespec struct instance.

This is for the first column.

Execution time clock choice

Read the man page for clock_gettime() to decide which clock is most accurate for our execution time.

clock_getres()

We also need to understand the resolution of these clocks. They're in seconds or nanoseconds, etc.

```
int clock_getres(clockid_t clk_id, struct timespec* res);
```

We again specify which clock ID, but now the value stored in the structure after the function lets us know the resolution. If it's in nanoseconds, tv_sec > 0, tv_sec = 0; if it's in seconds, tv_sec > 0, tv_nsec = 0. We need to investigate clock choices.

getrusage()

```
int getrusage(int who, struct rusage* usage)
```

We can ask for:

```
RUSAGE_SELF
RUSEAGE_THREAD
RUSAGE_CHILDREN // The one we want to use
```

When logging user and system CPU time for a process, include the CPU time consumed by children of the process, even if they have already been terminated.

We'll have a struct:

```
struct rusage
{
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */

    // ... other stuff
}
```

This is executed inside of the child process forked() off for a command. We need to decide when to use this syscall.

We can also get usage from /proc/[pid], and the stat value is in **jiffies**, not seconds, the number of timer interrupts since bootup. We'd have to convert jiffies to real time.

Additional

If the process did not exec a command or is the shell process itself, log the process' numeric ID in square brackets instead.

Example: ":" is a command that does nothing

We can choose to ignore this in our implementation. Note that in our README.

Make sure that log lines are not interleaved when two processes finish at about the same time; we can test that with commands like:

```
ls | grep 123
```

Since these are executing in a short period of time, they may overlap in or conflict in writing out to the profiling information file.

Where to start

prepare_profiling(): Open the file name for storing profile information

execute_command(): Add function to profile before create a process or after a process is terminated

Lecture #6: 26 January 2015

Orthogonality and Trouble

I/O on an Unlinked File

Suppose we have one process that opens a file:

```
fd = open("file", O_RDWR);

read(fd, ...);
write(fd, ...);
```

Another process does this at the same time:

```
unlink("file");
```

The UNIX design lets this happen, because by assumption and design, these are as orthogonal as possible. read()/write() deal with file contents, not file existence, which is what unlink() does.

How does this work? Simple: unlink() *removes the file name*, not remove the file contents. We can have an anonymous file, and it will simply work.

In fact, we could even do this inside of the same process, unlinking right after opening a file. We can think of this as creating a temporary file, cleaning it up ahead of time.

Catch: `ls` will say "no file here" `du` will not reflect the file disk usage

Ops team might not like this.

unlink() says "remove this file from the file system", and so it will be marked from removal. close() says "I no longer want access to a file", and so all we do is close our file descriptor.

`df`, which reports free space, will, however report it, as it's filesystem-based, whereas `du` is file-based.

How do we actually free up the free space then?

Answer: Kill the process. The file system doesn't reclaim the space of an unlinked file until the process exits. As long as someone is interested in a file, the filesystem keeps it around.

Unlinked Flash Drive

What happens if we try to read() or write() on a flash drive that has been removed? They'll fail, return -1, and set `errno` to EIO or something similar.

"We can only take orthogonality so far."

Zombie Processes

A parent process:

```
pid_t p = fork();

int status = -1;
if (p > 0)
    waitpid(p, &status, 0);
```

Suppose the child does:

```
_exit(37);
```

before the waitpid(), which waits for a process to _exit(). It's a sort of race condition. For full orthogonality, we don't want to worry about special cases. To handle this, our process table

```
19: [    exit status: 37  | dead: 1    ]
```

will keep the entry around, but it will store the exit status of the process. waitpid() can still look at the exit status. This is a **zombie process**. It's dead, (exited, but the Grim Reaper has not come along to collect the bodies. waitpid() thus **reaps** the processes.

Suppose A and B both wait on C, or A waits on B, and B waits on A (deadlock!).

To resolve this, waitpid() requires that waitpid() can only be called on your own children. Thus, you cannot wait on yourself, your parents, etc. There cannot be a loop with this condition, because the process creation process necessarily makes a tree, and trees do not have cycles. Violate this rule, and waitpid() fails.

But what if the parent does: "Oh boy, I created a child! Now I'm going to die."

```
if (p > 0)
    _exit(0);

GP
|
P (dead)
| \
C1  C2
```

The grandparent can still wait on and repeat the parent, but it cannot wait on its grandchildren. **Orphaned processes** go to the **orphanage**, found at process #1. As part of boot-up, process #1 is created. Orphaned processes are automatically re-parented to parent #1 by the kernel. init() is often the name for this process. Toward the end of its execution, it does something like this:

```
while(waitpid(-1) > 0)
    continue;
```

-1 is a special flag to waitpid(), which means "wait on any my children".

This is the default Grim Reaper.

"We don't want orphaned zombies. We want to put them to rest."

If init() fails to do this, the process table will fill up, and new process creation will fail. We simply don't let this happen.

Named Temporary File with Race Condition

Suppose we make a temporary file in /tmp, and it will have a name until we're done with it.

Something like `sort` will definitely need temporary files for large data sets.

sort.c:

```
int fd = open("/tmp/sorttmp", O_RDWR | O_CREAT, 0600);

write(fd, ...);

read(fd, ...);

// End our access to the file
close(fd, ...);

// Mark the file for removal
unlink("/tmp/sorttemp");
```

We have a huge **race condition** here.

Suppose we run sort twice:

```
a | sort | b | sort | c
```

We have no control over which `sort` instance begins first. One solution: we make different files for each sort instance:

```
// Generate a file name
char sorttmp[100];
struct stat st;
for (int i = 0; i < 1E6, i++) {
    int r = random();
    sprintf(buf, "tmp/sort%d", r);

    // Check if this file already exists
    if (stat(buf, &st) != 0)
        break; // File did not yet exist; can use name

    if (i == 1E6)
        error();
}

int fd = open(sorttmp, O_RDWR | O_CREAT, 0600);

write(fd, ...);

read(fd, ...);

// End our access to the file
close(fd, ...);

// Mark the file for removal
unlink(sorttmp);
```

Does this solve our race condition?

No! It's still possible that we have two processes run around the same time, and they both happen to get the same random number, same file, etc.

Whatever test we run, that test may be obsolete by the time we run it. It's inherent to the nature of race conditions.

Answer: We need the kernel's help with a system call that checks if the file already exists.

Answer: Use `O_EXCL` with open():

```
// Generate a file name
char sorttmp[100];
struct stat st;
for (int i = 0; i < 1E6, i++) {
    int r = random();
    sprintf(buf, "tmp/sort%d", r);

    // Check if this file already exists
    if (stat(buf, &st) != 0)
        break; // File did not yet exist; can use name

    if (i == 1E6)
        error();
}

int fd = open(sorttmp, O_RDWR | O_CREAT | O_EXCL, 0600);

write(fd, ...);

read(fd, ...);

// End our access to the file
close(fd, ...);

// Mark the file for removal
unlink(sorttmp);
```

To detect race conditions and do the right thing anyway, we don't use stat(), which is out-of-date. Use the check from open() instead:

```
// Generate a file name
char sorttmp[100];
struct stat st;
for (int i = 0; i < 1E6, i++) {
    int r = random();
    sprintf(buf, "tmp/sort%d", r);

    // Check if this file already exists
    int fd = -1;
    if (open(sorttmp, O_RDWR | O_CREAT | O_EXCL, 0600);)
        break; // File did not yet exist; can use name

    if (i == 1E6)
        error();
}



write(fd, ...);

read(fd, ...);
```

```
// End our access to the file
close(fd, ...);

// Mark the file for removal
unlink(sorttmp);
```

One more thing: Suppose the file permission bits were all set to 0. The creating process can still do I/O on it (it has the file descriptor to it), and we get exclusive access. But it's annoying to have non-accessible files, so we just use O_EXCL and 0600.

O_EXCL has open() fail if the file exists, even if the process could write to it. `root` can do anything though, including gain access to files with all zeroed permissions bits.

Example with gzip

Here's what gzip does:

```
gzip foo
```

It creates a compressed version of foo, gets rid of foo, and there is now a file `foo.gz` which has the compressed version. Otherwise, we'd be taking up more space.

gzip.c:

```
int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);

// Do the compression work

close(fd);
unlink("foo");
```

Suppose we have an error with write(). Then, we keep foo around, as we weren't able to create foo.gz. So, we get rid of the partially-created foo.gz:

```
int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);

if (write() < 0)
{
    unlink("foo.gz");
    exit(1);
}

// Do the compression work

close(fd);
unlink("foo");
exit(0);
```

Suppose `foo` is a large file, and we change our minds about compressing it. The user presses Control-c in the middle of gzip execution. We want the process to be interrupted. How is this accomplished? It needs to be able to happen at any moment.

Suppose Control-C happens before we delete foo, or during compression, meaning we have only a partial foo.gz. We want gzip to have this property: **all or nothing**. This prevents confusion.

One solution is to watch for Control-C from terminal:

```
int ttyfd = open("/dev/", O_RDONLY);
```

And we'll periodically watch for a Control-C sequence.

```
int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);

if (write() < 0)
{
    unlink("foo.gz");
    exit(1);
}

char c;
read(ttyfd, &c, 1);
if (c == 4)
{
    unlink("foo.gz");
    exit(29);
}

// Do the compression work

close(fd);
unlink("foo");
exit(0);
```

Problem: read() will hang as we wait for the user to input.

There's actually a special system call to check for if there's nothing to be read.

```
fcntl(fd, F_SETLL, W_NOHANG);
```

Which modifies this to:

```
if (readttyfd, &c, 1) < 0)
```

We could package this into a function:

```
check_for_interrupt(void)
{
    char c;
    read(ttyfd, &c, 1);
    if (c == 4)
    {
        unlink("foo.gz");
        exit(29);
    }
}
```

Which will NOT be called after the unlink("foo"), as otherwise we'll lose both files. This complicates the hell out of our source code, add this call to every single freakin' loop, etc. The code is hard-to-read.

We want just a little bit of work, such that everything else works.

This constant polling for Control-C is a non-starter. Rather than have check_for_interrupt() be everywhere, we want its functionality be in one place, which gets us to:

Signals

We know what we want to do when we get a signal like Control-C. We'll have a **signal handler** that will be executable by the kernel. In one spot, we'll have the code for cleanup in the result of a signal, and let the kernel know about it.

By convention, signals are assigned a small integer. Let Control-C be 3 here:

```
void handle_signal(int sig)
{
    // Our cleanup code
    unlink("foo.gz");
    _exit(1);
}
```

This is the easiest signal type to handle, one where we just exit the process rather than return to the caller after handling the signal.

So, now we have for gzip:

```
int main(int argc, char** argv)
{
    // Designate our signal handler
    // for a signal type
    signal(SIGINT, handle_signal);

    int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);

     // Do the compression work

    close(fd);
    unlink("foo");
    exit(0);
}
```

signal() system call

signal() is a system call that takes a signal type, and a function pointer to the signal-handling function.

Now, suppose we hit Control-C just after the `unlink("foo")`. We will again lose both files.

Or, suppose we have the Control-C before the open(). If we mistakenly run gzip in a directory where goo.gz was already created, the open() doesn't have a chance to see that foo does not exist. We again lose foo.gz, here our only copy.

We need to be smarter about where to check for signals:

```
int main(int argc, char** argv)
{
    // Cleanup may not be needed at all
    if (open("foo", O_RDONLY) < 0)
        err_exit();



    // Designate our signal handler
    // for a signal type
    signal(SIGINT, handle_signal);

    int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);
```

```
    // Do the compression work

    close(fd);
    unlink("foo");
    exit(0);
}
```

Now, let's add another signal handler, one that does nothing, ignore it:

```
signal(SIGINT, SIG_IGN);
```

Still buggy:

```
int main(int argc, char** argv)
{
    // Cleanup may not be needed at all
    if (open("foo", O_RDONLY) < 0)
        err_exit();

    // Designate our signal handler
    // for a signal type
    signal(SIGINT, handle_signal);

    int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);

    // Do the compression work

    // Ignore Control-C if it comes in here,
    // as we'd lose both files
    signal(SIGINT, SIG_IGN);

    close(fd);
    unlink("foo");
    exit(0);
}
```

Here's the problem. Suppose the signal comes in right before the open(). Suppose foo.gz was already there, but we didn't have permissions to write to it, and so we'd remove it and then create it. We thus lose data.

So we move signal() to after open(). But now about if we get Control-C before open()? How do we fix this?

Signal Masks

Kernel to the rescue, with another system call:

```
pthread_sigmask()
```

A signal mask is an array of bits, one for each signal. If it's 0, the signal arrives normally. If it's 1, the signal is held, sort of queued for the process:

```
pthread_sigmask(SIGBLOCK, &old, &new);
```

Where `old` and `new` are signal masks. We block the signals. We need to arrange for `new` to contain SIGINT, do this before open(), and SIGUNBLOCK after open():

```
int main(int argc, char** argv)
```

```
{
    // Cleanup may not be needed at all
    if (open("foo", O_RDONLY) < 0)
        err_exit();

    // Designate our signal handler
    // for a signal type
    signal(SIGINT, handle_signal);

    // Temporarily prevent a signal from coming in
    pthread_sigmask(SIGBLOCK, &old, &new);

    int fd = open("foo.gz", O_WRONLY | O_CREAT | O_TRUNC, 0600);

    // Unblock; end criticla section
    pthread_sigmask(SIGUNBLOCK, &old, &new);

    // Do the compression work

    // Ignore Control-C if it comes in here,
    // as we'd lose both files
    signal(SIGINT, SIG_IGN);

    close(fd);
    unlink("foo");
    exit(0);
}
```

Critical Sections

The section where we want to temporarily block signals is called a **critical section**. Then, we can have a flag that will let us know if the file was created:

```
int main(...)
{
    ...
    open(...)
    foo_gz_created = true;

    ...
}

void handle_signal(int sig)
{
    if (foo_gz_created)
        // Cleanup
    else
        // Nothing to clean up
}
```

Side Effects of Signal Handlers

Signals have changed our abstract machine. Between any pair of machine instructions, a signal handler can now be called.

Suppose our signal handler modified data in our program. Suddenly, we lose assumptions about the state of our data, as the signal handler might come in at any point and change things. Programs become much more difficult to analyze and understand.

For this reason, a signal handler is as bare as possible, as additions here may have us rethink the whole rest of our program. Ideally, a signal handler might just _exit() or affect only one global variable associated with signals.

The guiding principle: *Keep your handlers simple!*

But now yet another problem: Suppose GCC optimizes away a conditional:

```
x = 12;
if (x > 0)  // Optimized out
    printf("Ok");
```

This optimization assumes that no data was trashed by signal handlers.

Our gzip.c has a bug:

```
foo_gz_created = true
```

This line may be optimized out, as we never used it later in main(). How to prevent this?

Use `volatile` to discourage optimization

```
bool volatile foo_gz_created;
```

It can't help with everything though. We'll see more why later.

Reasons for Signals and a List of Them

Why might we have signals:

- Uncooperative processes

Control-C: SIGINT

- Invalid programs (e.g., division by 0)

Illegal (or privileged
in user mode) instruction: SIGILL

Floating point exception: SIGFPE

Invalid address: SIGSEGV, SIGBUS

We can now modify our programs such that we're never interrupted by segfaults or subscript errors.

- I/O error

No readers for a pipe being written to: SIGPIPE

A device may ask for I/O right away: SIGIO

- A child died

Parent can handle child death by looking for: SIGCHILD

- User signals

Control-Z (suspend a process): SIGTSTP

- Kill a process

This cannot be ignored or handled: SIGKILL

To send this signal, do kill [pid]. This is not a good idea under normal conditions, as our signal-handling code for cleanup will not be executed.

- User goes away

Hang-up: SIGHUP

Example: We log out after starting a process.

- Timer expires

When our timer goes off: SIGALRM

To set an alarm clock:

```
alarm(12);
```

After 12 seconds, the default behavior is to terminate, but we can handle it instead, and do something else.

This is **asynchronous notification**. In practice, if we want to handle these exceptional events without a bunch of extra modifications and easily-forgotten checks, this is the way to do it.

Asynchronous Signal-Safe Functions

Suppose our signal handler uses exit() instead of _exit():

```
void handle_signal(int sig)
{
    ...
    exit();
}
```

_exit() makes the process a zombie immediately. exit() does the cleanup process first.

We get a race condition.

Example:

```
char* p = malloc(1000);
```

Internally, malloc() maintains a heap data structure that keeps track of all the allocated objects and free space. It's very carefully arranged spaghetti. It moves one strand from free to in-use. We have to make both changes: no longer free, and now in-use.

Suppose our signal-handler does this:

```
void handle_signal(int sig)
{
    a = malloc(12);
}
```

Now our spaghetti will be messed up if we do malloc() in the signal handler, as it tries to allocate in the middle of another one.

In general, a signal handler should never modify global state. This can happen in many ways though:

```
malloc() // Modifies the heap
printf() // Uses malloc()
exit()   // Flushes the buffers for the printf()
```

And even:

```
sin()
```

Problem? sin() stores the value into a global var, asks for the sine component from the shared sin/cos function.

A signal handler can only call functions that are known to be async signal-safe functions:

```
_exit()
signal()
pthread_sigmask()
close()
...
```

POSIX maintains a list of these. Everything else cannot be in the handler.

Threads

Motivating problem: Processes are heavyweight, which leads to fork() being slow, and inter-process communication (**IPC**) is also slow (pipes, etc.).

In general:

- Processes are slow to create
- Processes are slow to intercommunicate

The problem that causes this: Processes are too isolated.

We want to create **threads**, which have less isolation, but are faster to create and intercommunicate.

Let's split up process resources into two categories:

Expensive stuff: Per-process, as too expensive to copy

Cheap stuff: Per-thread, as easy to copy, and we get some isolation.

There is at least one thread, as part of the initial process.

Expensive Resources

- Process ID

So we can wrangle individual processes

- Address space

Heap, instruction code

- File descriptors

Threads can share access to the same files. We have to be careful to coordinate their work and communication to avoid race conditions etc.

- Signal handler table

All the threads use the same handlers

- Working directory

`pwd` returns the same result for all threads. If it's change, it will be changed for all threads.

- umask

Default permissions for all new files created by threads

- UID/GID

User and group ID is shared between threads

Cheap Resources

- Thread ID

So we can identify threads.

- Registers

Or they'd stomp all over each other.

- Instruction pointer

Or else we'd have to track this

- Stack

As return addresses stored on here.

- Signal mask

While threads share signal handlers, they may not all use them. A common implementation dedicates one thread to signal handling, and the rest block them out

- Errno

A global variable for checking the failure details of a system call. We can't afford conflicting error information.

- State

Blocked/ready/zombie, etc. The threads need to know if they can run, etc.

Now, IPC is rapid, as address space is stored.

Lecture #7: 28 January 2015

Threads

Threads are like processes, but lightweight, and share memory, file descriptors, and a few other things listed above.

Downside of threads: They complicate our applications. A problem we won't fix today: synchronization becomes a bigger deal. We already saw this possible problem with pipes, where processes have to wait on each other. That waiting was the synchronization mechanism, and kept things well.

Scheduling Threads

It's like any scheduling problem.

Example: This class is scheduled by the registrar.

"One of the things about schedulers is that nobody notices them unless they screw up."

Case in point: CS 111 was originally in too small a room.

In computing, we have N CPUs and > N threads. This isn't a problem unless we have more threads wanting to run than there are CPUs. Now, we have to choose who runs (if at all) and when. To decide, we'll have **scheduling policies**, the high-level, abstract rules for resolving conflicts. (The registrar has them, but they're secretive about them.)

Scheduling policies tend to be independent of platform. Then, we'll need **scheduling mechanisms**, the low-level operations that the scheduler needs to do its work. Policies and mechanisms should be *orthogonal*. We should be able to implement any policy using the mechanisms we have.

Scheduling Goals

Application developers constantly complain about scheduling. Having metrics lets us know if we should pay any attention to the complaints.

- Fast scheduler

Decisions should be made quickly: less time scheduling, more time executing.

- High performance app

The right threads at the right time can speed up performance.

- Fairness

We should avoid overly favoring particular threads.

- Scale for CPU scheduling

There's a *long-term* scale. It's concerned with things like having too many threads that swamp the CPU. It decides which processes/threads are admitted to the system in the first place. Example: We don't more people into Disneyland once it's full up. We control this via fork(). When the fork() syscall comes in, the kernel can just decide to refuse the fork() request.

Then there's the *medium-term* outlook. Here the concern is usually with memory constraints. Which processes fit into RAM comfortably? We switch from one process to another, which requires paging between memory and disk. We want to avoid **thrashing**, which is when we spend too much time just moving data between primary and secondary memory.

Finally, the *short-term* view has consider this: we have a lot of easily-runnable threads; which shall we let run?

All of these scales call for *different mechanisms*. The first is concerned with processes/threads that may not exist yet; the second has to deal with memory; the third has to choose what to run.

As for policies, we could have the same general policy at all three scales, or implement different ones.

Scheduling Policies

Policies can be complicated. For example: we may have different policies for different job classes (different processes are divided into different categories).

Some categories as seen on SEASnet:

- System processes

These have high priority and are important.

- Interactive processes

Vim, Emacs, etc. We want good (low) latency.

- Batch processes

Here, we want good throughput, and latency is less of a concern, as they're rather background tasks.

Real-time Scheduling

Not seen in WeensyOS or on SEASnet, but this does exist in the real world. This is where timing must be excellent, or things don't work, as our program interacts with the physical world.

Hard Real-Time Scheduling

Example: Controller for a nuclear power plant.

The software *must* drop those control rods when we push the SCRAM button.

We cannot miss a deadline.

Another example: The brake system in newer cars.

Predictability is key and trumps performance.

Here, *not* missing deadlines is far more important than efficiency. It must work without fail. For this reason, caches are often disabled, as they make execution time unpredictable. It may hit or miss, and we lose predictability. Now, our program is much slower, but consistently so.

Soft Real-Time Scheduling

This is more popular these days.

Deadlines still exist, but are not essential.

Example: An explosion animation in a video game that drops a frame or two.

Example: Online videos when they suddenly flicker or pixelate.

One problem: We scheduled more time than available. We need to decide what to dump. One solution:

Earliest (makeable) deadline first (older ones may have already passed).

"By the way, is this the algorithm the way you do homework?"

This could be made better by having **priorities**. Higher priorities win out. But what if we still don't have enough time? We could do the **smallest tasks first**. Either one of these can cause problems. The busiest devices will be scheduled out alogether.

"All these characters looking scared, running around--and there are no explosions."

Rate-monotonic scheduling throttles everything down. If we're at 120% scheduling, we make everyone accept 80%. This is fairer. We get most of everything instead of missing out on pieces altogether.

There are many, many scheduling algos out there. Look 'em up.

Scheduling Mechanisms

Begin simply: Assume 1 CPU and n = 2 processes/threads:

Process 2: ----- -------- Process 1: ---- ------------

However, the scheduler itself uses up CPU time:

Process 2: ----- -------- ^ | | V Process 1: ---- ------ ------ Scheduler: - -- - -

There could be gaps both between and within processes when the scheduler needs to run. Our goal is to keep this small, but with enough time to afford us some intelligence.

As for the transitions themselves, we need to decide who chooses them.

When we do the up transition of choosing a new process, that's easy. The scheduler is part of the OS, and we can use the `RTI` (return from interrupt) instruction.

But the down transition is harder. How does the process decide when to run the scheduler? At any one time, only one thing gets to execute on the CPU. We need to switch off between the scheduler in the kernel and the processes.

Two major mechanisms here:

Cooperative Multitasking

Each process *volunteers* to call the scheduler every now and then, and is prepared to go out of execution. This relies on every process not being monopolistic with its CPU time.

yield()

We'll have a new system call, with signature:

```
void yield(void);
```

"This is a good time to give up my CPU access. I've already saved all my registers to RAM, etc."

It's sometimes called:

```
void schedule(void):
```

We sprinkle yield() calls thorughout our program, at points where we know can temporarily transfer control to someone else. Say, every 10 ms, the program is known to call yield(). Then, we make all of our programs do this. Imagine adding these yield() calls everywhere manuall. Instead, we'll do it for them.

In fact, there is no yield() system call in Linux. We'll see why below. There is in WeensyOS though.

Busy Waiting

Consider how we might deal with I/O cooperatively. We could do:

```
while (isbusy(device))
    continue;

read(device, ...);
```

We hold on to the CPU, and don't give it up to another process. We only do this when a device is expected to be ready very quickly (e.g., time to switch back and forth takes longer than waiting). Otherwise, we'll busy-wait for forever.

Polling

Fairer might be:

```
while (isbusy(device))
    yield();

read(device, ...);
```

Here, we give up control of the CPU.

The difference between busy-waiting and polling is this: in both, we constantly check whether a resource is available yet, but with busy waiting, we continue to execute the process that is waiting. With polling, we let another process run while we wait.

Problem: Consider hundreds of processes waiting for many different devices. We'll end up walking through each process and getting "still busy" all the time. We're wasting CPU time again.

That is, we may let another process run only to find that it too needs to wait fr a process. So we let another process run, but it too needs to wait on a resource (possibly the same one). If this continues, we'll waste all our time cycling through processes just waiting on resources. Now, we're busy-waiting at a sort of meta-level.

Put another way: busy-waiting is standing there while our food is microwaved, versus polling, where we come back periodically to check on it. Blocking is us going away and doing other things and not coming back until we hear the ding.

Blocking

This is what we implemented in WeensyOS.

```
while (isbusy(dev))
    wait_for_ready(device);

read(device, ...);
```

We'll tell the OS to not bother scheduling a process until the device is actually available. To make this work, we'll need to keep track of what processes are blocked by a busy device.

We'll have a **blocked list** in the middle of a wait_for_ready() call, part of the data structure about a device. Then, the OS will unblock the process associated with the process table entries in the blocked list, and have it run. This happens when the OS finds out that the device is ready. Presumably, the OS-controller interface handles this.

For this approach to work, the scheduler will need to ignore blocked processes when it's scheduling processes. Rather than looking at the blocked list and the process table, we just have a `blocked` field in the process table. We could have also have a `runnable` field that means not blocked, or has some other reason for why a process is non-runnable.

Blocking works because we avoid scheduling any blocked processes at all. No CPU time is wasted on them while they're waiting. Instead, they're in a sort of cyro while waiting for the future to arrive where their promised resource is finally available.

Pre-emptive scheduling

What if we don't want cooperative scheduling? Application programmers may not be trusted to be responsible with their CPU time. We want a scheme where the kernel can force processes to give up the CPU.

Here, the kernel is always in charge, and the processes can *involuntarily* lose access to the CPU at any time.

Here's one way to do it: each system call drops us into the kernel anyway, and each one of them will call yield(). The original

Mac OS did this. This is taking advantage of the fact that we're in the kernel while handling system calls, and so we can do scheduling while there.

Problem: A lot of user-level code may loop ([semi-]infinitely) without syscalls at all, and so we might hang indefinitely. The scheduler never got a chance to to fix things. We are still relying on too much on applications being well-behaved. Outside of embedded systems we make ourselves, this is a terrible assumption to make. (Embedded systems may schew pre-emptive scheduling for reliable, consistent timing, as well as performance reasons, e.g., scheduler overhead)

We will need help, as we so often do, from hardware.

Timer Interrupt

Somewhere on the motherboard has a clock wired to the CPU. At regular intervals, the clock sends a signal to the CPU, and the CPU traps. It checks the *interrupt service vector* and executes the corresponding kernel code. It's like we issued an INT instruction from a program.

The system is dynamically modifying our program by periodically trapping the CPU and having it run yield() automatically. On Linux, the default is 10 ms, as this is fast enough for us to not notice this latency. We have multiple levels of indirection to accomplish this.

"We have now solved the Halting Problem."

We cannot determine if the program will go forever or will stop. See also Goedel's Undecidability Theorem. In practice, our machines are finite state machines. The halting problem assumes infinite resources. We're pausing/resuming our process repeatedly, but we don't know when

However, that said, this current scheme won't stop programs with infinite loops. We normally have users/operations staff handle this, as they can always press Control-C.

We can also use SIGXCPU, which is sent when a program has almost run out of CPU time. We set a quota on CPU time.

The program could still be rebellious:

```
signal(SIGXCPU, SIG_IGN);
```

But the kernel should eventually put its foot down.

Done or not done

But what if we preempt something that needs to either do something completely or not at all? The example from last lecture was gzip. The files it manipulates and produces will be in an inconsistent state.

In general, pre-emption complicates synchronization. We'll have another 1-2 lectures just on synchronization.

Scheduling Metrics

Scheduling is well-studied, and much of our terminology comes from the days of heavy manufacturing. Example: Order for 100 Model T's.

Order arrives | Begin task | First output | Done [wait time]

[response time] [run time]

[turnaround/latency time]

Wait time: We couldn't start the new order until we retooled, finished the previous order, etc.

Response time: Time from when we got the order to when we produced our first output

Run time: The time used on actually completing the order

Turnaround/latency time: Time from order arrival to order completion.

Ideally, we'd want to *reduce the averages* for all of these times, as averaged over workload X.

We'll also want to know the variance/std dev of the these times. This is because *predictability* matters too, not just performance.

Finally, we'll consider the *worst case*, especially important in the hard real-time case.

Another metric is **utilization**, the percentage of CPU time used for "useful" work. The amount of CPU time used by the scheduler itself is not useful.

This is naturally accompanied by **throughput**, the number of jobs/second completed, or the CPU-seconds done per second. Normally the higher the utilization, the higher the throughput.

First-Come, First-Serve

We execute jobs in the order in which they arrive.

Job: Arrival time: Run time: Wait time: SJF wait: A 0 5 0 0 B 1 2 4 + Δ 4 + Δ C 2 9 5 + 2Δ 9 + 3Δ D 3 4 13 + 3Δ 4 + 2Δ Avg: 5.5 + 1.5Δ Avg: 4.25 + 1.5Δ

This produces:

```
AAAAA[switch]BB[switch]CCCCCCCCC[switch]DDDD
```

So B starts at 5 + Δ, C at 7 + 2Δ, D at 16 + 3Δ, and we end at 20 + 3 Δ.

See here how C holds up D. Consider someone doing thing complicated at a bank and people behind them who need to do simple things. This leads to the alternative.

Shortest Job First

Now we have:

A still starts, as it's the first to arrive, before anyone else. Then, we B, C, and have arrived, and choose the shortest.

```
AAAAA   BB  DDDD    CCCCCCCCC
```

Now, all our metrics are the same, but our average wait time is lower.

We're assuming here that we know ahead of time how long each job will take. This is true for something like Model T building, but not so for things like keystrokes into a text editor.

Discussion #4: 30 January 2015

We can expect multiple-choice questions, and we should print the code from our assignments, with at least the skeleton code.

"Which of the following will be MOST correct?"

State all assumptions during the exam.

Problem 1

Suppose we execute a privileged instruction in non-privileged mode and there is no trap. Instead, nothing happens; it is equivalent to a no-op. If there were true, could we implement system calls to enforce hard modularity

Could we implement system calls to enforce hard modularity for an operating system like Linux with acceptable performance? If so, explain how; if not, explain why not.

A trap: unexpected event, and the OS takes control.

Problem 2

Suppose we run the following shell command in an ordinary empty directory that you have all permissions to. What will happen? Describe a possible sequence of events.

```
cat < mouse | cat | cat > dog
```

`mouse` doesn't exist. `dog` is created, but empty. `cat` gives an error that no such file exists, but it doesn't halt execution.

Problem 3

getpid() is often implemented as a system call. Would it be wise to implement it as an ordinary user-space function whose implementation simply accesses static memory accessible to the current process? Explain the pros and cons of this alternate implementation.

Downsides: Not very robust, and introduces security issues, as a program can be compromised, and the PID will be compromised.

Problem 4

POSIX requires that a `write` system call requesting a write of fewer than PIPE_BUF bytes to a pipe must be atomic, in the sense that if multiple writers are writing to the same pip, the output data from the `write` cannot be interleaved in the pipe with data from other writers. PIPE_BUF is up to the implementation, but must be at least 512 (on Linux, it's 4096). A `write` of more than PIPE_BUF bytes need not be atomic.

Atomicity: Operation will either be fully successful, or fail altogether. The set of bytes (beneat the limit) will be written seemingly all once.

Part A

What's the point of the this atomicity requirement? What sort of programs works if the implementation satisfies this requirement, but does not work otherwise?

For operations smaller than this limit, we want to execute them quickly and at-once.

Part B

Suppose you want at write a super-duper OS in which PIPE_BUF is effectivly infinity (it's equal to 2**64 - 1, say). What sort of problems do you anticipate having with your OS?

There is no fairness between programs. A program could hog everything to itself.

Part C

Did your solution to Lab 1c rely on this requirement? If so, explain where; if not, give an example of what happens if the requirement is not met.

Yes, we relied on the profiling log lines not be interleaven.

Problem 5

Why might we not want to replace `kill` with `write`?

What if we have a zombie process?

Problem 6

Does it make sense to use SJF in a preemptive scheduler? If so, give an example; if not, explain why it doesn't make sense?

Pre-emption will interrupt current job and give control to the shortest jobs as soon as possible.

Answer: Not a good idea. What if there are so many short jobs that the relatively longer job will never be finished? We have the potential to lose fairness.

Problem 7

Mutex: we acquire a lock, and then no other process can interfere.

Suppose we have a shared mutex, a single blocking mutex (not simple) for reading/writing around a shared pipe object. Can a would-be reader starve?
If so, show how. If not, explain why not. If your answer depends on the implementation, explain your assumptions and why they matter.

Yes. Either use multiple mutexes or use semaphores.

The thing about mutexes is that they lock the whole thing. A semaphore lets you lock a specific portion.

Problem 8

If line 4 is omitted, we don't set the registers to current process. Problem: The register holds the number corresponding to which interrupt we wanted. interrupt() fails to do its job altogether. We might even "interrupt" ourselves, as the check if it's an invalid PID, ourselves, etc., fails.

Lecture #8: 2 February 2015

// Midterm this Wednesday! It covers *everything*, // including today.

Round Robin Scheduling

SJF and FCFS in their naïve version wait on a job to finish. Using **preemption** lets us have our CPU switch off between jobs: break long jobs into a series of shorter jobs.

One of the simplest preemptive scheduling schemes is **Round Robin**. We split up every jobs into (unit) lengths of <= 1.

RR scheduling for jobs:

Job Arrival Time needed Wait time A 0 5 0 B 1 4 Δ C 2 9 2Δ D 3 4 3Δ

Result:

```
ABCD ABCD ACD ACD AC CCCC
```

As time goes -->

Disadvantage: A lot of delta time, the time to context switch between jos.

Advantage: The average wait time: 1.5Δ But throughput suffers and utilization is less. The effect depends on delta's size.

Fairness issue: A lot of newly arrived processes can crowd out the original ones such that they never get to run. As long as new arrivals appear, the old ones are starved of CPU time.

This phenomenon is called **unfair round robin**, where new arrivals have priority: they go to the head of the run queue.

The alternative is the converse, called **fair round robin**, where new arrivals go to the tail of the run queue. This is how people often implement pickup games. This increases wait times a bit, but prevents starvation.

Priority Scheduling

We can assign processes **external priorities** (set by the operators, users, etc.) or **internal priorities** (set by the OS scheduler).

SJF: The shorter the job, the higher the priority of the job, i.e., assuming standard ordinals, priority = length of job.

FCFS: The more recent the job, the higher the priority of the job, i.e., priority = arrival time.

Suppose a new scheduler, Eggert's Scheduler, where the priority = length of job + arrival time.

Lower priority numbers means more important jobs. First priority > second priority > third priority, etc.

What's the fairness of this scheme?

Digressing for a moment, consider the `ps` command, which has a bunch of flags, and the output includes PRIO, which lets us see the priority number. Another number, called NICE, tracks the external priorities. PRIO is the combination of internal and external priorities. PRIO is determined by a function of what it knows it about processes wanting to run, and what the user told it. Traditionally, NICE goes from -19 to 19.

"Niceness is the negative priority." They're polite, the nicer, the more they'll yield. The default is 0.

There's a `nice` command that adds 5 to the NICE score, by default:

```
nice make
```

Make (and all of its children) will now have lower priority. To add 9 to the niceness:

```
nice -n9 make
```

To subtract 9 from the niceness:

```
// Only root can do this, as otherwise we could
// dominate scheduling
nice -n-9 make
```

Thus far, every scheduling algo we've done so far can be encoded in some priority system, e.g. longest job first is the

negative of the length.

Synchronization

The Issue

1. Pre-emptive scheduling

At any moment, the program we're running could be overthrown as king of the CPU and forced to languish.

1. Threads

Not only can an application be kicked off, a thread may be kicked out, even as other threads continue. What if a thread or process was in the middle of a very important action? We can lose the CPU at any time. It's like we're assuming the CPU is ultra flaky.

The general idea of scheudling is that we need to **coordinate actions in a shared address space**.

We have shared RAM:

[RAM] ^ ^ ^ | | | (CPU1) (CPU2) (CPU3) // Virtual or real CPUs

We're trying to avoid race conditions, when threads stomp on other threads. This is when behavior depends on the order of execution (a terrifying thought). We can't arrange or guarantee a specific order of threads, as the CPU's pretty autonomous.

Things get complicated: Each CPU probably caches different things, and there's no way to notice at a hardware level that a cache held by a CPU is out-of-date.

Ways to Prevent Race Conditions

Sequence Coordination

Suppose two actions A and B, and they step on each other's data. To fix this, we can arrange for A to finish before B starts. We insist that commands act in sequence, in a particular order:

```
A ; B
```

To implmement this, we can make sure that just *one* thread executes both. That's the simplest way to do it, but we can even do it for when they're being executed by different threads (e.g, waitpid()).

Isolation

We can require that threads have totally different objects, such that they never access the same object, and so we never have to worry about them overwriting each other.

HOWEVER, both of the above more or less nullifies multi-threading altogether.

Atomicity

Arrange for A to be executed all before B, or all after B. We avoid interleaving. We'll get A;B OR B;A. This is more complicated, but nice for the multi-threading.

Kinds of Atomic Actions

They are small, **indivisible** (and so they cannot collide or interleave)>. The smaller the action, the better, as they're easier to implement. If the're small enough, even hardware can handle it for us. Any other thread must wait for atomic actions on the same project to finish.

They lead to better, more-for performance.

Cheating Atoms

Atomicity is sometimes implemented in a "cheating" way.

Suppose an ATM has two actions, with an initial balance of $100.:

```
deposity_money(20); // Dollars
withdraw_momwy(4);
lookatbalance();
```

Then: thread 1 does A, thread 2 does B, thread 3 does C.

Check the bank balance as it's running though, and we might $95, because we did provisial +10 increase, etc. Internally, the data structures were messed up, but the outside behavior was fine.

"If the bank's customers can't see the problems, there are no problems."

Regardless of the actual mess behind the scenes, so long as the result is obtained as expected, where we can come up with a explanation assuming atomicity, it's fie.

This is the **principle of serializability**. Example: The TA can't find the bug in our program. It is related to observability. As long as the user does not notice, we're okay.

Example: ATM withdrawal/deposity

The API:

```
void deposit(int pennies);
void withdraw(int pennies);
```

Doesn't account for failure. Better:

```
    bool deposit(int pennies);
    bool withdraw(int pennies);
```

E.g.: attempt to deposit/withdraw negative amount, or withdraw more money than allowed.

Implementation (assuming single bank account):

```
int balance; // Globals start off as 0, apparently
bool deposit(int pennies)
{
    if (pennies < 0)
        return false;

    // Another problem: We can overflow
    balance += pennies;
    return true;
}

bool withdraw(int pennies)
{
    if (pennies < 0 || pennies > balance)
        return 0;

    // Account for negative balance -> 0
```

```
    balance -= pennies;
    return true;
}
```

Suppose we make balance a `long long`, a 64-bit signed integer type, and we account for overflow:

```
long long balance; // Globals start off as 0, apparently
bool deposit(long long pennies)
{
    // Recall that overflow will mess up a sum check,
    // so we need to be careful with the overflow check itself
    if (pennies < 0 || pennies > LLONG_MAX - balance)
        return false;

    // Another problem: We can overflow
    balance += pennies;
    return true;
}

bool withdraw(long long pennies)
{
    if (pennies < 0 || pennies > balance)
        return 0;

    // Account for negative balance -> 0
    balance -= pennies;
    return true;
}
```

Now suppose that these actions occur at approximately the same time on two different threads. This could happen:

Some other thread deposits, after the overflow check, and causes overflow. Some other thread could withdraw as well. We are suffering a race condition where the check is not guaranteed to happen before the action.

Let's re-eimplement withdraw():

```
long long balance; // Globals start off as 0, apparently
bool deposit(long long pennies)
{
    // Recall that overflow will mess up a sum check,
    // so we need to be careful with the overflow check itself
    if ( !(0 <= pennies && LLONG_MAX - balance) )
        return false;

    // Another problem: We can overflow
    balance += pennies;
    return true;
}

bool withdraw(long long pennies)
{
    // Temporary copy of balance
    long long b = balance;
    if ( !(0 <= pennies && pennies <= b) )
        return 0;

    balance = b - pennies;
    return true;
}
```

Val Schary, invented structured programming back in the 60s, taught Eggert this trick of using only less-than.

We fixed one bug (we'll never get a negative balance), but someone could deposit, and we lose a deposit if a thread doesn't see the deposit in time. We introduced another bug.

Instead of modifying our code, we want to identify the sections of code that need to executed atomically, with just enough isolation to prevent race conditions.

Critical Sections

A set of instructions such that while an instruction pointer is pointing to any of those instructions, no other instruction pointer (different threads have different %ip's) is pointing at the critical section for the same object.

As long as we know that we don't have two pointers, one each into a critical section, or two into one critical section, we're okay.

This supports:

1. **Mutual exclusion**, where an instruction pointer in a critical section excludes %ip for all other critical sections. This gives *safety*.
2. **Bounded wait**, where a thread in a critical section is obligated to finish as quickly as possible, as otherwise we lose multithreading. This gives us *fairness*.

Notice that these principles rather conflict: the easiest way to have mutual exclusion is to have very large critical sections. The easiest way to have bounded waits is to "ensmall" critical sections, where we don't lock out other threads unless absolutely needed.

We must be guided by the **Goldilocks' Principle** Safety first, but don't forget about fairness. Large, but no larger than it has to be.

Pipes in Threads

Sacrifice memory safety for performance. Threads will be able to examine pipe contents.

Pipes will be bounded buffers, and we keep track of where the data is, i.e., read and write ends:

```
#define PIPE_SIZE (1 << 12) // 2 ^ 12 bytes
struct pipe
{
    char buf[PIPE_SIZE];
    unsigned r, w;
}
```

We'll read/write single bytes, and wrap around our indices using modulo:

```
char readc(struct pipe* p)
{
    // Return a char, and move forward
    // the read end afterward
    return p->buf[p->r++ % PIPE_SIZE];
}

void writec(struct pip* p, char c)
{
    p->buf[p->w++ % PIPE_SIZE] = c;
}
```

Note that this code relies on PIPE_SIZE being a power of 2; it would not work for the case of 1000 bytes.

We still have problems. What if we have an empty pipe and we get a read request or a full pipe and a write request?

```
char readc(struct pipe* p)
{
    // If read end == write end, full pipe
    while (p->r == p->w)
        continue; // Busy waiting

    // Return a char, and then move forward
    // the read end afterward
    return p->buf[p->r++ % PIPE_SIZE];
}

void writec(struct pip* p, char c)
{
    // If the difference between the two
    // pointers is PIPE_SIZE, the pipe is full
    // This works even if w overflows and wraps
    while (p->w - p->r == PIPE_SIZE)
        continue;

    p->buf[p->w++ % PIPE_SIZE] = c;
}
```

But again, we have problems if we have multiple threads: simultaneous writes on an almost-full pipe, a read on an outdated pipe, etc. To fix this we'll demarcate some critical sections. We'll assume some primitives to accomplish this:

```
lock_t* aLock;

// Precondition: This thread doesn't own the lock
// Postcondition: This threads owns the lock
void lock(lock_t *);

// Precondition: We own the lock
// Postcondition: We no longer own the lock
void unlock(lock_t*);
```

If we don't own a lock, we will be forced to wait while some other thread owns the lock until we can acquire and lock() it ourselves. unlock() will happen instantly. Assume for now that lock() is busy-waiting until we can get the lock. This busy waiting lock is called a **spin lock**.

Let's use them:

```
lock_t l;

char readc(struct pipe* p)
{
    lock(&l);

    // If read end == write end, full pipe
    while (p->r == p->w)
        continue; // Busy waiting

    // Return a char, and then move forward
    // the read end afterward
    return p->buf[p->r++ % PIPE_SIZE];

    unlock(&l);
```

```
}

void writec(struct pip* p, char c)
{
    lock(&l);

    // If the difference between the two
    // pointers is PIPE_SIZE, the pipe is full
    // This works even if w overflows and wraps
    while (p->w - p->r == PIPE_SIZE)
        continue;

    p->buf[p->w++ % PIPE_SIZE] = c;

    unlock(&l);
}
```

"I try to write bugs into every line of code I write."

Fairness issue: Suppose we can't read because the pipe is empty. So we wait for nothing, but we acquired the lock, and so no other thread can proceed. We're stuck waiting. We can't simply move the lock() and unlock() around, as we may still have a race condition to check if we can read/write.

Major red flag: Potentially unbounded amount of computation inside of the busy-waiting while-loop. Let's unlock there:

```
char readc(struct pipe* p)
{
    lock(&l);

    // If read end == write end, full pipe
    while (p->r == p->w) {
        // Yield while waiting; assuming fairness
        // from OS scheduler
        unlock(&l);

        // We exit the loop, lock in-hand
        lock(&l);
    }

    char c = p->buf[p->r++ % PIPE_SIZE]

    unlock(&l);

    // Return a char, and then move forward
    // the read end afterward
    return c;

    // No unlock() should ever happen
    // after a return; it'll never happen!
}
```

Lock Grain

Problems:

1. We're still spinning with that damned while-loop.
2. Suppose two pipes, A | B with associated pipe P, and C | D, with associated Q. Notice that we have a single global lock. Any exclusive access to one pipe is excluding access to other pipes. This will not scale at all. Our lock is **too coarse-grained**, excluding too many accesses which are not really a problem.

To fix (2), we'll have a **finer-grained lock**, one per pipe. The lock will protect the minimum. This promotes fairness, but

increases memory costs and complexity.

```
#define PIPE_SIZE (1 << 12) // 2 ^ 12 bytes
struct pipe
{
    lock_t l;
    char buf[PIPE_SIZE];
    unsigned r, w;
}

char readc(struct pipe* p)
{
    lock( &(p->l) );

    // If read end == write end, full pipe
    while (p->r == p->w) {
        // Yield while waiting; assuming fairness
        // from OS scheduler
        unlock(&p->l);

        // We exit the loop, lock in-hand
        lock(&l);
    }

    char c = p->buf[p->r++ % PIPE_SIZE]

    unlock(&l);

    // Return a char, and then move forward
    // the read end afterward
    return c;

    // No unlock() should ever happen
    // after a return; it'll never happen!
}
```

An example of too fine a grain:

```
bool isempty(struct pipe* p)
{
    lock(&p->l);
    bool ise = p->r === p->w;
    unlock(&p->l);

    return ise;
}
```

Suppose 100 threads come in and ask isempty(). Then, they lock/unlock alternatingly, which effectively causes serialization.

To solve this, suppose we have separate locks for reading and writing:

```
#define PIPE_SIZE (1 << 12) // 2 ^ 12 bytes
struct pipe
{
    lock_t rl, wl;
    char buf[PIPE_SIZE];
    unsigned r, w;
}


bool isempty(struct pipe* p)
```

```
{
    lock(&p->rl);
    bool ise = p->r === p->w;
    unlock(&p->rl);

    return ise;
}
```

This is too slow!

We want to have: at most 1 writer, but as many readers as possible (as nothing is being modified).

Discussion #5: 6 February 2015

Implement a kernel module

Ramdisk: a disk that is in memory Support read/write to it (simple) Support read/write locking (which is about synchronization--hard)

disk | memory | cache

Kernel Module

A piece of kernel code. It can be loaded/unloaded into kernel dynamically to change kernel behavior wihtout need to reboot the system.

In order to execute the code we write, it has to be loaded into the kernel, which changes the kernel behavior. Example: device driver.

Why don't we compile everything ahead-of-time? Because otherwise we incur overhead: the entire kernel has to be loaded into memory. Also, we may need to support many different devices, each of which needs a different driver.

Thus Windows (which doesn't have kernel modules) takes up 20 GB on disk and 2 GB RAM. But a highly customized Linux kernel and glibc < 20 MB.

Code Layout

osprd.c and osprd.h are the module source files, and are the one main ones we'll modify.

osprdaccess.c is a user-level agent to access the Ramdisk.

spinlock.h has wrappers for spinlock.

lab2-tester.pl has basic tests.

The rest are scripts for QEMU and setting up environment.

// All of this is done in the custom Ubuntu image.

In osprd.c:

```
static void osprd_process_request(osprd_info_t* d, struct request* req);
```

See slides for details on req struct.

Has arguments for current device (defined in header) and a struct holding the request (defined in Linux kernel). The kernel

abstracts away the s

```
int osprd_close_last(struct inode* inode, struct file* filp)
```

This is called when the user-level agent releases the file. It frees all the locks, and wakes up threads waiting on this file.

```
int osprd_ioctl(struct inode* inode, struct file* filp,
                unsigned int cmd, unsigned long arg)
```

We'll implement locking here. Follow the spec's advice and go incrementally on it.

The way the user agent interacts with the Ramdisk is done via the sytem calls we implement. The user agent may say open() on /dev/osprd[a-d], but we need to serve this request with Ramdisk-specific sys calls like osprd_open. To setup this connection, look into the setup script:

```
mkdnod /dev/osprda 222 [0-3]
```

We give major and minor numbers that identify our device to the kernel. The osprd_ops struct has function pointers to all of our of osprd functions. Then, we call a system call add_disk(), which lets the kernel know about this device. When the kernel sees open(/dev/osprda) later, it'll look up the matching major/minor number, and then go to to the pointed-to function.

Spinlock

Spinlock: Acquire a lock; if occupied, keep waiting

Semaphore: Acquired a lock; if occupied, put process to sleep. Process/thread will be woken up later when semaphore released.

We don't use semaphores to replace all spinlocks, since we introduce overhead in context switching. If lock requirement time is in millisecond magnitude, spinlock preferred. If the time is longer than that, we use should use a semaphore.

"task = thread", and sometimes spec should say "thread" or "process", and not vice versa.

See slides on dynamic memory allocation.

Note: if kmalloc()/kzalloc() use more than PAGE_SIZE (4096 KB), we may get very strange behavior. Use alloc_pages() and free_pages() in this:

```
alloc_pages(GFP_KERNEL, order)
```

If order = 0 => one page If order = 1 => two pages If order = 2 => four pages

```
free_pages(addr, order)
```

Where addr is returned by alloc_pages().

wait_event_interruptible

This is is meant to take in a head for wait queue.

```
int wait_event_interruptible(wait_queue_head_t q, CONDITION);
```

If we make this condition false, a thread won't wake up unless we call the wakeup function. This eliminates the magic wake-up.

Wake-up will also check this condition and wake if true, ignore if false. The LXR specified in the spec is way old, and died a long time ago. The 2.6.18 code is over a decade old. To get the old code, go to kernel.org. We're on 3.18.6 now.

printk

Similar to printf(), but in the kernel. The info doesn't always show on screen though. To see that, use dmesg(), if the kernel crashes/hangs, we won't see that either.

Solution:

```
// Change level to always show on screen
#define myprintk(__str, ...) printk(KERN_EMERG__STR, ##__VA_ARGS__)
```

Or do:

```
// This will show a lot of noise from other kernel modules
echo 8 > /proc/sys/kernel/printk
```

We passed in a formatted string, followed by variables.

Note, actual fault may be somewhere after the printk(), as it calls schedule(). printk() inside of schedule() will cause an infinite loop. If code behaves strangely after printk() calls are introduced, comment them out.

Lecture #9: 9 February 2015

How to implement critical sections

Previously:

```
void lock(lock_t*);
void unlock(lock_t*);

typdef      lock_t;
```

How shall we implement these primitives?

Some simplifying assumptions:

- Single processor (single core)
- No interrupts (timer interrupts, signals, pre-emptions by kernel's scheduler, etc.)

Then, it's a trick question: We don't need lock()/unlock(). Control can never be suddenly lost, and multi-threading really means just voluntarily handing over control. There's no risk that our critical work will suddenly be stopped and another process take our place on the CPU.

Then, lock()/unlock() are NOOPs (in the portable library implementation). It's trivial.

Now, let's dump the second assumption. We can now be interrupted or signaled, with sudden, unpredictable loss of control.

Assume Machine A has both assumptions. Assume Machine B has only the first assumption, plus a third assumption that there is a third way to temporarily pause handling interrupts, e.g., make a pending queue.

Then, with this 3rd assumption:

// setPriorityLevel() lock: splx(6); // No interrupt unless of // level 6 importance or greater

unlock: splx(4);

This is very fast.

Finally, let's dump the first assumption. On a multi-processor machine, this won't work. We'll have a race condition between processes

Let's implement:

```
typedef int lock_t;

void lock(lock_t* l)
{
    while (*l)
        continue; // Busy-waiting, a spinlock

    &l = 1; // Unlocked
}

void unlock(lock_t* l)
{
    *l = 0;
}
```

This doesn't work. Checking and setting aren't atomic. Two different CPUs may both think the lock is available if a signal arrives just as we get past the while-loop.

Another issue: Assuming we have 32-bit ints, but the memory bus is 16-bits wide. And our load instruction is implementing as storing the top 16 bits, then the bottom 16 bits. If we store in the opposite order than we load, then there's a race condition just in the dereference, with half of the value and half of the previous value. We're assuming loads and stores are atomic. We'll get a value that no one ever stored, as we're loading in the middle of a store.

Atomicity in loads and stores are not guaranteed atomic in modern processors. On x86: Not guaranteeed for > 4 bytes Same thing for unaligned access on data that doesn't line up every 4 bytes.

We can see this with a union:

```
union {
    int a[2];
    char c[8];
} u;
```

Guaranteed to align on a 4-byte quantity as it possibly has ints.

```
u.a[0] = 12;
u.a[1] = 97;

char* p = &u.a[0];
p++;
int* q = (int *)p;
```

```
// Unaligned access!
*q
```

Little Endian on x86:

```
[   .  .   .   |   .  .   .   ]
                        ^   ^
                        |   |
                        q   u
```

We get 97 << 24. We get a mishmash of combined values.

Adding to the problem list: Small than 4 bytes? Watch out.

Example:

```
struct {
    bool a: 1;  // Bit field; I live in just 1 bit
                // Normally uses 1 byte otherwise
    bool b: 1;
    char c;

    int x
}

             padding   padding
[    x     | ... | c | .. |B|A ]
```

Then we use:

```
s.a = 1
s.b = 1
```

As instructions, we might have load, set, and then store again. The problem is if we have a race condition between processors, where both are loading and storing.

"I feel like I'm leaving the impression that you can't do anything, that you're toast, no matter what."

"Whenever you have multiple processors accessing the same memory, your natural assumption ought to be that it isn't going to work."

But on x86: If we have *aligned* loads and stores of ints, this is guaranteed atomic.

But in a strange way:

```
[CPU 0]    [CPU 1]
    |         |
    ---       |
       |      | BUS
-----------------------
            |
            |
           RAM
```

"You are engineers, you are in charge of the magic."

The CPUs will keep track of what the other CPUs are loading/storing, such that they never get a mish-mash of old/new values. It's atomic to be old xor new.

Summary: Atomicity fails if:

- Data is not properly aligned
- Data is too large
- Data is too small

Modern CPUs allow snoopping between CPUs, which allows them to compare valuces in cache for the same data and realize when they are out-of-date. This allows for cache coherence for loads/stores of 32-bit values under x86.

Back to lock/unlock

```
typedef int lock_t;

void lock(lock_t* lock)
{
    while (*l++ != 0)
        continue; // Busy-waiting, a spinlock

    &l = 1; // Unlocked
}

void unlock(lock_t* lock)
{
    *l = 0;
}
```

The race condition is gone at the source code level, (and there's still an overflow level) but there's still a problem at the implementation level. We have either a load and store, or an `incl`. But `incl` 's implementation does load, add 1, store, and is not guaranteed atomic.

Even so, standard checks like:

```
while (*lock++ != 0)
    continue;
```

will not work, as we need the load, store, add instructions. Even ones like incl (increment) are composed of multiple, non-atomic operations in the x86 microcode.

We can't do this. So how do we lock/unlock assuming only atomicity for aligned loads/stores? This is possible. With enough cleverness, it's actually possible to do this even without that assumption (Leslie Lamport figured it out, but it was complicated and slow).

What do we do when we face these problems? Software friends call up hardware friends and ask for help.

Special atomic x86 instructions

There's now a large set of these atomic instructions.

lock incl

Normally, `incl x` increments a long x. But, `lock incl` is guaranteed to be atomic. It sends a signal to the bus, and the other CPUs see this, avoids it, and then the bus gives an all-clear signal to all the CPUs again. It is thus significantly slower. We can package this into a C function.

```
void lock_incl(int*)
{
    asm("lock incl");
}
```

With this instruction in hand:

```
typedef int lock_t;

void lock(lock_t* lock)
{
    while (true)
    {
        int x = *lock;

        // Signal could arrive here
        // and mess everything up

        lock_incl(&lock);

        if (x == 0)
            break;
    }
}

void unlock(lock_t* lock)
{
    *l = 0;
}
```

This still doesn't work because of the possibility of a signal arriving before the lock_incl().

xchgl

This atomically exchanges memory with a register.

Modeled at the C-level:

```
int xchgl(int *val, int new)
{
    // Temporary variable to hold
    // old value for returning
    int old = *val;
    *val = new;

    return old;
}
```

This is all done as a single, atomic asm instruction.

Using this:

```
typedef int lock_t;

void lock(lock_t* lock)
{
    // Old value returned was 0, then got the lock
    while (xchgl(lock, 1))
    {
```

```
        continue;
    }

    // Return from lock()
}

void unlock(lock_t* lock)
{
    while (!xchgl(lock, 0))
        continue;
}
```

Problem:

Precondition for lock() is that we don't own the lock. It's an obligation on the caller. If we already own a lock, we get an infinte loop.

This is because xchgl() does not return ‘0’ (which gets us out of the while-loop) until the old value of the lock was 0, meaning that another process released the lock.

If we try to double-acquire, the old value will always be ‘1’, and we’ll loop infinitely while trying to lock

Precondition for unlock() is that we already own a lock. So we don't need to check that the xcghl() succeeded:

```
void unlock(lock_t* lock)
{
    xchgl(l, 0);
}
```

Faster (as stores are atomic):

```
void unlock(lock_t* lock)
{
    *l = 0;
}
```

The above code is ambiguous. Much clearer:

```
void lock(lock_t* lock)
{
    // Old value returned was 0, then got the lock
    while (xchgl(lock, 1) != 0)
    {
        continue;
    }

    // Return from lock()
}

void unlock(lock_t* lock)
{
    while (xchgl(lock, 0) != 1)
        continue;
}

// OR:

void unlock(lock_t* lock)
{
```

```
    xchgl(lock, 0)
}

// OR:

oid unlock(lock_t* lock)
{
    *lock = 0;
}
```

We're building larger locks from smaller ones, such that we can build atomic operations of approximately the right size. Then, we avoid race conditions but also get performance. If everything is a critical section then we lose parallelism altogether; sequential operations don't have to deal with all these synchronization issues.

Back to the ATM example. This snippet is NOT thread-safe (meaning that it cannot be multithreaded without risk of data-stomping).

```
int balance;
bool withdraw(int amt)
{
    // A signal/interrupt/pre-emption could arrive here,
    // and another request might go through, even
    // though the two withdrawals combined might lead
    // to a negative balance

    if (balance < amt || amt < 0)
        return 0;  // False

    balance -= amt;

    return 1;      // True
}
```

This can be made thread-safe:

```
int balance;
lock_t balance_lock;

bool withdraw(int amt)
{
    // START critical section
    lock(&balance_lock);

    bool valid = (0 <= amt && amt <= balance);

    if (valid)
        balance -= amt;

    // END critical section
    unlock(&balance_lock);

    return valid;
}
```

To improve performance, we'll reduce the size of our critical section.

compare_and_swap

C-model of the instruction:

```
bool compare_and_swap(int* val, int old, int new)
{
    // Check to make sure that old value matches expected
    if (*val == old)
    {
        // Update value
        *val = new;
        return 1;
    }
    else
    {
        // Failed; value did not match what we expected,
        // e.g., another atomic instruction modified it
        // in the meanwhile, and we need to try again

        return 0;
    }
}
```

We can now improve withdraw()'s performance:

```
bool withdraw(int amt)
{
    while (true)
    {
        int tempBalance = balance;
        if (amt < 0 || tempBalance < amt)
            return 0; // False

        // This is where the actual lock happens
        if (compare_and_swap(&balance, tempBalance, tempBalance - w))
            break; // True
    }

    return 1;
}
```

We're recomputing the new balance in case of sudden changes. This has less contention, as the code preceding compare_and_swap() can do whatever.

Trade-offs:

- Nicer performance
- If there's contention, it can chew up CPU, e.g., an expensive function is used to compute b - w
- Assumes state fits in 4 bytes (for atomicity)

For something more general, we'll need lock()/unlock().

Pipe Example

```
struct pipe {
    char buf[1024];
    size_t r, w;
    lock_t l;
};

char readpipe(struct pipe* p)
{
    while (true) {
        lock(&p->l);
```

```
        if (p->r != p->w)
            break;

        unlock(&p->l);
    }

    char c = p->bug[p->r++ & 1024];
    unlock(&p->l);

    return c;
}
```

Assume we're a fast reader and there's a slow writer, e.g. cat and sort. Then, we're constantly locking and unlocking while waiting for the pipe to be filled.

We need a way to tell the operating system: "Please don't schedule me to run until/unless there's data in the pipe."

Blocking Mutex

Mutex == "mutual exclusion"

It's like a spinlock, but when we try to grab a resource, the process/thread will sleep if the resource is not available. We don't spin. It'll be woken up upon the unlocking of the resource.

We can construct this higher-level synchronization primitive via the lower-level sync. primitive of a spinlock. It's internal to our implementation.

```
typedef struct bmutex {
    lock_t l;
    int locked;
    struct proc* blocked; // Linked-list of blocked processes
                          // waiting on this bmutex to be unlocked
} bmutex_t;
```

We're assuming the process table has a `next` field. Implementing:

```
void acquire(struct bmutex* b)
{
    while (true)
    {
        lock(&b->l); // Lock the spinlock
        if (!b->locked)
            break;

        cp->next = p->blocked; // Prepending to list of waiting processes
        b->blocked = cp; // current process pointer in register
        yield();

        unlock(&b->l);
    }
}
```

But look! We're yielding while we still have a spin lock. No one else will get that resource. Move it:

```
void acquire(struct bmutex* b)
{
    while (true)
    {
```

```
        lock(&b->l); // Lock the spinlock
        if (!b->locked)
            break;

        cp->next = b->blocked; // Prepending to list of waiting processes
        b->blocked = cp; // current process pointer in register
        cp->status = BLOCKED;

        unlock(&b->l);

        yield();
    }

    // We get here when we break
    b->locked = 1;
    unlock(&b->l);
}
```

And analogously:

```
void release(struct bmutex_t* b)
{
    // Need to wake up the waiting processes
    // We'll wake just the first
    *b->locked = 0;
    struct proc* p = b->locked;
    b->blocked = p->next;
    p->status = RUNNABLE;
}
```

But wait, race condition again!

```
void release(struct bmutex_t* b)
{
    lock(&b->l);

    // Need to wake up the waiting processes
    // We'll wake just the first

    *b->locked = 0;
    struct proc* p = b->locked;
    b->blocked = p->next;
    p->status = RUNNABLE;

    unlock(&b->l);

}
```

From TBP Tutoring with Mickey

Assume we have a shared resource, say a disk, and we guard this using a mutex. We could do so with a spinlock, but that wastes CPU time. A mutex itself is locked with a spinlock (for us acquiring the mutex) but we expect the time between locking the mutex's spinlock and unlocking it to be very short.

In general: We're serializing access to a shared resource for multiple processes/threads to prevent race conditions.

Here's the mutex:

```
// "Blocking mutex"
```

```
struct bmutex {
    bool locked;
    spin_lock_t l;
};
```

Then here's the acquire() code for acquiring a bmutex:

```
void acquire(bmutex* b)
{
    while (b->locked)
    {
        unlock(b->l);
        schedule()

        lock(b->l);
    }

    // If here, mutex available for acquiring
    // b->locked will be set to false by release()
    // The mutex isn't explicitly associated with
    // a process/thread; it's just available to one
    // at a time by design

    b->locked = true;
    unlock(b->l);

}
    // Lock the mutex in order to check if it's locked
    // Multiple processes/threads may be trying to
    // acquire()/release() the mutex, so we need to serialize that

    // While mutex is still held by someone else,
    // we'll unlock, sleep until it's released,
    // which means we're added to the queue of blocked processes,
    // (OS will handle our wake-up), and then lock
    // the mutex again

    // We need to lock again in order to see if mutex
    // was acquired in between the time we were woken up
    // and tried to acquired it for ourselves
```

Summary of Possible Problems

On x86:

- Longer than 4 bytes
- Unaligned access
- Smaller than 4 bytes

Detour: semaphore

What we have for blocking mutex, but not quite.

Semaphores, we have an:

```
int avail; // # of simult. locks allowed
```

It allows multiple threads to have a lock, up to a limit, where we prefer preformance or have a quota on number of

simultaneous users, etc.

When we have acquire(), we'd decrement the avail. count.

P: analogous to acquire; decrement avail; from "prolaag" V: analogous to release: increment avil; from "verhoog"

The guy who invented semaphores: Dijkstra.

Pipe with Blocking Mutex

Now:

```
char readpipe(struct pipe* p)
{
    while (true) {
        acquire(&p->b);

        if (p->r != p->w)
            break;

        release(&p->b);
    }

    char p = p->bug[p->r++ & 1024];
    release(&p->b);

    return c;
}
```

Problem: After the first release(), we still have control of the CPU, and we continuously acquire the lock in a cycle. But, the pipe is still empty. We only want to be awoken for a non-empty pipe.

Let's write a function that lets us know when the time is right:

```
bool are_we_there_yet(void)
{
    return p->r != p->w;
}
```

First problem: p, w is out of scope.

Okay, we so modify it:

```
bool are_we_there_yet(void* p)
{
    return p->r != p->w;
}


char readpipe(struct pipe* p)
{
    while (true) {
        wait_for(&p->b, are_we_there_yet, p);
        acquire(&p->b);

        if (p->r != p->w)
            break;

        release(&p->b);
```

```
    }

    char c = p->buf[p->r++ & 1024];
    release(&p->b);

    return c;
}
```

But this still chews up CPU time while waiting. We need a new idea.

Condition Variables

It's a blocking mutex, plus an extra boolean variable that keeps track of the condition (known to programmer, and noted in comments, but not known to the OS).

Its API:

```
// Precondition: Acquired the bmutex

// Args: Condition variable, blocking mutex
wait(condvar_t*, bmutex_t* b)
{
    // Releases b then blocks until some
    // other thread notifies us that condition
    // may be true now.

    // Then, it reacquires b and returns
}

notify(condvar_t * c)
{
    // Tell OS that condition is true
    // OS will wake up one process blocked while
    // waiting on condition
}

broadcast(condvar_t* c)
{
    // Like notify(), but wake up everybody
}
```

Modify pipes implementation:

```
struct pipe {
    char buf[1024];
    size_t r, w;
    bmutex_t;
    condvar_t nonempty, nonfull;
};

char readpipe(struct pipe* p)
{
    acquire(&p->b);

    while (true) {
        if (p->r != p->w)
            break;

        // When wait() returns,
        // we've been notified pipe
```

```
        // is nonempty, but need to loop
        // and check for other processes
        // that may have messed with pipe
        wait(&p->nonempty, &p->b);
    }

    char c = p->buf[p->r++ & 1024];
    release(&p->b);

    // Let reader-s know
    notify(&p->nonfull);
    return c;
}
```

Read-write lock

Readers just inspect data-structures; they're read-only. Writers typically read and write. Everything so far, even readpipe(), has been a writer (as readpipe() modifies read-end pointer).

If we have a lot of readers, and few writers, then we want a new lock type that allows many readers.

This is a read-write lock with the following characteristics:

- Allows many readers
- At most 1 writer
- If any writers, no readers

This can be done as spinlocks, blocking mutexes, or with condition variables.

Lecture #10: 11 February 2015

// Ooh, Eggert has a mic today. // I asked him randomly about he // got started with GNU stuff too. // He was at a startup at the time // in the mid-80s and needed to get // a development environment up and going

// They chose Emacs, but it didn't work, // so he sent in patches and got in touch // with Richard Stallman, who thanked him // and asked him for more

// Ah, sore throat/cold, thus the mic

Hardware Lock Elision

We're like to go faster, especially for our spin locks. Our problem is that the spin locks inherently make everyone else wait.

`xchgl` slows things down.

A new technique on Haswell and beyond is called **Hardware Lock Elision**.

We do locking, but cheat to get performance. We execute a variant of xghl, and give a prefix to it:

```
mutex -> [ 32-bit int ]

// 0 when not acquired; 1 when acquired

//////////


lock:
```

```
    // Value to load into the mutex
    movl $1, %eax

try:

    // Swap eax value with mutex
    xacquire lock xchgl %eax, mutex

    // eax should now hold zero
    cmp $0, %eax

    // Try again if failed;
    // jump if not zero
    jnz try

    ret

unlock:

    xrelease movl $0, mutex
```

Recall that we had the `lock` prefix too.

Everything done up until the xrelease is done provisionally in cache; nothing is stored in the memory until then. The CPU can check at the xrelease point and see if someone else acquired the lock. If they did, it jumps back to the xacquire and causes it to fail.

It's like it never happened, a dream. We're relying on caches having enough memory to track everything. Thus, this only works for a limited amount of work.

Deadlock

Deadlock cannot happen with really simple code. It happens with more complicated code.

Something like the deposit()/withdraw() example is a simple lock grab/release. It can't deadlock.

But:

```
deposit($3, acct#19);
withdraw($4, acct#27);

transfer($2, acct#19, acct#27);
```

Now we need to do something that involves making an atomic action on two objects.

Now consider:

Thread 1: tries to transfers from 19 to 27 Thread 2: tries to transer from 27 to 19

Now, thread 1 gets 19, thread 2 gets 27, and they then try to get a lock on the other account. They can't. Boom: deadlock.

This can't be prevented with fancier locks. It's a fundamental issue to lock operation. Hardware locks, conditional flags, etc. cannot solve this either. We'll just spin faster or self-block.

Deadlock is a global, non-obvious property that's not apparent from looking at any one part of a system.

By trying to solve one race condition, we introduced so many locks that we got another race condition: deadlock.

Can we always prevent deadlock or do we accept the possibility of deadlock?

Conditions for Deadlock

Mutual Exclusion

If one thing has acquired a resource, no one else can access it.

Circular Wait

It's a cycle in the graph of processes waiting for each other to release something. It could be a long, convoluted cycle. It could even be a process waiting on itself.

No pre-emption of locks

Many OS' offer the ability to override locks. We override the lock, force the other person off, and we grab it for ourselves.

If pre-emption is offered, no deadlock.

Hold and wait

One resource is held while waiting for another resource to become available.

Without this ability, deadlock is impossible. For example, a system that only allows one lock to be held at a time.

Preventing any one of these conditions prevents deadlock.

Solving Deadlock

Detecting Deadlocks Dynamically

Here, the OS tracks every process and the resources they hold or are trying to hold. It builds a directed graph and makes sure it's acyclic. If a cycle were to be established based on the new arc, then the lock request will be rejected, giving an error status like EDEADLOCK for lock().

The kernel is helping applications, but apps must now be modified to watch out for deadlock.

Redesign so that deadlocks cannot occur

One way to do it: **lock ordering**.

If a primitive needs more than one lock, we impose this lock ordering algorithm:

1. We assume that locks are ordered and numbered. On Linux, we can just look at the memory address:

```
lock_t a, b;
&a < &b;
```

2. We acquire locks in order. If all successful, we're fine. But if a lock is not available, instead of waiting on that lock, we release all locks and start over. This assumes for blocking mutexes that we can try to acquire without waiting: try_and_acquire(). It never blocks; it just immediately fails.

The only people that wait are the ones waiting for their first resource. No one holds and waits, so the fourth condition of hold-and-wait is broken.

Priority Inversion

Locking and scheduling can interact in unexpected ways.

Mars Pathfinder was lost in 1997 because of this bug. JPL's SWEs are the best in the world, but this happened.

Three threads, named by priority:

```
T_low          T_medium           T_high
```

T_high keeps the antenna pointed at Earth. T_medium does something like check the battery. T_low does some background work.

Suppose we begin with:

T_low runnable T_med waiting T_high waiting

T_low is running, so it grabs a lock. But then, we get a context switch to T_high, which just became runnable. It tries to grab the same object, but it can't, because T_low has it. T_high is blocked, and yields. We shoud get a context switch back to T_low, but suppose T_med is now runnable (woken up by some other previously blocking process).

The scheduler chooses T_med over T_low, and so T_low never gets to run--which means T_high never gets to run. This interaction between locking and scheduling is pernicious.

Some solutions:

1. Adding pre-emption: T_high can beat off T_low But we risk data corruption with this.
2. Temporarily elevate the priority of a process that holds a lock on a resource desired by a high-priority process. We borrow the priority of the other process.

Livelock

Our threads are running (not blocked), but no progress is made. This is even harder to debug compared to deadlock. Everyone looks fine, but nothing is really happening.

Example: receive livelock.

Suppose a bounded buffer that receives requests at up to 100 Gb/s

But we can only service them at 10 Gb/s

Our goal graph:

services done:

```
   ----------
  /
 /
/
  10 Gb/s

  (input load)
```

Service really has more than one part:

1. The actual computation
2. Interrupt service handler for packet-handling chews up more and more time as more packets flood in

The buffer may fill up, but we'll spend time tossing and accepting new packets. We'll always be doing something: handling a packet or computing (a little bit), but performance sucks:

```
     .-----.
```

```
    /       \
   /         \
  /           \
 /             \
```

Simple solution: Mask out interrupts once the buffer is full, and re-enable them afterward.

Event-driven programming

Locking code is a bug magnet.

We'd love to eliminate locks altogether, while still preventing races and attaining performance.

The competing approach, used especially in embedded systems: event-driven programming.

First thing we do: toss multi-threading. Only one thread now. We still want to do deal with asynch. events. Fundamentally, threads are used alongside partitioning of the work. Here, we instead break up computation into a series of events. Then, our program will consist of a set of event handlers.

At runtime, we'll have a queue of incoming events. Computation now pops off the first event from the queue, runs its handler, and then we loop through the queue.

An important rule: each event handler may wait for nothing. Do and exit. No waiting for read() to finish, etc. The gola is to finish quickly. The overall effect is that each event handler is its own critical section. There's no pre-emption or need for locks.

For handling priority, we can just have a priority queue. The event handler will start a read(), add that to the priority queue, and then exit.

TinyOS is an event-driven example. So why isn't this used? Simple: it doesn't scale to multicore systems. The proponents of this paradigm would say no to multicore altogether. Scaling would happen via many single cores communicating via the network.

File Systems Performance

"We get to talk about fast file systems without yet understanding how they work."

GPFS

A few years back, Eggert dealt with something that had 120 PB spread across 120,000 600GB hard drives.

So 600 GB and not something larger?

Student: "Something something parallelism?"

// LOL

Yes, and also: sweet spot between speed and capacity. These were 15K RPM drives, and also quite expensive.

First idea: We can't hook up all 120K drives to a single bus and CPU. We'll totally flood the bus. Instead, we'll form nodes of a CPU and bus controlling a few drives each. Then, we'll link the nodes together via a network. We'll give the illusion of a really fast, really big computer.

Stripe Like Crazy

One way to handle adding a new 500 GB file is to put it all on one drive. But then we'll be bottlenecked by that one drive. Instead, we split up the file across many drives, and then read them all at once later when needed. We can have all sorts of striping schemes to organize the data.

Data: [| | | | | |]

But how do we keep track of everything? One answer: we centralize all that information into a directory node. This is a popular approach, but then the directory node becomes the bottleneck.

Distributed Metadata

The data about the data, like where everything is, is distributed, rather than centralized.

Efficient Directory Indexing

For small directories, we could have a linear directory index that's like an array.

Problem: Listing 1 million files in a directory means ls must load all 1 million entries into RAM and then sort them.

Now, even deleting a file takes forever. We'd have to load a whole index into memory.

So instead of an array or hash table (which sucks at sorting) or a balanced binary tree (too bushy), we use a B+ tree, an n-ary tree with a large number of children per node.

Distributed Locking

We need to make changes atomically.

These locks live at the whole system level, not just in RAM on one node. It's not easy, but it's necessary.

Partition Awareness

Even when a network of nodes is split up, we want to remain aware of splits. This might happen with a physical cable being cut.

Easy solution: Count how many nodes on each side of partition. The larger partition wins ands keeps up read/write ops, but the smaller parition can only read now.

I/O Performance

Hard disk: cheaper per byte, but high latency

At 7200 RPM => 120 Hz => 1/120 latency worst case, 1/240 average case (4.1666 ms)

Seek time: accelerate read head, decelerate it halfway to a stop. A Seagate Barracuda LP 2TB @ 5900RPM has 5.5 ms average latency. 5.5 ms that is billions of CPU instructions. HDDs are unbelievably slow compared to our CPUs.

Some more stats:

- 300 Mb/s external transfer rate (for talking to the bus)
- 95 Mb/s sustained data read (pumping out data continously) (but it's the 300 Mb/s that appears in the ad).

Why is this slower, even though it's assumed to be a sequential read? Because that's as fast the data goes under the read head. The disk controller connection to the bus is faster, i.e., its cache is 300 Mb/s, but the disk to the cache is only 95 Mb/s at best.

- 32 MiB cache
- 512 bytes/sector (traditional)
- 50,000 contacts starts/stops.

The read heads float on a thin film of air while in operation, but they rest on the surface when off. This means 50,000 on/off

cycles until they wear out. SEASnet, etc. never bothers turning off the drives.

- 10^{-14} non-recoverable read error rate

1 out of 10^14 bits will be fail to be read properly, despite internal error correction (Read-Solomon, etc.); that's 1 bit out of 10 TB

- 0.32% AFR: the annualized failure rate. The whole HDD is toast.

Think about: it: 1000 drives. Every year, we expect 3 to fail. That's actually quite a few when we have a lot of drives. Data centers regularly switch out drives.

- 6.8 W average operating power consumption 5.5 W average idle (no spinning) 2 A startup current (12 V peak)
- 70 G operating shock. It can keep going up to 70 Gs without breaking.
- 300 G nonoperating shock. Shake it even more while off.
- 2.6 bels acoustic rate. How loud it is while operating.

Flash: Random access is faster, but more expensive

Example: Corsair Force GT, with a capacity of 60 GB.

- 300 Gb/s transfer rate
- 280 MB/s read
- 270 MB/s write (megabytes, not megabits)
- 1000 G operating.

HDDs aren't suppposed to dropped, but flash can. It weighs 80 grams, haha.

- 2.0 W max 0.5 W idle
- MTBF (mean time between failures): 1,000,000 hours

Eggert doubts this and the AFR, as they're just (optimistic) estimates.

The big downside of flash: it wears out. The exact number of lifetime read/write cycles varies by device. This is why we have **wear leveling**.

We want to read/write to blocks on the flash drive. The flash controller recognizes the fact that most file systems have hot spots, something that's updated often. On a flash drive, that's a problem if we always write to that one spot, and we'll wear out quickly. We'll lose capacity and lifetime. The flash controller lies to us. It spreads out writes among the blocks, so that they wear out evenly.

So how would we implement a wear-levelling algo? We'd need mapping between logical and physical blocks. And we store this on the flash drive, but we can't do it all on one block, because then we get hot spots. Historically, flash drives have a bad reputation for reliablity, due to the unreliable wear-levelling code.

Scheduling and File Systems

At any given point in time, our OS will have a bunch of things to get done:

A set of requests for each disk/flash drive. We need to choose which read/write requests to process first. Simplest one: FCFS: first-come, first-serve. Let's analyze the performance of this scheduling algo.

Assumption: Abstract device which acts like a disk (with seeks needed). Assume that I/O expensiveness depends on

distance between read head and desired read location. FCFS will turn out to be a bad choice.

// So that's why elevator algo popped into my head

Discussion #6: 13 February 2015

Lab 3 is a file system, and we have two weeks to do it. Diyu thinks that it is the hardest lab.

See slides for layout of our disk

Boot sector holds the MBR. BIOS in EEPROM boots this code. Though modern disks (with partitions) often have the MBR boot a VBR.

Super block contains magic number, which tells us which file system will be used. The OS checks to know which file-system to expect on this disk. It is meant to tell us about the properties of the file system.

The free block bitmap is actually composed of several maps, and tells us which blocks are free to use.

Next come the inodes, which stores metadata file about each file, directory, hard link, or symbolic link. These blocks store the inodes, which also track where files are in the actual data, which follow the inodes block.

Diyu does not know how Windows handles files.

Building the Disk

ospfs_data[] is the actual disk, and the rest of the skeleton code assumes its existence.

See slides.

ospfs_block() is designed to whether or not we overflow. It works by checking to see whether not we're overflowing.

If we're out-of-bounds on ospfs_data here, the result is undefined.

MODIFY the function to check if block number is larger than allocated amount. Otherwise, we'll get undefined behavior with that array access.

Superblock

os_blocks in struct ospfs_super is the number of blocks on disk.

Superblock is second block, and boot sector is only one block, the first. So we get to the superblock by acessing ospfs_data[OSPFS_BLKSIZE].

Free block bitmap

bitvector() functions are used to access the free block bitmap.

0: used 1: free

i.e., if (1) == true == free

question is: "Is this block free?"

bitvector_set(void* vector, int i);

This sets a single bit at index i. We would read block_bitmap[] from memory and pass it to these functions. Having the `vector` argument

bitvector_set() turns the bit into 1 bitvector_clear() turns the bit into 0

The index i is starting at 0.

bitvector_clear() bitvector_test(): returns 0 if 0, 1 if 1

Starting block: 2 Ending block: bs_firstinob - 1

Free block bitmap will show whole disk, but first two bits will always be 0, due to boot sector and superblock.

We're assuming the type of block_bitmap[] is a (char*).

C doesn't support operations with single bit granularity. We need these functions. We're checking 1 byte at a time if we do it directly, i.e. 8 bits.

allocate_block() and free_block()

allocate_block(): Fail if all blocks used up.

Just traverse the block bitmap linearly to find a free block, marked by 1. If none available (all 0), return 0. bitvector_clear(allocatedBlock) to mark as now used.

free_block(): Free that specific block number. Do this via bitvector_clear(blockNum).

Looking at the bitvector() functions, Diyu found bugs. Individually, the functions are fine. But what happens when we used the alloc_block() and free_block() functions? If we use bitvector() functions inside of them with multiple CPUs and/or threads, the block bitmap can get messed up. We don't have synchronization. If two CPUs both allocate, they'll both do bitvector_test(), but we may have a sudden context switch to the other CPU. Both think the block is empty and available, and both mark it as used. Then block 100 is returned to whatever called allocate_block() for BOTH CPUs and we have a problem.

We don't have atomic block bitmap setting or clearing.

See include/asm-i386/bitops.h

This has a bunch of useful bitmap functions:

set_bit(), find_first_zero_bit() [convenient!]

These are faster than our C-level instructions (because we have raw assembly now), but slower than what it could be if it did not have to be atomic.

Lab 3 assumes single CPU, so we don't need to worry about synchronization.

Inode

In OSPFS, we actually have two types of inodes in this file system.

The first is about file and directory.

oi_ftype tells use file type: regular, directory, or symbolic

Diyu says this was a perfect time to use enum rather than constants (more type safety then).

oi_nlink counts the number of links (0 means free).

The block pointers are where we point to the actual data.

ospfs_inode(ino_t ino) returns a pointer to an inode struct.

If it's an invalid inode number, return 0 (NULL)

Otherwise, we return the actual inode by accessing the inodes array at the right spot.

Accessing Data from Inode

The problem: We don't know the actual size of a file when we create it. We want to efficiently organize files and their data. We don't yet have a very nice implementation for this, and this is an area of active research. We have to track data, but we don't know how big to make our tracking data structures. Ahhhhh, this is where dynamic allocation comes in.

Most files are small. In our system, if a file takes up fewer than OSPFS_NDIRECT blocks, we'll store them directly in some set of blocks, the array:

```
oi_direct[NDIRECT];
```

Each element of this array is a pointer to a block of data.

If the file is larger than this first limit, then we have an indirect block that points to a block storing addresses. The block is 1024 bytes in size, and each 32-bit pointer takes up 4 bytes => each indirect block can point to another 256 indirect blocks. f So far, we can have files take up 10 + 256 blocks. If a file needs more space than even this, we add another level of direction: a double indirect block that has pointers to indirect blocks. An array of pointers to arrays of pointers.

Then, we have 10 + (double indirect * indirect) => 10 + (256 * 256) = 65,536 + 10 possible blocks

Looking at the ospfs_inode_blockno() function, we can see these arrays in action.

OSPFS_NINDIRECT is the max number of blocks supported by using just a single indirect block. If we need more than OSPFS_NDIRECT + OSPFS_NINDIRECT, we'll use indirect2 (the double indirection).

Look at the struct:

```
oi_direct[OSPFS_NDIRECT]
oi_indirect; // Just an int pointer to indirect block
oi_indrect2; // int pointer to double indirect block
```

In double indirect case, are indirect and double indirect block used, or just double indirect? Are all three members used?

All three blocks are used. The order of checks runs from: all 3 needed; all 2 needed; all 1 needed. It's an if-ladder. The data is split across them as needed.

Data for Directory

This is the second type of inode. (So that inode type is not too general). An inode for a directory contains an array that identifies the files under that directory. That data can include another directory data struct!

This is how we get recursive directories.

If we have file type directory, then the directory inode struct is also in the direct and indirect blocks.

Hardlink and Symbolic Link

A hardlink is a link to a specific inode (which identifies data on disk). We can have multiple files associated with the same inode. The oi_nlink member tracks the number of hardlinks to some inode. We free it when this count drops to zero.

A symbolic link is a string, a path to some file. An open on a symlink redirects the open to the file to the inode (which is the struct of data direct and indirect blocks we iust saw).

The inode for a symbolic link is new and has a path to some file.

The reason we still have symbolic links is because what if we want different permissions? Then, with different inodes, we can have different permissions set (since those are stored in the inode).

Also, what we have a path to some place on disk that doesn't use inodes, like /mnt/flashdisk, which is DOS-like? Then, we need a symbolic link.

Symbolic link struct is just a path. We still havea liknk count to know when we can free the inode used for the symlink. Also, we can hard link to a symbolic link. Strange, but possible.

We can have file systems coexist on the same machine. The user-level code (at syscall level) works magically, thanks to the kernel's sys_read().

Handling this system call, the kernel goes to the file descriptor that contains function pointers to functions for basic tasks like reading.

So we'll call:

```
file->f_op->read();
```

Where, f_op means "file operation"

1,000+ lines of code, just of skeleton code alone.

ospfsmod.c holds the structs like file_operations opspfs_reg_file_ops, which has pointers to functions. We'll implement some of those functions.

The system calls will expect these function pointers to be pointing to the right place, and it just follows them.

In Linux VFS, we have different object types, including the struct file and struct dentry, plus different inodes.

Under Linux, the syscalls will redirect to our functions. We can translate from struct inode to struct ospfsinode. They share the same inode number, meaning that the Linux inode and the OSPFS inode both point to the same physical data on disk. We just get the inode number from the Linux inode: i_ino

struct file points to current position in file.
This is how read() resumes reading at a particular point. struct file is kept separate from struct dentry so that we can keep track of file position.

Where's the interaction happening here between the Linux FS and our OSPFS?

f_dentry and d_inode are members.

struct file { struct dentry { struct inode } }

We're gathering information from the Linux structs to go into our file system at the right place.

mk_linux_inode() is used for the relationship between the Linux FS and ours. Note that there are two inode classes at play: the Linux one, and ours (which has different inode structs for files vs. directories).

Lecture #11: 18 February 2015

Disk Scheduling

Here's a simple model for disk I/O:

```
[              h              i       ]
0                                     n - 1
```

n blocks, indexed from 0 to n - 1. Head position at h, block to be read at index i.

The cost to seek randomly will be proportional to the distance between h and i => |h - i|

We'll do a continuous distribution so the points run from 0 to 1:

```
[       ]
0       1
```

If the head is at either end, our average distance is 1/2, and if the head is in the midle, the average is 1/4 (as the average is 1/2 on either side of the halves).

IF the head is at the 1/4 position, we have:

```
(1/4)(1/8) + (3/4)(3/8) = 5/16
```

And we get a curve; we want an average, so compute the area to get an average value of a function:

```
-        -
 -     -
   - -
```

"How do you do this? Calculus! That subject you fled from when entering computer science."

We'll end with a double integral (see scribe notes) that evaluates to 1/3.

Shortest Seek Time First

Now, say we have a set of read requests:

```
{20, 97, 512, 10923}
```

Where the numbers are the numbers of disk blocks, and assume the head is at 96.

Then, we'd go to block 97 first.

This is called **Shortest Seek Time First**, (SSTF)

- Best throughput
- Possible starvation

Niceness is (seek time). Not as nice if short seek time.

Many read requests near the current head may starve a read request for a block far away.

First Come First Serve

On the other hand, **First Come First Serve**: (FCFS)

- Fair (no starvation)
- Less throughput

Niceness is (arrival time).

Note that both of these are priority scheduling.

SSTF + FCFS

We can do a hybrid approach: just add up the seconds.

This gives us a new niceness value that is not quite as fair (though still no starvation), and that beats FCFS on throughput.

Elevator Algorithm

Suppose we don't know the arrival time, and all we have is the seek time and the block location, but we still want fairness.

SSTF but only in one direction. Think of read requests as people on different floors, and we service those that are along the way in our current direction. At the end, we reverse direction and go the other way.

-------> <------- -------> <-------

We still have a fairness problem. Those read requests at extreme ends wait longer than those in the middle (which may be picked up in either direction)

In a physical elevator, people are going in a direction, but for a disk, all it matters is that read requests get processed.

Circular Elevator

This modification has the the elevator always reading in one direction, and then wrapping around (the previous backtracks, not wraps around). This is fairer, and often preferred for disks.

-------> / / / / Overhead to reset read head / / / ------->

Anticipatory Scheduling

Suppose two programs A and B reading sequentially from different spots on disk:

A: 1, 2, 3, 4, 5 B: 1001, 1002, 1003, 1004

Suppose 1 comes in, then 1001, then 2, then 1002.

Notice the alternation between the two programs, and now we're jumping back and forth between two different spots, jacking up the latency.

To solve this, we can *dally*, waiting around for read requests after the first, to see if more requests for around the same block arrive. In the real world, this speeds up sequential reads. After some time, we switch to the other set of read requests.

The flow:

1. Read request received.
2. Purposely delay and hope/guess that a request for a nearby block will arrive soon.
3. More requests received, and at some point we service them.

We're relying on patterns like the fact that programs often read sequentially. There are even advisory system calls to let an OS and its schedulers know what a program plans to do.

Concurrent I/O

If we have multiple programs, each with different I/O needs, they may be satisfiable in parallel, and they may be able to be grouped into batches.

File System Design

A file system is a hybrid data structure that lives partly in RAM and partly on disk/flash.

FS design is data structures design.

We want the capacity of and permanence of secondary storage, and the speed of RAM.

Big, permanent, and fast.

Paul Eggert's 1974 undergrad research project:

Eggert once had a crazy professor during undergrad, who had heard about UNIX, developed the year before at Bell Labs. He wanted something similar. It was a paid research project to basically duplicate UNIX on the Lockheed SOE.

Write a text editor, and various other things such that we get a software development environment, including a file system, i.e., where to save the files.

All written in assembly, and there was no compiler. He had never heard the word "file system". In fact, he was later supposed to write a compiler too. The whole disk was about 1 MB in size.

He wrote something dead simple:

Files start at the beginning of sectors (512 bytes long), and are contiguous in content. If a file spills over one sector, another sector is allocated, and as much as needed is used, and the rest of the second sector is wated.

[directory | file | wasted space | file] 1024 1234 1536

To track where the files are, the first two sectors are reserved for tracking where the files are. Each one had three things:

- File name, 12 bytes limit
- Start sector, a 16-bit number
- Size of file (in bytes), a 16-bit number

He couldn't imagine any file larger than 65KB.

This can be done in a couple of days in assembly apparently, regardless of how bizarre the computer is. Later, it turns out the RT-11 file system was very similar and done in 1972, where RT stands for "real-time".

Advantages:

- Simple
- Good, predictable performance for sequential I/O to one file (The seek arm doesn't move much.)

Disadvantages:

- Internal fragmentation

We waste up to 511 bytes/file. 1 byte files waste almost a whole sector. This is space assigned to a file, but not used.

- External fragmentation

We have enough free space on the disk space, but we can't place it on the disk, because the free space is fragmented across the disk, and there's no contiguous set of free space large enough.

- Directory imposes limit on number of files

We're limited on how many files we can have based on number of entries in directory.

- Growing a file is difficult

We can only grow a file if the free space after the end of the current file is enough. Eggert required that apps specify ahead of time how much space would be required for their output. This is no problem for a text editor, as everything was edited in memory, and the size can be determined before write to disk.

- No user-created directories or subdirectories

We have only one big, flat directory.

// Eggert is lucky to have grown up while the whole // abstraction layers for modern computers was being // built. He's had a chance to learn it from bottom up.

"While I was sitting in a university computer lab and trying to reinvent UNIX, the personal computer revolution was happening. My competition was creating operating systems with funny names like CP/M and DOS."

FAT File System

This is from the late 1970s, short for "file allocation table."

The purest version of FAT models the disk as a big array of blocks. Rather than 512 bytes per block though, FAT chooses 4 KiB per block, which is 8 * 512-byte sectors.

Here, data does not have to be contiguous; additional data can just be allocated somewhere else. The file is being treated as a linked-list of blocks (CS 161 slides make sense!). We could have a `next` field in each block, but this uses up 4 bytes per block, which gives us only 4092 bytes per block, screwing up our clean base 2 allocations, which straddles I/O boundaries.

Instead, these `next` fields are stored in the FAT, the array of `next` fields, an array of 16-bit numbers (in FAT 16) stored in the blocks preceding the data blocks.

[| Superblock | FAT | Data]

The superblock will record important things like the block number for the root directory, and that gives us a starting point for getting to a file. Also recorded: file system version and size.

A file will have a 16-bit number representing something like 29, and so we go to 29, which has our data. We check the FAT again, which tells us the next data block for block 29 is found at block 36. We go there, gather more data, and then check the FAT again, which has a zero, marking EOF. To grow a file, we change the zero to point to a new block that was available and now allocated.

To know where to find that first block, we have the concept of a directory which is a file with contents:

```
[   name    | extension | block # of 1st block | size of file | type of file ]
[   name    | extension | block # of 1st block | size of file | type of file ]
[   name    | extension | block # of 1st block | size of file | type of file ]

[   name    | extension | block # of 1st block | size of file | type of file ]
    8 bytes    3 bytes          2 bytes                            1 byte
```

These are directory entries, and the file type is either a regular file or a directory. A directory maps file names to block numbers, and it too lives somewhere on disk. All zeros means unused.

We haven't solved internal fragmentation either.

Notice how many capacity we can have: 2^16 * 2^12 => 2^28 bytes => 512 megabytes, unimaginably large at the time.

2^12 == 4 KiB 2^16 == number of blocks trackable via 16-bit block numbers

FAT 32 expands via 32-bit block numbers. FAT 32 is what's used on my flash drive.

To figure out where the free space is (without loading whole FAT into RAM), there are -1 values in the FAT for free blocks. We just find the next -1, and mark it as a 0, and update the previous FAT entry for the file's previous EOF to point to this one.

Problems: Sequential I/O can be seeky: files may be allocated all over disk, fragmented and strewn about, and so reading is seeky. To solve this, we run a defragmenter that reads the *whole* filesystem and moves things about until they're contigous as possible.

Not only that, we risk corruption when we lose power, especially while defragmenting.

Now what about renaming? foo.c -> bar.c is easy and fast. At most, we lose the new name if we lose power. But a/foo.c -> b/foo.c is tricky.

If we zero out a/foo.c first and lose power before writing into b/foo.c, we lose the file altogether.

Else if we write out a/foo.c to to b/foo.c first, and then lose power, we'll end up with a duplicate. Later, if we free one instance, the other instance's block number will be the number for freed space. Later accesses can cause all sorts of crashes.

The original version of FAT just plain prohibitedym moving files from one directory to another because of these issues.

UNIX File System

Circa 1977, the biggest new idea is the **inode** (index node), which is a small object of fixed size that stores the metadata for a file. It's cached in RAM while a file is in use, and stored on disk otherwise.

We'll have 1 inode per file:

```
[       inode table |       data blocks ]
                        8 KiB per data block
```

Data in inode:

- File size in bytes
- File type
- Date
- Permissions
- Number of hardlinks
- 10 direct blocks with block numbers in each
- 1 indirect block with a pointer to a data block with block numbers
- 1 double indirect block with pointers to indirect blocks with pointers to data blocks with block numbers

Doubly-indirect: 2^13 * 2^11 * 2^11 Indirect: 2^13 * 2^11 Direct: 10 * 2^11

Sum of the three gets us to 2^35 bytes. We could have larger blocks for the indirect blocks, or have a third level of indirection.

Does this handle fragmentation better than FAT? Not really, as the inode blocks may point us to data blocks all over disk. However, the Unix file system wins out for lseek().

In FAT, getting to the latest byte in a file requires us to traverse a linked list: O(n). Under the Unix file system, lseek() is O(1), and so random access is faster.

To get to the file, we also have a superblock recording the root directory, and use that to get us to the file.

The inode tells us about a file, but it doesn't store the name. The assumption is that directory entries will store them instead:

```
   14 bytes     2 bytes
[ name      | inode number ]
[ name      | inode number ]
[ name      | inode number ]
[ name      | inode number ]
[ name      | inode number ]
```

Directories map names to inode numbers, and multiple names may map to the same inode number. As a graph:

```
        [   ]
       /     \
     [    ]
      | |
       ()
```

Why have this structure? Simply create a directory entry in another directory, (with the updated name), increment the hard link count, then delete the old entry, and decrement the hard link count.

For `git clone` on the local files, it's blazing fast, and just makes more links in the local .git folder to the cloned files.

We can do this via the `link()` system call. The original UNIX didn't have the `rename()` system call, assuming instead the use of `link("a", "b")` and then `unlink("a")`.

Things that can go wrong:

- link count was 1-byte (255), and nowadays at least around 16 bits, and so overflow was possible. To prevent this, link() simply fails if we're at max link count
- When we remove a file, we have to zero out a directory entry into a directory block, and then decrement the link count. Power can fail between these two operations and mess things up.
- What about loops or cycles within our file system? /clone/foo/clone/foo/clone/... This is simply prohibited. No cycles may be created. No hard links to directories are allowed, only hard links to files. This prevents cycles.

Suppose foo has a link count of 1:

```
(rm foo; cat) < foo
```

The flow:

1. We remove foo, which now has link count 0. The file's contents may now be freed and reused.
2. cat begins reading from freed space.

The answer: We only reclaim storage when:

1. The link count is 0

AND

1. Nobody has the file open

Here, the file is deleted, but we haven't yet reclaimed the storage, as the OS sees that cat has the file open. This is the OS' responsibility, not the file system's.

Berkeley Fast File System (FFS)

This one has a bitmap of free blocks right before the inode table, which can be fit into RAM. Thus, we can easily allocate new chunks contiguously while growing, as we have a view of all the free blocks at all times. This doesn't solve fragmentation fundamentally, but helps with it.

Discussion #6: 20 February 2015

// MacBook Air currently out-of-commission, // cause unknown. This is quite distressing.

// Currently on loaned CLICC laptop until it's fixed, // hopefully soon.

// At least there's always Vim... // This laptop has Xcode installed too, and Macs // in general have useful UNIX utilities like ssh, // thank goodness. I can work on SEASnet as needed.

// It's running Mountain Lion though. Mavericks // at least would help improve performance and // the (atrocious) battery life.

WeensyOS 2: Scheduling and synchronization

Scheduling

cursorpos is memory for console contents. PRINTCHAR is a number and color

Processes 2, 3 just do the same with different color and number.

sys_yield() yields control of the processor and lets a different process be scheduled to run. One of the things we'll do is improve the scheduler.

Todo:

- Implement a strict priority (replace sys_yield() with sys_exit()) AT1, AT2, AT3, ... Change the part in the "Yield forever" loop with sys_exit(), so that the process will be marked a zombie.

Choose 1 of these:

- Implement a system call for proceses to decide its priority and schedule the processe based on the priority it specified (again also replace sys_yield() with sys_exit())
- Implement a system call for processes to decide its share and let the process to run the amount of CPU time based on its share. Measuring the CPU time will be tricky.

Synchronization

interrupt_controller_init() is in x86.c, and it takes a boolean that lets us decide whether or not to accept hardware interrupts If we allow the clock interrupt, we will initialize the clock.

segments_init() is what sets up the handlers for different interrupt. See line saying "clock interrupt gets special handling". The actual handler is implemented in assembly, and sets up registers before calling the interrupt() function in kernel.c. The registers hold the code letting us know which interrupt we got.

Note there's a missing 4 in the first line, from around the 3rd/4th group. If we get an interrupt before the print, then we get another process scheduled before the value is printed, and so we lose the 4.

If we increase the frequency of the timer interrupt (via HZ value), then we get fewer values, and also fewer 4's. We're supposed to figure this out.

To implement a system call for the CPU time-sharing, we use the timer interrupt as a basic unit. We'll have a slice in the

process descriptor. With each timer interrupt, we decrement the slice count, and when it hits 0, we schedule another process. This slice count will be set based on processes' decided share. This is apparently the basic mechanic used in modern OS'. All the system calls are implemented in assembly. I hope we don't have to write them in assembly...

asm(assembly code : output : input : modification)

We have no output part here, just an input.

asm volatile("int %0\n" : : "i" (INT_SYS_EXIT), "a" (status) : "cc", "memory");

So if INT_SYS_EXIT == 48, then we have

```
INT 48
```

in the assembly code. 'i' means a constant number, and it and 'a' are the first and second arguments that we're placing into the formatted string for the assembly. If it were non-constant, it'd be in a register, so "a", "b", etc.

"a" means %eax register, so we'll put 'status' into %eax before executing the assembly code.

Midterm 1 Post-Mortem

Problem 1

Part (a):

"I have no idea what this code is doing."

Just change all '512' to '511' and change the buf from '513' to '512' and another buff access from '512' to '511'.

Part (b):

For this to work on both machines, we need to test if it's on a normal machine or a cheapskate machine. We write 512 bytes to a random sector, where this data's last byte is not 0. Read the disk sector to see if last byte became 0. If normal machine: 512 bytes for everything; else, 511 bytes for everything.

Problem 2

Create two pipes:

```
pipe(p1), pipe(p2);
```

p1: A->B p2: B->A

Parent process can be pipe A and then it forks to generate pipe B. For process A:

```
while (1)
{
    write(p1[0], "A", 1);
    read(p2[1], &buf, 1);
}
```

For process B:

```
while (1)
{
    write(p2[0], "B", 1);
```

```
    read(p2[0], &buf, 1);
}
```

Redirection with dup2() should also work?

Problem 3

Part (a):

"It's basically nonsense; I think it has nothing to do with operating systems."

'big': A very very very big file filled with NULL

```
grep x big
```

Very fast; no output. Uses lseek() to skip over null parts? I think via offset.

Part (b):

```
ls -l big | grep x
```

We'll see the listing, and then we'll see 'big', all very quickly. This is just printing out information from the associated inode.

Part (c):

```
cat big | grep x
```

This executes for a very very long time, and output nothing. Diyu actually tried this for 2 hours.

Prt (d):

```
cat big | ls -l big
```

This also just displays the listing. ls closes the pipe, and sends SIGPIPE to cat.

Part (e):

```
./big | ./big
```

Error message: cannot execute binary file Error message: cannot execute binary file

Part (f):

```
if if :; then:; else wc; fi
then date
else cat
fi < big
```

We end up seeing the date. Unwrap the nested 'if' to see this

Part (g):

```
:->>./big
```

bash: command not found

Part (h):

```
while cat > big; do rm big; done < big
```

Infinite loop and if we interrupt it, big is changed to an empty file. rm will just call unlink() and not actually remove the file, as there are still processes that have the file open. Problem 3 is graded by Eggert.

Problem 4

Part (a):

Can a system call using INT and RTI receive floating-point values?

Yes, it can. We just need to define calling conventions. See slides. for sys_call_double(double arg), we'll read the the upper bits from ecx, the lower bits in edx. The handler will get from register a and b to construct the floating point and then split the result between ecx and edx. To return the double, we do some bit ops:

```
return (double) (ret_high << 32 | ret_low));
```

We're just storing results to registers, and retrieving them from registers. We just need conventions on what to place when and where.

Part (b):

Can the same be used for SYSENTER and SYSEXIT? Yes, they don't actually interact with passing/returning values to/from syscall. They just mess with stack pointers, program pointers, and program segment. So if we can modify INT, then there's no problem with us using SYSENTER and SYSEXIT with this modified INT.

Part (c):

Implementing drand48() as a system call instead of a function call.

Pros:

- More random: Kernel can gether more entrop information compared to user: keyboard timing, mouse movement, hardware interrupt timing
- Safer: Harder for attacker to hjack this function. Not even root can modify kernel code.

Cons:

- It's slower

If answer was "No", grading will be based on alternative solutions.

- Make drand48() return an integer instead. Simple, correct => full credit

Problem 5

Part (a):

Suppose we use malloc() inside of the signal handler. It cannot be re-entered, and it is not atomic. This is a synchronization issue. The finding of a valid address and updating it as allocated is what's happening inside malloc().

Part (b):

int l = 0; locked_malloc(free) { while (l): l = 1; malloc(free); l = 0; }

Part (c):

Why won't this work? Because it will cause a deadlock. The signal handler also calls locked_malloc(), AFTER it already acquired a lock. Because we interrupted the thread that acquired the lock, it can't release it, as the signal heandler is stuck in the while (l) loop.

part (d):

Problem The signal handler takes over the current thread. The current thread is asleep and cannot release the lock. If it's multithreading, one thread will eventually release it.

Lecture #12: 23 February 2015

Inodes and Indirect Blocks

What is our worst case for internal fragmentation?

A few cases:

File size = 0: Null pointers or null indices to data blocks, but 48 bytes wasted for the inode.

File size = 1: A single direct block to a data block that has 1 byte used up, and the remaining 8191 bytes of the 8 KiB block are wasted, along the remaining 44 bytes of the inode.

Remember that 'big' file example from Lecture 1?

Here's how we can generate it:

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC, 0666);
if (lseek(fd, 1000000000, SEEK_SET)) {
    error();
}
else {
    write(fd, "Hello", 5);
}
```

So what happened with our inode for this big file? We created indirect and doubly indirect blocks, as moved our internal file pointer far forward, and then wrote 5 more bytes there.

This is called a **holey file**. Most of the indirect blocks are not actually used, except for that one data block pointers for the 5 bytes for "hello". The unused direct and indirect blocks act as zeros when read.

So how many bytes are wasted?

- The 44 bytes for the inode
- The 8188 bytes for the indirect block
- The 8188 bytes for the doubly-indirect block
- The 8187 bytes for the data block for "Hello" (1 trillion modulo 8192 to make sure)

Consider the `-s` option to `ls`. It means "size".

For 'big', it'll show 5 bytes, not 1 trillion + 5. We're taking advantage of the way the file system is implemented in order to efficiently determine file size.

Example:

We map a relational database into a 10^12-byte file. A hash function maps data into indices on the file. The file begins as all zeros, and so it's quite holey, but we bypass the overhead of truly writing zeros to disk. They're just placeholders. Over time, we write in actual entries and data to disk.

Cons:

- Space can run out

But, the underlying system can exhaust disk space before our database does. We said it's a 10^12-byte file, but maybe the disk actually ran out of space before we wrote in all the data.

- Slower

To get to our ith data block, we need to access up to 3 other blocks (in the inode) to find out where it is.

Wedding Cake Abstraction of File Systems

- Sectors

Fairly small; traditionally 512 bytes

- Blocks Small number of sectors; typically a power of 2

Why so many layers of file systems? To limit the damage caused by filling up a filesystem.

- Partitions /, /boot /usr, swap, /home, etc.

'swap' is not a file system; it's used for virtual memory)

/ is for misc. /boot is for the kernel /usr is for programs

Suppose we fill up /usr. We still want the kernel to work!

- Filesystems and Mounts

Each partition has a file system. In the kernel, we have a **mount table** that maps files (indexed by inode number) to file systems. See scribe notes for the overlapping trees diagram.

An alternative strategy is to expose the file system to the user:

```
C:\etc\passwd
```

Problems with this mount table setup. It's just pointers, so it could be

Bad mount table:

```
/etc         filesys 3
/etc/foo     filesys 4
/etc/foo/bar filesys 1
```

Which makes a loop back to the top-level file tree. How to prevent this? Well, the `mount` system call looks like this:

```
mount("/etc/foo", "/dev/rst/03", filesystem_type);
```

This fails if "/dev/rst/03" is already mounted, except for loopback mounts. These let us mount part of a current file system somewhere else. These must worry about loops, and some distros don't bother trying to protect us from loops.

We can support multiple file system types.

```
/usr ext4
/home vfat
```

We can hook in an arbitrary file sytem to a Linux system. OOP programming lets us do this. We specify an API for a file system, and treat a filesystem as an object to be implemented.

mount() needs to know the file system type, as it handles the actual gluing of the implementation into the kernel.

The VFS (virtual file system) is an OO implementation in C. It's a sort of base class for basic filesystems operations. So long as we implement a new filesytem that obeys this interface, we can load it as a kernel module.

An interesting filesystem is NFS:

Whent the read() call arives, we send off a request via the network to get the data from the server actually storing it.

In fact, VFS was originally invented for NFS circa 1985.

- Inodes
- File name components

file name resolution for something like:

```
open("abc/def/ghi");
```

uses a routine called `namei()` that translates strings to a (device, inode) pair. It works by looping over the file name. It maintains a current directory inode as it goes through.

It checks each directory inode, and finding the correct name, goes to that inode, and finds the next subdirectory via the next directory inode.

If a subdirectory is missing, we'll get an `ENOENT` error, leading to "no such file or directory".

If a subdirectory is not searchable due to our current search permissions (x bit needed), we'll get `EPERM` (no permissions error).

But where do we start in this loop over the name? We begin at the working directory, the directory implied for each process.

The name may also begin with root directory, which is also per-process. To change these:

chdir()

This changes the current process' working directory. We can change into and out of a directory.

chroot()

This changes our root directory. If we go lower, we lose the context of files at or greater than the higher-level directories.` It makes files go invisible. Going back is not possible.

How about chroot("..")? But we're at root, and kernel convention for parent of root is root: /.. === /

/.. === chroot("/") === noop

We should not use it unless we really need to restrict a program's view of the file sytem.

Suppose we want to gain root privileges to a machine:

```
$ su
Password: eggert
# // Root prompt
```

Well, under /etc, we passwd and shadow.

passwd has 'root' shadow has root's password. It's encrypted, but suppose we create a directory in oru directory:

```
/home/eggert/playpen/etc
/home/eggert/playpen/etc/passwd
/home/eggert/playpen/etc/shadow
// Plus the other files under /etc; just hard link
```

Now, if we do a chroot /home/eggert/playpen before we do so, every process, including children processes, only sees our playpen files, including our created passwords for root.

So what is chroot() really used for?

chrooted jail

Suppose we're about to run a potentially dangerous program, like apache. It's a web server that's regularly exploited.

We place apache and its files inside of chrooted jail, and it thinks it's inside root directory. Damage is confined to this jail. Attackers are skilled at escaping these jails though.

Symbolic Links

symbolic links file names (have slashes) file name components (no slashes) inodes

Symbolic links have a permission bits set that looks like this:

```
ls -l a/b
lrwxrwxrwx  eggert eggert 23 Jun 2015 b -> foo/bar
```

To make this link, we would have done:

```
ln -s foo/bar a/b
```

namei() subsitutes:

```
a/b/c
```

Becomes:

```
a/foo/bar/c
```

It's expanding names and resolving them. Why do we need symbolic links at all? Hard links came first. What can do we do with symbolic links that isn't possible with hard links?

- Symlinks can cross file system boundaries

Suppose we want /bin/sh to be the same as /usr/bin/sh, but these reside in different file systems. No problem: symlink.

```
ln -s /bin/sh /usr/bin/sh
```

If the name begins with a slash, we throw away the first part\ when expanding, as we're specifying with respect to the root directory.

- Symlinks can link to a directory

  ln -s /bin /usr/bin

In effect, we get all the files linked for free.

- Symlinks can link to nothing, to a nonexistent file

It can dangle, for a file we plan on creating later, but we create the link now.

- Can loop/cycle

(graph theory uses 'loop' and 'cycle' differently)

Example:

```
ln -s loop loop
```

Or:

```
ln -s a -> b
ln -s b -> a
```

So how can we handle looping? Is checking for loops going to cause us to encounter the Halting Problem? No, because this is just data, not a program. Adding conditional symlinks would introduce the Halting Problem. However, resolving continuously means we're only ultimately bounded by teh number of inodes on our system, which is huge. We'd iterate through with various algos to checks.

So, the convention is to have namei() (and thus open()) fail after 20 symlink expansions, and produce the ELOOP error.

Symlink Implmenetation

Do sym links have inodes?

Suppose yes. Suppose each inode has a field that identifies its type: regular file, symlink, directory, etc.

A symlink's data would be its symlink contents, placed in a data block, like "foo/bar".

Then, we could do hard links to symbolic links:

```
ln -s a b
ln b c
ls -l
```

Examining:

```
b -> a
c -> a
```

With a link count of 2 for a.

Suppose we did:

```
mv c /same/other/dir
```

Suppose another yes case: we hold the symlink string directly in the inode rather than in a data block.

This can be seen as an optimization of the first yes case to minimize disk usage for short link names. To see if this happening, we can run `ls -ls` and see if size = 0, length of 7 bytes. (name length is 7 bytes; disk space measures data block usage, not inode disk space usage).

Suppose no. We look at our directories instead. Special directory entries link a linkname to a string, presumably with a entry type field. Then, symlinks are notations in directories, not inodes. So, hard links are not allowed to them then.

Both of these are used, but the 'yes' case is more common at the moment.

What does it mean to have read/write/execute permissions on a symlink?

r: See what symlink expands to w: Able to change what symlink expands to

But in practice, the 'rw' bits are ignored for symlinks, even if they're all minuses. In reality, we can read a symlink, but we cannot write it.

```
b -> foo/bar
```

If we do:

```
open("b")
```

How does this work? How does ls work and know that b is a symlink? Answer: a few new syscalls.

lstat()

It's like stat(). We give it a filename and a pointer to a struct stat that fills that struct with information about the file. What's different is that lstat() does NOT follow symlinks:

```
lstat("foo", &st);
```

If 'foo' is a symlink, st will be filled up with information about the symlink.

readlink()

Similarly, we can read the contents of a symlink:

```
readlink("b", buf, bufsize);
```

unlink()

unlink() will not follow symlinks.

Next, what about "eggert eggert" is ls' output? It's the owner and the group. They normally tell us who's who (for the user/group bits). But these are still needed for disk space quotas; symlinks take up disk space too.

Special Files

What about:

```
ls -l /dev/null
```

It's a special file. There are two special file types:

```
c
b
```

Instead of a size, it has two numbers: a device driver and device.

```
crw-rw-rw  1, 3

Driver #1, device #3
```

It's actually very easy to write a device driver for /dev/null: throw everything away when writing; return 0 on read.

Another special file:

```
/dev/full
```

It gives a write error of "full" when we try to read from it.

Yet another:

```
/dev/zero
```

Gives as many zeros as we'd like when read from it.

mknod()

How do we make these?

```
mknod /dev/null c 1 3
```

'c' means 'character special file', device driver #1,

```
ln /dev/null haha
```

Read from it: "Haha, it went away."

mkfifo

This acts like a pipe. In fact, 'fifo' means "named pipe".

```
mkfifo foo
```

With special type 'p':

```
prw-rw-rw ...   foo
```

It sticks around, and all our pipe knowledge applies.

```
cat foo
echo blah > foo # in a background process
```

Produces:

```
blah
```

Dealing with Low-Level Failures

Suppose we have a bad sector, and we have data in that sector. Then, alarmingly, our symlink points to it.

Linux's approach: If at any time we encounter a low-level failure while trying to resolve a symlink, we get the EIO error for errno, and fail.

What if we want to destroy useful data? What if we want to prevent data from being read by other people.

Will this work?

```
chmod 0 file
```

Not good enough. We can simply mount the filesystem and do chmod 444.

How about:

```
rm file
```

Assuming there was only one hardlink and no processes had it open. open(), read(), write(), etc. no longer see it, but the data is still physically on the disk. We removed a directory entry; we didn't do the actual sector wiping.

shred

A better solution: the `shred` command. It does a pass of zeros, then a few passes of random data, and then it unlinks the file.

```
shred file
```

Even if the attacker can see the raw sectors, the data is illegible. Why 3 times? Because the erasure by the magnetice read/write heads isn't perfect. With good-enough tech, we can recover data. If we're truly paranoid, we can shred it, but even those bits can hold quite a bit of data. We can use DNA sequencing techniques to build up the original files from the bits. Melting then needs to follow.

But this still isn't good enough. shred doesn't affect the indirect blocks. We can still find out the pattern of holes in the file, the size of it, etc. In addition, some file systems write a new block of data right where the disk head is, and mark the old one free.

So how do we fix that? We have to overwrite the space marked as "free".

```
shred   /dev/dsk/03
```

This is is why shredding free space is needed.

Also, all of this is slow. Ridiculously slow.

How can we do this quickly? Encrypt then throw away the key.

Lecture #13: 25 February 2015

Recall the BSD Fast File System (FFS):

```
[ boot block | superblock | bitmap for free blocks | inode table | data blocks|
```

Suppose we want to test if our file system is working well, how might we test it? One way to do it is with invariants.

Invariants for File Systems

Invariants for file systems are boolean expressions that should always true. Non-interesting example: 0 < 1

We want to characterize what makes an implementation valid. Examples (many others possible):

1. Every block is used for exactly 1 purpose. Weaker version: <= 1 purpose (some blocks may be unused or wasted)

No block should double-up, because otherwise we lose information.

1. All referenced blocks are initialized to data that's appropriate for that block.

For example, an inode referenced by an actual directory should not have a negative size value in thtat field. NO random data. What about the data for a regular file? That could be anything. For a directory block, the data should follow the directory entry format and be decidedly non-random.

1. All referenced data blocks are marked as used in the bitmap.

Otherwise, we may overwrite a block already containing data in the mistaken belief that it was available.

1. All unreferenced data blocks are marked as free in the bitmap.

Otherwise, we may be "out-of-disk-space" when we really aren't.

Now that we have all these rules, what about what happens when the rules are broken?

Consequences of Violating Invariants

1. If we violate (1), then everything will be hopelessly mangled; soon, nothing will work. This is a total disaster: everything might crash, silently lose data, etc.
2. If we violate (2), we get similar disastrous results. Pointers in the superblock will go to invalid areas, etc.
3. If we violate (3), then a data block used for a regular file may be overwritten as part of growing/writing another file. We just lost data! Even worse: overwriting a directory block.
4. If we violate (4), then a lot of unused blocks will be seen as use. We'll end up with a lot of permanently unused blocks; we waste disk space. It's a sort of block leak. Our efficiency is less, but no data is loss, no crashes occur, etc. Violating this is not as big a deal.

When we make specifications, we should always think about the consequences of violating them. If we're in a rush, we'll prioritize (1) through (3).

Performance Issues

One way to see a file system is as a data structure living partly in RAM, and partly on disk/flash. The RAM can be seen as a cache for what is on disk/flash.

Suppose we have some processes:

P1: write(fd, "hello", 5)

P2: read(fd1, buf, 5)

To make this read() really fast, we can have the write() write to cache (RAM) first instead. Then, the read() bypasses the latency of the disk/flash.

Other performance-improvement strategies:

- Batching (hollect a set of data and then write to disk after some specified metric)
- Dallying (collect a set of data in cache and then write to disk after a set time)

Problem: RAM is volatile. We lose the data if the power is lost. Data can be lost if the data was held in cache, but not yet written out to disk/flash. This is a result of these performance accelerations (which are pretty common).

// Eggert trying to unplug the computer at the front of the room, // but it's locked away

But how do we know that we lost a write()? Tree falling in a forest sort of question. Answer: There's a later part of the program that depends on the write() succeeding (to disk). Later cascading failures will make it obvious to us that something went wrong.

Suppose the write() answered back success (because it wrote to RAM, but before it wrote to the disk). Then we expect its results, but if we try to use them, we have a problem.

Out-of-order Writes

Another example, where the user hasn't yet seen any signs of success:

P1: write(fd1, "hello", 5) write(fd2, "there", 5)

We're writing to separate sectors on disk for separate files. Suppose our scheduler uses a circular elevator algorithm that does the "there" write first, and then the "hello" write.

["Hello" "there"] ^ | disk head

fsync()

One solution:

- Complicate the API

We have the POSIX syscall:

```
int fsync(int fd);
```

This guarantees that that file has been written to disk from cache. This is *specific* to this one file.

This too can fail: I/O failure where the disk has bad sectors, was disconnected, etc. Another problem exists. Suppose:

```
if (fsync(fd) != 0)
    error();
if (write(fd, "hello", 5) != 5)
    error();
if (fsync(fd) != 0)
```

```
    error();
```

But we cleverly avoid straddling block boundaries by placing a buffer on a boundary:

```
if (fsync(fd) != 0)
    error();
if (write(fd, buf, 8192) != 8192)
    error();
if (fsync(fd) != 0)
    error();
```

So we avoided writing to a second data block, but we don't get to avoid writing to an inode block: last-modified time, size, etc. The file system design is forcing us to do two writes: one write to data block; one write to inode table.

One way to cheat this: mount -o -noatime -nomtime

Don't bother updating the timestamps, so as to increase speed.

fdatasync()

Ths is just like fsync(), and arranges to update the data on disk, but doesn't bother scheduling the metadata to be written to disk, e.g. disk

```
int fdatasync(int);
```

We can get massive speed improvements by avoiding moving the disk arm to the inode table.

But disk controllers can lie to us. Sometimes, they wrote to their cache, but not to their actual platter of data. How could we test for this? We would need to actually power off the disk as part of our test program.

"Look you're running Windows: it's gonna crash anyway."

Testing is expensive for verifying disk controller reliability.

SEASnet uses Network Appliance boxes, which are very expensive, but very reliable. $1000/TB vs $50/TB.

What about

```
int close(fd);
```

? It does not synchronize cache with disk. This syscall does not guarantee that the file has been written to disk. Why? Performance. It's common for one process to close a file, and then another process opens it again soon afterward, right from cache. Any writing to disk from cache with close() is OS-dependent behavior.

sync()

```
void sync(void);
```

This takes all files anywhere in cache and schedules them to be written to disk. This is regardless of which process issued this syscall. sync() returns right away, but it does let the OS know to prioritize writing data from RAM to disk.

rename()

Example:

```
rename("a", "b");
```

What rename() does:

1. Get working directory's inode
2. Go to inode table, read it into RAM, and look up the data block for the directory
3. Read the directory's data into RAM
4. We change the "a" to "b" in the name field in the entry.
5. Eventually, the cache is written to disk.

This is all done in cache.

For this example, if we lose power between (1) and (2) or between (2) and (3), or even after (3), we don't lose data. We simply don't see the rename() or it succeeded all the way to disk.

But what if had:

```
rename("x/a", "x/b");
```

Here, one of the directory entries may have been written to disk, but not the other one. How do ensure that the directory was updated? Open it after renaming it and then use fdatasync(), which needs a file descriptor:

```
rename("x/a", "x/b");
dirfd = open("x", O_RDONLY);
fdatasync(dirfd);
```

Important (dangerous) assumption: Writes from cache to disk are atomic.

Another example:

```
rename("x/a", "y/b");
```

x and y are different directory blocks on disk. The order in which we do it has different consequences:

```
[ inode ]   [ x dir-entries ]   [ y dir-entries ]
link: 1
```

If we do y first, then lose power, the link count is inaccurate. We have a hard link count of 1, but two hard links to the same file. the minute one is unlinked, the data will be freed, and the other hard link will be dangling.

Solution order of steps:

1. Read inode
2. Read X's data
3. read Y's data
4. Write to inode that link-count = 2
5. Write Y's new data
6. Write X's new data
7. Write inode again (link-count = 1)

We are really afraid of power losses interrupting somewhere. Sadly, this doesn't preserve all of the invariants.

Crucially, there will be a temporary time when the link count will be 2, when the actual link count is 1. Consequence: we

waste some storage (because link count is still 1), but no crashses are causes. That's not too bad a tradeoff to make.

As an application to make sure that the rename() hit the disk:

```
rename(...);
dirfd = open(x);
fdatasync(dirfd);
dirfd = open(y);
fdatasync(dirfd);
```

fsck

To inspect a file system and check for issues:

```
fsck
```

It works at a low-level, reading the disk directly, checking for incorrect link counts, including an orphan file. The orphan file will be put into a directory called (by convention), /lost+found

Problem: Suppose fsck is part of the root filesystem, the kernel will mount the filesytem in read-only mode, and fsck will check for problems and let the kernel know. A second reboot will be required. Because fsck works around the file system, it can only be used on a filesystem that is not currently mounted (not in use and not being updated by the kernel), so it's common to use it after a reboot.

If root filesytem is possibly corrupt, and fsck is there, the kernel can arrange for the FS to be mounted in read-only mode, and then fsck runs over it, and lets the kernel of any fixes to be made. A second reboot will be required.

Summary:

Problem 1: Write success visible to user and expected Problem 2: Some side effects may not be saved to out-of-order ("ooo") writes

Interaction Between Scheduling and Robustness

Suppose we have a set of pending I/O requests: read(), write(), etc.

We have scheduling algorithms for ordering this I/O, but suppose our I/O requests need to be in a particular order:

```
(3) write 39216
(x) write 2196
(2) write 37215
(1) write 2196
```

(In parantheses: as actually scheduled by disk controller). Our problem is when I/O on metadata is out-of-order; that's what can mess up our filesystem.

We want something close to SSTF without losing correctness but retaining most of its superior performance. Some solutions:

1. It's okay to reorder data blocks (assume no fysnc(), fdatasync())
2. Remember dependencies in non-data blocks: this before that before that, etc., but the orders of these ones doesn't matter. The low-level system implementing I/O must respect these dependencies.

This is the compromise taken in BSD filesystems. All of this requires a lot more work on our programming end.

Robustness Terminology and Theory

Three main goals of a file system (for robustness):

1. Durability

The file system keeps working despite failures, in the underlying hardware, e.g., losing power.

1. Atomicity

When a user makes a change to a file, they either see a change made completely or not at all, even if power is lost.

```
All xor nothing.
```

1. Performance

No one is going to use the file system if it's too damned slow. Main metrics:

- Throughput (operations/second)
- Latency (How long for a request to finish?)

Atomicity

We're going to assume that the device underlying our atomic implementation does not itself have atomic writes. That is, if power is lost during a write, the data that was written is indeterminate: junk, mishmash, nothing, etc.

As a timeline:

```
"A" ---------"?"---------> "B"
     write("B");
```

We have a non-atomic write:

```
nawrite("B");
```

And we want to implement an atomic write

```
void awrite(char x);
```

using nawrite(). This is an atomic abstraction atop a messy underlying device.

// This is precisely the kind of CS challenge that enthralls me // and why I love CS

We might try:

```
void awrte(char x)
{
    // Lock-out all readers
    lock();
    write(x);
    unlock();
}
```

This will work while the power is connected, but not if it was lost suddenly. Instead, we implement:

Golden Rule of Atomicity

Never overwrite the last copy of your data.

We'll have two copies of the data:

I: "A" ---------"?"---------> "B"

II: "A" ---------"?"---------> "B"

And we'll do:

```
write(I, "B");
write(II, "B");
```

I : A ? B B B II : A A A ? B

We still have a problem: I and II will disagree if we lose power and come back to:

I : B II : A.

We always choose the first guy in this case.

Problem! If we reboot into:

I : ? II : A

Simplest solution: make a third copy:

time ->

I : A ? B B B B B II : A A A ? B B B III: A A A A A ? B

Our new deciding algorithm:

- If all they agree, excellent
- If one disagrees, majority rules
- If all disagree, we must be at:

  B ? A

So pick I, the first one.

From a theoretical sense, we have made atomicity from mud, something profoundly beautiful. However, we've tripled our I/O operations and nuked performance. In practice, we make some more assumptions of our disk hardware, thanks to the fact that they have **error correction**.

Each block has many error-correcting bits associated with it (not visible to user-level apps), and the disk will see these disagreeing checksums. There are many different sets of assumptions on physical drives. The most popular one is:

Lampson-Sturgis Assumptions

1. Storage writes may fail

The data may be written, but it is wrong, or the write may corrupt other blocks. Neighboring tracks are adjacent to the current track, the disk head may be slightly misaligned, and so hits the nearby track, etc.

1. A later read can detect the bad block

The checksums will catch bad blocks from previous hardware-caused I/O failures.

Read-Solomon acccepts the small probability that this may not be true.

1. Storage can can spontaneously decay.

HDDs don't last more than a decade. Flash drives can wear out.

Possible causes:

- Cosmic rays: How often might a bit be flipped by cosmic rays? It's a trade secret. Eggert guesses it's on the order of 1 day to 1 week.

"Maybe it's the color of something--or it's your grade on the final exam."

- Drops
- etc.

1. Errors are rare

"They're sufficiently rare with various probabilities..."

We're hopelessly toast otherwise.

1. Repairs can be done in time

We swap out a bad drive, for example. These assumptions were made for big data centers with a dedicated operations staff.

We end today with two big ideas for implementing durability and atomicity:

Commit Records

These apply to filesystems like locks to data structures.

With locks, we locked a struct, then accessed it, then unlocked it.

With a filesystem, we might need to write to 10 blocks. Instead of overwriting the original 10, we write a new version as a copy (perhaps next to it). Elsewhere, we have a commit record that tells us 0 (use the old version), or 1 (use the new version). As long as we can write the commit record atomically, we've gotten effective atomicity for all 10 blocks.

Journaling

Our file system consists of a big set of cell data, and our changes should be atomic.

Here's how to do it:

1. We have another area on-disk that holds a **journal**.
2. The journal begins by logging what changes are planned
3. We append a commit record, signing on the dotted line, and then updating the disk at our leisure. If we crash or lose power, we re-read the journal and see if there were any committed plans that weren't yet completed.

Discussion: 27 February 2015

This is a rather free week, so it's time to write the final report (2-3 pages long), which is a summary of an topic.

The articles reference papers that we should read for ourselves. In addition, we should read 2-5 related papers.

To find related papers, see the "Related Works" section of the paper. Most are not on operating systems, but we'll focus only on the OS aspect.

Final Report Format

All in LaTeX with the IEEE Transactions template.

Work inside the file bare_conf.tex. Don't forget to modify the headers. Can use title like "A Survey of [Topic]". Use @edu email.

\par ends a paragraph apparently.

IEEEtranBST has references.

Introduction: (no more than 1 page)

- Introduce the topic
- Motivate its importance
- Brag (indirectly) about novelty and contribution of paper
- Outline the paper's sections and ideas in each

Background:

- Give background knowledge for topic needed for the rest of the paper, assuming reader is an undergrad who has already taken 111.
- Not too much detail needed here

Method (at least 1 page)

- Summarize papers here: avoid splitting up and summarizing each paper separately.
- Instead, understand the ideas behind the papers and *compare* them:

  "Paper 1 does ... Paper 2 optimizes ... step of Paper 1 to increase performance by ..."

  "Paper 1 has the benefits of ... However, the drawback of it is ... which Paper 2 addresses by ..."

Conclusion:

- Short and sweet

References:

- Use LaTeX
- General rule: Avoid citing a paper more than twice in the same paragraph.
- Use the author's name(s) to get around this: "Sampson et al. did ..."

Style Suggestions

// I now understand why Jackie laughed her way through 183EW

- Avoid long paragraphs (> 20 lines). Each one gives one idea.
- Clearly structure using sections, subsections, etc. I highly doubt I'll need more than 3 levels of nesting, but that depends on the papers. We can just use manual formatting (or various packages that doubtlessly exist) with bolding and italics.
- Use academic English for the paper. Avoid "cool" or "nice" or "new" ("novel" instead; "challenges" instead of "issues")
- Use impersonal "we" instead of "I"

// Oh lord, is today just going to be a LaTEx tutorial?

To create a BibTex citation, just find it via Google Scholar, then do Cite->bibtex. Verify it, then add it to the the .BIB file.

At the of the .TEX file:

```
\bibliographystyle{./IEEEtranS}
\bibliography{./IEEEabrv, ./bibFileName}
```

In paper, just use \cite{shortHand} in paper.

To get PDF, just do:

```
latexmk -pdf file
```

No .tex exetension on file

Lecture #14: 2 March 2015

// Today should bring virtual memory as a topic, // one I've been looking forward to for a long time

Atomic Updates to File Systems

Previously, two big ideas:

1. A commit record
2. A journal

More generally, when we implmement an atomic action, we build it from lower-level constructs. The workflow:

BEGIN (mark as starting a transaction) - PRECOMMIT phase: Write down enough to know what we were starting to do - COMMIT phase: Commit to making a change - POSTCOMMIT phase: Improve performance by saving the changes to the journal of the cell data matrix END (done)

Some variations:

Main Memory Database

The journal is on disk, with no cell data on disk; instead, the cell data is held in RAM, with all changes logged to disk.

Walk backward through journal (newest to oldest), find the most recent write to the block of interest. We can read the journal to reconstruct the cell data.

Advantages:

- Access to cells is very fast (RAM-speed)
- No seeks when writing

Disadvantages:

- Uses up RAM
- Re-initializing cell data takes time, so recovery after crashes can be slow. In the worst case, we have to rescan the whole journal.

Have backup power systems for this approach.

Another approach: Toss the journal, and just keep everything in RAM, and have redundant copies everywhere across a network.

Write-head log

Two types of log; this is one of them.

This one logs our planned writes into the journal, all our planned actions to the cell data. Then, we add a commit record to the journal, and then we install the new data into the cell data.

```
- Log planned writes
- Commit
- Install new value
```

We can also make an abort record, which become NOPs when the journal is replayed.

Recovery:

- Scan forwards (in time) through the log, looking for committed but not yet installed changes

Write-behind log

Alternative approach. We log the old values of the cells we plan to overwrite.

```
- Log old cell values
- Install new values
- Commit
```

We can also make an abort record, which become NOPs when the journal is replayed.

Recovery:

- Scan backwards through the log, restoring old, uncommitted values to cell data (ones where we started to install new values, but a power failure interrupted, so the whole installation sholud fail)

  We scan backward to find the first ununinstalled change (because they need to be done in sequence), then we scan forward to install the changes in order.

Log Approaches Compared

Write-head advantages:

- If crashes happen a lot, this is more likely to preserve our last attempted action. As long as we get to the commit, we're safe. There's a smaller window for the crash to interrupt the new values.
- Less disk I/O; we don't have to log the old cell values. The value of old cell values is unimportant here.
  - Old values are typically cached anyway.

Write-behind advantages:

- More conservative/cautious on accepting new values
- Recovery is faster, as we know where the log is. We go to its end, and scan backward, undoing any uncommitted changes.
- More recovery possible

Increasing Parallelism with Atomicity

Cascading Aborts

For complicated atomic actions, we may worry about the possible bottlenecking.

P1: ----|action|---- P2 ----| wait |----

Support P1 is doing something complicated (merging files, etc.), and P2 is doing something simple, like sizeof(file). We can let P2 "peek" at the filesystem and see what P1 is working on.

We tell P2 a preliminary file size, but later we may tell it that the original file size we gave it was incorrect. We want parallelism to let P2 run ahead and then we can update it later. If safety is our top priority, then we avoid all these tricks altogether.

- Tell P2 the file size and it begins doing things
- P2 is later told "Oops, wrong file size"
- P2 aborts its transaction

It's cascading, because P3, P4, ... might have asked P2 for its processed data, which was dependent on P1's tentative data. We can automate these dependencies and force the chain of aborts.

Pros:

- More automatable

Compensating Actions

Told preliminary information, then later told it was incorrect. We have an action that compensates for that error.

Example:

- Told P2 prelim file size of 1000 bytes
- Tell P2 "Oops, it was actually 800"
- P2 adds a compensating action to make original story incorrect, e.g, write an extra 200 bytes

This is harder to automate, and it requires app-level support, as the application must specify the compensating actions, based on what it does.

Pros:

- More flexible

Memory Protection

Unreliable programs are often made so by bad memory references.

This morning, Eggert reviewd a proposed patch to the GNU C Library, with changes to 25 places to fix alloca(). It takes an integral argument, and subtracts the stack pointer.

If the number is small, the patch uses alloca(), and a combination of alloca() and malloc() otherwise.

Solutions:

- Fire all our current programmers, and hire better ones!
- Help from the hardware, which will catch any bad pointers. (Emulator works too). This is faster; we can even parallelize pointer checks alongside regular execution.
- Help from a compiler and other runtime checks. This lets us check subscripts, null dereferences, etc. We often catch

errors closer to their source, and so it's more debuggable, but it greatly slows performance.

Software doesn't know how things map to in memory, which is why we need help from the hardware.

Base/Bounds

A simple approach. We divide up RAM into regions, which are devoted to the OS and different processes:

[OS | P1 | P2 | OS | P3 | ...]

When a process runs, each process has two registers associated with it: a **base** register and a **bounds** register. With each memory read/write, we check that the address falls between the two:

```
P1_base <= address < P1_bounds
```

Disadvantages:

- Contiguous RAM for each process
- Hard to share. We could try to overlap these memory regions, but then we could only overlap for two processes for pipes, for example.

Now suppose pipes are done by kernel with traps to directly copy memory to be shared to different regions. And we also know how much RAM is needed, so can we allocate contiguously. Even with these assumptions, we have a problem.

What if we have two instances of a process, e.g., with a parent and a child?

Then somewhere in that process' instructions, we have:

```
jmp 0x10008
```

Which could also be a call, etc. This is an **absolute address** that works for one instance, but not the othere; the kernel would disallow access to that other region for the other instance.

Position Independent Code

Solution:

Require **relative addresses**:

```
jmp *+38
```

This is PIC code, position independent code, which is slower and less efficient, as the compiler is now restricted in which instructions it can use.

One solution to avoid PIC:

We take our memory address and add a base value to it. We check that the base + reference value is between P1_base and P1_bounds.

```
[ P4 ]
0    n
```

And we check:

```
P1_base <= address + P1_base < P1_bounds
```

Then, our address is a virtual one, one between and 0 and n, where n is the size of our allocated region.

Segmented Memory: Multiple Base/Bounds Pairs

To address the problem of contiguous RAM, we have 8 pairs. We'll have a convention:

```
text [base   Devoted to read-only code and data
      bounds]

data         Devoted to initialized read/write data

stack

  .
  .
  .

shared memory
```

These pairs can point into different, potentially scattered regions of physical memory. We physically have registers to hold the pairs.

Downsides:

- External fragmentation (enough room, but not all in one spot, and now everything must be moved around).

To grow a stack segment for one process, for example, we may have to move it to a different region of memory and then grow it. This copying takes time.

The virtual addresses themselves are now split into two parts:

```
segment # | Offset in segment
```

- Segment sizes are now limited by the offset. We don't have as much address space as we would have had by directly addressing physical memory.

Memory is split into domains.

Paging

Paging is to segmentation as FAT is to RT-11.

Again, a virtual address with two parts:

```
[ Page # | offset in page ]
   wide        narrow
 20 bits      12 bits
```

A **page table** maps our **virtual pages numbers** (VPN) into **physical page numbers** (PPN).

Then, we take concatenate PPN with our offset, and we get our physical address:

```
[ PPN | offset ]
```

The page table is held in RAM, and the %cr3 register points to it:

%cr3 -> [PPN | other stuff] [PPN | permissions]

Stack Smashing on Steroids

Early x86 had 'rw' but not 'x', as 'r' was treated as implying 'x'.

Nowadays, we have 'x' as well, to help prevent buffer overruns, where we overwrite the buffer to place malicious instructions and then place a pointer to its start.

By making all the pages composing the stack 'rw'-only, but not 'x', the malicious attack is foiled.

To get around this, attackers, even when limited to only writing to the stack, can use **return-oriented programming** (ROP).

We assume we know the victim program:

```
[ photoshop | ...]
```

We examine it in memory (read-only), and we collect all the instructions in it (including treating data as instructions). We're treating instruction strings like snippets of code, a DNA sequencing problem.

Looking at them, we see something like:

```
instruction, instruction, return
```

We get a huge amount of subroutines, all the byte routines. We can categorize them: adds two words to stack pointer, store into %ebx, etc.

If all together these regular instructions are Turing-complete, then we move about our return addresses to point to our cleverly subverted instructions. When the machine returns, then we jump into those instructions, now in a malicious order, even though they don't realize it. It's likely cleverly rearranging people's schedules and tasks such that they cause a secure building to be compromised, just through their regular, "safe" actions.

// Damn, that is so cool.

Page Table Size

4 bytes in page table represents 4096 bytes of virtual memory; 1/1000 of memory as overhead here.

Suppose awk (an interpreted langauge) starts small and wants to grow.

Its page table at first:

```
[ entry ]
    .
    . // unused, with zeroed permission bits
    .
[ entry ]
    .
    .
    .
[ entry ]
```

Here, 3 physical pages are in use.

Now, it grows, using malloc(LARGE_NUM). malloc() uses system calls and the OS allocates physical pages and updates the page table.

So how big does the page table need to be? Enough for 2^20 entries, 4 bytes wide, so each page table takes up 2^22 bytes or 4 MiB.

Now, this is 4 MiB *per process*. With 1000 processes, we've already exhausted RAM with just our page table storage.

(2^20 because each entry has a offset of 12 bits into the physical page for the exact byte)

x86 does support this mode, but instead we normally use:

2-level Page Table

To account for this problem, we have our virtual address divided into three parts:

```
[ 10 bits   |   10 bits   | 12 bits ]
  1st level     2nd level    offset
```

2^10 (1024) entries for top level page table, whose entries themselves index into a second level page table, which in turn gets us to the physical page level. %cr3 points to top-level.

Now, our overhead is smaller most of the time, 3 pages (~12 KiB) rather than ~4 MiB.

Would 3-level page tables be wise? Probably not on 32-bit, but on x64, yes, maybe.

Swap Space

One of the coolest advantages of virtual memory:

Programs can use more RAM than physically available. How? By exploiting the next level of our memory hierarchy.

We reserve enough space on disk/flash as process memory, with RAM being a cache for what's on disk. We use our page table to keep track of the mapping between RAM and disk. When we're lucky, it's all in RAM.

We have a 0 in the page table to mark an entry as unused, and trap when that happens.

[RAM] // 2 GiB | | V [disk] // 16 Gib

This area on disk is called the **swap space**.

Performance can be very bad. This requires that locality of reference to be practical, where we only access virtual memory all around a region.

Without this, we get **thrashing**, where we spend most of our time moving memory between disk and memory for each process. Depending on the application, we get thrashing once we go >= 1.5 physical memory size.

Some people deactivate this feature, to prevent thrashing.

Page Fault

When we don't have the requested data in RAM, we have a **page fault**. The process traps into the kernel, which must deal with this. We'll have:

```
pfault(va, cp, atype)
// pfault(virtualAddress, currentProcess, accessType)
```

Access type is read, write, etc.

Implement:

```
void pfault(void* va, struct proc* cp, int atype)
{
    // Check if page is on disk and just needs to
    // be moved into RAM, OR if the virtual address
    // was an illegal reference

    // int swapmap(va, cp) tells us:
    // 0 : invalid address
    // address in swap space (on disk) otherwise

    // Assuming kernel maintains its own page table
    // in software-defined format for what's valid
    // for each process

    if (swapmap(va, cp) == 0) {
        // Segfault: signal segmentation violation
        kill(cp, SIGSEGV);
    }
    else {
        // Get this page from disk into RAM,
        // but RAM is already full-up:
        // "Free memory is wasted memory"

        // Pick a victim physical page currently
        // held in RAM, and boot it out of RAM
        // We'll need some removal policy.
        vppn, vva = removal_policy();

        // Write out VPPN (victim physical page number)
        // to disk, so that we don't lose it

        // Read in desired page to same physical location

        // Update page table
        page_table[vva] = 0;
        page_table[va] = vppn; // Virtual address = used up PPN
    }
}
```

Lecture #15: 4 March 2015

// Midterms are finally back... // and here comes the curve:

80-89: ..... 70-79: ..... 60-69" ..... 5 ... 20-29: . [exactly one sad hash mark]

median: 55 mean: 56.9 s.d.: 13

Target mean is 50.

// Why must he hand it out during class... // During 9th week!

Review: Page Replacement Mechanism

Recall:

Page table maps virtual addresses to physical addresses that specify actual locations in physical RAM. Meanwhile, we have swap space on disk.

IF we get a page fault (where a virtual address mapped to a physical page number not present), the kernel takes over. It

evaluates if it was an invalid page number altogether, or if it referred to a page in swap space but that was not in RAM.

Note that pages currently in RAM may have a garbage version in swap.

We now need to *swap to disk*, choosing a **victim page** in RAM (referred to by physical page number). PAges on disk are referred to by virtual page number.

Steps:

1. Choose a victim
2. Write out victim to swap space
3. Read new page into physical RAM
4. Update page table

Page Replacement Policy

We need a policy for selecting victims, one that is:

- Efficient and fast (in kernel, which handles this)
- Avoids thrashing (spending all our time swapping out pages rather than doing actual computation)

These are competing goals.

Comparing Policies

But how do we test and compare policies? We gather data from real world programs, and keep track of every page access, in order. This can be stored simply as **reference strings**, a (ordered) list of virtual page numbers. That is enough to compare our policies.

We'll have a toy example:

5 virtual pages, indexed 0, 1, 3, 4

3 physical pages, indexed A, B, C

Example:

0 1 2 3 0 1 4 0 1 2 3 4

Went through, went back to beginning, then to end, then though again.

FIFO

First In, First Out.

It's certainly very simple and efficient.

Let's test FIFO. We assume the values are initially in swap, with RAM holding garbage data.

The top row is VPN requests, and we track what each physical page is storing. A 'x' stores where we had a paging happening.

```
0 1 2 3 0 1 4 0 1 2 3 4
```

A ? 0 0 0 3 3 3 4 4 4 4 4 4 B ? ? 1 1 1 0 0 0 0 0 2 2 2 C ? ? ? 2 2 2 1 1 1 1 1 3 3 x x x x x x x x

9 page faults

So maybe we buy more RAM?

With an additional piece of RAM (lettered 'D'), we actually get 10 page faults.

This is the **Belady's Anamoly**, something that can happen with FIFO, and utterly counter-intuitive.

LRU

Least Recently Used

This particular reference string demonstrates how a heuristic like LRU doesn't always work out. This one pages out the page that was used the longest time ago, with the prediction that something that hasn't been used for a while may not be used again soon.

// See scribe notes for this, as too messy to try to copy // down via typing right now

To choose the victim page, look at the reference string, and see which of the virtual page numbers currently held in RAM was accessed least recently, it appears furthest down the reference string.

10 page faults

To implement LRU, we can modify our page table, adding a timestamp field. To pick a victim, we look at the timestamps and pick the oldest one.

But this isn't used, because it's crazy from a hardware point of view. We'll have to load and store for the page table, in addition to our regular load or store on our physical page. That's 3 operations instead of 1! Also, we're searching linearly.

So in practice, we don't actually implement a full LRU scheme. We implement an approximation instead. The timestamps are to the nearest second or so instead, and may even be wrong.

Remember, any change to the page table means hardware changes, as this is part of interaction with RAM.

Instead, in the kernel, which maintains it own page table, we add in an approximate time column. This is not visible to the hardware. The software page table will only be updated when the kernel has control. This happens with clock interrupts or with a system call. More importantly, it can do so when a page fault occurs.

To reset the clocks, the kernel can zero out the page table entries, which will cause a kick back to the kernel next time, which then updates its software page table. See scribe notes

Oracle

If we had access to the Oracle of Delphi, it'd consult the future, and tell us which to swap out, based on which page won't be needed for the longest.

```
0 1 2 3 0 1 4 0 1 2 3 4
```

After the initial loading, '2' won't be needed for the longest time. The end result is 7 page faults, and this is the optimal result, constituting our upper bound.

Can we consult an oracle while page faulting? Run it twice! Run it once, save its reference string, and then use that to inform your next run.

Applications like payroll, bootup sequences, etc., can all exploit this. A compiler could also accomplish for some things.

Is this one reason why browsers are much faster on second run? Additional reasons would be things are cached in RAM, etc.

Paging Optimization

Many others besides these two.

Demand Paging

Two ways to start a program:

1. Load the first P pages into RAM, then run

Advantages:

```
+ Better throughput if program uses these
  first P pages a lot initially.
```

1. Load just the 1st page into RAM (entry point here) then run it. We get a lot of page faults, but we deal with them as before

Advantages:

```
+ Startup is faster
  (We'll see initial output faster.)
```

We might strike a compromise between them, loading $1 < p < P$ pages.

Dirty Bit

The cost of a page fault:

Minor:

- CPU for policy decision
- Trap overhead

Major:

- Write out victim page to disk
- Read in the faulting page from disk

To cut down on costs, we can skip writing out the victim if we know it hasn't changed since being read in.

Now, in the hardware page table, we store just 1 bit: dirty (1) if RAM-cached copy differs from the copy in swap, and clean (0) otherwise. We initialize this to 0, and change it to 1 when a write happens to a page.

Assuming victims are mostly clean, this can greatly improve performance.

To make things easier:

- Have permissions all set to r-x in the hardware page table
- In the kernel, we flip the dirty bit in our software page table
- We set the 'w' bit in the hardware page table
- We hand back control to the process, which does the write
- Later, when we page out the victim, the kernel knows it needs to be written out.

Notice the general pattern: If the kernel tracks something, we need the kernel to be in control to make those updates to its internal trackers.

fork() Optimization

Suppose Emacs wants to fork() and have the child run 'date'. Problem: the fork() will have to clone all of Emacs' virtual memory, which might be huge, say 6 GiB. We don't have time to copy all of that.

So a clever optimization is to have the child point to the parent's stuff instead, by cloning just the parent's page table. We take the write bit for both, if *either one* wants to write, we get a page fault, and the kernel makes a copy, then marks that page as writeable. Why zero out writes for both? Because otherwise, the child and parent will see inconsistent pages, due to each other's writes.

This is **copy on write** (COW).

AHHHH: This is what Eggert was reference way back when I asked Eggert if a child had separate copies of everything.

This is a variant of demand paging, and so the child can do what it does, including a short time to execvp().

Performance is still not as good as it could be, as we're still copying the page table.

vfork()

To get around this, we clone, but we freeze the parent until the child does a execvp() or an exit(). The child shares everything from the parent, but the parent is frozen, such that we don't get memory stomping between the two. The child will have to be careful with the parents' data structures though.

It's a controversial syscall, apparently. Emacs does use vfork() instead of fork(), if it's supported.

malloc() Implementation

FINALLY!

How does something like malloc() work?

```
void* malloc(size_t size);
```

Remember that it returns a pointer into *virtual* memory.

mmap()

This is a system call that lets processes modify their own page tables in particular ways. Many arguments to it.

A typical task: Map a file's pages into RAM. The data is mapped into a virtual address and the page table is updated. Nothing is moved into physical RAM, we just have updated page table entries. read() and write() are still needed though, not just for streams.

Problem: Our app is now at the mercy of the OS page-replacement policy, which can't possibly match our own custom tuned programs. malloc() does the equivalent of:

```
mmap(..., "/dev/zero", ...)
```

We get zeros when we read. This would not work with /dev/null

Apache

Apache has code like:

```
for (;;) {
```

```
    int fd = accept(...); // Get a request
    read(fd, ...);
    handle (fd, ...);     // Apache code
    close(fd);
}
```

Problems:

- The server is limited to one request at a time

To improve:

```
for (;;) {
    int fd = accept(...); // Get a request

    fork(); // Have the rest done by children

    read(fd, ...);
    handle (fd, ...);     // Apache code
    close(fd);
}
```

Would vfork() help here? Because we're again freezing the parent and limiting it to one thing at a time.

Better solution: use threads instead:

```
for (;;) {
    int fd = accept(...); // Get a request

    pthread_create(); // Have the rest done by child threads

    read(fd, ...);
    handle (fd, ...);     // Apache code
    close(fd);
}
```

A thread doesn't need to clone anywhere near as much as a cloned child process. Problem: more brittle, we can't afford a misbehaving thread.

Performance issue:

We make a thread for each requests: 1000 threads, but we only have 8 CPU cores.

"You now have 1000 different little birds in your nest demanding these CPUs."

Context switching now increases greatly as a cost, as we're constantly scheduling and switching between threads.

Even faster: Just have one thread per CPU, and each thread has an event queue. It pulls off the first item, handles it, then moves to the next. This is **event-oriented programming**, a.k.a, asynchronous I/O. We never switch; the threads are always running.

We may never do a write(). Instead, we do an aio_write(), which sort of runs as a background process, and we can check later if it finished. Again, all these threads share the same thread table, decreasing synchronization costs.

Distributed Systems and RPC

Before, we used *virtualization* to enforce hardware modularity, constructing virtual machines of sorts that limited what the apps that ran atop them could do.

Now, we'll distribute systems. We'll have the guts of the machine we try to protect on another machine altogether.

An app runs on a **client** and our "kernel" runs on the **server**.

With virtualization, an app sees:

```
pid_t p = fork(); // "A function call!"
```

With RPC, we see:

```
int n = read(fd, buf, sizebuf);
```

But the data is actually held elsewhere. A function call that is implemented elsewhere.

RPC (Remote Procedure Call) vs. Syscall

RPC:

- Caller and callee don't share memory The server cannot directly place results of an RPC into a client's memory
- No call by reference (no addresses can be shared, as each has its own memory)
- Caller and callee might use different hardware architecture (No callbacks, e.g., function pointer to be followed when something occurs. Data sizes or Endianess may not even agree)

Different machines with different data: sounds exactly like the Telemetry Data Chain I need to figure out for Rockets

Marshalling (serialization/pickling)

In the RAM of the client, we have some beautiful data structure. We want the server to be able to see it. To transmit it, we need to define a format that sends them in a serial way to the server. The server sees this parading messages arriving and reconstructs the data structure, based on the rules of the format.

Or, the client can ask the server what it wants, and then convert data before sending it. Most Internet protocols are Big Endian, because the machines at the time of the protocols' design were Big Endian.

Discussion: 6 March 2015

Don't hack the template for the report, but we can get rid of the Abstract section if we want.

Don't be perfectionist with the report.

Lab 3 due date was extended by 1 day.

No guideline for this report. Freaking Eggert and his lack of documentation.

// Lol, everyone asked for Lab 4 skeleton, // but no one has read the spec yet

OSP2P protocol is Lab 4.

Lab 4

OSP2P is similar to BitTorrent, with tracker and peer nodes.

The original code is terrible; we need to hack it for scalability and security.

Downloader can ask for anything from uploader; lack of permissions checks.

We are now using client-server model for achieving hard modularity, including RPCs.

Each peer connects to a tracker, and registers itself with the track (RPC: 'ADDR'). Then, the uploading peer registers each file it has available with the tracker. (RPC: 'HAVE')

When we want to doownload, the tracker acts as an intermediary, telling the downloading peer which uploading peer has the file available. Then, the peers will connect to each other.

We'll apparently use telnet to connect to the servers.

Linux network working socket layer user interface

To communicate with another machine, the other machine must have an OS that supports networking.

Socket: An abstraction of an end point of IPC via a network.

Sockets are also file descriptors, but when we write to it, it's sent to the other machine. It's a very nice abstraction.

socket() creates a socket and returns its file descriptor.

We can send or receive information with the socket.

bind() binds a socket to a local address. listen() will allow a new connection to a socket.

socket()

```
int socket(int socket_family, int socket_type, int protocol)
```

socket_family: IP address (for the Internet; we won't use Bluetooth, for example)

socket_type: TCP (we'll use TCP here)

protocol: Put 0 (only used for when socket_type is SOCK_RAW)

bind()

```
int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen)
```

sockfd: The socket we made with socket()

addr: Specify which address and port (family is same as socket_family type)l

addrlen: Length of 'addr'

If we use INADDR_ANY is used for the 'addr' argument, then we bind to all local interfaces

listen()

The socket will now be listening for incoming connections requests using accept()

sockfd: socket created by socket() backlog: length of queue for pending connections Once backlog is full, other requests will just be dropped/refusesd.

accept()

This creates a connection, based on the first request in the listening socket's internal queue.

It will return a new file descriptor.

Q: What does this represent versus the old file descriptor?

Possible that this accept() returning a new file descriptor lets us process the queue of requests without having to do the listen() setup work.

connect()

Skeleton Code

MD5 is what we'll be using for data integrity. An extra-credit portion asks us to use this to verify that we got the right file.

writescan.c (with header osp2p.h): helper functions for I/O and string parsing. Look HERE before implementing things. Don't forget to look for useful helpers!

osppeer.c: Download/upload code

sscanf and its output brohter take a string IP address like "192.168.1.1" and convert it for storage as an address inside of struct in_addr.

getpwuid() return the information of the current user, based on UID returned by getuid().

Then alias is name and time, to get something like user3920. Diyu finds the presence of an alias unnecessary.

If this fails, then use "osppeer #" as a default.

The minute we see labels, we know there's a goto...

The argprocess-goto process is just a while loop.

If we don't give an address and port, the default one is used.

In our commandline invocation, we specified which files to download. Once we're done with that, we upload files while other peers request files from us.

The code differs from what the spec says.

This is on the file downloading and file registering part.

htons() takes a port, and it converts an unsigned short integer from host byte order to network byte order:

```
[h]ost[to][n]etwork[s]hort
```

What is this needed for? Endianness.

Big Endian places the MSB (most-significant) first. Little Endian places the LSB (least-significant byte) first

0xABCDEFGH

Might be stored in memory as:

0xGHEFCDAB

Note that a single hex char stores 2 bits, half a byte., and Endianness only affects the order of bytes. Their individual bits are not flipped.

htons() just prevents this by converting to the network standard.

Note that no connection formed until accept(). listen() just notes requests.

Scalability and Security

Parallelizing Downloads/Uploads

Currrently, downloading (as seen in main) is sequential. We want to make this parallel. This is true also for uploading. Again, we want to parallelize this.

Two main ways to do so:

Fork off children processes, and creating more threads.

Processes:

-

Threads:

- Much less overhead

QUESTION: How were file descriptors affected on processes vs threads?

Apache: Limits the number of threads per process to the number of cores, and each thread services one connection. When we need more, we spawn a new process.

Buffer Overflow

There are buffer overflows. Find them and squash them.

Permissions

Our current code totally allows peers to request—and receive—important files like /etc/passwd.

We should not allow files to be requested from outside the current directory.

For example, is constraining our request path to not begin with '/' enough?

No, because they could feed a malicious one like:

```
/   /    / /etc/passwd
```

Though we could iteratively strip to the first alphachat.

Bad peers could also try:

```
~/...
```

We'll need to prevent both.

Nobody knows how the tracker works.

Diyu is trying to set it up now.

Lecture #16: 9 March 2015

Media Faults

Disk drives go bad. Flash goes bad too: it wears out.

A good paper to read: Mai Zhang et al.: "Understanding the Robustness of SSDs under Power Fault" (FAST '13).

// This paper changed Eggert's view that flash is more reliable

Flash drive problems

These happen in the presence of power failures.

- Bit corruption

Lampson-Sturgis assumes this is expected, and rectifiable (if done soon enough), but not too bad.

- Flying writes

The record is stored correctly, but in the wrong place. The checksums are all right, so it's harder to catch

- Shorn writes

The first part of the write succeeds, and we get a successful return message, but the last few bytes weren't actually written/somehow ignored

- Metadata corruption

Recall that flash systems have a logical layer to implement wear levelling, where we map logical blocks to different physical ones. Problem: the map from logical to physical can itself become corrupted.

- Dead device

We wrote everything, but now the device is dead, and can no longer be read.

- Unserializability (buggy controller)

WRites are sent to the flash controller, as a set of requests with an order in which they are to be done, but the controller may have its own optimizations and do things in a different order. If the controller is buggy, then the logical level will make no sense because of the actual physical writes.

To test this, the researchers set up a test harness with their drives, wrote test programs, and periodically interrupted the power to see what would happen.

Solutions:

- Test different SSDs for all these problems, and buy only the reliable ones. (13/15 failed these in the research)

Manufacturers don't do this, because reliability doesn't matter that much for most people. For those who do, like with Network Appliance, Isilon, etc., do do this.

- Design a file system that overcomes these problems

RAID

Example: **RAID**: Redundant Array of Independent Disks, designed by Dave Patterson, a UCLA alumnus.

We get a U-shaped curve for $/TB vs. capacity. The current sweet spot is around 2 TB.

RAID 0 exploits this with **concatenation** of multiple physical disks to make a larger virtual one. We pretend four 2 TB physical hard disks are one massive 8 TB HDD. It's much less expensive than an actual 8 TB drive.

RAID 1 uses this for **mirroring**. We have two 2 TB physical disks, and use these to implement a single logical 2 TB disk. The two physical disks mirror each other for redundancy. Even if one of the drives fails, the other one is still good-to-go.

- More reliable
- Reads are faster (we can have our two disk heads reading different parts of the file at full speed)
- Write twice (a bit slower, but writes can be parallelized)
- Uses half of our storage on redundancy; we need twice as much disk space as as actually needed for data

There's a rebuild procedure to copy from the good drive to the bad one (~5-6 hours for 2 TB at 100 MB/s).

RAID 4 (2 and 3 exist too) reserves one of the N drives for holding parity numbers. Suppose we have 6 physical drives. Like RAID 0, the first 5 drives will be concatenated into one massive drive:

```
[A][B][C][D][E]    [XOR parity drive]
```

The final drive will implement XOR as a parity check: A^B^C^D^E. If the parity drive fails, we replace it, and rebuild it by XOR'ing the data drives.

If another drive fails, the same procedure works:

A^C^D^E^(parity) A^C^D^E^(A^B^C^D^E) => B

(XOR is commutative and A XOR A == 0)

While in **degraded mode** (a drive is being rebuilt), we had better not lose another drive, or else we're screwed.

Some ideas to improve this:

- A **hot spare**, an extra physical drive that contains anything, and if there's a problem, the system immediately rebuilds to the hot spare. This works even if the ops chief is out on vacation. This shortens the vulnerability window for catastrophic data loss.
- More parity drives. The more parity drives we have, the more drives we can afford to lose. We'll need a more complicated parity function, but that’s fine.

All of these RAID techniques can be stacked: we can build a wedding cake: mirrored, then concatenated, etc.

RAID 5 is RAID 4 + striping. We take drive [A] for example, and split it into 6 pieces. There's a little bit of A, B, etc., on each drive. And parity is also split across drives. Each drive contains 1/6th of the A drive, and 1/6th of the parity.

Why use RAID 5? Performance. RAID 4 sucks for performance, as the parity drive is a hotspot and/or bottleneck during writes (but not reads, as we don't need to change parities then). The parity bits have to be updated.

SEASnet uses RAID 4.

RAID 4 vs. RAID 5: Suppose we want to grow our virtual disk. With RAID 4, we simply tack on a new drive, which begins (virtually) as all zeros. The parity check is now A^B^..^F, and F is 0, so nothing to changed. RAID 4 is thus easy to grow, whereas RAID 5 will need a whole rebuild. RAID 0 and 1 also easier to grow. Thus, RAID 4 is better from an operations point of view.

If we can't make the L-S assumptions, we'll have a harder time.

Hardware RAID vs Software RAID

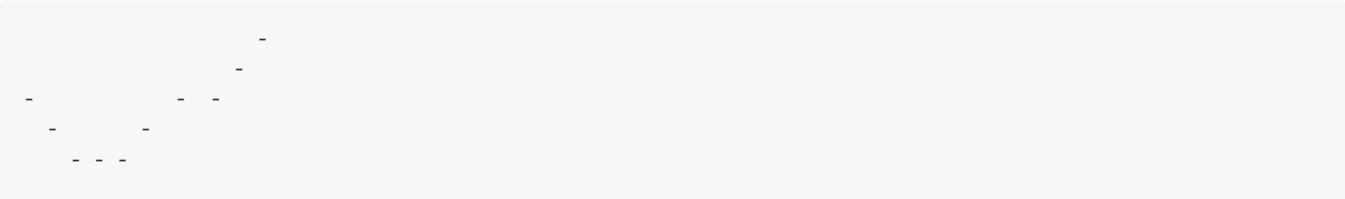
In hardware RAID, the OS doesn't know about the all RAID stuff; the RAID controller pretends to be one massive disk. In software RAID, we get higher performance, as the software knows more about application needs, but it complicates the OS, which needs to coordinate the drives setup.

RAID typically uses uniform disks. When we have different sizes, we often pick the minimum, and when we buy the next one, if the lower size is no longer made, we buy the next size up, which wastes some space, but's acceptable.

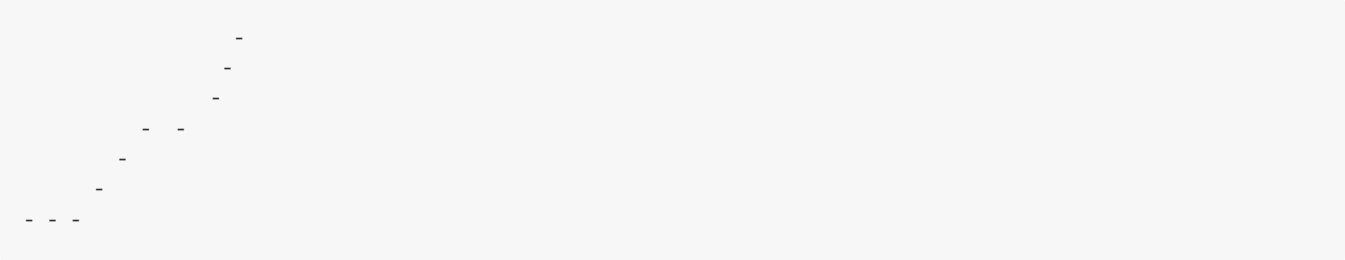
Parity drive could be an SSD, as it's an expected hotspot, so we could make it faster. Normally, hybrid systems uses the SSD as a cache. Journals are often in flash, with cell data on HDDs.

RAID Robustness

Failure rate of 1 disk drive: Bathtub curve (actually quite jerky). It starts high, then shallows out, then rises gradually.



Suppose we have RAID 4 with 6 drives, then what's the failure rate of the whole system? Our goal is to keep it very low, and this is doable, thanks to our ops engineer swapping in spares. But if we assume a mission to Mars with no replacement drives, then we can't keep the failure rate low indefinitely. We'll eventually have a second failure in degraded mode. This failure rate is actually higher, as the chances of 1 of 5 drives failing is more than 1 of 1 drives failing.



A poorly-managed RAID can be less reliable than a single drive. Same thing can happen with UPS.

RPC Failure Modes

Recall that with RPC:

- Client/server cannot trash each other's memory
- Messages can be be corrupted (checksums are probabilistic)

Easy resolution: We get back a "What did you say?" response, and resend the issue.

- Messages can be lost (transmission error, too many packets, etc.)
- Network can be down or slow
- Server can be down or slow

  From the client's POV, it's hard to tell if

  - The network is slow
  - The network is down
  - The server is slow

- The server is down

The three above problems (including lost messages), can be solved with:

- If no response, resend; keep trying. This is an **at-least-once RPC**, often used for lower-level operations, where it's okay to repeat an operation, e.g., disk writes. Such operations are called **idempotent operations**, where repeated applications of the idempotent operation will deliver the same result.
- If no response, give up, and report an error to the user. This is an **at-most-once RPC**. This is often used for financial transactions (e.g., to prevent double transfers).
- Exactly-once RPC: transaction IDs may be resent, and the server will ignore one of the double transactions. This is harder to do, and leads to the Byzantine problem

HTTP

Examples of RPC: HTTP

Client to the server: GET /HTTP/1.0 \r\n\n\n ... Server to the client: HTTP/1.1 200 OK \r\n ...

SOAP is a system for building RPC atop HTTP.

X Window System

Another example: X window system. The X server handles display, keyboard, mouse, etc. The client might request to draw a pixel at some coordinate with some color: Draw(12, 23, "blue"). The X server draws the pixel, and sends back an "OK, pixel drawn" response. X is "pixrect on wheels".

Obvious issue: Performance! SEASnet lags like hell over SSH -X.

Some solutions:

- Change the APIs to have higher-level operations. Instead of drawPixel(), we have drawImage() or drawRect(), etc. X in fact does this, which improves performance.
- Batching: Send several draw() requests at once, rather than draw-wait-OK. This approach is called **pipelining**. We'll have several outstanding requests from client to server at any one time. This improves performance, but what if one of these requests fails? The client has to be prepared to deal with failures. Requests cannot be depend on preceding requests, and error handling is trickier to prevent cascading failures.
- Cache on client side on the server state, so that we can avoid asking the server unless it's something we don't know about, as the roundtrip overhead of server-client can be quite awhile. We cache answers, questions we just asked, recent responses, etc. This improves performance, but we get the classic cache problems of: complexity, cache coherence (cache-server mismatch).
- Guess what queries will be asked, and pre-fetch them. This is what Google Chrome does with pre-loading top search results from the Omnibar. This again improves performance, but we encounter the same problems of complexity and cache coherence.

Network File System (NFS)

Recall the stack:

App | V Linux kernel | V VFS (virtual file system layer) | V Filesystem implementation (NFS) | V <---> network <---> NFS server

NFS protocol looks like Unix syscalls for files (they're not exactly the same):

Client request (dirfh is directory file handler, name is something like "abc", with no slashes, and should refer to a file in that

directory)

```
loopup(dirfh, name)
```

Response:

```
file handle + attributes
```

A **file handle** is a unique ID for a file in the network file system (an integer). NFS returns an file handler for the "abc" in that directory referred to by dirfh(). Attributes means metadata like last modified time.

Recall that namei() resolves a filename to unique ID: "/usr/bin/abc". For Cal's FFS, we'd get an inode. For NFS, we might get something like a pair of (device #, inode #). namei() would have to do each part of the name to get to the file on the server. Note that the pair will probably be a packed-together large integer: 930|39024829408, where we have dev#|inode#.

"Suppose we have one last assignment, due Friday. Suppose."

NFS servers are written in the kernel, not as user-level apps, as we have no syscall for getting a file just from its device and inode number. This is easy at the kernel level.

Stateless Server

NFS' big idea for dealing with this is the idea of a **stateless server**.

Client | V <------ Network ---------> Server | V Files

The idea is this: The contents on the server don't matter. We can lose it, reboot it, etc. without worry. Corollary: the server can't keep track of who its clients are. Its knowledge on client state is ephemeral, sticking around only until reboot or crash.

We're making the server simple from a client POV. This makes the client "fat", as it needs to do more now.

More NFS primitives:

Request:

```
create(dirfh, name, attr)
```

Response:

```
file handler + attributes (file created)
```

Request:

```
mkdir(dirfh, name, attr)
```

Response:

```
file handler + attributes (directory created)
```

Request:

```
rmdir(dirfh, name, attr)
```

Response:

```
file handler + attributes (directory removed)
```

read() and write() also exist, and are in the same vein, but for reading and writing. rename() also exists.

Suppose we have Client 1 write() to file1, and Client2 does a rename() on it. Any problems? No, because rename() doesn't change the file handler. There won't be a communication problem.

But what if now Client 2 does remove(file1), then Client1 does write(file1)? These can clash: the server will report: "No such file". Now in UNIX, this is annoying, which is accustomed to unlink() and and write() at the same time, where the storage isn't reclaimed until the last process with that file descriptor still open exits and closes it.

But without state, NFS servers make it harder to implement UNIX semantics.

Two standard solutions:

- If unlink() and write() are both on the same client, the client kernel generates a funny request like rename("file", ".nfs930284"). The write will happen on that instead. Then, when the write() is done, the .nfsblah file will disappear via a remove() RPC.

This has issues. Suppose the client crashes. Then, .nfs junk files will be leftover. We saw this before with orphan blocks. It's up to applications to cleanup their mess.

BUT what if our assumption that the NFS syscalls originate from the same client is not true? Conflicting processes on different clients will lead us to a violation of UNIX semantics, and we'll get errno set to `ESTALE`. This means that the file handler is stale; the server says that the file is not one it knows about.

A worse problem: The server doesn't think the request is stale.

Client 1: remove(file1), then create(file2), where file1 and file2 have the same file handler

Client 2: write(fh, ...), but it's again the same file handler, which leads to writing to the wrong file. To solve this, NFS has a serial number, which is part of the file handler:

```
device#, inode#, serial#
```

In general, we are dealing with at-most-once and at-least-once RPCs.

Lecture #17: 11 March 2015

Security

// Today is the last lecture of CS 111, Winter 2015 // So sad...I looked forward to this class for a year, // and loved it as I went through it.

// And now it's over

"Let me begin by telling you how to turn in assignments past Friday's due date."

CCLE runs a script to check if the assignment was submitted before the due date.

The nominal hard due date: 2015-03-13 23:55

So we pass something like:

```
http://ccle.ucla.edu/----cs11----/? turnin _cs111_
```

This is a turnin script, found in like /usr/.../bin/turnin. It checks its arguments, etc., and we can confuse it about the timezone ('tz'), e.g., `tz=America/Juneau`.

The turnin script takes `name=value` arguments and puts them into the environment, rather than passing them into turnin. It can be set to whatever. Eggert is actually the maintainer of the `tz` system.

However, CCLE's server doesn't have this bug, though other submission systems do.

What happens if we do `tz=../../../etc/passwd`

Security is a big problem; it's adversarial computing.

Our adversaries are malicious and seek our the flaws in our systems. Examples from recent tech history:

- Sony hack (November 2014)

"It caused a terrible movie to be watched by too many people."

- Supreme Council on Virtual Space

Reporting directly to the Ayatollah, they manage virtual space in Iran, and attacked the BBC's websites in 2012, taking it down worldwide for a few hours)

UCLA does have courses on security:

- CS 136: computer security
- CS 183: cryptography

Today will just be a brief intro to computer security.

In the real world, our concerns can be divided into:

- Attacks via force

Gun-armed attackers, or physical break-ins to server rooms, etc.

- Attacks via fraud

Operating systems mostly deal with this.

"Some of you are doing this right now, attending this class without being enrolled in it."

"We're the nerds, we get to worry about this."

Main Forms of Attacks

- Against privacy

The attackers want information that we have and want to keep secret. This is unauthorized release of information. Example: Sony hack

- Against integrity

Tampering with other people's data. Eggert has to worry about grades being changed. This can be fraudlent information, or just pure information destruction.

- Against service

The attackers want to make our computer useless, by overloading it in some way, e.g., denial of service (DOS).

General Goals

We want to:

- Disallow unauthorized access

If someone is not allowed to change grades, they should not be able to. This goal protects against attacks on privacy and integrity, and indirectly helps with protection against attacks on service.

This is hard to test; it's our attackers who find this. Testing is essential here, and hard practically.

- Allow authorized access

This is often forgotten about, but onerous security systems are less likely to be used well.

This is easy to test: impose the security system and see if users can still get access.

- Efficiency is good-enough

We can't afford for our security to get in the way.

This is also easy to test; users will complain if things are too slow.

Security Testing

So how do we test our security system? We want to test for disallowing unauthorized access.

Let's:

- Create test clients that try to do unauthorized access
  via malformed data ("This is not what we want to do with our lives. It's boring."). Fuzz testing is rarely done well by developers, though it can be tested.
- Hire a tiger team (hackers trained to break into systems) To have an even harder time, give them the source code. This is expensive though.

Threat Modeling and Classification

There are many, many holes in large software systems. Eggert is constantly bombarded by emails about security problems.

Half an hour ago, Eggert got an email about `setenv`. In C, we can change the environment:

```
setenv("PATH", "/usr/bin:bin", 1);
```

The bug in glibc:

```
setenv("PATH", NULL, 1);
```

What happens? The environment is set into a strange state, and getenv will give back an unexpected pointer into internal memory

When it comes to bugs, we need to *prioritize* them. To do so, think about our adversaries. Who are they? What are they likely to try to pull off? Based on that, we prioritize our bugs: what's most likely to be used, and which one is most urgent? As a practical matter, no design is perfect, and so security work never ends.

Common threat types:

- Insiders

People can misuse their authorized access. Registrar's backups are held on magnetic tapes taken to who knows where. This should be top of the list, as it's one of the worst. Edward Snowden must have shocked NSA greatly.

- Social engineering

This is how an outsider can pretend to be an insider, bypassing all our external defenses.

Kevin Mitnick was a SoCal hacker of corporate telephone systems. He'd go to a telephone pole, call the central office for the telelphone system, and he'd ask for the password. He was assumed to be a repairman.

- Network attacks

Computers attached to the Internet must worry about a whole menagerie of malware: drive-by downloads, etc. ~10% of websites have these (2013 estimate) and are often compromised sites, not purposely malicious ones.

- Device attacks

Our machine will use a bad device masquerading as a good one. For example: a USB drive containing malware. Drives are stolen all the time, and at Baghram (USAF base in Afghanistan) AF base, repairmen bought USB sticks from the local bazaar, which had stolen and loaded with malware.

General OS Security Functions

Almost any OS will need these features/primitives.

Authentication

We need to check user identity. Is an actor who they say they are?

The most common technique: passwords.

So what if SEASnet refused to let use use passwords that were already stored? BUT, if so, an attacker could find out that a password in use. Solution: Don't tell them that’s why you're saying "No".

The major categories of authentication techniques:

- Based on what we *know* (passwords, etc.)
- Based on what we *have* (RSA SecureID, etc.)
- Based on who we *are* (biometric authentication)

"If we stole someone's thumb, wouldn’t that be something we have?"

"Oh, the boundaries are somewhat blurred, but you don't need their thumb."

The lab he used to work in had a thumb print reader that was too easy to fool. They switched to iris reading.

It is possible to manufacture (physical) keys from photographs of them.

To bolster all this, we use **multifactor authentication**, ideally from different categories. Very few people here use this, sadly.

Attacking Passwords

Some possible attacks:

- Guess them (takes forever)

A defense: rate-limiting. Too many attempts in a short period of time, or captchas, etc. all slow the amount of attacks.

Another one: complexity requirements.

- Snoop on them

Passwords are transmitted. Any router on the way can intercept and see the password. Even easier: someone over the shoulder, or a camera that saw the keystrokes. Never type password in a public place, e.g., LAX.

- Fraudulent service

Fool the user into authenticating on a false website, e.g., udla.com. Eggert's favorite: microsoft.com, where the 'o' is Cyrillic.

Defenses: Buy all the domain names; certificates (which are signed off on by a certificate authority, but sadly, they're incompetent); etc.

External and Internal Authentication

Real-world analogy: Bagram Air Force Base. The external perimeter has fences, armed guards, etc. We'll need proper ID, etc. to get through the few entry gates. This is external authentication. It's also expensive: a call to central, etc.

Computer equivalent: SSH key exchange.

Now, we get a badge that lets us traverse parts of the base, though not all of it. For someone like us, we'll get an in-person escort. This is internal authentication. It's also required to be cheap and fast. Glance at the badge, that's it.

Computer equivalent: We've logged on, but now we must check if the user has R/W/X permissions on various files.

Integrity

We need to check for tampering.

The most common technique: checksums. These have to be stored in a secure location, and files can be in a slightly-less secure location

Authorization

Assuming a user has authenticated, we still need to track what they're allowed to do, i.e., their *permissions*, stored as a few bits in each inode.

Methods of Access

There are two major ways to let people access resources:

- Direct access

Map into address space (e.g, mmap()).

Pros/cons:

```
+ Fast
- Hardware must support access restrictions we want
- Not flexible
```

Example: We have a memory-mapped graphics buffer that represents a 1024x1024 screen, with 3 bytes per pixel:

```
char screen[1024*1024*3];
```

We can have apps write directly to parts of the screen, but if we had a windowing system, then we only want the app to access a *subwindow*:

```
[       |allowed|   ...   |allowed| ... |allowed|     ]
```

- Indirect access

We give the application a **handle** that's opaque: a small int, a restricted pointer, etc. This handle can be used by syscalls only; and syscalls are handled by the kernel, which themselves determine access.

Pros/cons:

```
+ Flexible (even circular/elliptical windows)
+ Easier to revoke access (rather than map/unmap)
```

In UNIX, users are often called *principals* (as users often have devices working on their behalf). Then, we have *resources* (like files).

We can imagine permissions as an array with users on the rows, and resources on the columns. We track if they have access via 1 or 0. But we have a third dimension: the operation types: read, write, execute, etc.

user : optype : resource

So for an organization like SEASnet, we have a few thousand users, rounded to 10,000. And say 10 million files and 4 operation types => 400 billion bits in this array => 50 GB to track who can do what. This is not an option; performance would be terribe if we stored on disk and cached in RAM, etc.

To cut down on this, we need to *look for patterns*. For example: Eggert will have permissions on everything Eggert/*. Also, Eggert, Palsberg, etc. work together and share a lot of files. The entropy in this 3D array of bits is much less than it seems at first.

Our goal is not simply compressing this structure, creating a compact representation that's easy to process and read.

Access Control Lists (ACLs)

With each resource, we associate a list of users and groups authorized to access that resource, and the operations they can perform on that resource.

Example:

/home/eggert/ Root, eggert: rwx Others: r-x

We walk through the list and look for the first match.

UNIX/Linux has a simple approach: users/group/other We have rwx bits for each one, but it's rather inflexible. A more genreal approach is to let users specify their own ACLs. Our Linux servers allow:

```
setfacl
getfacl
```

Btw, drwxr-xr-x+ means that there is an ACL on a file, and we can use getfacl to find out more.

Users can now set up ad-hoc groups (e.g., "This is our new exclusive club.").

Capabilities

In this approach, we associate access lists with principals, not resources.

Example: We have a shell running on our behalf. We authenticate and build a capability object that we present to the OS for approval. It includes an unforgeable unique ID. To give this capability to another process, we simply copy over that whole bit pattern (modifying it to add/remove permissions, if needed)>

Linux does both. It has ACLs, and uses capabilities as well.

Suppose:

```
fd = open("foo", O_RDONLY);
chmod("foo", 0000);         // No one can access this file now
read(fd, buf, sizeof(buf)); // This read() still works, as our permissions
                            // for reading were obtained as a capability
                            // when we open()'ed it, though subsequent
                            // open()'s will fail.
```

So how might we make a secret file? Don't do this:

```
touch secret
chmod 600 secret

echo 923-234-3252 > secret
```

An attacker can do an open() before the chmod, and then does a read() after the echo.

"I have a 20-lecture course and only 17 lectures."

Auditing

We need to record accesses, e.g., login records.

We can catch imposters, see what they did, etc.

Correctness

All our regular OS functions must still work without our security making things too slow.

Trusted Computing Base (Kerckhoff's Design Principle)

Named after a Belgian cryptographer (whose life was pretty secret) who promulgated some principles:

- Minimize what needs to be secret
- Argued against security via obscurity; the system should be secure even if the adversaries get ahold of the blueprints
- Keys must be secret, but nothing else must be

In operating systems, we try to minimize what we have to trust. So, we'll trust just the kernel, or even just its core.

Problems:

Pointed out by Ken Thompson (Turing Award winner and do-designer of UNIX). Famous paper:

"Reflections on Trusting Trust"

He built a bug to get into any UNIX system in the world:

login.c:

```
if (strcmp(user, "ken") == 0) {
    uid = 0; // root!
}
```

To hide it, he modified GCC:

gcc.c:

```
if (compiling "login.c") {
    insertSecurityHole(); // The buggy code no longer
                          // need be in login.c

}
```

Now to hide this change:

```
if (compiling "gcc.c") {
    insertGCCSecurityHole();
}
```

The binary version of GCC won't reveal it's been poisoned.

Solutions: Keep old versions of GCC, disassmble the GCC output, do everything ourselves, etc., but they are counteracting attacks, e.g., modify GDB. If we don't know when he made these nefarious modifications, we can't do much.

The big idea: our **trusted computing base** is much larger than we think, and we should put everything into it that we need. At the end of the day: Whom do we trust? How much should we trust?

Not only do we need a threat model, we need a trust model.

// Applause. That's all folks.

Discussion: 13 March 2015

Old 111 final from 2004:

Problem 2

Does an older process always have a lower PID?

No, because it wraps around once we fill up the process descriptor table.

Problem 3

Is it possible for a Linux-based multiprocess application to have deadlock and livelock simultaneously?

No. Livelock means tasks are being completed, but slowly. Deadlock means tasks are being blocked from completion (threads are waiting for each other). This is by definition.

Analogy: We want to go to downtown from Westwood, but we always pick the route with the least traffic. However, a stupid GPS means that we'd never get there if we kept re-routing.

Problem 4

IS it straightforward to implement link(), unlink() and rename() on FAT32?

link() creates a hard link unlink() removes a hard link rename() is like mv

Recall the FAT format:

```
[ boot | super block | FAT1 | FAT2 | Data blocks ]
```

In the FAT table, we have an index that is either the index of the next block or EOF. The first block ID is stored in the super block. Directories are also stored in data blocks.

link() is hard

rename() is hard

unlink() is easy

rename: simply renaming is easy, and moving fiel to another directory using rename: hard

The reason link() is hard here is the lack of inodes, which gave another layer of indirection.

Problem 5

Explain why the variable time-stamp resolution system of FAT32 might lead better overall performance. Conversely, give arguments for why Solaris' tmpfs uniformly high resolution might give better performance overall.

tmpfs might be better overall because it's all RAM-based, and therefore operations on it are are far faster than ones split across RAM and HDD, like FAT32.

FAT32, however, avoids the overhead of constantly updating files' metadata down to the nanosecond. Note how FAT32 only keeps data of last access, not its time, which means it's faster for many reads in one day.

However, Solaris is better than FAT32 for many writes. (Also interesting: Solaris is written in C++).

Problem 6

In special case of pipe to pipe, can cat have the kernel link the input pipe directly to the output pipe?

How do we glue two pipes together?

send_file() sends data from one file descriptor from to another, all via the kernel, and blocks until it's done.

Could create a new system call sys_glue() that merges two pipe buffers together. This would be done by the kernel rearranging them until they're contiguous in memory. Problem: Buffer grows larger and larger.

External fragmentation would make this difficult, e.g., enough free space, but not enough *contiguous* free space.

Problem 7

In 32-bit x86, we have 2-level page tables.

In x86-64, we have 3-level page tables.

But we want Linux to be portable, so it assumes that the page table is 3-level. The three levels: pgd, pmd, pte. They get 10, 0, and 10 bits for their offsets (within the tables). And 12 to offset within page.

In PAE (physical address extension) mode, they have 2, 9, and 12-bit offsets. Also, in PAE, every table has 64 bits for every entry (to suppport more than 4 GiB of RAM).

We're comparing how much memory is needed for the page tables themselves if the lower 2 GiB of virtual memory is occupied. We compare the size needed in (a) original i386 mode, and (b), PAE mode.

In 32-bit mode: We use half of the first-level page table, which means 1024/2 * 32

And we use 512 of the second-level page table: 512 * (1024) * 32

We know to use half of the first level, (as we were told half is used), and then we assume all of the second level pointed to by the first level is used up fully. THen, we multiply by how big each page level entry is.

Total is 512 * 32 * (1 + 1024)

For 64-bit mode, similar logic

Problem 8

Indirect blocks are rather similar to intermediate page directories in multi-level address maps. Explain the main differences in implementation goals and strategies between the two schemes.

Differences: Indirect block is more flexible. We can have many of them, and then support larger sizes.

Problem 9

This is a trick question: it's the opposite of what we were taught in lecture as a solution to priority inversion.

Here, if we have the thread needing a lock held by a lower priority process, then taking on its lower priority just means both processes will be scheduled over.