CS 111 Lecture Notes (@jlin97)

Lecture 1 (09/01/17): Introduction

Logistics:
Prereqs: algorithms, data structures, assembly language, software construction, 131, 151B computer systems architecture, 118 networking
Hours/week: 4 lecture, 2 discussion (lab), 9 outside study → 19.4 hours
CS111 is hard not because each individual piece is hard but because there is a LOT of material to cover
OH: M 10-11, W 13:30 – 14:30, 4532J Boelter
Projects: almost like a 2nd course
Grading: 20% midterm (5th week), 30% final exam (180 min) open book, open notes (tip: bring notes), Labs, 5% research report (3 pages on some leading edge OS topic
This class will require a TON of reading, deadline for projects 23:55
Labs are all SOLO, a lot of solutions are online
Textbooks: AD is optimistic, SK is pessimistic
Lectures will mainly be pep talks to get us awake at 8 in the morning

Now to get onto the material:
System: OED (1928) I. an organized set or connected group of objects II. A set of principles, a scheme, method
More definitions: a set of interconnected components that has a specified behavior observed at the interface (boundary between system and environment) w/ its environment
Biggest problem in OS, trying to figure out where interface should be

Operating system: 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 (American Heritage 4th ed. 2000)
Encarta – 2007 defines OS as master **control** program in a computers
Wikpedia: 2017-01-06: system software that **manages** computer hardware and software **resources** and provides common services for computer programs; in other words, it controls resource **allocation** and **management** to make the system whole

$ ls -l big
-rw-rw-r-- 1 eggert faculty 922337203685477500 Oct 6 11:31 big ← holey file, takes very little space
$ time grep X big
real 0m0.009s, which is around 10^{-2} s → 10^{21} B/s
All of Internet bandwidth combined can allow for a file transfer rate of: 2 x 10^{13} B/s
All of US freight full of microSD cards: $1.2 mil/gallon allows for 0.5 Zb/s = 0.5 * 10^{21} B/s
How is grep so fast? It doesn't use open+read, it uses system call lseek (has options SEEK_HOLE, SEEK_DATA) which makes it sort of like the world's simplest database

Problems with computer systems:
1. incommensurate scaling: not everything scales at the same rate, diseconomics of scale (star network)
    - if a central hub wants efficient connections between components, it's an $O(N^2)$ problem, 8,16 port hubs are cheap, but 96-port is not because of the complexity involved in retaining efficiency!

- Other repercussions: wastage (pin factory)

2. emerging properties arise as you scale, qualitatively different, unanticipated e.g. Taroma Narrows bridge, the emerging proeprty was resonant frequency
3. propagation of effects: happens too often in OSes
e.g. Microsoft SJIS-encoding for Asian languages was used to fit a ton of Asian characters for ASCII, went over and leaked into file systems part of OS. Microsoft's fix was to make file system SJIS aware, but this made the OS more complex (bad)
4. tradeoffs: waterbed effect – can't make one part of OS really fast without making another part slow
e.g: space time tradeoff; instead of quick/merge sort, keep an array of pointers, sor tthem and read from there. Increases efficiency BUT uses more memory
5. complexity: Moore's law, # transistors on a chip increases proportional to log scale
Kryder's law: disk drive capacity increases as well

HW: write a program to scan 10 terabytes of disk sequentially

Lecture 2 (11/01/17): Abstractions and bootstrapping

Management mechanisms, coping mechanisms to understand complexity
Abstraction: to reduce to only the points we care about
Modularity: break the problem into pieces, each piece is a smaller independent part of the problems
You can focus on only the piece that you care about

Interface vs implementation; system vs environment; we should make it so that we change implementation but not interface (this doesn't always work this way in practice though because interface isn't always stable)
e.g. fid = open("file", O_RDONLY) ← standard interface since the 1970s
Let's say we want to add a new capability to the function, how? Interface evolution
We start with the interface we like and improve upon it. The function we want to add is to be able to open remote files

Ideas suggested by class:
1. New function, fid = remote_open("code.google.com", "file", O_RDONLY)
Downside: requires changes to all our apps, it's a hassle
2. Change the API for open, fid = open("code.com", "file", O_RDONLY)
Hard implementing two functions at low-level, may not support function overloading
3. Add a flag to the function; open("file", O_RDONLY | O_REMOTE, "code.com")
Flag will tell system to use new connection with these parameters
5. putenv("FILESERVER=code.com")
6. fid = open("file0code.com", O_RDONLY)
7. fid = open("code.com/file", O_RDONLY) ← might get away with this
8. fid = open("/u/class/cs111/file", O_RDONLY) ← mask the server as a directory

Suppose we don't believe in interface stability, abstractions, modularity. Thought experiment: anti-OS, not having an OS may be an advantage for NSA. We don't have to build an OS but we gotta get an app to work. Sample: paranoid professor, afraid USC has bugged our computers, stealing our data.
Proposals for ideas can't be over 10000 words. We want to count words using Linux, wc.

Project: build word counter that is 1. paranoid 2. as fast as possible

Computer, no Internet connection, battery-powered. Important questions: where is the file on disk? How do we access this?

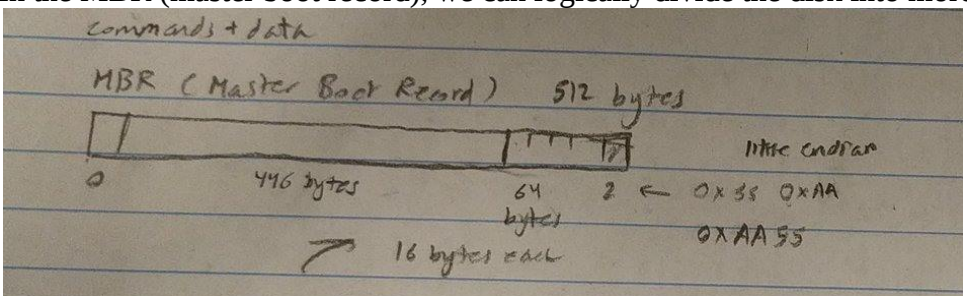
It'll be on sector 1000000 → the read head will read until null byte '\0', and sector size will be 512 bytes, and the bus is 16 bit. The controller will be attached to drives. Before there used to be SATA (Serial Advanced Technology Attachment where data was transferred serially rather that in parallel but now, PATA (also known as IDE) is used. (Parallel ATA). PATA is faster and has a hotplug signal but downside is synchronization overhead.

1. Bootstrapping → when you power on computer x86 processor runs in 32 bits, memory is in two pieces: DRAM and ROM, kept in a device called EEPROM (DRAM is volatile which is bad)
In computing, a bootstrap loader is the first piece of code that runs when a machine starts, and is responsible for loading the rest of the operating system. In modern computers it's stored in ROM.

2. Commands + data
ip = 0xffff0, at this address ROM has a program called BIOS, set by ROM manufacturer
ROM (normally) 1. tests system, 2. looks for devices trying to find instructions telling it what to do, 3. finds bootable device, 4. reads the device's first sector into RAM at 0x7c00 … 0x7dff and 5. jmps into 0x7c00

The MBR is the information in the first sector of any hard disk that identifies how and where an operating system is located so that it can be loaded into the computer's main storage or RAM.

In the MBR (master boot record), we can logically divide the disk into more sectors/regions



Describes an area of disk (partition)
Partition descriptor has: start sector, size (# sector), type (1 byte that says "I am bootable")
BIOS runs MBR which runs VBR (Volume Boot Record), which is the 1st sector of the bootable partition
Chain loading loads grub (Grand Unified Boot Loader) which loads actual Linux drive

Boot loader source code

```
for (i = 1 ; i < 20; i++)                    // want to execute at 0x7c00
   read_ide_sector(i, 0x10000+(i-1)*512);

read_ide_sector(int s, int address);          // this chunk sets up disk controller for the next command
while(inb(0x1f7) & 0xc0)!=0x40) // 0x1f7 control/status register
          continue;
```

```
outb(0x1f2, 1);          // 1 sector
outb(0x1f3, s&0xff);  ← grab last bits
outb(0x1f4, (s>>8)&0xff);
outb(0x1f5, (s>>16)&0xff);
outb(0x1f6, (s>>24)&0xff);
outb(0x1f7, 0x20);
while((intb(0x1f7) & 0x20) != 0x40)
        continue;
insf(0x1f0, a, 512/4);
```

Discussion 1

OS is hard, but its interesting and useful
lab0: warmup
1. have a workable linux system
VMWare workstation / Virtual Box, HW will be graded on Centos 7
lnxsrv09 seasnet server ← test lab again
2. read manuals: open(2) ← second section of open man page (man 2 open)
3. write a program that copies from stdin to stdout

1. file descriptor: integer associated with a file; file in file out

```
int ifd = open("file.in", O_RDONLY);
int ofd = open("file.out", O_CREAT | O_WRONLY);
char buffer[1024]; // 1024 just a convention seems like
while (read (ifd, buffer, 1024) != EOF)
   write (ofd, buffer, 1024);
close(ifd);
close(ofd);
```

scanf(" ", …); ← will eventually call read → read(0, …);
printf will eventually call write(1, …);

Reasons for using scanf, printf, over read, write: 1. better formatting 2. scanf, printf is a level of abstraction that helps hide low-level details and helps with portability

-- input = filename, use filename as standard input
Signal is inter-process communication; signal is sent to process to notify the process some events have occurred. When hardware detects an exception, a signal will be sent to the handler.

Lecture 3 (18/01/17): Modularity and virtualization

Last time: learned how to get a program starting
booting: read_ide_sector, used to load the program (what else does it do?)
But we never got around to what the program actually does
The idea is that we will read in the executable code for the following program, we will put it on a certain number of sectors on disk, we'll jump to main, where main will count the number of words in the file

```
#include <stdbool.h>
```

```
// counting words in buffer, repeatedly counting the words
void main (void) {
        long nwords = 0;
        // we want to count each transition into a word, want to count the edges, by keeping track of
        whether we are in a word or not
        bool inword = false;
        unsigned s = 10000; // will only support 2^32 sectors, each sector has 2^9 bytes = 2^41 bytes
        // to fix should we use larger sectors?
        while (true) {  // each time through the loop we'll read a sector
                char buf[513];          // array is local to the loop
                read_ide_sector(s++,  (int) buf);
                for(int j = 0; j < 512; j++) {
                        if( buf[j] == '\0' ) {                  // bug: this program will infinite loop if no EOF
                                write_out(nwords);
                                // before, we had return; which is a bug
                                // because we jumped into this code there is no return address
                                while (true) continue;
                        }
                        // if current character is alphabetical, then it's the start of a word
                        bool thisalpha = isalpha (buf[j]);
                        nwords += !inword && thisalpha; ( equal to ~inword & thisalpha)
                        inword = thisalpha;
                }
        }
}
```

Occasionally in embedded systems you want to have infinite loops, in this case the number will be shown on the screen infinitely

Memory mapped I/O, CPU writes into region, other parts of the computer
We start in 0xb8000, built-in into the hardware
Monitor has 80 characters, 25 rows vertically, 80x25 grid
16 bits per character for the monitor:

- Low order 8 bits is ASCII
- High order 8 bits is color 7 which is short for (grey on black)

```
void writeout (long n) {
        //easiest way to output to the screen
        //start with last digit (just take the remainder of dividing count by 10)
        unsigned char * screen = (unsigned char *) 0x68000 + 80*25*2/2+8;
        // shifted to the right 4 spaces
        do {
                screen[0] = n%10+ '/0'; ← ASCII 0
                screen[1] = 7;
                screen -= 2;
                n/10;
        } while (n != 0);
}
```

Problems contained in this code:
1. integer overflow
2. read_ide_sector_ has multiple copies, what if there's a bug in one of these
    1. next to the word count main function
    2. master boot record
    3. in EEPROM

Master boot record doesn't need to call read_ide_sector if read_ide_sector is already in ROM
read_ide_sector has to be at a well-known location in ROM, 0x7f000. Stick with the chosen address
ROM contains BIOS: basic input output system

Downside: suppose we come up with new disk devices, old functions won't be useful anymore, Linux uses BIOS but never uses any of the slow code anymore on modern hardware. This approach is inflexible.

3. busy-waiting, CPU keeps asking disk controller, "Are we ready? Are we ready?" CPU tied up constantly querying the disk, would be more efficient if it could do other work in the meantime

If a program polls a device say every second, and does something else in the mean time if no data is available (including possibly just sleeping, leaving the CPU available for others), it's polling.
If the program *continuously* polls the device (or resource or whatever) without doing anything in between checks, it's called a busy-wait.

I. Solution: CPU should do two things at the same time. It can count words of the previous sector waiting for the current sector to read. We'll have two buffers, CPU cycles expended to count words will be free, the amount of time for I/O is the total amount of time used
Technique name: **double buffering**, because it relies on two buffers, trading memory for better CPU performance (triple buffering won't help)

II. Solution: INSL instruction, grabs data into CPU and ships it off to RAM, it occupies the bus twice, disk controller → CPU → RAM, less bus bandwidth to deal with other activities
Better: DMA: Direct Memory Access, CPU still has to talk to the disk controller , but disk controller does all the heavy lifting itself by accessing RAM.

4. Want to run several simultaneously which is not enough, but newer devices, desktops are using something completely different. Unified Extensible Firmware Interface. UEFI
- We have much smarter firmware; instead of having a BIOS, which looks for a boot record and it's done, firmware has a lot of stuff it can do.
- Standard format for bootloader (EFI). Operating system independent.
- Also has Globally Unique Identifiers (GUID) for partitions (128-bit numbers); idea behind this is for when drive unplugs and plugs disk or removes drive; firmware will remember the device's GUID.
- Also has a GUID Partition table (GPT), which is interpretable by the firmware
- EFI partitions use MS-DOS file formats
- UEFI boot manager in firmware, which is configurable via NVRAM, reads GPT tables, can access files in DOS formats, and can run EFI programs (in a sense it is like an operating system that opens other operating systems)

We would like our word count program to run on this system too. RIP.

6. Port to UEFI

Is this Lab 2? Take what it would take to make that kind of improvement. We've got code and the code works but the code will be too much of a pain to change. We have a paranoid program but it's too much of a pain to change. Do-able if you're willing to sit and write a whole bunch of code but it's going to be a lot of work. It's going to be too much of a pain to take some of these ideas, and make the same changes for other programs too. Pain to reuse code in other programs. For 5. we don't really have a mechanism for this feature at all. The code we've written now assumes there are no faults in the hardware, assume that the data will come off the disk and there will not be any read errors at all. We're going to have to go through our program and fix it so it has a lot more **error-checking** and recover from faults.

This kind of design, although its fun to see how a program boots from scratch, its a bad model for writing real world applications. This is a bad way to go. We need a better approach, one that will let us change code easily, reuse code, without burdening application devs about worrying about this kind of stuff.

We want to have a way of writing software, where the software just … works.
Basic technology that we are going to use to get this kind of flexibility is **modularity**.
Break code into pieces with well defined interfaces between the pieces so that we can start plugging and playing. The core part of our word count program will be sitting in one place, read_ide_sector, writeout in other pieces. The goal is a software engineering thing.

Suppose you have N lines of code and you have K modules. You have a large program. Number of bugs is proportion to number of lines of code. Time to find bug a bug is also proportional to number of lines of code. Debug time: # of bugs * N = $O(N^2)$. BUT if you have modules, each bug will be easier to fix. Bug will be in a particular module and each module will have # bugs/K per module * (N/K) * (K) = $O (N^2/K)$ which speeds it up by the factor of K.
Every line of code is a module? O(N) LOL, some bugs cross modules, and this simplistic analysis is not realistic. But the idea is that modularity is important in software systems. Modularity can be good or bad. How do you tell? If someone proposes to break up your program into modules, do you consider it a good or bad strategy? You need metrics.

1. Performance; does adding modules significant help or hurt performance? Generally modularity hurts performance because it builds walls within the system. And modules may need to access the internals of another module for maximum performance. But because of this modules say, "Ok I'll do it the slow way." Thus, hurts performance but **not too much**
2. Robustness; how well does the system tolerate violations of the interfaces that you set up, tolerance of faults/errors. Faults not only in the hardware but bugs in the program. You want a way of modularity such that a failure in one module doesn't fail all modules.
3. Flexibility/neutrality/lack of assumptions; the basic idea is that we want our module interfaces to not put in arbitrary restrictions or limits on how we can link modules together. We should be able to hook up three random modules of a system together if we want. Apple module links with Google module
4. Simplicity; easy to learn, use, and has a short manual

All these four are competing objectives; if you look at the word count program it doesn't do well on any of the objectives except simple. Break: think about life and its mysteries

HOW TO IMPLEMENT MODULARITY:

0. Don’t do it, save yoself
1. Function call: caller/callee modularity, where its the line between the caller and callee that is the modular interface. Example:

```
int fact (int n) {
   if (n == 0)
      return 1;
         else
            return n*fact(n-1);
}
```

x86 code gcc generated, without optimization:

```
fact:
        pushq  %rbp           # push the old base pointer onto the stack
        movq   %rsp, %rbp     # copy new stack pointer
        subq   $16, %rsp      # subtract 16
        movl   %edi, -4(%rbp)
        cmpl   $0, -4(%rbp)   # see if n == 0
        jne    .L2
        movl   $1, %eax       #if it is return 1
        jmp    .L3
.L3
        leave             #rsp = rbp; pop into rbp
        movl   $396, 104(%rsp)
        # reaching into the caller’s frame and replacing whatever is there with 396, maybe some piece
        of important information and overwrite it completely. Callee has the ability to mess with caller,
        caller can’t even trust callee to return correctly
        ret               #pops the return address into rip
.L2
        movl   -4(%rbp), %eax
        subl   $1, %eax
        movl   %eax, %edi     #edi contains the
        call   fact
        imull  -4(%rbp), %eax
```

With function call modularity, what can go wrong?
- callee can mess with caller, or caller can set stack pointer to 0 and call callee too
- callee can loop forever
- stack overflow
- callee can mess with caller’s registers as shown below:

```
.L3
        leave
        popq    %rbp
        addq   $1024, %rsp
        jmp    %rbp
```

Function call modularity is soft modularity and relies on trust. It is the most popular kind of modularity in software systems but not good enough for operating systems. We want **hard modularity** by which we mean the callee is insulated from the caller. No matter how badly the caller operates, the callee should still be able to run. In order to get this, I can't rely on function call modularity. How can we get hard modularity?

If you don't trust program, run it on someone else's computer
1. Use different computers for each module, when the modules need to communicate with each other, they can send emails or via a network API (used in cloud computing, network computing)
- Problem: gives you hard modularity but cost is pretty large, only works if modules are large and can do a lot of computation, and requires little communication between each other

Easier way? Instead of using different computers, use the same computer but we'll write an interpreter
2. Write an interpreter, which runs each untrusted module. Write a C program that is an interpreter that can execute x86-64 instructions. Test module inside interpreter. Sounds ridiculous due to overhead.

```
int main ( ) {

}
execute_insp (char *ip) {
        //implements addcp   ← checks memory address, and if fails returns -1
        //but doesn't solve the Halting problem, but you can halt the program with a time limit
        //we want reliability more than performance
}
```

Using an interpreter is slow so we need hardware instructions so that we can go make method 2 as fast as the actual hardware.

Discussion 2

We only need to implement read only, write only, verbose and command (most difficult part)
Read about fork, catch up on lecture reading, fuck I'm behind

```
a = 0;
pid = fork();
p = pipe (      );       ← if you have a pipe outside, the pipe will be duplicated in the child process
if(pid == 0) {
        // child process
        // you have a new process in this part, has the same memory status, is a copy of the parent
        process
        a = 0;    value of a will initially still be 0 but this a is a copy, not the same a
else if (pid > 0) {
        // current process
}
```

After execvp, nothing else below it will be run?

Options has a list of options; for each one, the second parameter is either req_arg, no arg, or optional.

You can use opt_index to move backwards and forward, to find all the params

After you return from the child process, the current process can track the data structure to find the specific pid (WTF)

For lab 1b, implement all other options except for –profile. They are very simple, except for pipe? Pipe function will have two params, an array with two elements and an int

First to use pipe, you create an array, which has two elements. Each of the elements represents a file descriptor, fd1, fd2.

If you close(pipefd) in the current process, it will close the pipe in the current process only not the

Read end of pipe in child process, use write end of pipe in parent process

dupd(pipe[2], 0); automatically closes and dups, safer because it works atomically
close(pipe[1]);

Use waitpid() and a loop to check whether all child processes are done
We create three child process, we can create a data structore to keep track of all the information. The command that will be run by the child process is called, and then we call wait()

– close, pretty trivial
– abort, just copy the code from lab 0, create a pointer and assign it to NULL, cause a segmentation fault
– catch N, same just copy the code from lab 0,
– ignore N, in signal(SIGINT, SIG_IGN, the signal is ignored)
– default N, in signal(SIGINT, SIG_DFL, the signal is set to default)
for own handler(?) use signal (SIGINT, handler)
– pause, even more easy, just pause(void);

Most important bit: pipe, and how pipe works
winter16/cs111/scribe/3c/index.html

Lecture 4 (23/01/17): OS organization

What are some goals we have when trying to organize an operating system?
- Protection: want it to be secure
- Robustness: ability to cope with errors during execution
- Utilization: have it doing useful work vs useless work / nothing at all
- Performance: answer a question efficiently (maybe at the expense of utilization)
- Flexibility
- Simplicity

Incompatible goals, you can’t have all 6, maybe only 2, have to put up with deficiencies in some areas to have a happy medium for allocate

Classic way to **organize** operating systems: classic way we organize any computer program. Have modules and have them connect to each other through function calls. Some operating systems operate this way and work, but function calls are very bad for **protection and robustness.** We need a better way to organize our OS.

Try to look at this problem more abstractly, in one of the readings of the book, it mentions three fundamental abstractions that are a better technique than function calls. Used not only to gain protection and robustness but also in many future cases in class.
Fundamental abstractions: memory, interpreters and links

Memory: has load and store options, read and write.
What's the complete size of all the memory? Not just total size, what's the word size? The unit of size that you use in dealing with that memory x86-64, word size is 64 bits, but underneath, its the cache line size which depends on the manufacturer (typically 64 bytes) If OS is going to insist on buffers that aren't 64 bytes you'll have trouble.
What's the throughput? Latency? Volatility? (when the power goes away does the memory go away or survive?) We have to give the illusion that all memory is non-volatile
Linear vs associative memory: linear: have to specify the address and system gives it; associative uses key value pairs
Coherence: multi-level memory; cache contains parts of lower level memory but what if it doesn't?

Interpreter: includes instruction pointer ip, environment pointer (context under which OS operates), repertoire (includes the built in operations of the machine but also includes functions)
Normal execution vs exceptional execution (throws or traps/interrupts)

Links: sort of abstraction used for I/O devices. Instead of having load/store, you have send/receive when using I/O bus
Memory could be seen as a special case of the links abstraction but it's more complicated than that

How do we use these abstractions to do something better than function calls when it comes to protection and robustness?

Function calls are soft modularity; we want hard modularity.

2 ways to get hard modularity using these abstractions.
1. **Client/service organization:** e.g. SEASNet server, you have two different computers. They have a network and the client sends a read request to the server and the server responds back. We're using links abstractions. We get hard modularity in a straightforward way. Client/server cannot mess with each other, except through their messages.
Advantages: we limit error propagation and we do it by having no shared state, even if one side loops or crashes, the other side can still continue working (robustness)
Downsides: hassle to set up, and takes up more resources

Real world problem: client/server configurations or network problems. We'd like less hassle and cheaper. We turn to method 2:

2. **Virtualization:** you just have one computer but it pretends to be two or more
One computer + one CPU, but we make it seem as though there are many. We want the boundary between different parts to be secure to have hard modularity

Easiest way to achieve this: virtualization via emulation. We're using x86-64 or ARM hardware. The hardware is the interpreter. One way we can get virtualization is by slowing things down a bit. We write a C program that will emulate ARM and run on SEASNet. If something bad happens, emulator can catch the bug and protect the system. Example: qemu; can run programs and varieties of instruction sets. Advantages: quick for debugging. Problem: it's very slow (DS emulator)

An emulator is doing just that: It is *emulating* a piece of hardware using software code. Since the GameCube games don't run natively on your PC hardware, the emulator software has to read each instruction, then translate it into instructions that your PC can understand.

We need help in order to get full speed execution from instructions while still having safety of emulator. We will try to get help from Intel and AMD.
Instead of virtualization via emulation, we'll have **virtualization via hardware support**. If you look at the instructions being emulated, many of them are fine. In hardware, we'll add a filter and use it to check whether an instruction is safe. If the instruction looks weird, then the hardware is going to give up. This method divides instructions into unprivileged – run at full speed (addq $1, %eax), and privileged instructions which don't run at all. (at attempts to run these instructions, hardware traps will activate). At rare cases when privileged instructions are attempted, the system has to reboot (bad)

You have interrupt service vector. (sometimes called trap array). Basic idea: there are different traps that you could run into. Hardware could have convention that index 0 is error divide by 0. Whenever the hardware traps, it looks into the service vector and gives a pointer that points to a code to be run when this trap occurs. It could be that some of these errors could be errors we could fix. Perhaps what this code wants to do is to put infinity in the register when at an instruction that tries to divide by zero.

Trap not only stores instruction pointer, but saves stack pointer, flags, stack segment, code segment, error code

Where should the trap save this information? We can't save it in registers, because we don't want to reduce what we can use on registers. We save them on the stack, run as if each were called by a function call. Traps are more expensive than function calls. For traps **you have to save a lot more** than function calls. A trap is a less common event but it uses more memory.

How do traps give us hard modularity? We put the trap array in protected memory so apps aren't allowed to access it. The trap handles themselves need to be in protected memory. Traps are a means of protected transfer of control.

For function calls, you have the call and return operation. You push and pop on the stack.
When we're done with whatever needed to be done, we need to resume with normal execution.
(trap)+rti, pops all traps from the stack. The trap handler can execute privileged instructions.

Hardware keeps track on whether their instructions is privileged or not in flags. Traps are too slow to send to memory. By the time trap handler is done, memory has changed.

```
char bof[511];
read(0, buffer, 512);
```

We'll look at machine code for read, return address is sitting on top of stack.

```
read:
   halt
   ret
```

For every system call, we could pick a privileged instruction but there are 255 possible errors so that's not going to work. Linux has the convention to use the instruction INT that simulates a hardware interrupt. Takes an operand, we give it 128

```
read:
   push $12
   int 128       // ok we'll take that first one and we'll mean system call. This in effect is the syscall trap.
```

The original reason traps were put in was because programmers wanted to keep the program running when the program crashed. → eventually for when program crash or I/O error (traps and interrupts) → when OS were being developed, this method was reused for virtualization

```
On x86 (faster approach)
   sysenter
        sets cs eip, ss, esp to values (protected) set n model-specific registers
   sysexit
        sets eip ← edx
             esp ← ecx
             cs  ← ss   from model-specific registers
```

Less traffic between CPU and memory because you're placing values in machine-specific registers. Goal: making system calls go faster.

```
x86-64 (even faster)
   syscall     args                        // can see syscall as instruction
        rdi rsi rdx r10 rr r9                                   (input)
   It destroys rcx and r11 and puts result in rax               (output)
```

Suppose you want a syscall with 9 arguments, you just put the rest in r8 which points to an array in memory.

In Linux, to get system calls to work, have to use a VDSO (virtual dynamically linked shared object). The idea is that system calls in Linux are not sitting in application when set up. They are brought in dynamically by the CPU. The vDSO (Virtual Dynamically linked Shared Objects) is a memory area allocated in user space which exposes some kernel functionalities at user space in a safe manner.

```
$ ldd /bin/sh   ← list my dynamically linked libraries
linux-vdso.so1 → (0xf7744000)      ← the shell has this piece of memory accessible to the command
that is supplied by the kernel (can contain syscall) Functions are not originally there but are linked in
libc.so.6 → /lib/libs.so6 (0xf74b5000)
/lib/ld-linux.si,2 (0xf7447000)
```

syscall are just more instructions. Specify argument → you get results back but they operate differently. Kernel's job is to execute system calls and can execute hardware at full speed but difference is that kernel can run privileged, but applications can only run unprivileged

Privileges take longer, you want to run as few as possible

Now you have what is called a layered system. What's in hardware vs what's not. ABI is the **application binary interface** that divides ordinary instructions from system calls. If you take a look at how many C programs, stdio level: getchar putchar; syscall level read+write. Use **protected transfer of control** at each layer. x86 design has this property; may have intermediate layers between kernel and application so that the system would be immune for layers to hurt each other
0 kernel 1 virtual memory 2 files 3 application ← ring structured OS
Downside: every time you need to cross a boundary you're going to need to call a trap. It's not only the overhead of the trap, the problem is that the file system code will want to do interesting access in the VM. If these interfaces don't let them, then the file system will be slow.

Linux: monolithic kernel, within the kernel, every part trusts every other part, which leads to improved performance (downside: loss of reliability, extra levels of protection)
All you need is one bug in a rarely used I/O device that can access any part of kernel and mess up entire system

Trade-off between: security and performance or robustness and performance

So… now, when we're talking about organization by virtualization, we need to to think about ALU. If we're thinking about what user code can do, when we're talking about the ALU, it has full access. It can do adds and subtracts and multiplies however it wants. Next resource: registers, when we're doing virtualization, user code will want to deal with registers. We have full access to rbp rsp rax rip registers. But there are some registers that we don't want to allow full access to, the flags that we will allow partial access to. Other registers that will get no access. Next we have primary memory, does loads, stores, pushes, pops. Users get full access to user memory. There will be other parts of memory where we don't get full access: ISV, kernel. The line between the two memories depends on the OS, software controlled, whereas for the register separation, it's determined by the hardware.

I/O: applications have no access (!?) The only way to do I/O is to make system calls, the reason being that I/O tends to be the most complicated, has the most special cases so we allow all I/O to be privileged.

Figure out how to get this all to work
Process = program in execution in an isolated domain
Typically run on virtualizable processor, that can be set up in order to pretend you have control but all you have are syscalls

Way to create a process is by issuing a system call, pid_t fork(void); You can see what processes are being run with $ ps -ef.
int execvp(char const *file, char const **argv); allows you to run another command in another process

fork and execvp are kind of opposite to each other. Fork → the new process is the child = the parent except the child has different pid, also every process has a parent id, and the child's parent id will be the parent, and the parent's id will be the grandparent

File descriptors: each process has an array of file descriptors. After you do a fork, child and parent have different copies of the file descriptors. You can see the file descriptors as being pointers.
Accumulated execution times

File locks: child has none, parent has all
Pending signals: child won't, parent will get the signal

execvp is the opposite, only thing that has changed is that you have a new program; you're no longer running the shell. You have new data, new registers. Signal handlers are reset.

We're trying to take a complicated notion and we're trying to split it up into simple flexible pieces

Lecture 5 (25/01/17): Orthogonality, processes, and races

Before: We were talking about process model in Linux and how processes are created/destroyed
pid_t fork(void); clones current process, and returns -1 on failure, returns 0 in child, returns process id of child in parent

int execlp (char const *file, char *const *argv); never returns except returns -1 when it fails

Process destruction: int kill (pid_t p, int sig); (doesn't actually kill the process, it sends a signal to the process and the process might die or it might not)

_Noreturn void exit(int status); → your current process is exiting with this exit status; only bottom 8 bits matter) → doesn't actually kill the process either
There's a technical name for the process that's still there but doesn't run anymore: zombie

_Noreturn void _exit(int status); _Noreturn void _EXIT → does less cleanup before exiting
You should use _exit (or its synonym _Exit) to abort the child program when the exec fails, because in this situation, the child process may interfere with the parent process' external data (files) by calling its atexit handlers, calling its signal handlers, and/or flushing buffers.

Actual way of removing processes:
pid_t waitpid(pid_t pid, int *status (pointer to the victim's status will be placed when that process is finally destroyed), int options (WNOHANG, returns -1 if still running));

Why can parent only wait for children to die? You want there to be a clear line of responsibility for process destruction. You don't want processes to try to kill each other; management argument
You could wait for other processes which could be waiting for you, could check if there's a cycle but it'll jump complexity to O(N)

This procedure of destroying processes is sometimes called "reaping" the child process

```
bool printdate (void) {
        pid_t p = fork(); // look at the return value for fork to check whether we are parent/child
        if (p < 0) {          //always check for an error
                return false;
        }
        if (p == 0) {
                alarm(10); // ten seconds from now, it will send a signal
```

```
                int f = open("foo", O_WRONLY )      // we can execute any code we want in the child
                before we actually exec the child
                char *args[] = {"date", "-u", (char*)0};
                //execvp("/usr/bin/date", (char* []) {"date", "-u", NULL});
                execle("/usr/bin/date", args);
                _exit(23);   // _exit is used because we don't want to do cleanup code
                int status;
        }
        if (waitpid(p, &status, \0) != p)
                return false;
        return WIFEXITED(status) && WEXITSTATUS(status) == 0;
}
```

child process:
date.c

```
int main(int argc, char **argv) {
        clockgettime();
        gmtime_r();
        printf();
        exit(0);
        // alternatively it can return 0 beecause crt is run that has the line exit(main(2, args));
}
```

There another school of OS design that doesn't like this model of cleanup stuff between fork() and exec(), they would like to have something simpler? There is a system call: int posix_spawnvp (does both fork and exec packaged up in a single function.

```
int posix_spawnup (
        pid_t *restrict pid;  // says to the caller that you are not allowed to make this pointer an alias for
        char const *restrict file;          //any piece of memory already being used
        posix_spawn_file_actions_t const * file_acts;  // an object describing something you want done
        posix_spawn_attri const *restrict attrp;              to a file
        char *const *restrict argv; (array of arguments that you want child to have)
        char *const *restrict envp;
```

Microsoft uses this, it has better performance, but it's complicated! Microsoft's idea for building this: Processes are heavyweight things and you don't want to have one process fiddling with another process

Moving on to files:
File can be : 1. regular files (array of bytes), 2. directories (mapping from file name components to file attribute), 3. /dev/null ← always empty 4. /dev/full ← always full, 5. /dev/zero ← when you try to read you succeed and get 0's , when you try to write to it it goes away, 6. /dev/tty0 you get the keystrokes, 7. /dev/dsk/0 ← a file but if you write to it, you're writing to disk 8. pipes (communication pathway from one process to another)

What are some operations you can do with a file? You should be able to create them
int open(char const *, int flags, … (0666)); flags include O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, take one of these and you can OR in a whole bunch of other flags

There is an alternative model for file opening: creat(char const *f, mode_t m); == open(f, O_RDWR | O_CREAT | O_TRUNC, m)
We pass a bogus process id or an id that's not your children. Same for close if you pass an integer that doesn't correspond to any file. You should never just close, check the return value of close. Why are fds ints? Wouldn't it make more sense to use pointers? File pointers can't be used for IPC, and should be used for general purpose buffered I/O only (printf, fprintf, sprintf, scanf)

How to model OS resources in a user program? There are several different ways of doing so. You can use opaque handles, which are ints, or define a struct file* but never to explain what the type is (incomplete type) struct file;. Another possible way is using transparent pointers, struct tfile*, where struct tfile {char *name; }
Unix/Linux tends to like the opaque handles (special case of an opaque data type, declared to be a pointer to a data structure of some unspecified type) approach. Transparent pointers are more efficient but they make the system more brittle, because apps have access to system's internal structures.

stat (char const *file, struct stat * sp)

Process table has process descriptors and exit status, you can manipulate elements using system calls. File descriptor table can be seen as one part of the process descriptor table.

How does file redirection work?

```
pid_t p = fork();
if ( p == 0 ) {
        int fd = open ("f", O_RDONLY | O_CREAT | O_TRUNC, 0666);
        if (fd < 0) error();
        if (dup2(fd, 1) < 0) error();
        if (dup2(fd, 2) < 0) error();
        // dup2 manipulates the file descriptor table by copying an element to another spot
        close(fd);
}

int fd = open("file" , O_RDWR | O_CREAT | O_TRUNC, 0666);
if (0 <=fd ) {
        if (unlink("file") <=0 ) {
                ssize_t n = write(fd, "wyzzy", s);
        }
}
```

link, open, unlink, rename try to be as independent as possible from read, write, close
You can create files without names but downside: applications are harder to manage

Discussion 3

Lab 1B tips:
Have to close all unused fds before wait, otherwise there will be an error
Signal is not thread safe

C vs C++ programming: In C we need to end the C-string with null byte always

```
fun() {
   int[] array = {1,2,3};          // after this function is over, it's illegal to access this memory location
   return array;                   // again, you can try to access it but it will be unsafe
}                  // to prevent this, you should use dynamic allocation, but don't forget to free
```

Lab 1C tips: (has been modified recently) what you need to do actually is easy. Just follow TA's suggestion and we'll be okay. Save some time and write less. Just write the main points; that's enough

struct rusage ru;
You'll use getrusage(RUSAGE_SELF (use RUSAGE_CHILDREN to get info about children), &sec);
Important attributes: timeval ru_utime, and timeval ru_stime;
Remember, you always need to get the first two information. And print out in human-readable format.

Then you need to compare your implementation with that of bash …

Requirements
1. For the test cases they have to be detailed
2. Need to test bash or dash

Run command with simpsh, then convert it into standard command in bash or dash
Write script, and you need to add the command "times"
First line is the command
Second command says "times"

	user time	system time
parent process		
child process		

Wall time: program starts to run
t1 starts running
t2 ends running

t2-t1

For a program, if you sleep, the wall time will increase, but the user/system time will not increases
The time sleeping is not part of user/system time

You use time command to get all the information
For parent process, you use getrusage(RUSAGE_SELF, the structure);
getrusage(_CHILDREN, process);

commands to test
spec test case, test close and invalid file descriptor, test segmetnation fault and catch

Run bash and afterwards, run dash

vi .profile
export PATH=/usr/local/cs/bin:$PATH

We will cover midterm next time, so we will cover 2A right now.”During the exam, sometimes he may not know the answer. He just makes the problems so relax”

Lab 2A tips:
gcc -o lab2a lab2a.o -lpthread ← need to use the pthread library
In some cases, you will see -I/___/___ which means include the header
(1) /usr/include/lib, (2) “ “ (3)

First link the libraries, create threads, run functions, and then join threads at the end
Gcc atomic: provides locks

Lecture 6 (1/30/2017): Signals, scheduling and threads

What can go wrong with file descriptors?
We talked about comp org. and processes before, and some of the things that could go wrong are:
States and programs where dealing with file descriptors and you mess up.
You close(3) and then try to read(3, buf, 512), read returns -1, and sets errno to be EBADF, otherwise errno is set to jmp.

Suppose in between the time you close file descriptor and you read, you fd = open("foo", O_RDONLY). If open happens to return 3, read will work but read from the wrong file. (common mistake)
You may have a program that does a bunch of reads and then loops and reads more, if you never close, you have what is called a "fd leak" but your program starts out working, at least initially.
fd = open("/dev/usb", O_RDONLY), when device is no longer there, read(fd, …) returns -1 OR I/O error changes errno to be EIO.
read(fd, …) returns 0; at EOF // you don't know it doesn't work
read(fd, …) but user is away, so it waits, is frozen, which is good. The program that wants to run isn't accessing CPU resources. It "blocks."

There's a different class of problems: race conditions (behavior that depends on timing)
You have a potential bug lurking in your system, and you can't reproduce the bug
(cat a & cat b) > c, causes race condition based on which one gets access to the CPU faster, if you check c you'll see it'll have intermixed bytes from a and b
(cat > a & cat > b) < c, takes c and splits it into a and b in a nondeterministic way
But sometimes you might want race conditions, benign outcome
(apache1 & apache2) >> c, will contain timestamps from both of them which is the desired output

gzip foo.tar → reads foo.tar → writes foo.tar.gz → removes foo.tar
Suppose you're running this program and the user types ^C because the process is taking too long.
Remove foo.tar.gz XOR foo.tar if it's done writing foo.tar.gz, you want process to do either 1 or 2 but not neither or both

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

```
    int fdi = open("foo.tar", O_RDONLY);
    int fdo = open("foo.tar.gz", O_WRONLY |O_CREAT | O_TRUNC, 0666);
    int fdk = open("/dev/tty", O_RDONLY | O_NONBLOCK); // keyboard input
    call signal here
    do {
        // checks for keyboard input, but this won't work, will wait for user to type in something
        // so we add O_NONBLOCK
        if(read(fdk, &c , 1) == 1 && c == 3){ // 3 is the letter for interrupt
            unlink("foo.tar.gz");
            return 1;
        }
        n = read(fdi, ….);                    // Might work, but is not optimal solution
        compress;
        write(fdo, ….);
    } while(0 < n);
    but actually we should call signal here
    close(fd0);
    unlink("foo.tar");
}
```

This approach is called "polling", every time you run a certain number of loops, you ask if ^C has been pressed. Downside is that you need to keep modifying the program every time it may be necessary (any big loop)
Better solution: signal, way to deal with unusual events, and main part of program doesn't have to worry. In signal handler, don't have to worry about signals or polling. Goals: might be dealing with uncooperative programs (infinite loop), invalid programs (illegal instructions: SIGILL, floating-point exception, division by zero, invalid address could be SIGSEGV, or SIGBUS)
^C internally is an integer constant SIGINT
Used where we don't to disturb computation or keep polling (which is done in a loop)
You can also use signals for I/O errors (SIGIO, SIGPIPE). SIGPIPE sent when you try to write to a pipe, but there are no reads (Example: a | b, where b exits and a does a write after, write returns -1 and keeps trying again, but this is resolved when a signal is sent)
Other signals: SIGCHLD – child died (signal handler to call waitpid immediately right after a child dies to find out what happened to the child; without this, you have to use waitpid alone which uses polling)
User signals other than ^C, SIGKILL(9), can't catch this signal (some signals are uncatchable because otherwise uncooperative programs may catch them): changes the process into a zombie, SIGSTOP (stops the process), SIGTSTP (^Z) , SIGCONT (continues)
SIGHUP, sent to user when program goes away (user logs out), SIGALRM, alarm clock, can exit whenever.
Downsides of signals: change to processes' abstract machine, between any pair of instructions, a signal can arrive → signal handler function is executed (function is specified by the application)
Writing a solution with signal for the gzip race condition problem:

```
signal(SIGINT, handle_control_c);
void handle_control_c (int sig) {
    pthread_sigmask (block SIGINT);  // ignore signal
    unlink("foo.tar.gz");
    _exit(126);
}
```

SIG_IGN, ignores signal and doesn't run handler code. Useful when code is done writing to the new .tar.gz file, SIG_IGN is called and .tar.gz isn't called anymore. Instead, .tar is removed
signal(SIGINT, SIG_IGN); → critical part, "section" of program → signal(SIGINT, handle_control_c);
pthread_sigmask(block SIGINT) (what you were blocking) □ critical part of program □
pthread_sigmask (old set);
pthread_sigmask is used instead when we want to guarantee that we unlink at most once. We add this this within a signal handler, and this is so common that it is the default behavior.
Signal handlers can arrive between any machine instructions so you should only write coherent "super instructions" inside. Suppose: you want sig_handler to output a message.

```
void handle_signal(int sig) {
   FILE *f = fopen("file", "w");  // fopen calls malloc → inspects and modifies the heap data structure
   fprintf(f, "caught signal %d\n", sig);
   fclose(f);
}
```

malloc shouldn't call malloc again, when you call signal handler, malloc is called. Inside signal handler, malloc shouldn’t be called again. Signal handlers should only call async-signal-safe functions, which are low-level safe functions that don't inspect local memory

```
static sig_atomic_t volatile interrupted;
void handle_control_c (int sig) {interrupted = 1};
for (int i =0 ; I < 100000000; i++ ) {
  if(interrupted) break; → POLLING, you can have polling on top of signal handlers
}
```

Signals
Pros: can manage processes better, fixed some robustness & performance issues
Cons: processes are less isolated, can be signaled at any time → race conditions, notoriously buggy

Less isolation than process without async calls, we'd like a model where the program is in control of its own instructions (more predictable instruction pointer)
We use threads: gets rid of all memory isolation, all threads can all access the same memory. For performance in doing things like signals, pipes.
Threads in a process share as much as possible. Share memory (address space), code, file descriptors, owner, pid, ppid, heap
They don't share instruction pointers, registers, stack, threaded, errno

8-core CPU, 1000 threads, what to do? Resource allocation problem
CPU runs a scheduler which schedules the CPU! We take our 8-core CPU and we say core 0 runs the scheduler. Cores 1-7 run actual programs. This doesn’t work for embedded systems with max 1-2 CPU cores. Scheduler isn't run on a core but runs in the gaps between programs. CPU will run thread one for a while, and one it stops, it'll execute scheduler code. Mechanism (what set of instructions actually gets executed) and policy issues? We'll use sys calls; whenever a thread wants to give up control, it uses sys calls to determine scheduling.

Lecture 7 (02/01/2017): Scheduling algorithms

#include <pthread.h> /usr/included/pthread.h
Int pthread_create(pthread_t * thread, pthread_attr_t const * attr, void (*start) (void *), void *arg);
(*start) runs at the start of the function, arg is the argument passed in

void pthread_exit (void *value);
void pthread_join (pthread_t thr, void ** value) // waits for thread like waitpid waits for process
void pthread_kill (pthread_t, int); // kills the process, might send a signal to the signal handler(?)

Process is a bundle of threads
Cooperative threads: a way that threads can cooperate (give up CPU when not needed). When you have more threads than CPUs, one thread might want to run a CPU-intensive process and other threads will have to wait. Solution: threads yield the CPU to other threads (for fairness) At every system call, CPU will not be in the kernel; when kernel decides to return back to the user, it chooses which non-running thread to return to.
What can go wrong with cooperative threads? Infinite loops without system calls. Fix: every time your program could do more than 100 ms worth of computation, make sure you have a system call in the loop. What's a system call that does nothing? Nothing happens but behind the scenes, gives other threads the opportunity to run.
#include <sched.h>
int sched_yield(void); // in C

Maybe there's a way of still letting people write apps but not force them to add a system call to every loop. A better way, let the OS do it for you. Hardware timer interrupt: there's a circuit on the motherboard connected to the CPU via bus, and every 10 ms, CPU traps. Traps to trap interrupt vector and kernel decides which thread to return to. Before returning, save registers as a process table entry in the process table. Downside of clock interrupt: hurts computation with having to save registers and switch threads every interval, code has to worry about what other threads are doing all the time because the current thread could be affected by their behavior
For infinite loops: you can have a timeout, alarm(100); One major problem with timeout, you can't tell what number to pick. Alternative: user-initiated interrupt (^C), or don't use loops (or keep an upper bound for # of loops)

Cooperative threads also include: cooperating access to shared memory and I/O devices. You don't want two threads to run commands to the disk controller at the same time.
3 basic techniques:

1. Busy waiting: use this sort of technique in boot code. Look at disk controller, wait for it to become ready. While it's not ready, loop

   while (!ready()) continue;
2. Polling: more computationally intensive, don't want to run sched_yield() a ton

   while (!ready()) sched_yield();
   For very fast devices / short waits, fast operations use busy waiting. For slower operations, polling is better.
3. Blocking: probably the best for longer waits

   while (!ready(dev)) wait_for(dev);

What this means: when a thread is not running, OS will know because in process table entry it will say thread is waiting for device to be ready. Another column holds the thread state (running, runnable, blocked)
Which thread to run next? Issue of policy, not mechanism. Scheduling problem – you have policies and dispatch mechanisms. What sort of policy should we have? e.g. airline scheduling is a major problem; lots of CPU cycles squeezing out the best airline schedule given all the constraints. Factory-floor scheduling.
Simple subset; take the best ideas from conventional scheduling and take the best ones that are cheap

Scheduling scale: how long term are we going to have to worry about what our scheduler is doing?
Long-term: which processes are admitted to the system? Called admission control
Medium-term: which processes are in RAM? If you have too many programs trying to execute simultaneously, the amount of RAM they asked for might be more than physical RAM available, some processes will be stored on disk, frozen.
Short-term: which threads have a CPU? Preemptive – clock interrupt, cooperative – yield()

Priority queues (multilevel):
- System thread queue (high priority)
- Interactive threads (associated with user programs, next priority)
- Batch threads (payroll, data administration; low priority)
- Student thread (lowest priority :'( )

Real-time schedulers: used when you have apps that have real-time constraints
Hard real time (nuclear power): you cannot miss a deadline, equivalent to the system dumping core, predictability trumps performance (e.g. caches disabled), prefer polling to interrupts
Soft real time (video playback): some deadlines can be missed if there's too much work, for rendering each frame you have 1/30 of a deadline. Look at all threads running, each thread has an associated deadline. Take the earliest deadline first → simple scheduling policy. Alternative: rate-monotonic scheduling: we have lots of thread, 1/N of the CPU to each thread, you look at the current CPU and if it can't have it done within allotted time, you throw it away (assign higher priorities to threads that need to run frequently / more CPU)

Every timeout / scheduling policy has its downsides.
Metrics to determine how well your scheduling policy is working: order arrives → execute order (begin) → 1st output …build more → done
Wait time: time from order arrival to executing order
Response time: time from order arrival to 1st output
Turnaround time: time from order arrival to done
// Basic idea of latency: how long do you have to wait after done to next order execution
Context switching time: overhead between end of first order and start of next order execution
Average (wait response turnaround) time, also can calculate variance in ("" "" "") time
Throughput: how many jobs of useful work can we do per unit of time? == utilization (e.g. 90% utilized)
Achieved when only 10% of the time it is either waiting for the next order to arrive or execute

Fairness and utilization are competing goals. Fairness wants low variance but throughput wants low average times.
Simple scheduling policy: first come first served (FCFS); priority = arrival time (no yielding!)

Jobs	Arrival time	Run time	Wait time (FCFS)	Turnaround time (FCFS)
A	0	5	0 seconds	5 seconds
B	1	2	4 seconds + δ	6 seconds + δ
C	2	9	5 seconds + 2 δ	14 seconds + 2 δ
D	3	4	13 seconds+ 3 δ	17 seconds + 3 δ

AAAAABBCCCCCCCCCDDDD
Avg. wait time: 5.5 seconds, avg. turnaround time: 10.5 seconds
Recall: utilization: what fraction of CPU time is spent doing useful work?
Utilization = 20 / 20 + 3 δ

Convoy effect: the slowest truck will cause the remaining trucks to wait, bottleneck; long-running job delays later jobs
Shortest job first: AAAAABBDDDDCCCCCCCCC, wait time changes
C's wait-time: 9 + 2 δ
D's wait-time: 4 + 3 δ
Avg. wait time becomes 4.25 + 1.5 δ, better turnaround time
Shortest job first is always better, but we'll discuss its problems next time

Discussion 4 (02/03/2017)

Lab 2A involves scp, compare and swap? Spin-lock
_sync_lock_test_and_set
Know how different locks work, mutex

Question 1: our own code can try to deal with the special case and deal with it?
The answer is yes

Question 2: describe all possible outcomes

Question 3: pro: no overhead of switching to kernel mode, con: not safe because it's not read only memory

Question 4: atomic size is really large, and takes up a lot of memory

Question 6: it doesn't make sense, longer tasks will not be processed

Question 7: tip: if the question says, "if your answer depends on the implementation" then say it depends on the implementation. Answer: depends on the scheduler. Everyone will have the opportunity to get the lock.

Question 9:
Tip: the question about the locks will be the hardest one.
Tip: talk about key points and combine w/ questions
Tip: questions will contain hints

Lecture 8 (02/06/17): Consistency; critical sections

Scheduling – we covered FCFS (more fair), SJF (better avg. wait time)
Fairness – no job waits forever; a boolean, a system is either "fair" or not
We can try to combine FCFS and SJF: use preemption (based on a quantum – 10 ms in Linux, divide up CPU time into individual units)
e.g. robin robin (RR) scheduling: avg. wait time: 0 s, avg turnaround time: 12.75 s, for an example with jobs ABCD arriving 0, 1, 2, 3 and taking up run times 5, 2, 9, 4
Overhead of context switching might dominate performance

For round robin, bunch of jobs waiting for their turn, new 1ms jobs show up in a stream, then old jobs waiting will starve. Fix: new jobs are placed at the end of queue

Priority scheduling: user-assigned (static) priority, system-assigned (dynamic) priority
On Linux, there's a user-assigned priority called "niceness", user can call $nice in the shell.
$nice make linux ← will run command with niceness 5 greater than it normally is
niceness ranges from -19 to 19. Program is set to 0 by default
$nice -n 3 make linux ← adds 3 to "niceness"
nice: determines how late in the queue a new job will be placed

---

Synchronization: threads' biggest problem!
Default behavior of a thread is unsynchronized! All threads have access to all memory and can cause chaos by default
Yet with multi-threaded applications it usually works anyway 99.99% of time

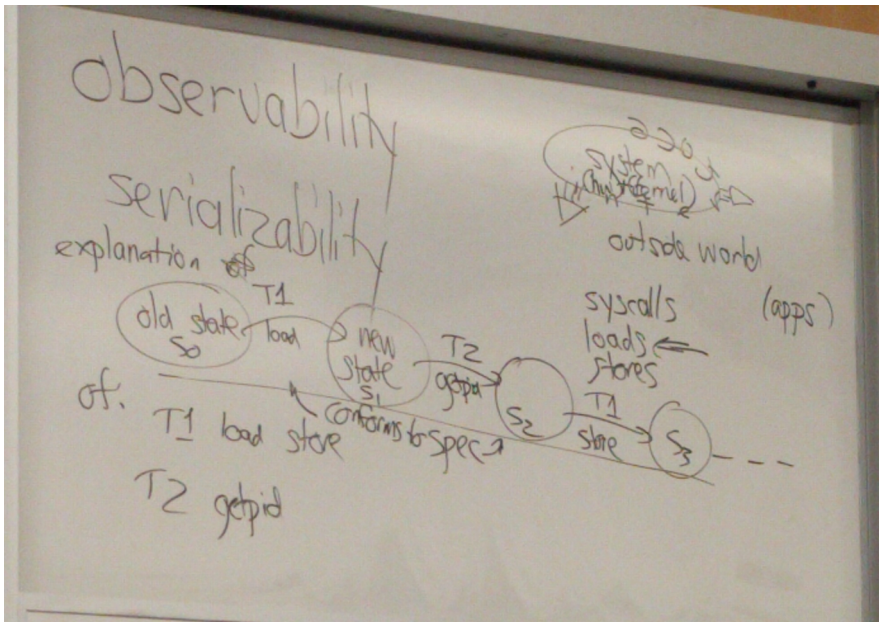
When we were using multiple threads we are coordinating actions in a shared address space where two threads can affect the same chunk of memory at the same time. We have to maintain data consistency and do so efficiently in terms of utilization and latency (sometimes these goals will compete with each other but we'll try to find the best of both worlds). We want this coordination to be clear and simple too.

Problem: race conditions – will only matter if they result in observable outcomes that are bad

Observability says: observers from the outside world (apps) are looking at system (kernel + hardware). Apps are making syscalls, loads, stores (look at return values, values that have been loaded, to make observations)

Serializability: old state → new state, difference is that a thread has taken an action. You should come up with a list of steps the threads have taken and create an explanation of what each thread on its own could observe. e.g. T1 did a load, a store, T2 getpid
List of sequential series of events can be produced even if happening in parallel

Isolation: two different actions that cannot possibly affect each other, you don't have to care the order the two are run in
Atomicity: actions X and Y are related but when trying to do them simultaneously, system arranges for X to finish before Y starts or vice versa




```
unsigned long balance;          //10000
void deposit (unsigned long amt){
        balance += amount;
        // bug: overflow possibility
}
bool withdraw(unsigned long amt){
        if (amt <= balance){
           balance -= amt;
           return true;
        }
        return false;
}
```

```
T1: deposit(1000)
mov    balance, %rax
add    $1000,%rax
movq   %rax, balance

T2: withdraw(2000)
movq   balance, %rax
cmpq   $2000, %rax
jg      fail
subq   $2000, %rax
movq   %rax, balance
```

Only a few instructions, most of the time will be fine but could have bad implications for specific cases

Why don't you just declare threads to be synchronized (if writing in Java)?

```
synchronized void deposit...{}          // that's fine but how does Java do it?
synchronized void withdraw… {}          // makes these functions atomic
```

Two critical sections: areas of code that are supposed to be executed indivisibly
At most, one thread's ip is in a critical section
Don't want critical sections to be too large (lose performance and parallelism due to bottlenecking)
Don't want "" to be too small (or you don't fix the problem)

To find a <u>minimal critical section</u>: look for writes to shared state → look for dependent reads, use them to grow the critical section → look for reads to large objects (> 8 variables)

```
struct {
   int a, b, c, d, e, f;
} s;
```

read s; // in between the times you read s.a and s.b someone could write s.a.
You end up reading a struct that is a mix of a set of old and new values

Footnote: watch out for small objects too!

Atomic operations are allowed to have “benevolent” side-effects as long as you can’t observe them.
e.g. update a cache that is invisible to user

```
bool transfer (unsigned long amt, int from, int to) {
        // critical section for two accounts at once
}

bool audit_all_accounts () {
        // needs to inspect account while there's a transaction going on
        // have to lock out all other activity in the bank
}
```

Enforcement of critical sections: simple version: 1 CPU syscalls for all actions, no preemption, no hardware traps. Every syscall is a critical section → done

Temporarily mask traps in front of a critical section, unmask afterwards
p_thread_sigmask, p_thread_sigmask

For multiple CPUs, there are more problems
1. mutual exclusion: no two threads in critical section, any thread in c.s. will exit quickly
2. bounded wait: just as important; will prevent starvation (otherwise thread may stay in critical section forever)

```
struct pipe {
        char buf[1024];
        size_t r, w;
}

bool write (struct pipe *p, char c) {
        if(p->r==p→w==1024)  return false;          // after midterm , we'll see mutexes
        p→buf[p→w++%1024] = c;                      // how to effectively synchronize this
        return true;
}

int read (struct pipe *p){
        if (p->r==p->w) return -1;
        return p→buf[p→r++%1024];
}
```

Lecture 9 (13/02/17): Synchronization; deadlock

```
struct pipe {
   unsigned char buf[N];
   size_t r, w;
}
bool write (                          // you can disable interrupts to ensure atomicity
   disable_interrupts();
   if (p -> w – p->r == N) {          // if number of bytes written – number of bytes read
      enable_interrupts()             // PROBLEM: enable_interrupts() hadn't been run, we add it
      return 0;
   }                                  // cooperative multitasking: we execute no instructions when
   p -> buf[p -> w++ % N] = 0;        // entering or exiting critical section and ensure CPU is
   enable_interrupts();               // not given up in the middle of a crit section
   return 1;
}
int readc (struct pipe* p) {
   disable_interrupts();
   if (p->w == p->r ) return EOF;            either returns EOF or an int from 0 to 255
   int x =  p -> buf[ p->r++ % N];
   enable_interrupts();
   return x;
}
```

Multiple threads all acting on the same pipe, acting on the same data structure, pipes are going to start running into each other
We should implement a critical section around readc or writec in order to fix these race conditions, assume simplest case is uniprocessor. Only problem is interrupts, so disable them in critical sections
Multiple cores or CPUs: we need a better technique, Eggert will give us an abstraction on how to solve this problem

Comparisons of lock implementation:
System calls: easy to implement, but slow due to context switching (kernel will trap)
Regular code: still race conditions, but faster than system call
Hardware support: caching (fast, but need **cache cohererence**)

```
typedef  ______ mutex_t;
void lock (mutex_t *m);       (precondition: you can't currently own the lock)
void unlock (mutex_t *m);    (precondition: you have to own the lock)
```

Adjustments to code with mutex locks:

```
#define N 1024
mutex_t m;       // this is course-grained locking, imagine only one mutex in entire program

bool write (
   disable_interrupts();
   lock(&(p->m));
   if (p -> w – p->r == N) {          // if number of bytes written – number of bytes read
      unlock(&(p->m));                // don’t forget to unlock
      enable_interrupts()             // PROBLEM: enable_interrupts() hadn't been run, we add it
      return 0;
```

```
    }
    p -> buf[p -> w++ % N] = 0;
    unlock(&(p->m));
    enable_interrupts();
    return 1;
}
int readc (struct pipe* p) {
    disable_interrupts();
    lock(&m);
    if (p->w == p->r ) unlock, return EOF;          either returns EOF or an int from 0 to 255
    int x =  p -> buf[ p->r++ % N];
    unlock(&m);
    enable_interrupts();
    return x;
}
```

Coarse-grained locking: you create a bottle-neck. We'd rather have finer-grained locking. Will encourage parallelism and better performance but code is harder to understand.

```
struct pipe {
    unsigned char buf[N];
    size_t r, w;
    mutex_t m;   // finer-grained locking, each pipe has its own mutex
}
void lock(int *m) {   while(*m) continue;  *m = 1;   }
void unlock(int *m) {   *m = 0;   }
```

Race condition, both threads gain access to the lock at the same time. We ask for help from Intel
Intel says: loads and stores are atomic for words only (1, 2, 4, 8 ; have to be aligned). Address is a multiple of size

lock incl x // adds one to x atomically, CPU talks to the bus and RAM, no caching

More expensive instruction than normal incrementing but we get the atomicity that we want
Alternatively:

xchgl %eax, (%rbx)

```
void lock_incl(int *p) {
    (*p)++;
    //asm(incl lock % *p)          // actual instruction performed, asm is an atomic instruction
}
void xchg(int *p, int v) {          // exchanges data in memory location with v
    int ov = *p;
    *p = v;
    return ov;
}

void lock(int *m) {
    while(xchg(m, 1)) continue;     // tells other CPUs not to touch a specific section of memory
}
void unlock (int *m) {
    *m = 0;
```

}

Another type of lock:
lock (&m);
x = f(x); // pure function, so we can alter x rather than just x = x+1;
unlock (&m);
bool cas(int* p, int old, int new) {
  if (*p == old) {
    *p = new;
    return true;
  }
  return false;
}
```
while (1) { int old = x; int new = f(x); if (cas(&x,old,new)) break; }
```

Multiple methods at the hardware level of synchronization
New type of lock, not in textbook: hardware lock elision (Intel Haswell)

```
        movl    $1,   %eax
try:    xacquire lock xchgl %eax, (%rbx)
        cmpl    $0,   %eax
        jnz     try
        ret
```

// exchange our atomic lock instruction, except it doesn't sent that everyone get out signal
// just keeps going

```
unlock: xrelease movl $0, (%rbx)
          ret
```

If someone else used a lock, **xrelease** backs up to the previous acquire, performs time travel

Problem with using a lock to unlock and lock a pipe:
We need to pull in order to do reads and writes. We need to avoid polling, blocking mutexes (you try to get the lock and if you can't, you go to sleep) If not wait for lock to become available
Can be implemented atop ordinary spin locks

We declare new struct: the **blocking mutex**
The blocking mutex builds upon the simple spinlock by using a spinlock to access the mutex, and then checking the **bool locked** field check if the mutex is locked or not. This can allow for processes that would have otherwise been spinning to instead *yield()* or *schedule()*.

```
typedef struct bmutex {
   mutex_t m;
   bool locked;
   struct process *waiting;     // linked list of waiting processes
   struct process **waiting_tail;
} bmutex_t;

struct process {
   bool blocked;
   struct process *next;
}
```

```
void acquire ( bmutex_t *b) {
   for ( ; ; ) {
      lock(&b -> m);
      if(!b -> locked) {
         b->locked =1;
         unlock(&b -> m);
        return;
      }
      self->next-> 0;
      *(p->waiting_tail) = self;
      b->waiting_tail = &self->next;
      unlock(&b -> m);
      self->blocked = true;
      yield();
   }
}

curproc->next = b->waiter;
b->waiter = curproc;
curproc->locked = true;
unblock(&b->l);
yield();
return;

void release(bmutex_t *b) {
   lock(&b -> m);
   b->locked = 0;
   proc *p = b->waiting;
   if(p) {
     p->blocked = false;
     b->waiting = p->next;
     if( !b->waiting) {
          b->waiting_tail = &b->waiting;
      }
    }
    unlock(&b->m);
}

acquire(&p->m);
if(p->w – p->r  ==  N) {
   release(&p -> m);
   yield( "until p->w – p->r != N ");
}
```

Condition variable: blocking mutex with a logical condition where the app decides condition, true or false

! Call yield() when scheduling spin locks so that they don’t take up whole CPU slices

Lecture 10 (02/15/17): File system performance

Condition Variable (contd.)

API conduar (represents specific boolean expression, e.g. the pipe is full)

- condition that they stand for up to the application

```
sample primitive;
void wait(conduar *c, bmutex_t *b) {
        //precondition: b is already acquired
        //releases b, blocks until some other process notifies
        //reacquires b, then returns


        notify_cond(conduar *c); //notifies one
        broadcast_cond(conduar *c) // notifies all waiting
}


struct pipe {
char buf [8192];
size_t w;
bmutex_t b;
conduar_t pipe_is_full;
conduar_t pipe_is_empty;
}


void write (struct pipe *p, char c){
acquire (&p->b);
while (p->w - p->r == 8192){
wait(&p->pipe_is_full, &p->b);
        }
        p->buf[p->w++%8192] = c;
        release (&p->b);
        notify_control(&p->pipe_is_empty);
}
```

Semaphore

- blocking mutex, but with an int (not a boolean)
    - int represents the number of available resources that can be used simultaneously
- e.g. starts with 10; meaning that <= 10 simultaneous users
- a blocking mutex is essentially a semaphore of 1 (binary semaphore)
- Functions:
    - P
        - “prolaag”
        - down, acquire
        - P(&s);
    - V

- “verloog”
- up, release

Deadlock

e.g. cat ---> read (0,&buf,512); write (1,&buf,512);
What would make cat very efficient would be a new system call: copy
copy(0,1,512) //instead of putting anything in the buffer (RAM), write directly from file descriptor 0 to 1

Problems with this proposed system call

- read will lock(&fd[0].m), and then unlock(&fd[0].m)
- write will lock(&fd[1].m), and then unlock (&fd[1].m)
- implementation of copy will have to:
    - lock(&fd[0].m)
        - copy(1,0,1024) //will lock fd[1], and now cause a deadlock in both copies
    - lock(&fd[1].m)
    - do the copy
    - unlock(&fd[0].m)
    - unlock (&fd[1].m)

This will not work:

- Say that at the point directly in between the two locks, another copy come in
- one solution: lock ordering &l < &m
    - acquire locks in order
- another solution: if a lock is not immediately available, unlock <u>everything</u>, and try again

Deadlock is race condition (won’t always happen: depending on timing of a system)
Dependent on 4 conditions

- circular wait (one process holding x waiting for y, another process holding y waiting for x)
- mutual exclusion
- no preemption of locks (forceful break of locks in a situation when we don’t have lock)
- hold + wait (ability to hold one lock while waiting for another)

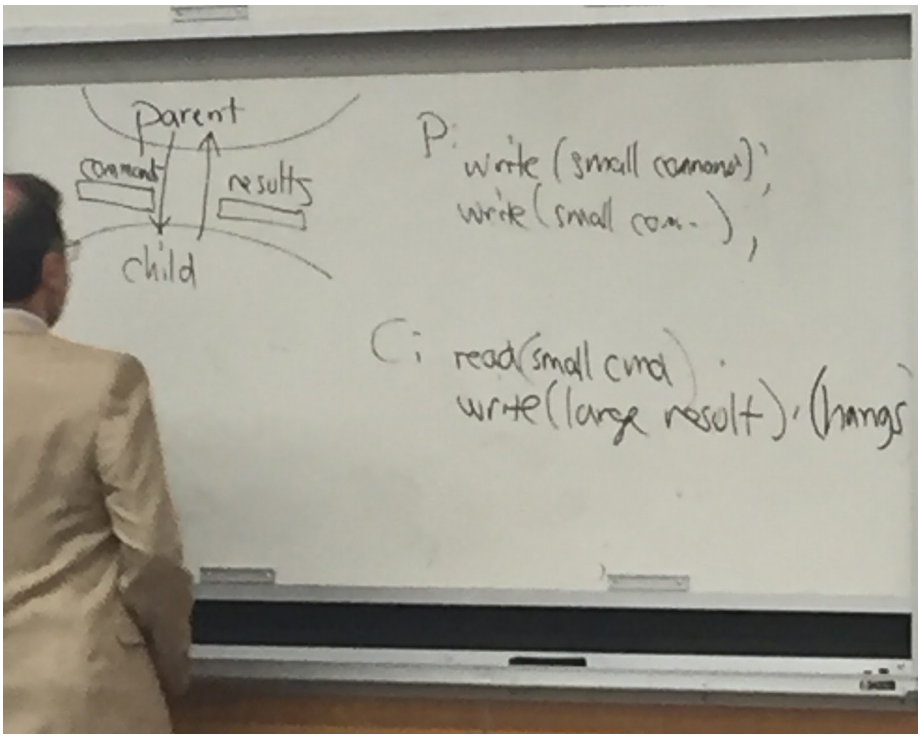
In Linux, there is dynamic detection of circularities

- will fail the system call when a circular loop is detected
- Failure message EWOULD_DEADLOCK
- this will not prevent with deadlock with spin locks (they are too simple, don’t know other processes waiting for lock)
    - so then make the detection time based (like any way to break infinite loop, if taking too long kill it)

More Complex Synchronization Problems

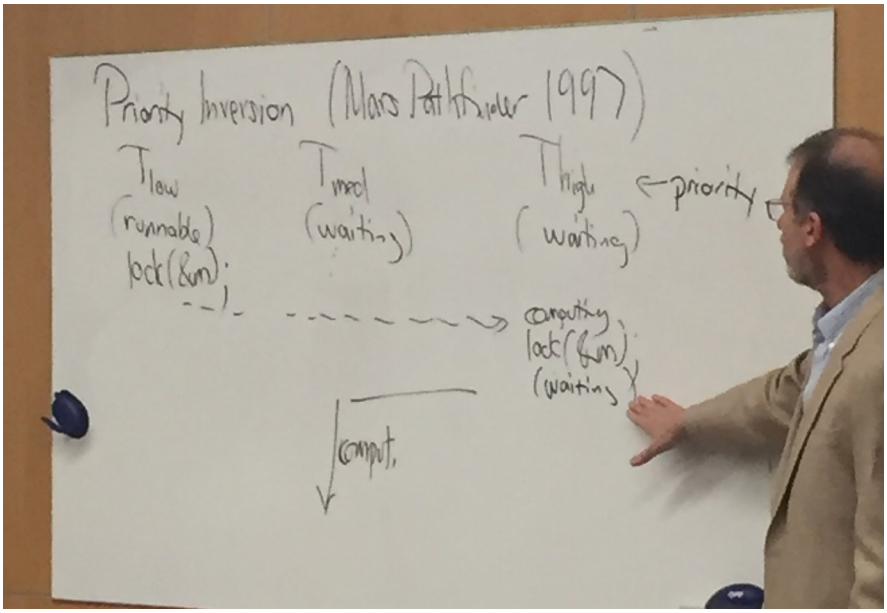
- example: parent forks, sends commands to child and retrieves results from child
    - P: write (small command); write(small command);

- C: read (small command)
- write(large result) (HANGS if buffer is filled)
- now parent also hangs



Priority Inversion

Mars Pathfinder 1997
Combination of synchronization + priorities:



Result: highest priority process will never be run

- solution, preemption, or check which process took lock

Event-Driven Programming

```
for (;;){
e = get_next_event();
handle_event(e); //handler must finish quickly
}
```

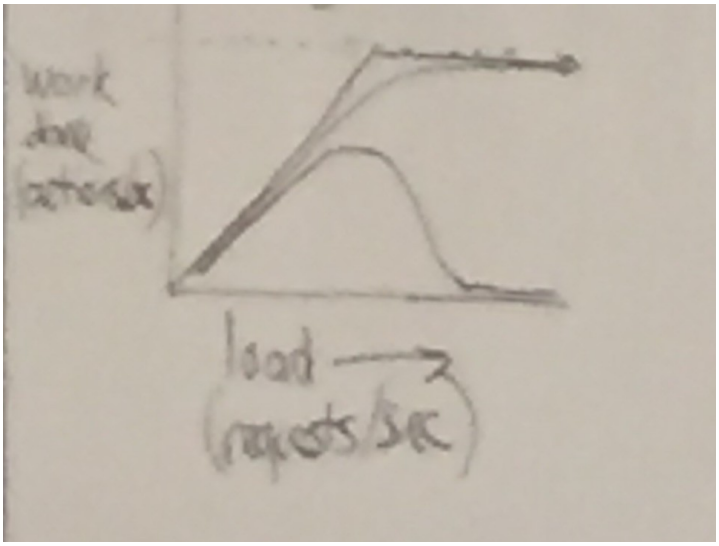
In a sense, the inverse of threads
Threads: take code, sprinkle in locks
Event: take code, partition complex portions into separate pieces

Problem: gives up on multi-core systems
- new school of thought: cores perform independent functions

Another problem: doesn't entirely remove race condition situations

Livelock

- CPU always doing something, but no meaningful work is being done
- receive livelock
    - scenario: get request, start handling it, generate subevents, get another request
    - eventually, overhead of subevents will make efficiency shite
    -
    - To fix: discard events more enthusiastically (e.g. disable interrupts when system is loaded



File System

- 120 PB (120,000 TB)
- 200,000 drives (each 600 GB)
- GPFS - general parallel file system

striping
- to access a single file as quickly as possible, put bits of file on different disks
    - advantage: can read/write each section concurrently

distributed metadata
- e.g. ls -l file

- -rw-r-- eggert eggert 2017-02-20 file
    - all this additional info is metadata (including data location information)
    - this metadata is also put on disk
    - if metadata is all stored on one disk, then there may be delays
    - instead, multiple copies of file's metadata throughout disks

efficient directory indexing

- if we had an array of 20,000,000 directory entries, each of which is comprised of name and additional info, this will be shit in terms of time
- better data structure: unary tree (idk look it up)

distributed locking

- no single node "owns" an object

partition awareness

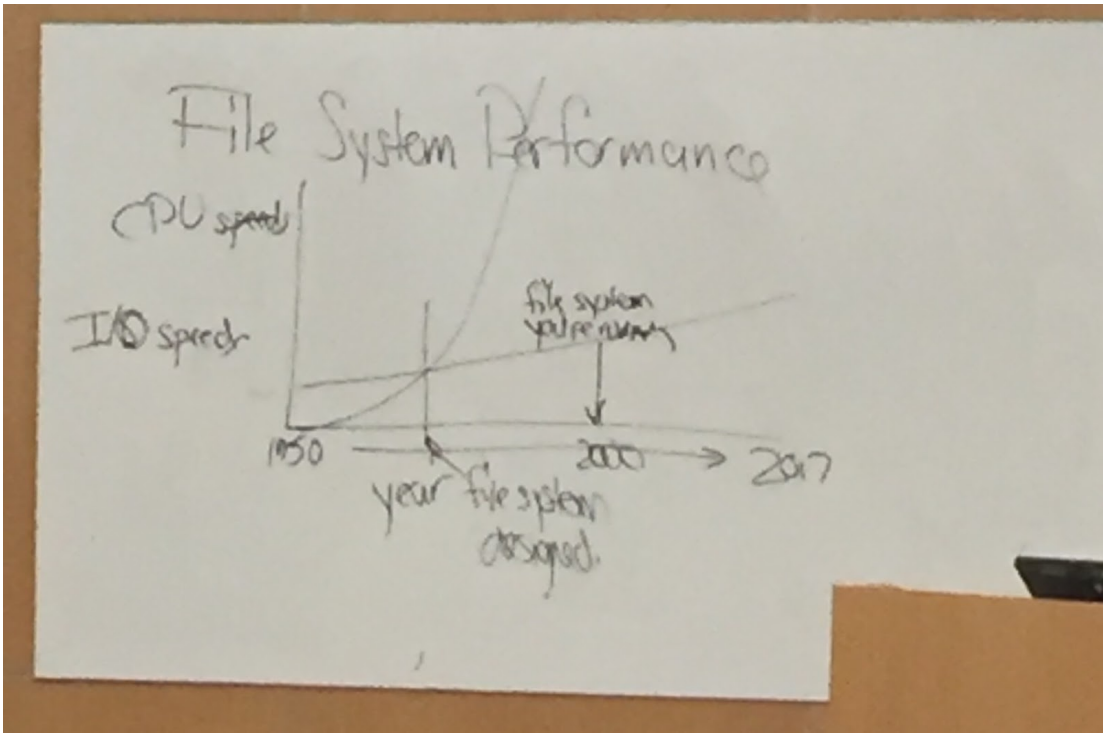
-

File system stays up during maintenance

-fsck (file system consistency check)

- done while booting

File System Performance



Lecture 11 (02/22/17): File system implementation

Last time: file systems and performance
Today: file system organization, next time: file system robustness; we want access to data to be fast but we don't want to lose our data when it crashes
Organization tries to bridge these two opposing goals. To make a reliable file system, make a slow one. We want as much robustness as we can without sacrificing performance.
Taking a look at performance: devices we have to deal with
What are the metrics? How can you tell if file system is performing well?

4. Bytes/sec I/O (throughput) how much data are you putting in and getting out of the system?
5. Utilization: fraction of total capacity you are using (0 to 1)
6. Latency (when you want data from the file system, how fast it can respond to your request?)
    a. e.g. 10 ms delay

We can have high latency to get high utilization but hard to get latency AND utilization. To get low latency, don't move the disk arm to faraway places, keeps bytes/sec up.
File disk devices include:

7. Hard disks: Seagate Barricade: 2 TB
    a. 300 Mb/s external controller rate at which disk controller can send data over the bus to the CPU, how fast data is flowing through spin head
    b. 95 Mb/s sustained data rate how much data you can actually get over the drive over and over again
    c. 32 MB cache
    d. 55 mm avg. latency (random access)
    e. 5900 RPM spindle speed ~ 98.33 Hz
    f. 512 bytes per sector
    g. 10^{-14} nonrecoverable read errors/bit (every time you read data you read bits, what's the probability that bit wasn't read because disk drive failed internally)
        i. $8*10^{-5}$, for 1 GB
    h. 0.32% AFR (annualized failure rate; probability if you use drive 24/7, that it will stop working entirely during the year)

    And less importantly:
    i. 50000 contact starts and stops (frequency at which disk head is lifted up; after this, the drive is dead)
    j. 6.8 W avg. operations
    k. 5.5 W idle, if disk head is just spinning
    l. 2 A startup current
    m. 70g operating shock
    n. 300g nonoperating (disk powered off is more likely to survive a drop)
    o. 2.5 bels (acoustic output)

8. Flash drive: 60 GB
    a. 3 Gb/s
    b. 1000g shock
    c. 2 W operating
    d. 2 Gb/s I/O read/write
    e. 50000 IOPS (I/Os per second, assuming 4k byte writes at a time) throughput figure
        i. 50k * 4k = 200000000 B/s = 200 MB/s transfer rate ~ 1.6 Gb/s (why so fast)

ii. Flash drives tend to be higher performance, latency is a lot better than disk drives
iii. This wears out unlike HDDs

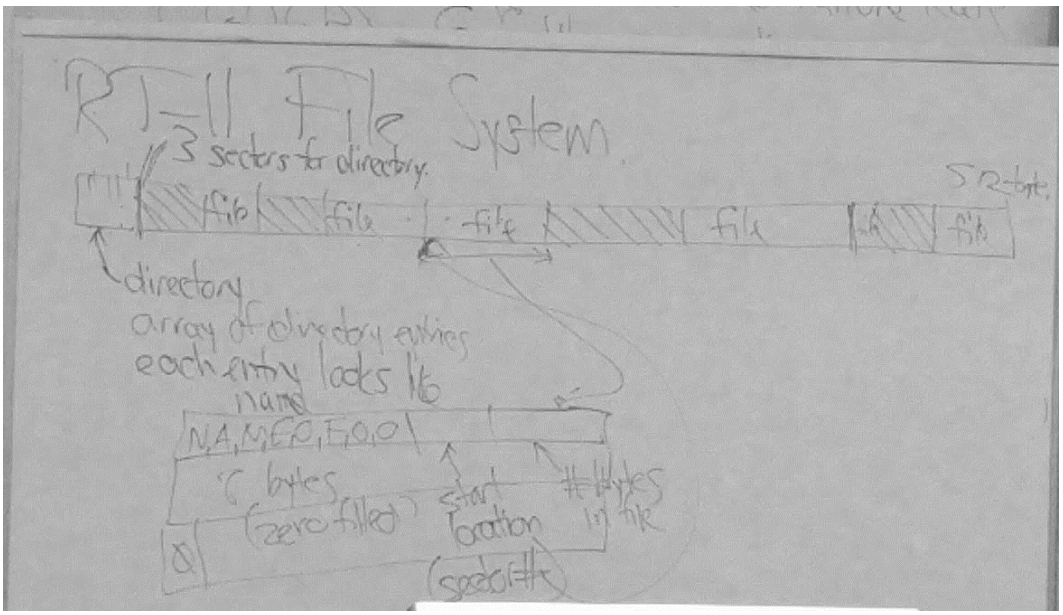
What we have to do is design a file system? There are techniques for performance on secondary storage.

9. Batching: instead of reading data one sector at a time, you read several sectors all at once
    a. Read/write contiguous sectors on a disk low level batching
    b. getchar(), reads one byte from stdin, read(0, buf, 8192) high level batching
10. Dallying: not responding immediately to requests but rather collect, then inspect and look for opportunities for batching
    a. aims to increase throughput at the cost of latency
11. Prefetching: grab data from disk/flash and stick it in RAM before it is needed
    a. when well executed, improves both latency and throughput
    b. subset of speculation
12. Speculation: try to guess what the application will do next, and based on that, start actions on I/O (move disk arm)

When does prefetching hurt performance? When you guess wrong. You want to guess right often enough that it pays off. Assume **locality of reference**, assume applications will stick around in the same neighborhood

13. Spatial locality: accessing i means likely to access i+1 block
14. Temporal locality: accessing block at time t means likely to access the block at time t+1

Start discussion on file system organization. Eggert designed system similar to current RT-11 (real-time) file system as un undergrad, didn't know it had been in use already
Put files on file system, each file will be a contiguous array of blocks
There will be gaps between files, and a map that keeps track of where all the files are
Directory tells you names and location in the file system
It is an array of directory entries, each entry looks like:
Name 8 bytes, zero-filled, start location of file, number of bytes in file
Eggert's file system had a limit of 3 sectors for directory, but RT-11 system file system keeps it as variable



Deleting the file, just change the first char of the name into a null byte (not a problem)
In this file system, files are contiguous (implications: simple, high throughput, predictable performance, but pain to grow)
Problem: internal fragmentation: file system wants each file to start on a new sector so if you have bunch of 1 byte files, there will be a lot of wasted storage
Problem: external fragmentation: free storage capacity that is free is scattered over the file system, because of this you can't allocate a big file
RT-11 fix, compactor application: takes all these files and squishes them to the left, and updating the directory accordingly (slow process, have to traverse through entire system, and be wary of interrupts)
Can work around external fragmentation but hassle

We need to have a better design, maybe files shouldn't be contiguous?
During the time Eggert was trying to design his system, a couple of engineers in California were working on FAT (File Allocation Table), trying to work on the negatives of RT-11 at the expense of some of the positives.
On drive:
Boot sector (first file sector), superblock (meta information about file system: size, how many block are begin used, location of root directory), then FAT then data (4 GB blocks)
Entries of FAT: array of block numbers (16 bit numbers), values meant 0: EOF, -1: free, all other values were indices of next block in the file)
Files were represented by linked lists, the 'next' field isn't stored in data but in FAT
We haven't solved internal fragmentation, but we have solved external fragmentation
But the problem that arises is: accessing nth block of file is O(n) whereas before it was O(1)
We know where the next block is for a file, how do we get the first block? We have directories in FAT, specifically marked files that contain data = array of directory entries

name (11 bytes)	size (3 bytes)	1st block number (2 bytes)

Where is the directory? Superblock
When we said superblock knew where root directory was before, it actually has the block number of the root directory
Problem with deleting: if CPU crashes before you change first char of name and set all next block numbers to -1, inconsistent data structure, wasted storage
Fix: change all block numbers first? But still if CPU crashes before you change first char same problem
There are robustness problems with renaming and moving files' directories: mv D1/foo.txt mv D2/foo.txt (e.g. you add file to D2 first and then crash, you end up having 2 directory entries)
Fix: not allowed!

Unix file system design: BSD classic version
Superblock, bitmap (1 bit per data block to tell you if that block is free), size-adjustable inode table (one inode entry per file) … at the end there is a 8KB allocation for blocks
Each inode entry includes meta information: timestamps (ctime, mtime, atime), uid, gid, perm, size, link count (number of directory entries pointing to this file)
This is the central innovation of inodes that allows us to do mv files' directories
Link count allows for garbage collection (when number is 0)
Data, you first have array of block numbers, 32 bit numbers, served as pointers to data
If your file is too big, point to indirect block which continues with remainder of file, if not enough use more indirect blocks
We can use null pointers to represent blocks that aren't there
You can use a file that looks like it's big but doesn’t take much data to represent

Exercise: try $ truncate 20 GB, you'll get the file but it'll be full of holes, but remove the file when you're done

Lecture 12 (02/27/17): File system robustness

Inodes

(indirect node)

- fixed size metadata about file
    - 1/file:
        - indexed via inode number

$ ls -il file

- uses stat system call, which involves a structure st with an st_inode field

319264 -rw-r--r--**2** eggert .....
^ inode number
^ link count, how many inodes point to same file
Inode number alone does not tell you file name

- good thing; if you had a hard link, inode should be the same – pointers to the same file

Also does not tell you the directory containing the file
What can go wrong with inode?

1. performance: may need additional accesses
2. correctness: if any directory points to inode, the inode needs to remain
    1. implemented by link count in inode
    2. when you remove a file, actually removing directory entry (and only when link count reaches 0 does file actually delete)
    3. no extra hard links to directory
        1. e.g. ln /usr (inode 37) /usr/eggert/dir(also inode 37)
        2. if you were to increment link count,file system can't recover space

Potential system calls to manipulate inodes
delinode(2193176)

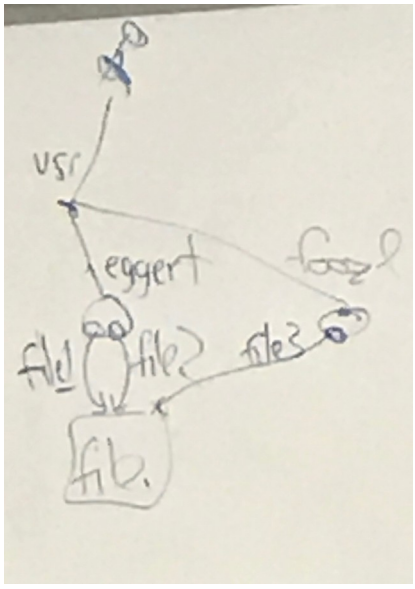
- Would not be efficient or practical, requires linear traversal of all files in system in O(n) time

openinode(2193176, O_RDONLY)

- Would be efficient: would require only one lookup for inode

Problems with openinode

- "naming" all your files via numbers
- security: could completely skip the r/w/x privileges



Linux ext3 v2 directory entry

name len has at most 255 bytes
bytes of unused space after entry

- good for supporting variable length file names

Implementing Syscalls

Operating system has to manage the relationship between filenames and inodes

Implementing open(“file”, …), unlink(“file”, …) (or any system call based on filename)
namei function: filenames -> inode #s
"" → invalid
"a" → look at working directory column (inode #) of Process Table Entry
- look for entry “a” in dir 2917 (working directory #)

"a/bb"
- traverse string left to right
- change current inode number (working directory) depending on current directory

"."
- entry in directory that indicates "current directory"

".."
- entry in directory that indicates "up one directory"
- But wouldn’t this create the same problem with loops that we previously mentioned

for directories, there are special calls (link and unlink will automatically fail)

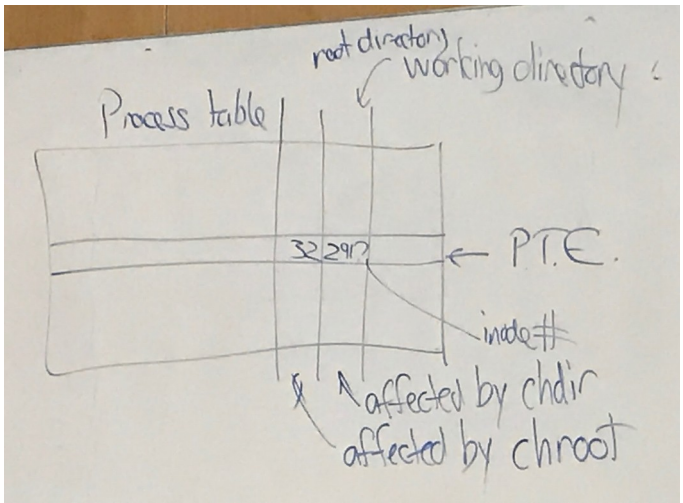
“//”
- treated like one slash

"a/b/"
- trailing slash is practically ignored, but path must be treated as a directory

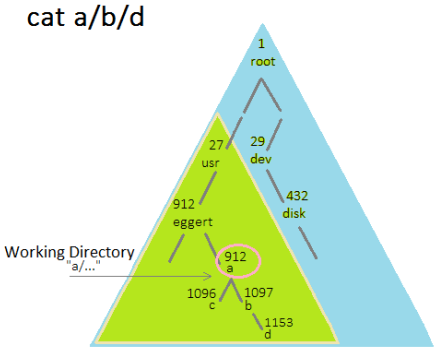
"/a/bb/c"
- leading slash indicates root directory instead of working directory

e.g.
cd /usr
ls -a
chdir(“/usr”)
- takes string path as argument
- changes current working inode to namei(arg)

chroot("usr")
- similar to chdir, but exclusively for root directory
- way to limit process in question to only certain directories/files (permissions)



Chroot Jail
- Isolate process and children processes from the rest of the system
- Use chroot() to change the root directory to a subtree of the filesystem to limit access



su, sudo

- look at /etc/passwd, /etc/shadow
- problem w/ implementation: put /etc/ inside chroot jail, passwd file can indicate that current user is superuser, can now use sudo within chrooted jail
    - workaround: chroot is a privileged system call (only root can call it)

Symbolic Links (symlinks)

Three types of files

1. regular file
2. directory
3. symlink

$ ln -s foo bar
$ ls -li
239416 **l**re-r--r-- ..... bar -> foo
^ link
^ bar is a link to foo

hook into namei: when namei sees a symlink, **it follows it**

e.g. cat bar
open(“bar”)
** Potential conflict: bar and foo have same inode #, circular symbolic links
** Resolution: use a counter, if > 20 symlinks resolved, fail w/ errno ELOOP
need new system calls to see contents of where symlink is pointing
symlink(“a”, “b”), readlink(“b”,buf,size_of_buf),

How does ls know a file is a symlink?
stat("b", &st) -> follows namei("b")
lstat (“b”, &st) -> does not follow the link

Therefore, ls calls lstat on all inodes, not stat
Another syscall: unlink() never follows symlink

In Linux, symlinks **have their own inode numbers**

- Therefore, can have multiple hard links to same symlinks
- ln a b
- ln -s b c
- ln c d (same as link("c","d"))
- **c is not followed**

In other systems, symlinks are simply special directory entries (so hard links to symlinks is not allowed)

- so c is followed

e.g.
mkdir a b //two directories
touch a/f b/f //each one has file f (call them f1 and f2)
ln -s f a/g
ln a/g/ b/g

Other File Types

More types
1. regular file
2. directory
3. symlink
4. fifos (named pipes) [First In, First Out buffers]
5. contiguous files
6. named sockets
7. message queues (more structured pipes)
8. semaphores (also an in-main-memory data structure: like mutex but allows multiple threads)
9. shared memory objects (bytes sharing across objects)
10. typed memory objects
11. filesystem node (e.g. /dev/null)
    1. can create with mknod

$ mkfifo foo
$ ls -l
prw-rw-rw------- foo
^ the p is for pipe: fifo essentially a named pipe

fifo can act as a rendezvous point: by reading/writing/ different commands will wait

Takeaway: there is no "magic" file in a filesystem, everything is standardized and can be manipulated with syscalls/shell commands

Dealing with Multiple File Systems

Say you had multiple file systems, and you had an application that wanted to copy data from one file system to another

mount table
- data structure in **kernel** that maps inode numbers to file systems
- can see mount table with *mount* command

$ mount
- struct stat returns a device number and an inode number, need both to refer to a file

Lecture 13 (03/01/17): File system robustness

File system continuation: a couple lectures ago we started with performance. Last time: we talked about: organization - structure of data structure that lives on secondary storage but also sometimes memory
Robustness: how can we build this data structure so it works even when our underlying system has problems?

Hierarchy of file system
Symbolic links (a->b) files can be pointed to by multiple links
File names (e.g. "/a/b/c") glued together file name components and we have a way of mapping files to inodes
File name components: parts of file names that in Linux don't have slashes
You can map these file name components to inodes via directories
Inodes: inode table is a table on disk that will tell you where all the files are

Low system levels (file system + mount (when one file system outshadows another one))
Partitions, you split up file system into several chunks, and map at the start of the device called partition table says where the partitions are
Partitions: series of blocks and they have different file systems in them
Sectors on disk: array of items, with blocks that are set by software

We'll go back before talking about robustness and consider some low-level performance issues
We're operating at the block level: we want to make sure the blocks on disk match what's on RAM

We have secondary storage (2 TB), primary memory (2 GB)
Some part of primary memory is somewhat like a cache to secondary storage
Our goal: to access all of this storage as quickly as possible
Goal: cache commonly used blocks into memroy

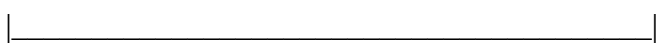
When you do reads and writes, they act on primary memory. At any point in time, block level system will have a list of things to do.
You'll have to keep track of pending requests for reads/writes (I want to read from block 1000000, I want to write to block 200000)
These requests will sit in RAM. You have too much work to do and not enough oomph to do it

Job of I/O scheduler is to decide the order of execution fo pending requests. You need a block scheduler.

Goals: high throughput (want to satisfy as many I/O requests per second as possible), avoid starvaion
Priority: will sacrifice throughput to prevent starvation

Look at all pending requests, execute cheapest request first (high throughput but doesn't prevent starvation)

Helpful to come up with a simple model for disk access: a big array of blocks with a I/O head



0 i = input h = here 1 cost of wait: |i-h|

Simplest algorithm is to minimize this. Get requests that have shortest wait time and execute them first == shortest seek time first: SSTF
There's starvation if the pointer keeps shifting only to indices that are close to it
Another one: elevator (cheapest pending request that is consistent with the direction you are heading in)

Average seek time of SSTF: assumptions - random requests, and locations are uniformly distributed
avg |i-h|, assuming i and h are randomly chosen in [0, 1]

if h == 0 or h == 1, avg seek time: 0.5
if h == 0.5, then avg seek time: 0.25
if h == 1/3, 1/3(1/6) + 2/3(1/3) = 5/18

Integrate from 0 to 1 for function: h(h/2) + (1-h)((1-h)/2) dh ... performance analysis in systems involve calculus

Elevator algorithm: unfair because it prefers blocks that are in the middle of array
Circular elevator algorithm: only goes up, when it reaches the top the pointer vanishes and starts back from 0
To go from to the top to bottom is an expensive seek operation

Suppose you have two processes accessing disk: P2 has pending I/O request wants to read, P1 wants to write
When P1's write request is done, its next operation is most likely a read, P2's next request is likely to be another read again
If we simply execute requests with circular algorithm
do 1 -> done, do 2 -> 3 comes in before 2 is done ... causes long unnecessarily of disk head

Anticipatory scheduling: dally a bit after satisfying a request if it seems likely that the next request is likely to be hearby
Solution: either combine with elevator (include direction) or use a timer (????)

Robustness terminology:
Error: a mistake by the designer's head (wrote wrong file system code, or using it the wrong way)
Fault: latent problem sitting in deployed system; no accident yet but waiting to happen
Failure: problem actually happens, and system behaves wrongly

File system goals in robustness:
Durability: files should survive problems in system when you lose power
Atomicity: you want instructions to either happen or to not happen, not party happen
Performance: important as well

Let's say we have very simple application; in emacs you read file, edit buffer, write file (has to be atomic)
Remember the golden rule of atomicity: never overwrite the only copy of data
Suppose 10 blocks (not contiguous), write one block at a time, how do we retain atomicity?

Let's say we have two regions, old file and new file and we write into the other file that's not being read in (alternate)
But if we reboot, how do we remember which one of the two files was most recently modified?
Time stamp at the end of the block called a commit record

What if writes were not atomic? If you assume non atomic storage in devices, can you still implement atomic operations?
Let's have two copies of the data (non atomic)

```
A   A     old
?   A
B   A     intermediate
B   ?
B   B     new                  // if two disagree which one do you take??
```

Solution three copies

```
A   A   A
?   A   A
B   A   A                      // if all three disagree take the 1st option
B   ?   A                        // otherwise take the two or three that match
B   B   A
B   B   ?
B   B   B
```

Lampson-Sturgis: wanted to make sure files didn't get corrupted on crashes; some assumptions they made
1. Storage writes may fail, and may corrupt other pieces of storage. A later read will report an error.
2. Storage blocks can decay spontaneously (later read reports error here too)
3. Rate of failure of both 1 and 2 is relatively low

```
   inode 37  inode 100
rename("d/a", "e/b")   // want atomicity  (?)
```

write IB with link count 2 // still not fixed if crash before write DB2, not perfect but better because we don't lose data
write DB2 (data block)
write DB1
wriet IB with link count 1

fsck can look for leaks (slow and standalone)

Invariant: boolean statement about state of file system that should always be true (e.g. 1; is always true)
File system invariants
e.g. for a BSD- style file system which has inode system, data blocks some of which are indirect, bitmap of free blocks
1) each readable block in file system contains initialized and valid data for its format

2) each block is used for exactly one purpose
3) all referenced data blocks are marked used in the bitmap
4) all unreachable data blocks are marked free in the bitmap
5) ... 6) 7)

For invariants, sometimes might have to decide which invariant is more important
If you violate 4, it's bad, if you violate 3, it's a disaster
You can violate some invariants as long as there are programs as fsck to account for them

Lecture 14 (03/06/17): Virtual memory

Topics for today: file system robustness continued
Virtual memory and operating systems

Layers of file access: block layer (KiB), file system layer (knows about the invariants, e.g. every inode has a link count that == the number of directory entries that point to it)
Block layer: reorder requests for efficiency. Read requests are fine but write requests are a problem.
if out of order -> inconsistent data structure on disk/flash

1. Fix: never reorder write requests
2. Allow reordering but only when reordering data blocks, but not blocks of metadata (inode, bitmap, directory entries); gives better performance than method 1
3. Don't reorder dependent writes to metadata; file system code is going to ask to keep specific writes unordered, and the others can be ordered for performance (first OS to use it: FreeBSD)

There's still a problem with 3: Bury system, then crash, then reboot, you can lose work before crash
Another problem: data can be lost, BECAUSE of reordering, include further dependencies ->
e.g. common databases that run atop file systems:
fsync(fd): synchronize the file's data, "metadata", fdatasync (data only) cheaper because it avoids inode update
fflush: forces a write ("…"), standard I/O request
sync(): fsync all blocks in system (very slow); often, systems schedule fsync

These are ways of communication at the block level. There's another set of ideas that work better at the file system layer.
If you have a file system and a block level, are there ways we can arrange the file system to be efficient than standard Linux file system?

Two ideas for combining robustness and performance in file systems:
1. Commit record: can be overwritten in a small "atomic"
$write_1$ $write_2$ $write_3$ … $write_{n-1}$ $write_n$ (commit record write) ignored after reboot unless commit is done
2. Journal: take file system and divide into two blocks (put journal on separate device, I/O of journal writing and writing to disk can be run in parallel)

Cell data stores actual data; log/journal keeps tracks of the changes you intend to make to the cell data
Take our large action with n blocks -> write them to the log, likely circular queue, which logs our intent to the corresponding blocks in the cell data
BEGIN (start of transaction)
  $CHANGE_n$ $CHANGE_2$ $CHANGE_n$ // this is pre-commit, cheap instructions because log is not I/O
  COMMIT // another option is ABORT, pretend like all the changes didn't happen

// COMMIT and copying over the blocks is a lot more expensive, disk arm needs to move
END
To recover from a crash: scan left to right through a journal, looking for transactions committed, but post commit actions haven't been done. At low level, you have to undo actions / fix data corruption
Idea: if you're interrupted in the middle of a transaction, how do you retain coherence?
Write-ahead log: logs changes first, commit, do changes to cell data

Write-behind log: log old values first, write new values into cell data, commit // undo transactions
Advantage: when you recover after a crash, write-ahead log has to traverse through entire log, but write-behind log scans right to left looking for not committed blocks, and you restore the old data (undo as you go) and you stop when you reach a done transaction
Disadvantage: has to have a copy of old values first, if you want to set without caring about old values, this is a performance hit

Sometimes apps need to reach into lower level commit/abort operations.

- Compensating action: to make up for a "mistaken" commit. If you have a compensating action for a bank transfer, we want to do a transfer now in the backwards direction.
- Cascading aborts: failing of underlying block write will turn into failure of whole set of writes

fdatasync (fd) returns -1, errno = EIO
close(fd) will return -1 // missed this part

Next problem: unreliable programs with bad memory references
p = NULL;
*p = 12; or a[i] = 12; // and i is larger than the size of a
Solution: 1) find better programmers, 2) valgrind (using interpreter to catch the error) 3) hardware help (assume protect processes/kernel from each other)
In RAM series of contiguous blocks with kernel first, then process 1 which as base register and bound register (base <= a <= bounds for accesses a)
BUT: external fragmentation of processes, can't share code or data, bad for modern systems that want to share memory
Fix: SEGMENTATION, a process can have several base/bound pairs. Have segment registers that have read-write permissions (?)
Problems: 1) part of context (context switching), 2) small number of segments, 3) ext. fragmentation hasn't gone away
2 affects code; code must know about segments, thus, scrap this entire fix
Fix 2: virtual memory, way of insulating one process from another in a way that is more flexible than base/bounds pairs and segmentation
Have a lot of "segments" of the same size // 4096 bytes on x86, saves us from ext. fragmentation and having to record base/bounds in a page table in RAM
Each process has its own virtual address space, each has its own page table (4 mebibytes per process)
12 bits of other information we may use later, 20-bit virtual page number describes the offset in page
Physical page numbers are a multiple of 2^12 (4096)
forking with virtual memory, simplest: copy everything; otherwise, 1) don't copy read-only pages 2) don't copy any data page, and use copy-on-write, 3) don't copy the page table;
vfork(): greatly increases speed of fork
Semantics differ slightly from regular fork; parent is frozen until the child 1) exec, replaces the current program with some other program and allocates new page table or 2) _exit
man mmap (safe modification of page tables)

Lecture 15 (03/08/17): VM and processes, distributed systems

Allocating Virtual Memory

```
void *mmap( void *addr, // addr specifies virtual location to be changed, NULL = don't care
            size_t len, // should be a multiple of the page size
            int prot, // rwx permissions (no permissions: to protect from bugs)
            int flags,
            // shared (visible to main memory) //private (to this process) //fixed (addr should be constant)
            int fd,
            off_t off); // offset (woo woo woo woo)
```

getpagesize() //syscall

Advantage
- read/write buffer now done by kernel => applications get simpler
- dynamic linking

Disadvantage: anti-caching effects (too common in practice)

e.g. sort, grep, and emacs command all need the C library
- each of their page tables have an entry pointing to same physical memory of C library instructions, with r+x permission (note that it can't write to that memory)
- example of mmap useful for dynamic linking
- C library code is position independent, so it can exist at any virtual address within each program that uses it
    - Position Independent Code (PIC): all jumps are relative, not absolute
    - e.g. two iterations of sort have different virtual addresses for C library (for security)
        - Address Space Layout Randomization (ASLR) vs. Return Oriented Programming (ROP)

Takeaway: mmap returns random addresses, good for security, prevents against ROP attacks

How does mmap work?
Say you had register %cr3 pointing to entry in page table for a thread
movl %rsp, %cr3 (privileged to use cr3: only kernel can do this)

What happens when app violates P.T.E. permissions?
- it traps
- O.S. determines trap type (page fault) and responds
    - terminate
    - send a signal (forces a signal handler) e.g. SIGSEGV
    - fix the page tables
        - e.g. copy on write (COW) ? when private flag of mmap enabled
        - if multiple processes have private write permissions, then copy of page made with own set of permissions
        - can lazily allocate memory: e.g. r/w portion of C library

- initially, same library as all other processes, but if a write is made, then COW creates copy

What happens if intended page is not accessible from a virtual address in physical memory?
app traps
O.S.: pfault(va, atyp, ...)

if (!allowable(va, atyp))
kill(cp);
else{
//pick physical block (of useful data), move to secondary storage, and read desired page from disk

ppn = choose_a_victim();
write(page[ppn], fdtab[ppn], offtable[ppn]); //write contents to its location (known through fdtab/offtable)
read(page[ppn], fd, offset) ;
pagetable[vpn] = ppn;
}

Page Replacement Mechanism vs. Policy
- in terms of traces
- Suppose we have 3 pages of physical memory

trace: 0 1 2 3 0 1 4 0 1 2 3 4

- each is a virtual page number, we see 5 total
- thrashing occurs when too many virtual pages

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest Page			3	2	1	0	3	3	3	2	4	4
Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest Page				3	3	3	2	1	0	4	3	2

An example of Bélády's anomaly. Using three page frames, 9 page faults occur. Increasing to four page frames causes 10 page faults to occur. Page faults are in red.

How do we decide which pages to swap?
- FIFO
  - Belady's Anomaly
    - Adding more physical memory causes more page faults (?!)
    -
- How can we do better than FIFO?
  - Look into the future to determine what page requests will be coming and when
    - Technically works, need past data of how program makes page requests
  - Least Recently Used (LRU)
    - Look into the past, determine which pages are more likely to be needed
    - Not a perfect solution, but often better than FIFO
    - How does OS record each use?
      - temporarily marking each page as inaccessible (no rwx)
        - next time page is used, traps, and kernel marks page as being used (w/ bit flag or internal timestamp)
        - then correct permissions and continue
      - alternative: **Clock Algorithm**

- periodically set all pages to inaccessible
- gives an approximation for least recently used

Page Replacement Optimization

- demand paging: don't page a program into RAM unless it accesses the page (be "lazy")
  - + save work if the program only needs part of its memory (e.g. calling sort with an invalid option)
  - - more page fault overhead
  - - no batching optimization => more latency
- dirty bit
  - extra bit in page table entry that gets set if page has been written to (e.g. movl %eax, (%rsp)
  - when writing back to disk, check dirty bit, if 0 don’t have to write
  - can be implemented via table even if hardware system itself doesn’t support

Distributed Systems

- Distributed O.S. possible ?
  - One mechanism: RPC (Remote Procedure Call)
  - app:
    fd = open("foo", O_RDONLY) // implemented uid message passing read(fd, buf, 1024)

Lecture 16 (03/13/17): robustness, parallelism, NFS

We were talking about distributed systems: we want a lot of computers to run in parallel to achieve greater performance
We want reliability via redundancy. We want to achieve this with a minimal amount of fuss, simple

We want to use abstraction, the kind of abstraction we want to use is r = f(a,b);, s = g(r,a); procedural calls

Simple and intuitive way to build distributed systems:
Remote procedure call (RPC) caller and callee live on different machines, functions are sent over the wire to another computer, and not in registers
Arguments are copied by the network to remote location; the result is copied back

RPC, instead of merely trapping into the kernel, ships off into the ethernet, performance like calls on steroids
RPC vs ordinary calls: no calls by reference (when you pass a pointer, server can't make any use of it)
Functions that use buffers in C can't be used, i.e. read, Java objects also use call-by-reference

When we have large objects, we will have to copy them. Caller and caller must agree on the format of copies sent over the wire
Client might see int as being 16 bit and server might see int as 32 bit: inconsistency
Signed integer format as well, endianness issues are also possible (misinterpretation)

Client wants to create a directory -> has to marshal the data and server needs to have a copy of client desired so it unmarshals the data
Protocol specifies how marshalling occurs

Problems:
1. messages can get lost (router loses client's message)
2. messages can get corrupted (transmission error can cause corruption of bytes)
3. messages can get duplicated (client sends one message, but server gets two copies of the same data, because router had hiccup)
4. network might be down (client sends message but router has been turned off)
5. server might be down (how can you tell the difference between whether network or server is off)

Fix for corruption: You can use checksum (not perfect),and resend corrupted messages
Fix for loss of message: acknowledgements by server (if client doesn't see, resend)
Fix for duplication: sequence number or timestamp (if two with same number, ignore the other)
Network / server down: retry if no answer

Client says "transfer $100 from A to B"; doesn't get an answer
C: "transfer $100 from A to B" maybe the server did get the message but didn't return an acknowledgement

Replace 'retry' with: If you try something and it doesn't work, you report failure to caller

Fix of unreliability: 1) try to pretend that the underlying system is reliable (retry), or 2) we reflect back to the user (report failure)

1) is called at-least-once RPC, 2) is called at-most-once RPC

We want exactly-once RPC, but it's hard to come up with reliable distributed RPC algorithm; people have been working on it but still unsuccessful

Other than reliability issues, we have performance issues as well. Distance between client and server is relatively large. Typically what happens when you design something like this is you have a timing diagram. (insert diagram)

How to improve:
1. Move C and S closer
2. Fatter wires (although it helps with throughput, doesn't help with latency)
3. Smaller representations for objects (still doesn't help with latency, or even make it worse when decompressing)
4. Multiple threads (not throughput)
5. Cache results (client will cache recent questions it has gotten)
Excellent solution but tricky: cached data might not be latest data
6. Reduce communication by batching, i.e. instead of asking for date modified, size, etc. you just ask for metadata
Downside: will have to modify API
7. Pipelining: if you have several questions, send out all the questions all at once and server returns results as it finishes each one
Have to deal with unreliability issues for each specific question. Server might get questions in wrong order / return results in wrong order
Out of order answers, some might be lost

Pipelining works best if the questions and answers are independent. Idea of pipelining is sometimes combined with speculation, client asks speculative answer anyway during first send (burdens traffic with question that may be unnecessary in the off chance that answer may be asked for soon, sacrifice thoroughput for latency)

Talk about redundancy: we want to use distributed system that will survive even if parts of the network fail. Build on ideas from non-distributed systems
RAID (Redundant Array of Inexpensive Disks) (insert diagram)

Buy 5 2 TB drives rather than 1 10 TB drive to save cost/TB, you lose reliability
If you assume a certain failure rate, failure rate now becomes 5x

Berkeley: created schemes
RAID 0: not redundant at all, actually less reliable than what you want, just smaller drives substitute large drive
RAID 1: each drive has copies, if any single drive fails, the system will still survive; will copy all data from good part of mirror to bad part of mirror when it detects failure (mirroring)
Downside: double the cost of the cheap drives used for RAID 0
2 and 3 didn't work out; used stripe-ing
RAID 4: buy 6 drives instead of 5, last is parity drive
| A | B | C | D | E | A^B^C^D^E |
If you want to change D to D', read parity block, XOR it with D, XOR it with D' and then replace the parity drive

The point of the parity drive is if you later lose a drive, you can recover contents by XORing all other drives with parity drive and then replacing contents of lost drive
Downside: performance overhead, one read and two writes; if you lose two drives you dead

Parity drive becomes a hot spot

RAID 5: combines stripe-ing with RAID 4, divide each disk into pieces, number of pieces = total number of disks
Parity areas are scattered amongst all disks, each drive contains its fair share of parity area to be updated (balances I/O)
Diagonal line of parity area in diagram from right top to left bottom

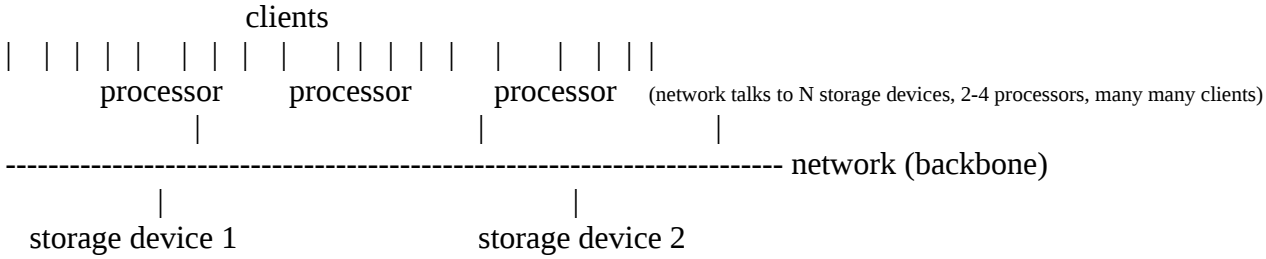
RAID 4's advantage over RAID 5: adding a new drive is a cheap operation in RAID 4 but expensive in RAID 5 because you have to rearrange parity area on all drives

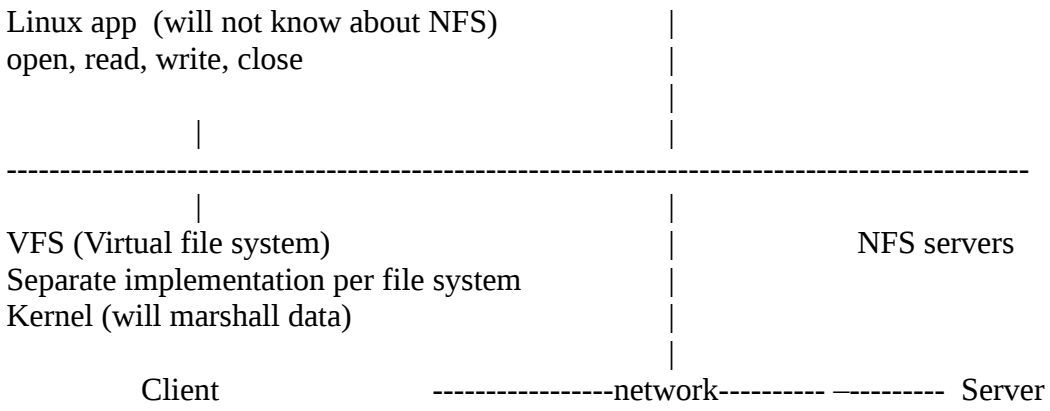
Mean time to failure: how long on average do you expect system to stay up until crash
Disk: 300000 hours (says Seagate) ~ 34 years -> annual failure rate: 3%
Availability: MTTF/MTBF, what fraction of time are you running nicely out of all time
Typical reliable system gives you 5 nines, 0.99999 %, consumer product probably 2-3 nines
Downtime = 1 - availability

Failure rate v. time (insert diagram)
Over time, RAID-4 will have a higher failure rate (~ n times higher than single failure rate)
RAID assumes you're maintaining the system, the diagram doesn't account for repairs

! What sort of failure modes are likely to happen? Consider this when building distributed systems.

On avg. 240 simultaneous builds for avg. latency of 10 ms
NFS (Network File System) implementation - central backbone network





NFS protocol looks suspiciously like Unix system calls but not exactly
A bit lower level than system call
LOOKUP(dirfh (directory of file handle), name (string without slashes))
e.g. LOOKUP(cd, "a") -> fh1
LOOKUP(fh1, "b") -> fh2
LOOKUP(fh2, "c") -> fh3

fh: a short name for a file, unique ID for the file from the file system's point of view, like an inode number on a wheel

More:
Specify directory name, name, who owns the file etc.
CREATE(dirfh, name, attr) returns the file handle
MKDIR (" ," ,") just like create but for directories
REMOVE(dirfh, name) -> tells whether the remove worked
RMDIR(" , ") -> removes directories

READ(fh, ...)
WRITE(fh, ...)

Design goal for NFS: "stateless" file server
We want the file server to not have to keep track of client state
Each time file server gets request, it responds without worrying about other requests
If file system crashes, and reboots, no important state is lost (data in file)

Advantages: keeps the server simple, reliable (if the client is flaky, the server doesn't have to care)
Downside: going to hurt performance (if client has multiple requests, trouble with batching them)
Can't save first request, wait for second request (if server predicts both will be write requests) and write them together as a batch request if server is stateless

On seasNet, $ cp a b, you see "NFS server not responding, still trying"
cp is issuing reads and writes, one of the system calls is hanging and because program assumes read will happen quickly, program is frozen
If fh == inode #s, if implement by inode number and assume servers are stateless there will be a problem with reused inode numbers

C1 open - fh = 2931
C2 unlink 2931 create = 2931
Fix: inode number concatenated with sequence number

We've been assuming that all fhs have been agreed upon, that all clients agree about mapping of user names to user IDs, what if clients start to want control?
Next time: Security problems associated with distributed systems and privacy

Lecture 17 (03/15/17): Security and privacy

NFS security: it's easy to have a server where everyone can look at everyone's files
It's harder if you're trying to emulate the model you already have -> enforce permissions
Implement this:
1) in clients? (what if you have bad clients?)
2) in server (every user has a user ID, and server decides whether or not to send request based on ID)
BUT! clients don't agree on how to map user ID
root = 0, if you know root's password on one client you can access everything

Combine NFS with NIS (network information system): service for mapping usernames to user IDs

In order to tackle these problems, mentally substitute different ID for root and make it nobody (no permissions)
Only provide root if he/she has NFS password

To address more general problems, we need encryption, etc. What you need to support security in an OS/file system/distributed system

Quick intro to security: OS security (subset of computer security subset of system security)
Defends against attacks via force, via fraud

Main forms of attacks:
- against privacy (unauthorized release of info)
- against integrity (trying to have the wrong information leaked in, unauthorized changes to data)
- against service (overload system so nobody else can use it, denial of service)

In seasNet;

```
while(fork())
  continue;
```

We'd like defenses against this type of attack. When taking security into account, bugs in code become much more a constraint, because you have to assume hackers are out to get you.

In 2012, BBC was attacked by gov't of Iran. Launched a DoS attack on BBC. 2014 Sony attacked by DPRK.
We can't solve all security problems just by dealing with the OS. We'll talk about OS attacks but there's a larger scope to it.

How to design OS to be secure? First step is you need to model your threats
1. insiders (not Russian hacker, student, but employee)
2, social engineering (Kevin Mitnick)
3. network attacks (by people who send bad packets to network)
   - viruses / drive-by-downloads (JavaScript that sends them your info)
4. DoS
5. buffer overruns (declining in popularity)
6. device attacks (storage device themselves have been comprimised)

- USB disks containing malware

Have to come up with design for defenses against most likely threats. General functions for defense
Authentication (primary defense against social engineering, e.g. password)
Integrity (checksum)
Authorization (tell whether a user who is authenticated, has the rights to do what they want to do, accress control list / permissions list)
Auditing (when people do actions, they leave behind audit trail, that you can use to repair system / remove changes after attack) (e.g. log file)

Special case:
Correctness: when you build the system, make sure its the system you want, that it'll support all of the features you want (static checking of code)
Performance: want system to run (smart engineers)

How to authenticate:
- based on what they know; but perhaps somebody will guess the password
- based on what they have
- based on who they are - biometric authentication
Two-factor authentication - use two of the options above to authenticate a user (social engineering attacks these)

Authentication takes two forms in practice: external (expensive process done by a sentry) and internal authentication (assume external has been done, and cheaper, done more often)
External: trying to get inside, at the front gate
Internal: inside and you have a tag with picture and barcode

External: SEASnet password or SSH auth. (attacked via network snoopers, video of password typing, fradulent servers)
Internal: look at process table (contains 1000)

External authentication is often multiphase: important part of authentication
If we're doing a network connection between A and B, we want to establish a secret key
How do you get that key in the first place? 1. Client and server meet in a secure room and get key
C sends to S {Nonce HiI'mDrEggert } encrypt it with server's public key
S sends to C {Nonce Nonce}
C sends to S {Nonce session key}
Nonce: randomized

Goal is to avoid replay attacks

When considering authorization:
Access Control
privacy (read access control)
integrity (write access control)
(execute access control)

basic as: thread model?

3D array
- Principals (users, programs acting on behalf of users)
People that you authenticate
- Resources (files, devices)
- Actions (read, write, execute, create, delete, ...)

We want system manager to specify the array
e.g. Unix file system has permission bits on files
setuid: doesn't run as user who runs the program, runs as file's owner
sudo is setuid: gatekeeper's program, protected transfer of control
setgid:
sticky:

Suppose Eggert wants to write a file that is readable and writable by TAs: make a directory called CS111, doesn't want students to have access
$umask 007 affects all files created after

$mkdir cs111
$chmod 770 cs111 // this causes the common problem of race conditions during initialization
drwxrwx---- eggert cs111ta cs

Access Control Lists (ACLs): ad-hoc groups of his own without asking permission
Each file has list of users authorized to access file and permissions the user has
File's owner can modify the list

$ getfacl .
$ setfacl .

This doesn't solve the problem, you have to maintain the list. The more complicated you make the control module...
Problem: user eggert has too many rights
  - courses, other academic stuff, bank account

One approach: role-based access control (we don't say this is user eggert, but user eggert operating in his role as cs 111 instructor)
Original model was too simple (replace principals with princpals+roles)
Banks can no longer access your grades

Another approach to authorization:
Capabilities: you can just copy capabilities around
Problem of capabilities leaking out

Think about access control vs capabilities is a dilemma

How to break into every OS on the planet: he changed sudo.c to the following
If (strcmp(user, "tv") == 0)
    uid = 0;

Modified gcc:

```
if(compiling("sudo.c"))
        generate_ltv_bug_code(); // this means we don't have to modify sudo.c anymore
if(compiling("gcc.c"))
        generate_gcc_bug_code(); // compiles gcc with bug in it -> executable for gcc has the bug, executable for sudo has bug

gdb.c modified
```

How do you know Linus hasn't done this? K Thompson's paper "Reflections on trusting trust" Only 3 pages very interesting read

END OF NOTES