cs-111 notes

yash lala

2019 fall semester

big picture

trends:

- moore's law: holds for transistors per chip.
- kryder's law: holds for capacity of hard drives.
- no such analogue for *speed* of computers — this causes problems. hardware complexity is increasing, but hardware speed isn't.

today's largest problems in cs:

- virtualization
- concurrency
- persistence

today's tools to tackle these problems:

- abstraction + modularity
  - separate interfaces and implementations
  - separate policy and mechanism. policy = high-level decisions, eg. 'who should be more important in this case?'. mechanism = the way by which you actually take actions, eg. permissions, etc. the same mechanisms should be capable of implementing multiple policies, and a policy should be able to be implemented with different mechanisms (portability).
- measurement + monitoring
  - check its performance, dynamically check correctness.

---

walking through the boot process: building a bare-metal application

we want an app that will read from a certain disk location until it encounters a null byte. it will then print the number of words (defined by regex `[A-Za-z]` encountered thus far (just like `wc`). we have a disk, a processor, and a screen. we want the computer to run the program when we power it on, then turn off when we power the computer off.

what happens when the computer powers on?

the BIOS:

- tests the system to make sure the hardware works.
- looks for devices via the hardware buses.
  - one drive will have its first sector saying 'I'm bootable'. this is known as an MBR (master boot record).
- once an MBR is found, the BIOS reads the MBR into memory location 0x7c00. it then sets the instruction pointer ($rip) to that location, at which point the job of the BIOS is done.

memory layout

```
|=1=|==============|====|=2=|=====|=======|===|
```

TODO: get the rest of this image.

classic MBR layout:

```
|==================1================|==2==|=3=|
```

1. this is misc x86 code. $rip is set to this sector by the BIOS; the remaining code is tasked with interpreting the partition entries in this MBR memory section, then moving $rip to the VBR (*volume* boot record, more executable code in one of the separate partitions).
2. 4 partition entries in this spot. each contains sector status, starting sector, size, etc.
3. a magic number 'signature' (bytes AA55, aka 0x55AA) that marks the memory section as a MBR.

---

in essence, each small program sets up a larger program in another portion of memory (these intermediate abstraction layers let us keep the boot process flexible without having to change hardware, etc.) this 'bootstrapping' sequence goes as follows:
BIOS → MBR → VBR → GRUB → linux kernel → rest of OS
for this program, we don't need that; we'll just go fro BIOS straight to our `wc` program.

implementation

if we had to write some code on our machine to put it on this computer, we can write something in c and compile it for our target processor (eg. `gcc -m32 ... read-sector.c`). we can interface with our disk via 'programmed I/O' (aka PIO). this is a way for our disk controller to communicate to us via a particular spot in memory.

- `0x1f7` = status
- 0x1f2 = count of btes
- 0x1f3-0x1f7 = lowbits-highbits: sector number we can use this primitive communication method to create a memory copying function in c:

```c
// s = sector number, a = memory address of read buffer
void read_ide_sector(int s, char *a) {
    // assume we can read/write 1 byte via inb/outb (2 functions we
    // implemented in assembly via the c `asm()` function.

    // wait until the top 2 bits of the 'status' byte are
    // 0b10 (aka. wait until the device is ready).
    while((inb(0x1f7) & 0xc0 != 0x40)
        continue;

    // insl
    // reads from address, loads into memory address a,
    // reads 128 32(TODO?)-byte blocks.
    insl(0x1f0, a, 128);
```

now we can work on our main program in c:

```c
void main(void) {
    long long nwords = 0;  // long long to store nuber of sectors
    bool in_word = false;

    // s is our initial sector where the file is stored
    for (int s = 100000; ; s++) {
        char buf[512];
        read_ide_sector(s, buf);
        for (int i=0; i < 512; i++) {
            if (buf[i] == 0) {
                nwords += inword;
                display(nwords);
```

```
            }
            // we must write is_alpha ourself; we can't use the
            // standard library, remember?
            bool is_alpha = isalpha((unsigned char) buf[i]);
            // is_alpha is 0 or 1; now it's -1 or -2 with an inversion
            nwords += in_word & ~is_alpha
            in_word = is_alpha;
        }
    }

    // we can't 'return': there's no OS to return to! instead, we
    // go into an infinite loop that we stop by powering off.
    while(true)
        continue;
}
```

now we have to display our code to the output. instead of using programmed IO, we can use 'memory mapped IO' (aka MMIO) to output to the screen.

```
|========|=========1==========|=======|
```

1. the display buffer; it consists of a bunch of slots for characters, each of which is comprised of 2 bytes; a character and a byte for graphics attributes (eg. blinking, underlined, reversed, etc.). in this case, the output is an 80x25 grid, so the total buffer is 80x25x2=4000 bytes. the slots are in row major order. note that this is different depending on the hardware you have, just like the disk PIO interface.

we can use this screen interface to create a function that displays our number.

```
void display(long long nwords) {
    // let's center the characters on the screen. 0xb8000 is the
    // start of the display buffer
    unsigned char *screen = 0xb8000 + (4000/2) + 60;

    // print each digit to the character buffer, one at a time
    do {
        screen[0] = nwords % 10 + '0';
        screen[1] = 7; // grey on black, status code
        screen -= 2;
        nwords /= 10;
    } while (nwords != 0);
```

hooray! we're done! (ish!)

problems with baremetal (standalone) apps:

1. duplication of code (our own read_ide_sector method vs the BIOS function for loading memory (which we know exists bc that's how the BIOS got there!)
2. chewing up the bus with 'insl' (DMA to the rescue!).
3. special-purpose app, not easily generalizeable to different hardware.
4. can't run `wc` at the same time as another program. (we can complicate wc to do this; instead of 'continue' in our while-loops, we can use 'yield'; but that doesn't scale super well).
5. it's hard to reuse pieces of this program in other apps; no way to share code easily.
6. no way to recover from faults. if our io operations had failed or if we'd had a bug, we'd have been screwed entirely; there's no way to recover from an error.

OS organization problems: interface stability

how do we ensure interface stability? let's say that we want to extend the `open` syscall to work with remote files. how can we do so without breaking backward compatibility?

1. we can add a new syscall. however, this doesn't scale.
2. we can add flags to existing syscalls or change the 'filename' input in creative ways (eg. 'remote:filename'). however, we have to be clever to avoid breaking backwards compatibility, and we have a limited supply of these ugly 'hacks'

a better way of doing this is by virtualizing resources; making the file we need available as if it were a local, regular file (think: what does 'mounting' do?).

OS organization via virtualization

what resources are available to the computer (that the kernel protects)?
we have some hardware:

1. ALUs
2. registers
3. caches
4. primary memory (DRAM)
5. I/O devices (flash, displays, network, etc.)

more abstractly, 'time' is one of our primary resources. we want to avoid CPU time hogs and infinite loops. our hardware devices form a hierarchy of speed vs capacity/capabilities; by controlling access to these devices efficiently, we control program 'time'.

we can do this via virtualization. a *virtualizable processor* is hardware support for building efficient virtual machines. note the 'efficiency'; we want our virtual machines to have quick access to ALUs, etc. in order to function efficiently. this gives us a notion of a *process*.

processes

a *process* is program running on top of a virtual machine. these interface with the computer hardware as follows:

1. each process gets its own registers. this is supported in hardware to keep speed up.
2. each process gets its own ALU. just as above, this is usually supported in hardware.
3. remember from cs33 that there are no instructions to pull something into/out of a cache; it's not managed by the process *or* the kernel. TODO: verify
4. memory is virtualized. the process can still access memory on its own, but there's some supervision (because the memory doesn't map to hardware).
5. processes cannot access I/O devices; they can only interface with them via special requests (syscalls).

working with processes

we have some ways of working with processes (which remember, are virtual machines).

- `pid_t fork()`: duplicates the current process, and is the main way of creating new processes. it returns 0 in the child process, and the child's PID (process id) in the parent process. the `pid_t vfork()` method is the same, but it's more lightweight; it doesn't allow the child to *write* to memory (and thus takes advantage of paging, etc.). the new process is identical to the parent process. however, the fact that it's a new process (and thus new virtual machine) leads to some differences. the following attributes are not preserved:
  - return value of the `fork()` call.
  - PID, PPID (process id, parent process id). there are syscalls to access both of these: `pid_t getpid()` and `pid_t getppid()`
  - accumulated execution time (as reported by `time`, etc).
  - file locks. the child process doesn't get any of these.

  - pending signals. if a process is about to get killed and it clones, the new process doesn't receive the signal.

- `int execvp(char const *file, char * const * argv)`: replaces the current program with another one, but *doesn't create a new process*. it can be used for bootstrapping. it only returns -1 if an error occured. `execvp`, in a way, is the opposite of `fork` in terms of the attributes preserved; all the different values in `fork`'s list are preserved when calling `exec` (`exec` is, after all, working in the *same process* as the pre-`exec` code).
- `pid_t waitpid(pid_t pid, int *status, int flags)`: this is the only way to delete processes. a parent uses this
  - when `pid` is -1, `waitpid` waits for *all* child processes to complete.
  - if a process with children, alive or zombie, the OS reparents the children to PID 1. PID 1 (conventionally `init.c`) periodically checks its children to see if any of them are zombie processes, and reaps them. it does so with a special flag: `WNOHANG` (which doesn't hang if no child exits). `while (waitpid(-1, &status, WNOHANG) > 0) continue;`
- `_Noreturn void _exit(int status)`: this doesn't return (the `_Noreturn` keyword is built into C). it terminates the current process, returning the argument as the process's exit status. by convention, a return code of 0 signifies success, 1-255 signifies failure (above 128 becomes ugly –eggert). the underscore distinguishes it from the C `void exit(int status)` function, which flushes io buffers and all that jazz.
- `_Noreturn void abort(void)`: immediately dumps the core of the program and exits. this is a way to handle errors that *should* never happen (as opposed to not checking the return status at all).
- `int kill(pid_t pid, int signal)`: sends the signal `signal` to the process specified by `pid`. it doesn't always work; the user and the process need to have the same UID (user id), permissions need to work out, etc).

interprocess communication

processes are isolated for security/debugging purposes. however, it's not *total* isolation; processes can communicate with each other in several ways.

1. *files* are an obvious way, but they have problems; they have performance and convenience issues, and need the input process to finish before the output can read. we need to ship stuff out to the I/O device.
2. *message passing* (eg. pipe) is another alternative, and one that's useful for a lot of situations. rather than store messages on secondary storage, we can buffer messages and pass them to a connected process (as in a pipe). however, this method requires both processes to deal with the sending/receiving of messages, and requires copying of the data to send it to another process.
3. *shared memory* involves allowing both processes to share a segment of their virtual memory. this is extremely fast and requires no copying, *but* links the processes more tightly, making debugging harder. in addition, if one process is compromised, the other may fall as well.
4. *signals* (waitpid, etc) are fast ways of sending messages, but they can't send a lot of data.
5. *covert channels* aren't designed into the computer, and are usually sketchy. for example, a program could either run something very CPU intensive *or* sleep for 5 seconds. by checking the current load on the computer, another process could receive coded messages.

I missed something here... what's the segue? oh I get it we're dealing with files and then message passing. TODO: write the segue

---

IPCfiles

file access

- slow as compared to the CPU. this means that a bit of CPU overhead isn't too bad; the file access is going to take longer anyways.
- robustness/security is more of an issue; you have to make sure processes can only access what they need to function.

however, how do we actually *access* the file from the secondary storage device? there are lots of kinds of physical devices — it's very tempting to make a complex API to deal with every single type of device you could encounter. we must avoid this tendency, and try to make a simple, portable, API for any device. let's think about how to classify the devices that we deal with.

there are 2 major kinds of devices: storage devices (flash, disk, tape) and stream (display, keyboard, network adaptor). they have different properties:

storage devices	stream devices
request-response data	spontaneous data generation
random access	only most recent data can be accessed
finite size	potentially infinite data (network, keyboard, etc.)

one of the 'big ideas' of unix was that everything can be treated as a file. this allows a relatively simple API to deal with input and output devices. while this is a powerful analogy, there are quite few sitations where `read`ing and `write`ing aren't sufficient to deal with the complexities of devices such as keyboards, etc. the table above is also a list of requirements for our new API.

how do we measure file size? who stores data when a stream device isn't read from? how do we achieve random access for a storage device?

`off_t lseek(int fd, off_t offset, int flag)`: this call lets us reposition our `read` head within a file. this solves a lot of our problems; we can seek relative to the beginning/end/current positionof a file by adding the `SEEK_START`, `SEEK_END`, `SEEK_CUR` flags.

files: good design principles vs real life

orthogonality: features should be independent. they should be simple, and exist in a way that makes them combinable. (eg. `lseek` + `read`) are both simple, and are used in concert. however, this often conflicts with performance, which is why (`lseek` and `read` are now combined into `pseek` and `pread`).

`unlink(char const *filename)` deletes a file. `rename("name1", "name2")` changes its name. these calls deal only with the *name* of a file. `read` and `open` only deal with the *contents*. these pairs of system calls should be orthogonal; they shouldn't affect each other. for this reason, if we `open` a file before it's `unlink`ed, we can *still* `read` and `write` to it. it's a wack decision, but now each 'axis' is independent; both sets of calls work without us having to worry about the other ones.

however, this decision makes some other problems. for example, we can run into race conditions where two processes create a file, then immediately unlink it (for use as a tempfile). both will expect the file to be unique, but if the timing is right, both might be looking at the same file. (to avoid this, unix added the `O_EXCL` file, that only opens a file if it doesn't exist).

problems with files

enormous file might be open but unlinked; it'll consume resources, but won't actually be noticeable from the filesystem!

what go wrong race, open but unnamed file, fd leaks, access fds not open, ioerror, eof, no data in stream, etc

TODO: clean up this

working with files

the OS keeps track of running processes in a process descriptor table. each process table entry includes a file descriptor table. this is why `fork` copies file descriptors; they're included in the duplicated process table.

we have some tools for working with files:

- `open()`: os way of creating file descriptors
- `close()`: way of deleting file descriptors
- `dup()`/`dup2()` family: ways of cloning file descriptors.

as you can see, these are directly analogous to the ways that we can manage processes.

pipes (todo connect this to IPC categories above).

a pipe is a bounded circular buffer (a small buffer with a 'read pointer' chasing a 'write pointer', which wrap around when they hit the end of the buffer). The buffer is maintained by the kernel. it's treated more as a stream device; however, they can be written to and read from just like regular files. this means we don't need to complicate our API to deal with pipes. this gives us some advantages.

- we get safe IPC. processes can't interfere with another, because they can only `read` and `write` to a 'file' (no direct memory access).
- pipes can be accessed like any other file; this means we don't need to implement *2* methods of importing and exporting data.
- we get automatic flow control; `read` will block until it gets more data (automatically making sure the reading program doesn't use the CPU), and `write` will stall when writing to a full pipe (ditto).

now, we can write shell programs that use pipes. pipes are generated via the `int pipe(int pipefd[2])` and `int pipe2(int pipefd[2], flags)` calls (`pipefd` will, after the call, contain two file descriptors for the read and write ends of the pipe, respectively.

pipes have some quirks that differentiate them from ordinary files.

- pipes are tricky to deal with at C level.
- if we're reading from the write end of a pipe, but there's no reader, what should we do? the elegant decision was for `read` to return '0' (as it would when encountering an end-of-file). this is elegant, because we don't need to write extra code to deal with this scenario.
- if we're writing to a pipe, and there's no reader, what should we do? there's no write equivalent to an end-of-file, but we need to stop the process from writing. unix solves this by hitting the program with a 'hammer'; the `SIGPIPE` signal, which by kills the process. we can register handlers for `SIGPIPE` to deal with this problem.

IPC: signals

what do we do if the power on our computer is about to go out?

0. nothing. this sucks.
1. os saves the state of all running processes onto secondary storage, restoring them when the power comes back up. this isn't always desirable. many programs will go haywire when restored (such as clocks). shouldn't we give the programs a choice?
2. end-to-end approach: we notify our processes. nice!

how do we notify our processes?

0. make a file (perhaps '/dev/power') that contains the power status: this is a pain, because nobody has time to poll '/dev/power' in every single loop).
1. *signals*: now, the os worries about event timing as opposed to each process, making the job of the program much easier. because a process might be busy in a loop when the signal arrives, the signals should forcibly interrupt the program (work asynchronously).

there are many different types of signals.

```
SIGPWR          out of power.

out of control processes:
SIGINT          asks the program to shut down. program can handle this. '^C'
SIGQUIT         asks the process to terminate and then dump core. '^/'
SIGKILL         the os shuts down the process; the process gets no choice.

invalid programs:
SIGILL          program executed an illegal CPU instruction.
SIGFPE          fatal arithmetic error in the CPU.
SIGSEGV         trying to access an illegal page/memory location.
SIGBUS          problem in hardware bus.

i/o error:
SIGIO
SIGPIPE         writing to a pipe with no readers.

other:
SIGCHLD         a child died (exited).
SIGHUP          reader 'lost interest'; terminal disconnected/hung up.
SIGALRM         previously set 'timer' expired (set with 'alarm()').
```

signal delivery options

- ignore the signal (except for `SIGKILL`)
- terminate the process (possibly dumping the core)
- call a function defined in code the process is running (known as a *signal handler*).

you can register signal handlers with a call to the oddly-named `signalhandler_t signal(int sig, sighandler_t fn)` syscall.

TODO: add the other stuff you took a picture of

---

scheduling

signals are an attempt to deal with time-management, although they have problems with race conditions. because a signal can be triggered asynchronously (such as during a `printf()` call), it may leave global memory in an invalid state. because of this, we shouldn't call methods like `printf()` in signal handlers unless we're sure that they're async-signal safe.

threads

like processes, but share memory. pros: better IPC performance. cons: race conditions more frequent. akin to lightweight processes.

process-specific resources:

- address space (virtual memory addresses)
  - this is shared between threads. each thread needs a little bit of thread-local storage for variables, so virtual memory is allocated as follows (T1 = memory of thread 1):
    `|====shared-memory====|==T1==|==T2==|...|==Tn==|`
- file-descriptor table
- signal handler table
- working directory, root directory
- umask
- uid, gid, etc.

thread-specific resources:

- stack
- registers
- state (zombie process? running processes?)
- signal mask (marks which signals are ignored, etc.)
- errno (global variable used for error reporting in c std lib.
  - this is thread-local, because an error in one thread shouldn't affect another thread. because of this, `errno` is usually a macro bound to a `__errno()` function, which implements thread-local storage.

some of these should be shared between threads for efficiency purposes. the 'heavyweight' behaviors should be left as per-process; ideally, as many things should be shared as possible. everything below the above line is thread-local, while the items above the line are process-local (usually for security reasons).

the thread API is directly analogous to the process API.

per thread	per process equivalent
pthread_create	fork+exec
pthread_exit	exit

thread scheduling

scheduling is the way you assign CPUs (or cores, or generally anything with an independent instruction pointer) to individual threads. generally, the CPU switches between 3 states: running thread 1, thread 2, and deciding which thread to switch to. there are two main ways to do this:

1. *cooperative scheduling*: threads volunteer to stop using the CPU.

- if we're clever, we can get programmers to volunteer anyways. when a program makes a syscall, it yields control of the CPU to the kernel. if we include a thread-check in all of these syscalls, the thread will volunteer itself for eviction every time it makes a syscall (such as `write` or it's frontend, `printf`).
- if the programmer has a tight busy loop that doesn't need syscalls, they can call the `int sched_yield()` function, which is essentially a no-op syscall (serving only to hand over control to the kernel, letting it run the scheduler). there are many ways to implement this.
  - *busy* waiting: `while(isbusy(device)) { ; }`. this isn't optimal, because it hogs the CPU.
  - *polling*: `while(isbusy(device) { sched_yield() }`. hands over command to the kernel. still not optimal, because it doesn't scale; if there are 1 thread working and 999 threads all polling, the polling threads will just keep passing control around between them and immediately exiting.
  - *blocking*: don't even return from the syscall until the device *is* ready. each thread descriptor gets a special bit that specified if the thread is busy or yielding. yielding threads don't transfer control to other yielding threads. you already know examples of this: `write` blocks until the write works.

2. *preemptive scheduling*: the kernel occasionally preempts the threads, forcibly taking control of them and running the scheduler.
   - dealt with using timer interrupts. the motherboard hardware has a comparatively low-frequency clock that activates a special pin on the CPU. this serves to give control of the CPU to the OS.
   - necessary; because of the halting problem, we can never know if a program's in an infinite loop. ideally, this is a fallback.

most life, most operating systems rely on 1, resorting to method 2 when enough time passes. there are some applications where this isn't the case, mainly dealing with 'realtime scheduling'. with *hard* realtime scheduling (such as embedded chips), missing a scheduling deadline is a loss in correctness, and predictability trumps performance. because of this, these applications disable caches, and use polling, not interrupts (can't wait for the hardware clock to tick). with *soft* realtime scheduling, some deadlines can be missed. if there's CPU to burn, an 'earliest-deadline-first' policy may be used. this doesn't work well under high loads (one late assignment makes the next one late, etc). if not, we use 'rate-monotonic' scheduling, in which every thread gets a fixed quota of CPU (5% for one second), and just has to deal with it. most operating systems combine these and use *priority*

scheduling, in which there are separate queues for different processes. each queue can use a different scheduling method, but the overall choice of queue is tiered (eg. root gets priority).

scheduling metrics

- response: the time it takes between your request and when the computer starts to execute.
- turnaround/latency: the time between your request and a response.
- throughput: the rate at which you do your work/use your resources.
- utilization: the percentage of time that your CPU is doing useful work.
- fairness: resource starvation for a thread is impossible.

scheduling methods

single-cpu policies

assume there are 4 jobs, a, b, c, and d, that arrive in a certain order and run for a certain time. assume every context switch takes 'a' (TODO: change to delta) time, and assume there's no preemption.

job	arrival time	run time
a	0	5
b	1	2
c	2	9
d	3	4

what scheduling methods can we use?

- first come first served. pros: this favors long jobs, because it means fewer context switches. it's fair, because starvation is impossible (assuming finite runtimes). cons: suffers from the convoy effect (the entire process will slow down due to a few slow threads, just like a convoy held back by the slowest individual).
- shortest job first. this assumes approximate thread runtimes are known. pros: average wait/turnaround time is better than FCFS. cons: unfair, resource starvation can happen for long processes.

what if we allow preempting threads?

- round robin. this is like FCFS, but uses preemption. once a thread is done with a quantum, we forcibly stop it and context-switch to the next thread in the queue. we keep going until all jobs are done. pros: wait time is by far the best. the policy is fair if new jobs are added to the end of the queue. cons: overall utilization goes significantly down due to the frequent context switches.

more generally, we have a large category of scheduling methods that fall under 'priority scheduling'. in this, jobs are given priorities. in unix, processes have a property called 'niceness', an integer ranging from -19 to 19. higher priority jobs have lower nice numbers (aka they're not as nice to other processes). only root can lower niceness. because assigning priorities only based on niceness isn't fair (because it leads to resource starvation). most common operating systems use it as a priority *factor* rather than an end-all. for example, `niceness - time since last run` is often used as a priority value. in this way, the scheduler is 'dynamic'; priorities change as time goes on.

multithreading

writing robust multithreading programs is hard. even when a machine instruction *looks* atomic (eg. `addq %rax, balance`), it can be implemented by microinstructions that can be interrupted by a change in the `rax` registers! messed up synchronization can happen with

1. two CPUs running simultaneously
2. one cpu with preemptive scheduling
3. signal handling (which, remember, is asynchronous)

critical section: a sequence of instructions that have to be executed together, without interference from another thread or signal handler. they're hard to recognize; how do we find them? one method is to make every object-oriented method a critical section; each object has a 'lock' that prevents other threads/handlers from running. this works just fine, but when you deal with more than one object, the locking gets complicated. alternatively, we can think of each thread as a sequence of actions. we can try to guarantee that certain actions are isolated; they're *atomic* actions, which can either be 'not started yet' or 'done', with nothing in between.

problems with critical sections:

- enforcement of critical sections
  - mutual exclusion: one thread in a critical section excludes another from entering.
  - bounded wait: a thread cannot starve waiting to enter a critical section.

the above two problems are only hard if you have preemption or multiple CPUs. otherwise, we can just block signals (proofing us from signals) and block the timer interrupt (proofing us from preemption). how else can we make sure threads work nicely together?

synchronization

this is hard, because threads are by default unsynchronized. how might we implement synchronized behavior?

0. only deal with single threaded code.
1. event driven programming: 1 CPU, many threads. avoids the need for synchronization to begin with, in a way.
   - your program consists of a loop that waits for input. in response to an input, an 'event handler' is executed. the event handler should be fast to avoid thread preemption, and shouldn't require waiting on other threads: this enables the entire event handler to be put into a critical section.
   - pros: simple to implement, works great. cons: forces you into an event-driven program architecture, not parallel (multicore CPUs aren't really used). we can split our program into 2 event-driven processes to deal with the multi-core issue, but it's still a hack at the end of the day (plus, IPC is more of a pain).
2. load-store synchronization: pass messages between threads by writing to/reading from a location. this is problematic, because you don't know what machine instructions and library calls are atomic.
   - we can't use *large* objects (such as storing/reading strings), because threads could be preempted while writing to the shared location.
   - we can't store small objects, such as bit fields, because they could be stored inside 1 register or other machine alignment field. this means that the entire block of bits could be read/written at once, leading to invalid state. check out the following for an example of a bit field stored in a single 'block' (aka unsigned int):
     ```
     struct v { unsigned int bit_a: 1, bit_b: 1, bit_c: 0; }
     ```
   - for x86_64, writing objects with an alignment of 4 or 8 bytes is atomic, but for other architectures, it might not be. this means that if, on an x86_64 machine,
     ```
     a = 1;   // initial setting
     a += 10;  // in thread 1
     a += 100; // in thread 2
     ```
     a will be either 11 or 101; no 'jumbled' value can be output (as might happen with string interleaving).
   - we must be wary of compiler optimizations; most compilers will assume consecutive reads of a variable will yield the same result. to mitigate this, C gives us the `volatile` keyword, which signals to the compiler that a variable may asychronously change value at any time. it's used for read-write coherence-based synchronization.

using these primitive methods of sychronization (primarily read-write coherence based synchronization), we can make more complex atomic operations. let's write some thread-safe syscalls.

```
struct pipe {
    char buf[BUFSIZE];
    unsigned int r, w;
}
```

```c
// |-------r=======data========w-----|

typedef volatile int lock_t;

int readc(struct pipe *p) {
    lock(&l);
    int r = (p->r == p->w) ? -1 : p->buf[p->r++];
    unlock(&l);
    return r;
}

int writec(struct pipe *p) {
    lock(&l);
    // unsigned arithmetic guaranteed wraps around, modulo 2 to the n.
    // as buffer size is also a power of 2, this will always work.
    int r = (p->r == p->w) ? -1 : p->buf[p->r++];
    unlock(&l);
    return r;
}
```

the problem with locking is that eventually, we have to hope that reading/writing to a `lock_t` is atomic. if we do something like this,

```
void lock(lock_t *l) {
    // wait for another thread to release the lock.
    // (1 is locked, 0 is unlocked).
    while (lock == 1)
        continue;
    *lock = 1;
}
```

we could have a problem. if both threads call `lock()` at the same time, both will acquire the lock. we need some kind of atomic way of setting something.

x86_64 hardware engineers were asked to implement an atomic machine instruction. they gave us `xchgx %eax, (%rbx)`. this atomically exchanges `%rax` (in this example) with `mem[%rbx]`. it's guaranteed to either succeed or fail. it's atomic in 2 ways: ... TODO what?...

we can use this to implement an atomic exchange function!

```c
int xchgl(int *p, int v) {
    // asm is equivalent to:
    // int t = *p;
    // *p = v;
    // v = t;
    return asm("%xchgl %rax...etc etc etc");
}

// now we can use the function to implement a better lock.
void lock(lock_t *l) {
    while(xchngl(l, 1) == 1)
        continue;
}

void unlock(lock_t *l) {
    *l = 0;
}
```

this lock is commonly called a *spinlock*; the CPU 'spins' idly until the lock is released. we can improve our

implementation; our implementation is *coarse-grained*, locking every pipe in the system if even one pipe is being read from. we can eliminate that bottleneck by making the locks pipe-specific: just add a lock to each `pipe` struct, making the locking mechanism more *fine-grained*.

aside from controlling granularity, we can increase thread performance by what eggert calls the *goldilocks principle*; each critical section should have as little code as necessary, but no less (critical sections are, of course, bounded by locks).

TODO: copy code in from ur pictures

```
// improved; now, only the while-test is inside the critical section
void do_cosine(struct s *p) {
    do {
        double d0 = p->di;
        double d1 = cos(d0);
    } while (!casd(&p->d, d0, d1));
}
```

this works because our double was small enough to be locked by a hardware primitive (wrapped with casd — aka 'compare and swap double'). for larger objects, we need to manually add locks.

we can shrink the scope of locks by differentiating read and write locks. we can allow: n>0 readers (with no writers) or 1 writer (with no readers).

```
typedef {
    int readers;
    bool writer;
    lock_t l;
} rwlock_t;

bool readlock(wrlock_t volatile *p) {
    lock(&p->l);
    // note: '->' takes precedence over '&': the following is a pointer to
    // a lock.
    if (p->writer) { unlock(&p->l); return false; }
    p->readers++;
    unlock(&p->l);
    return true;
}
```

blocking mutexes

we can do better than spinlocks; if we can't acquire a lock, then we should put ourself to sleep to save CPU time. to do this, we can use a *blocking mutex*; let's implement one below.

```
typedef struct {
    bool acquired;
    // this forms a queue of blocked threads.
    threaddescriptor_t *blocked_list;
    lock_t lock;
} bmutex_t;

void acquire(bmutex_t *b)
{
    while(true) {
        // we're essentially using a spinlock to protect a smarter lock;
        // the thread queue.
        lock(&b->lock);
```

```
        if (!b->acquired) {
            b->acquired = 1;
            unlock(&b->lock);
            return;
        } else {

            // let's change our thread descriptor to mark ourselves as
            // 'blocked'. this way, the CPU won't wake us up and
            // immediately put us to sleep. this will make sure the kernel
            // prefers to wake up threads doing useful work.
            self->blocked = true;

            add_self_to_blocked_queue(b->blocked_list);
            unlock(&b->lock);
            yield();
            break;
        }
    }
}
```

semaphores

we can make a better locking primitive; a *semaphore*, pioneered by (E.W Dijkstra (*ijk*, like matrices)). semaphores generate blocking mutexes to allow no more than n threads to run simultaneously. it's 'locked' when the thread counter is 0, and 'unlocked' when the counter is $0 <= counter <= n$. we can implement this pretty easily; just change the `bool acquired` in the above `bmutex_t` definition to an `int acquired` starting at `n`, and replace `b->acquired = 1` with `b->acquired--`. you get the gist; just allow for n number of threads instead of just one.

conditional variables

blocking mutexes aren't sufficient to implement `pipe`, `getc`, `putc`, etc? no! if we're waiting to read from a pipe, we don't care if another thread is also reading to a pipe; we should just wait until somebody *writes* to the pipe (in which case we can try to read again). this need for conditional locks led to the creation of *conditional variables*. to use them, we need:

- a boolean condition defining the variable (in the designer's head)
- a blocking mutex that's protecting the expression

the API for conditional variables is as follows (assume `condvar_t` is a conditional variable):

1. `wait(condvar_t *c, bmutex_t *b)`. precondition: `b` must be acquired, so way the condition testing will work out without any race conditions. `wait` releases lock `b`, then blocks until some other thread says that the condition is true.

TODO: copy in from pictures.

hardware lock elision

this avoids most of the spinlock bottleneck by exploiting modern hardware support for *speculative execution*. using the intel x86_64 TSX extension, we can use the `xacquire` prefix. remember, the `lock` prefix makes the next instruction atomic.

```
lock:
    movl $1, %eax
again:
    # xacquire *assumes* that we will not (TODO: will?) take the jump below the
    # comparison, and will go straight to the return, running speculatively
    # (doesn't check for lock, doesn't notify other CPUs, etc).  when the
    # computer realizes it *doesn't* have a lock, it rewinds to the step below.
```

```
    # this effectively gives us a fast spinlock that's atomic.
    xacquire lock xchgl %eax, l     # l is address of our 32 bit int mutex
    cmp $0, %eax                    #
    jnz again
    ret

unlock:
    xrelease movl $0, l
```

deadlocks

we can use locks to prevent race conditions. however, locks are also subject to race conditions. for example, assume two resources, a and b. if thread 1 acquires a, thread 2 acquires b, then thread 1 tries to acquire b, and thread 2 tries to acquire a, both threads will wait forever. deadlocks are hard to deal with, but are subject to four main conditions:

1. circular waiting: each thread waits for another thread's resource. if no 'loop' of threads can form, no deadlock can form.
2. mutual exclusion: a resource can only be accessed in a non-shared way. if we have no critical sections, then a deadlock can't occur.
3. no preemption of locks: lock preemption involves forcibly taking a lock from another thread, then alerting the thread via a signal/killing the thread. it's complex and frowned upon, but it does help avoid deadlocks.
4. hold+wait: a thread can hold one lock while waiting on another. if this *isn't* possible, you can't possibly have a deadlock.

how can we solve this?

1. dynamic deadlock detection: any time you acquire a lock, the OS looks through all locks and looks for cycles. if it finds a loop, it tells you that the acquire failed. this is expensive, but is done; in linux, the kernel will set `errno` to `EWOULDDEADLOCK` (but then again, many programmers don't bother check the result of pthread methods; this means the mathod isn't super nice to use).
2. lock ordering: take all the locks in your system, and order them (typically by mem address). every thread with a critical section must acquire locks in increasing order. threads wait only on their first lock; if any lock afterwards fails, they release all of their locks. this is low tech, but works. however, it pushes the responsibility onto the application writer.
3. manual approach: used too often, not preferred. if two threads hang for long enough, you heuristically kill offending threads.

deadlocks are tricky, and can pop up in all kinds of places (even code without mutexes!). for example, assume a parent process and child process that communicate with each other via two pipes. if both processes are especially talkative, both might fill up their pipes with data, at which point both processes will block, waiting for their writes to complete.

alternatively, priority scheduling can cause problems (such as one that took out a mars rover). assume we have three threads with three priorities, T-low, T-medium, T-high. if the low priority task TODO the standard solution is with 'priority inversion'; a low-priority thread becomes high-priority while it's borrowing a lock that a high-priority thread needs.

we can also run into a deadlock variant called a *livelock*. suppose we have a router that can retrieve packets. if it receives an interrupt, it will accept the packet, and store it into a buffer. if the buffer is full, it will discard the packet. what we will find is if we receive a large number of packets, the work done goes down. why? because the interrupt priority has a higher priority than the packet-processing mechanism, we spend all of our CPU time accepting packets into our buffer (then later throwing them away). this process of being overwhelmed is a form of livelock. the standard solution is to keep the interrupt scheduler at high priority normally, and to make it low priority if the system is reaching peak capacity (we won't receive some packets, but hey — we were just going to evict them later anyways if we didn't have time).

file systems

file systems are data structures on primary and secondary storage that store and find blocks of stream data. in this way, they're a specialized subset of databases.

I/O performance metrics

- throughput describes total requests per second for the whole file system (reads, writes, etc).
- latency describes the delay between the I/O request and the response.
- file system designers also have to worry about utilization, which is the fraction of storage capacity that's doing useful work.

improving I/O performance

we mainly use speculation: the OS guesses what I/O will be done. the storage hierarchy gets slower as we go down (from registers to disk to network, etc). to speed things up, we can exploit locality of reference, in which we take advantage of patterns in how programs use data. this comes in two main forms: *spatial* locality (in which accessing `a[i]` means there's a likely future access of `a[i+1]`), and *temporal* locality (in which accessing `a[i]` at time `t` means there's a likely access of `a[i]` at time `t+1`.

- **prefetching**: the OS guesses a later read, and caches the memory block in a smaller, faster, memory tier (such as a cache).
- **batching**: when we fetch `a[i]` into cache, we also pull in the surrounding block of data.
- **dallying**: when asked to write a block, the OS caches the write, hoping the user will write to the memory block again. after waiting, it writes the block once as a unit. this causes some complications, as we can't guarantee that we've actually written to disk (causing problems in the case of a power outage). to deal with this, we have a few system calls:
  - `sync()`: this is an older, uglier call. it writes all files on the filesystem. because this would take a long time, it doesn't actually block until the writes have finished. to mitigate this, we have:
  - `fsync(int fd)`: this writes the specified file descriptor to permanent storage. waits until the file is written; this can take some time. we have an even newer call to deal with some complicatedness.
  - `fdatasync(int fd)`: this only syncs the file data, not any associated metadata, such as modification time, etc. was included for use in databases, because synchronizing these things ends up being complicated.

FAT file system

FAT = file allocation table. designed by microsoft in the 1970's, very simple.

```
|1|=2=|===3===|=================4================|
```

1. empty: system was optionally bootable, so one sector was left empty for the MBR.
2. superblock: metadata for filesystem. described filesystem size (in numbers of blocks), version number, etc.
3. file allocation table: array of 16 bit block numbers. block numbers could be:
   - 0: end of file
   - -1 (0xffffff): unused block
   - anything else: index of next block in the same file. in this way, the FAT is a linked list in disguise. this level of indirection lets us expand files relatively easily without running into fragmenting.
4. file data. each sector contains as much data as it can hold.

some files are directories. directories are arranged in a tree, and the index of the root directory is the superblock. each directory lists files, with an 11 character filename and pointers to the first data block in a file (from which we can follow the linked list to get the rest of a file).

because this isn't very efficient (especially with `lseek()` — how would we seek backwards through a linked list?), traditional unix took a different approach.

unix: ext3 file system

file system format:

```
|1|=2=|==3==|====4====|==========5=-=-=-=-=-=|
```

1. empty. see 'FAT' section of notes.
2. superblock. see 'FAT' section of notes.
3. a block bitmap: 1 bit per data block in the file, that tracks whether a block is in use or not.
4. inode table: full of inodes, each of which is much smaller than a block. 1 fixed-size inode is stored per file, and each contains metadata about one file.

directories

each directory contains a list of names and corresponding inode numbers (in fact, directories are the only place we'll find file names; they are the only way to find a file — literally 'directories'). the superblock contains the inode number of the root directory. each directory entry was similar to FAT, but has been improved since ext3v2:

```
|=1=|=2=|=3=|=4=|==5=========|
```

1. 32 bit inode number.
2. 16 bit directory entry length, lets us expand dir entry length.
3. 8 bit file name length
4. 8 bit file type
5. file name. the name doesn't take all of the remaining space; there's unused space as well (that can be used for caching inode numbers, etc).

now, if we want to delete a directory entry, we can just expand the entry before it; if we want to create another one, we can just remove some of the empty space in part 5, and insert a new directory entry inside. because dir entries are variable length, looking through a directory necessarily takes *O(n)* time.

inodes

inodes can't use this approach; they must be a fixed size for easy indexing. they are structured as follows:

```
|=1=|===2===|
```

TODO: copy inode data from pictures (indirect inodes, double indirect inodes).

the inode concept provides a layer of indirection that has some pros and cons, and mixed results.

mixed results:

- files can exist even when their directory entry is deleted; the file doesn't go away until the last process with an open file descriptor to that file disappears, so processes reading from a file that's being deleted will still keep reading (the file is briefly nameless). we need to keep track of how many files are open by *refcounting*; each inode has a field that keeps track of the number of dir entries + the number of open files pointing at the inode. when we open/link to a file, it increments the number, and when we unlink a file, we decrement the number. when the number of references to the file reaches zero, the file is deleted (same as refcounting in garbage collection). a *hard link* is just two directory entries pointing to the same inode.

pros:

- this primitive system allows for a level of 'compression' with holey files (aka files that are mostly null bytes). any blocks in the inode that are all zeroes can point to the same all-zero block, meaning we don't actually need to fill a large portion of disk with zero bytes.
- renaming/moving a file is more robust.
  - in FAT, to move a file, we have to delete it from 1 directory and then write its block numbers into another directory. if the power cuts in the middle, the file is gone. if, instead, we add the page numbers to the next directory and then delete it from the old one and the power cuts in the middle, we have two files pointing to the same blocks. both will affect each other, and if one is later deleted, the other will be pointing to unallocated blocks (danger!).

  - with inodes, we first `link("d1/a", "d2/b")`, then `unlink("d1/a")`. if the power goes out, worst case we just have two hard links to the same file (two dir entries pointing at the same inode). this doesn't corrupt the system.

cons:

- it's hard to get back to a filename given a certain inode; we'll have to search through all directories and compare inode numbers.
- the overhead of filename -> inode lookup can get heavy, because we have to traverse many directories to get to the end inode (imagine if we had to do this for every system call involving a file!).
  - this is why the `open()` syscall exists; it takes care of the inode lookup once, then returns an integer (file descriptor) that refers to the kernel's open-file-table entry; this entry points at the inode, meaning using the file descriptor instead of a path doesn't require expensive inode lookups.
  - this is also one reason the current working directory exists. the kernel's process table entry includes the inode of the current working directory (for resolving relative paths) and the root directory (for resolving absolute paths).
- if we allow linking multiple directory entries to an inode, we could get loops in our directory structure that would be difficult to detect.
  - this would make many programs misbehave, so unix forbids hard links with directories; directory inodes may only have one filesystem entry.

this last point gives us a problem. we would like a way to point files at other files to allow for complexity in our directory structure. unix designers implemented **symbolic links** to solve this problem. a symlink is a file whose contents are a file name; this file name will be resolved to an inode. this doesn't have the same problem as hard links; because symbolic links are a specific file type as opposed to just two normal files that happen to share the same inode, programs can *choose* if they want to follow symlinks, allowing them to avoid dealing with loops if they choose. because symlinks have to be resolved every time the link is used, they're relatively expensive, but they're convenient. symlinks also have problems; they can point to a nonexisting object (a dangling symlink), or cause recursive problems. to solve this, `namei` (the OS's default way of turning filenames into inode numbers; it will expand the *name* recursively until it gets to an *i*node) only expands a certain number of symlinks until it gives up (usually 20) and assumes the problem is recursive, returning `errno` as `ELOOP`. hard links aren't an explicit filetype, so it's hard to count them (you'd have to search every other file on the system to see if one had the same inode); that's why we can't do the same for hard links.

more file types

we've already discussed directories and symlinks. there are a few more interesting filetypes.

- **FIFOs / named pipes**: listed as `p` in `ls -l` output.
- **character special devices**: listed as `c` in `ls -l` output. these work as unbuffered windows into device drivers drivers. they serve as a way to interface with hardware, just as we used PIO when we were discussing bootloaders. to create, eg. `mknod /dev/ser1 c 59 27` (the last numbers are arbitrary and hardware dependent; they're how to interface with the hardware and are provided by the vendor).
- **block special devices**: listed as `b` in `ls -l` outout. these work similarly to character special devices, but are buffered. they often restrict writes to the block size of the device.

multiple filesystems

how do we deal with multiple filesystems? the naive approach is to line them up side by side, a la windows. eg. `A:/etc/passwd`, `B:/etc/passwd`. unix has only one 'root' filesystem, and maintains a **mount table**, which keeps track of what filesystems are mounted at what place (mounting attaches the base of one file system to a directory in the root file system so the files can be accessed from the root fs). this gives us a problem; before, files could be identified by their individual inodes. inodes aren't necessarily unique across devices, so we must have a 'device number' as well to specify our currently looked-up file; `dev_t` as well as an inode number `ino_t`. the mount table maps inodes to device numbers; when we're using `namei` to resolve a name, we look up the name in the mount table. if it's found, it must be a device root; we'll resolve the later file name components in the new device specified in the mount table.

this mount table lookup is fairly expensive, because it's needed for every name component. we stick with it (as

opposed to adding a device num in the directories themselves) because it makes mounting more modular; we don't have to un-entangle filesystems when unmounting; we can just remove them systems from the mount table.

file systems can have different types. linux deals with this by using OO programming in a system dubbed the 'Linux VFS (virtual file system)'.

TODO: copy in VFS diagram

tldr: levels of a linux filesystem

software-level structures:

1. symlinks
2. absolute+relative filenames (namei, etc)
3. file name components+directories
4. inodes
5. file systems+mounts

hardware-level abstractions:

6. partitions
7. blocks
8. sectors/pages

problems: doesn't deal with network file systems, file systems that span partitions, or RAID arrays. the fact that we have these many abstraction layers makes overriding them tricky; for example, if we're trying to securely `shred` a file. even if we `shred /dev/sda`, the flash memory's control unit will have its own internal algorithms designed to wear-level (mitigate the wearing out) of drive sectors; to override this, we need to give the controller the `TRIM` command, which writes everything over with zeroes, or the `SECURE ERASE`. this is, however, subject to the implementation of the device, and isn't always secure.

file system scheduling

hard disk drives have to move their head between different blocks on a disk. let's say the cost of a read/write is proportional to the distance between the head and the place to read/write.

- **shortest seek time first (SSTF)**: finds the request closes to the current head. best throughput, but can lead to starvation.
- **first come first served (FCFS)**: fair, but bad throughput.
- **elevator algorithm**: SSTF only in one direction. when you hit the end, reverse the direction (like SSTF with a cam, aka an elevator). has good throughput and fairness, but latency will be much less near the middle of the disk (points near the middle will get swept up by the elevator on the way up *and* down).
- **circular elevator**: when the elevator hits one end, make the read head 'wrap around' to the other end. don't read anything on the way back; treat it like a loop. it's less efficient, because the seek back to the start of the drive takes time, but it's more efficient.

these algorithms are optimized for disks. more generally, we can dally (as with caching) our writes, allowing for writing sequential blocks. this is also good for SSDs.

file system robustness

in this class, we're dealing with robustness in the case of power failures. we want durability, atomicity, and performance. first, let's define problems:

- failure: problem at runtime, eg. power loss.
- fault: latent problem in HW or SW.
- error: problem in the head of the user or programmer.

modifying data

golden rule of atomicity: never overwrite the only copy of your data.

note that copying is a bit difficult; if we accidentally end up with two copies of our data, we should know *which* one to choose after a power failure (the old version or the new version). we shouldn't have to rely on heuristics.

if we assume atomic writes of a single block, we can build a file system based on this assumption (for example, we can copy a to b, putting a bit in block c that tells us which file is correct in the event of power failures that lead to duplicate files. this is true for some hardware devices, but not others. even with an unreliable device, we can make atomic writes using 3 blocks, as follows:

assume we have 3 blocks, and are trying to write B into an A block.

```
block   data over time
------  --------------
0       A ? B B B B B
1       A A A ? B B B
2       A A A A A ? B
```

algorithm: if interrupted, take the best of 2/3 blocks, and assume that's the copy you should have. if all three disagree, choose what's in block 0.

TODO: copy in more pics.

modifying filesystem structures

now, we know how to copy the data of a text file. how do we deal with renaming *filesystem* data, such as directory entries?

what if we lose power while removing the old entry? our solution is to be very conservative; while we're making the temporary 'hard link', we make sure to increment the link count on the old file. now if we lose power, the worst case is that our directory/file's link count is out of sync with the filesystem, and now that inode will never be deleted (if there's 1 directory pointing at it, it'll think there are 2, etc). to deal with this, we'll have to periodically manually walk through all of our directories and manually update every single inode's count (using `fsck`, the filesystem-check tool). `fsck` is a bit of an ugly solution though; ideally, our inodes and directories should never get out of sync in the first place. how can we make our filesystems more robust without periodic maintainance/resyncing?

- a **commit record**: take a leaf out of the filesystem and use a third storage location. assume the rename didn't actually happen until another (commit record) block is written. however, this leads to more writes and more storage space use.
- a **journal**: like a commit record, but records everything into a well-known location in the filesystem. if we have to blocks write *A'* and *B'* to disk, we first note we're "writing *A'*" and "writing *B'* into the journal. TODO: expand on journaling file system. look up what cell storage is.
- **write-ahead log**:
    1. log planned actions in journal
    2. commit in journal
    3. install new values onto disk
    4. mark done in journal
- **write-behind log**:
    1. log old values in journal
    2. install new values onto disk
    3. commit in journal

file systems: scheduling vs. robustness

if we store writes in our journal, they may not be written in the correct order. even if we *do*, our disk-management chip may very well cache the write. this is a problem if we have writes that depend on each other for robustness. this optimization clashes with robustness; while we can bypass the chips and OS scheduler, it slows us down unacceptably. the solution is to write down write dependencies; the OS I/O scheduler respects these dependencies when writing (although it does mean we need a more complex I/O device interface).

virtual memory

overview

in virtual memory, programmer doesn't manage memory directly. the OS maintains a 'page table' for each process, mapping each address in 'virtual memory' to a certain hardware memory address. sometimes, the hardware memory address may not even exist, in which case the OS generates the memory 'page' on demand (either reading from disk/swap).

- frees the programmer from worrying about where physical memory is.
- prevents processes from accessing each other's memory.
- lets processes/threads share memory.
- lets virtual memory be larger than physical memory. this point isn't entirely true. fetches from memory will require fetches from exponentially slower disks. this constant page-fetching bogs down the computer, essentially freezing it in a phenomenon known as **thrashing**.

alternatives to virtual memory:

0. hire better programmers.
1. enable subscript checking, null pointer checking, etc. in the software. this is popular in languages like java, python, javascript, etc. essentially, we let our software deal with the memory.
2. base+bound registers (simple - only 2 registers): hardware support for bounded pointers, where each register can only modify hardware memory between two bounds.
   - assumes continuous memory
   - prone to fragmentation
   - each program must relocate every time it runs; it can't share memory.
3. segmented memory. memory is given in distinct segments: each address consists of a segment number and an offset in the segment. the hardware maintains a segment table; the memory is divided into 'segments' of different sizes, each of which can be given to a program. programs can't access other segments, because their addresses consist of segment offsets. when given a different segment number, programs should still work. however, this still assumes continuous entries, and is prone to fragmentation (still deals in real memory).

virtual memory lets us map other objects to other memory objects.

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
```

- maps a regular file or device object into virtual memory (creates a page table entry into the file), and returns a pointer to the place.
- `int prot`: defines the permissions (rwx or none) in that page table entry. the options implemented depends on the hardware of the machine; the execute bit is a "high cost" option hardware wise, so may not be implemented for some devices.
- `int flags`: defines whether memory should be shared between processes, etc. eg. `MAP_SHARED`, `MAP_PRIVATE`, etc.
- `int fd` most commonly is `/dev/zero`, used for lazily allocating large zeroed-out blocks of memory.

there's a lot of nuance to `mmap`: it can be used to share virtual memory between processes, can be used to expand virtual memory, dynamically link libraries by loading shared libraries into memory, and much more. consider printing out the manpage for the exam.

page faulting

sometimes you lack permission to read from a page. sometimes the page table entry is invalid. the hardware treats this as any other trap; it interrupts the program and switches to the kernel. what should the kernel do?

1. terminate the process
2. we could send a signal to the program while also messing with the program's stack, deliberately adding a 'signal handler', changing `$rip` and `$rsp` registers as needed. this is somewhat useful (say we run out of stack space. this would cause a page fault (known as `SIGSEGV`, as a relic from the era of segmented memory). our signal handler might include a call to `mmap` that expands the stack, fixing our problem. this

is, however, a hack at best; if we cause an error by some other reason (eg. access a dangling pointer), the handler will *still* expand the stack, not fixing our problem. it's a coarse approach at best.
3. the kernel claims the page table access *was* valid; it updates/fixes the page table entry, and then re-executes the offending instruction in the client program.

option 3 sounds fairly useful. however, once our physical memory fills up with pages, we've got to find some way of deciding what page to evict from memory as we bring in a new page. we want to evict the page that will be used the farthest in the future. this means we need a *page replacement policy.*

page replacement policies

notationally, we have *reference strings* to represent the sequence of page numbers accessed by a program.

first in first out (FIFO): exactly what it sounds like, the kernel keeps a table of when each page was brought into RAM, and evicts the last one. subject to a number of problems, such as *belady's anomaly* (where for many page replacement algorithms, adding more physical memory actually causes *more* page faults than if your program had run with less memory).

least recently used (LRU): the page that hasn't been touched for the longest time is evicted. this is *usually* more efficient than FIFO, especially for realistic reference strings. however, implementing this is a problem; the kernel doesn't know when a page is accessed (the CPU only switches to kernel mode when it recieves a page fault). there are a few ways to address this:

- put a clock value in each page table entry (PTE), and update it in hardware whenever the page is accessed.
- the kernel, every now and then, deliberately marks all PTEs as being invalid, then uses the resulting sequence of page-faults to maintain 'clock values' in *software* (not in the MMU).

page replacement mechanisms

TODO: include diagram about physical page to virtual page/swap (you have a pic). heck maybe just embed these in the pdf because this is going to suck to transcribe.

```
int swapmap(int process, int virtual_address) {
    // returns the disk address associated with the virtual address.
    // if the virtual address isn't given to the process, fail.
}

void pfault(int proc, int v_addr, int accesstype) {
    if (swapmap(proc, vaddr) == FAIL) {
        // send segfault or kill process
        kill(proc);
        return;
    }

    // use our removal policy to choose the pages to be evicted.
    int evicted_proc, evicted_v_addr = removal_policy_method();
```

TODO

optimizations around paging:

- **demand paging**: don't load *any* page into ram until a page fault occurs for it (empty page table at start). bring in the current page (c startup routine), then bring in other pages as needed (the main subroutines, etc). this is good in many cases (eg. if a 1000 page program loads, finds an invalid argument, then exits, we don't have to load all 1000 pages), and inefficient in other cases (eg. the program needed all of those pages; we could have copied them efficiently, in bulk, but instead we did it one block at a time).
- **dirty bit**: each page table entry is given a bit. 0 means the page is identical to the contents of swap, 1 means that the page has changed (been written to). this means that different processes can share pages without a dirty bit. this can be implemented in hardware, or in software using a technique similarly to the one described in the LRU policy method:

  - when a `rw-` page is first loaded, it's *always* given only read permissions, and can be split among programs as needed. if a write leads to a page fault, the kernel sets the software dirty bit, changes the page to `rw-` permissions, then returns.
  - this can be used to implement is copy-on-write, in which page tables aren't actually duplicated until they're written to (for example, a `fork` doesn't actually duplicate physical memory at first, making it a comparatively cheap operation. this helps in fork+execs, where there's no point cloning memory that's immediately going to be replaced).

drive failures: RAID

computers are subject to many hazards: power loss, drive failures, user error, configuration error, OS errors, cosmic rays, etc. we've already discussed power loss when discussing file-system robustness. now, we're going to mainly focus on drive failures.

drive sectors fail relatively often, so drives usually contain replacement sectors that are swapped in by the drive management chip. virtualization was an effective tool against memory failure; we can analogously virtualize physical storage devices into a virtual devices. one of the first ways of doing so was: **RAID** (redundant array of inexpensive/independent disks). was originally cost saving, but turned out to have a lot more applications.

- RAID 0: lash together two drives into one 'virtual' drive.
  - we can concatenate drives into contiguous data, (1-N blocks on one device, N-M blocks on another device), or...
  - we can *stripe* drives (consecutive blocks alternate between devices). this gives us faster contiguous I/O, as we can parallelize writes to adjacent portions of storage (as most writes are). each high-level write is really two lower-level writes.
- RAID 1: pair two drives with identical data. now, we now have no single (drive) point of failure.
  - however, this isn't very efficient, and doesn't scale beyond 2 drives. after a few revisions, berkeley developed a new version.
- RAID 4: store `d` drives and one parity drive. no single point of failure, easy to grow, but parity drive is a read/write bottleneck.
  - if we have drives `a`, `b`, `c`, and parity drive `p` the parity drive contains the data `a xor b xor c`. if we want to write something to just `b`, we don't need to re-read all the drives, either: `p = p xor b` (think about it. xor is cool).
  - if a drive fails, we can use the parity drive to reconstruct the data (or if the parity drive fails, we can just recalculate its contents).
- RAID 5: tries to eliminate the parity-drive bottleneck in RAID 4; it's like a bunch of RAID 4 drives 'stacked on top of each other'. each drive is divided into blocks; every block, the parity drive rotates (for block `a`, it's in drive `4`. for block `b`, it's in drive `1`). now, there's no single write bottleneck. analogous to striping RAID 4.
  - however, it's difficult to grow the system later on; it'll require a lot of rewriting if you ever want to expand the space.

however, RAID assumes that when a drive fails, it's repaired before the next drive fails. if another drive fails during the drive rebuild process, we're screwed.

note that operator error is almost always the reason for data loss. if a system is supposed to survive a drive failure, deliberately pull out a drive just to make sure the system actually works.

TODO: copy in the metrics (mean time between failure) etc he mentioned today. URGENT.

distributed systems and RPC

look at xorg.

remote procedure calls (RPC) differ from ordinary calls.

- caller and callee don't share address space. pros: hard modularity. cons: no pass by reference. we can only send raw data.

- caller and callee might be on different architectures, so might degree about data-type lengths, endianness, etc.

we can overcome these problems by **marshalling/serializing/pickling**, in which we convert an internal data structure into a string of bytes. examples of protocols for doing so include XML, JSON, or an IEEE 754 bit pattern.

RPC robustness issues

RPCs are subject to different forms of failures as well (and are immune to others).

- neither side can trash the other side's data structures, because no shared memory. this provides some intrinsic protection.
- messages can be corrupted between the client and the server.
  - we can use better chceksums; if corruption is detected, we can resend the request.
- the network might be down or slow, or the server might not be listening. how do we differentiate between the two? what do we do if we call and get no response?
  - if we keep trying, we call our method 'at-least-once RPC' (suitable for idempotent apps, where double requests aren't a problem).
  - if we return an error, we call our method 'at-most-once RPC' (suitable for transactions, where we might want to query our app to see what to do in case of failure).
  - if we keep moving, we call it 'exactly-once' RPC. these names aren't exactly rocket science.

RPC performance issues

marshalling and unmarshalling causes a CPU overhead, and transmission delay is non-negligible. while blocking on calls, our CPU isn't doing useful work. there are a few strategies to avoid this as well.

- asynchronous calls: after we execute a call, we immediately work on the next step, without waiting for the call's response. however, this assumes that calls are independent, complicates failure handling, and runs into problems when the responses might arrive out-of-order.

TODO: copy in pics and diagram yo

distributed file systems

network file system (NFS)

NFS protocol v3: a means to access files with TCP/UDP over a network. client process calls the kernel via syscalls such as `read()`, `open()`, etc. the kernel has an internal abstraction called the virtual file system (VFS), as discussed before. this vfs allows for different implementations of these syscalls; different modules handle ext4 file systems, btrfs file systems, and even remote nfs file systems (abstracting away fs differences behind the syscall interface). in this way, the kernel takes care of RPC; if the network goes down, the syscall can return an error (eg. `EINTR`) or just fail. main NFS methods: `LOOKUP(dirfh, name)`, `CREATE()`, `MKDIR()`, `REMOVE()`, `RMDIR()`, `READ()`, `WRITE()`, `CREATE()`. these are mapped by the kernel to the regular syscalls.

the NFS protocol specifies certain ways of dealing with files given a unique NFS file handle. the NFS file handle uniquely identifies a file on a server (it survives renaming the file), and persists even if the server reboots (so that clients will keep working even if the server crashes). it usually consists of a local inode number, a device number, and a *serial number*. the serial number is needed because NFS servers are supposed to be stateless, so the server doesn't know which clients have the file open — if another user deleted the file and reused the inode, we should still detect the new file with a different handle. because of this, each inode has a kernel-internal serial number; every time the inode is created/deleted/reallocated, the serial number increments. this ensures that handles are unique in *time* as well as across devices.

the NFS server caches common lookups and files. this means that NFS reads and writes are not guaranteed to be in sequence or even atomic. remember that on a local machine, the kernel makes sure that `read`s and `write`s appear in the same order, because it handles the cache. now, we have multiple caches to deal with. by convention, the server pauses and syncs for `open`s and `close`s, but this makes them unusual operations (if there's

not enough disk on the server to sync the cache writes, even `close` might fail; and programmers almost *never* check for that. this means that code dealing with NFS tends to have to plan around it.

the original NFS assumes the client-server kernels are trusted. to solve this, the NFS file server creates a dummy account with *no* permissions. any NFS file access uses this account's permission (meaning it can only access world-writable files, etc). now, we can use client authentication (kerberos, etc) to gain more priveleges.

computer security

the real world defends against force attacks (someone comes in and takes what you want) and fraud attacks (someone impersonates you and takes what you want). computer security mainly focuses on fraud attacks. attacks usually affect:

- privacy: unauthorized release of data.
- integrity: unauthorized modification of data.
- service: unauthorized restriction of services (eg. DoS).

how do we accurately deal with 'authorization'? security has some goals:

- allow authorized access (positive)
- disallow unauthorized access (negative)
- good performance.

when the first point fails, it's immediately reported as a loss of functionality. if the third point fails, it's immediately reported; nobody will buy the system. the second point, by its own nature, is seldom reported (malicious actors don't turn themselves in). these are the ones we have to keep an eye on. we do this by **threat modeling** and classification (one of the first things you do when designing a system). what classes of attacks can exist?

- insiders: threats can come from within. buying off the lowest-paid member (eg. janitor) isn't too hard to do, and a good security system should be mindful of this instead of only protecting from external attacks.
- social engineering: we can trick people into giving us their passwords (eg. kevin mitnick).

computer security uses a few key methods to mitigate (although never entirely suppress) these problems.

- authentication: are you who you say you are? dealt with via passwords, credentials, etc.
- authorization: what can you do? dealt with via access control lists, etc. even after a user is authenticated, they can only perform certain actions (eg. permissions, etc).
- integrity: is our data compromised? dealt with via checksums, etc.
- auditing: even if we're not attacked now, have we been compromised before? dealt with via access logs, etc.

correctness and performance. TODO.

authentication

prevents *masquerading* as another individual. for efficiency's sake, is divided into two subcategories.

external authentication is used to admit users 'into' the system. it can consist of: a trusted login agent (usually the OS; beware, this can itself be compromised), passwords or identity tokens, biometrics, multifactor authentication, etc. we can bootstrap authentication; we can use something the user *is* (eg. fingerprint) to give the user something to *have* (eg. id card), which can be used to give the user something to *know* (eg. set a password).

internal authentication: external authentication can be slow, so we keep faster, more temporary forms of authentication inside the system'. in unix, for example, user processes have a specific UID in their process descriptor; this identifies what user spawned them (and can be used to give them the permissions of that user). we'll get back to this in the next section; authentication and authorization go hand-in-hand.

this becomes more complicated when we consider security over a network. if symmetric key, the key can be leaked and/or we become vulnerable to man in the middle or replay attacks (copy the traffic, try again the next day). we can mitigate this with a *nonce*, a piece of 'nons'ense data mixed into a message. this is intended to avoid replay attacks (a sends a random nonce to b, b has to encrypt it and send it back to a to prove that it can

encrypt messages with the shared secret). even then, we need some way to 'mix' the message with the nonce (appending it, obviously, would be easy to break). in practice, we could use a secure checksum function (aka a **secure hash**, eg. SHA) and a **hashed message authentication code (HMAC)** algorithm to combine the nonce and the message. we can improve this with asymmetric key algorithms (you should remember them, if not look them up they're cool), which are typically a lot slower than symmetric encryption. usually, we use public key encryption with a nonce; once we have a shared nonce, we can use it to set up a symmetrically encrypted connection.

covert channels: unintended, eg. check cpu business and send message.

authorization

how do we enforce authentication?

we can give authenticated processes direct access to resources (eg. `mmap` resources into the process' address space). however, this means resources can easily be corrupted due to race conditions or errors. alternatively, we can give programs indirect access to shared resources, and control the ways in which they are used (eg. syscalls to the kernel). the second approach is less efficient, but gives us more robustness, allowing for more neat security tricks when used with authentication.

in unix, for example, user processes have a specific UID in their process descriptor; this identifies what user spawned them (and can be used to give them the permissions of that user). really, the programs have two UIDs; the 'real' UID (aka the id of the user that started the program) and the 'effective' UID (which is used for most file operations and actions). the two can differ, and allow for some level of controlled privilege escalation. we can flip a file's 'setuid' bit, eg:

```
-rwsr-xr-x 1 root root 63K Nov 13 08:58 /usr/bin/passwd
```

now, if a user runs this program, the program will run with an effective UID of root. we have to be very careful what we put in this program (eg. we shouldn't turn on the setuid bit on shell scripts, because the user might run us in a chroot and give us compromised shell utility binaries (eg. cd).

access control lists (ACLs): who can do what action to which resource? say we have a space (x axis: *principal* (security lingo for the 'user', the one performing the action), y axis: resources (eg. file), z axis: actions (eg. rwx)). a standard way of mapping this space is with an ACL (access control list), where each file has a list of permissions allowed for different sets of users. ACLs are controlled by the owner of the file. in essence, resources have metadata specifying how they can be accessed.

note that this can cause some problems users can have *too many* rights. if a user has permission to compile a command and *also* delete their own home directory, a compromised makefile could cause them to delete their home directory (users are all-or-nothing). to rectify this, we can augment ACLs with **role-based access control**, in which users can assume *roles* (eg. 'grader', 'server admin'). access rights are associated with roles (so eggert's makefile won't accidentally delete our grades), and only end users can switch roles.

capabilities: objects (tickets) granting access to other objects. unix file descriptors are somewhat like this, because they can be passed around within a process. more generally, we can take a hard-to-forge description of a file (encrypted) and send it to another process. this token carries metadata; anyone with the token can use it to access the resources. more flexible, but control is harder; a single corrupted agent can pass the tokens to a 'bad' agent.

IT trends

book: 'the rise of serverless compputing'

over 2/3 of most enterprise-level IT infrastructure & software spending is spent on cloud computing.

used to be **infrastructure as a service** (IaaS). a service provider gives you virtual machines on a network. you provide an OS (via a bootable image), configuration, apps, etc. the host machine has a physical host 'OS' running a *hypervisor*, and guest OSes on virtual machines that you get to control.

this has some problems.

- scalability: if you run out of resources, you have to call the provider and buy another server. you have to figure out how to start up the other servers, parallelize the tasks, reconfigure applications, etc.
- fault tolerance and performance guarantees that you have to negotiate, appropriate service monitoring. this is a lot of paperwork.

these might not work well for new trends, such as the *internet of things* (IoT, also known as edge computing). this involves billions of devices (eg. smart thermostat) connecting to servers. to an extent, we can mitigate this via *fog computing* (more 'dispersed' cloud computing; sets up a multilayer cloud, in which tiny, cheap, local servers are set up and interface with nearby devices. when they can't resolve something, *they* forward it to the central server, helping latency and server load).

to scale up, engineers started sacrificing control over more of the software stack with **platform as a service** (PaaS). like IaaS; you still do provisioning, but you need less expertise on the OS, etc. unfortunately, you still have to scale the program.

they upped it even more: **backend as a service** (BaaS). usually used for mobile apps. gives a software framework that runs a mobile client and a backend server. now, you don't even worry about what the program is running.

and lastly, we sacrificed *all* control over where/how the program is running and how to scale with **software as a service** (SaaS). the provider supplies the entire stack on the server side. by necessity, they only provide support for a few applications

however, all these still depend on us buying and thinking of how to interface with servers. providers tried to abstract this entire process away with **serverless** computing.

- hides server usage from the developers entirely.
- runs code on-demand (in their servers) as a part of billing.
- automatically scaled up as workload goes up.
- billed only for running time (ideally; billed separately for persistent data that we store on their cloud, minimum bill to register for the service).

variant: **function as a service** (FaaS). most common form of serverless. each function is the unit of computation (eg. you ask to run a single python function). each one is executed in response to a trigger (eg. HTTPS request suppling arguments to the server). they run for a few seconds to a few minutes (time limited), and have no persistent state (you can get them to talk to an external database server, though). parameters and return calls are sent as a JSON object that encodes the parameters, etc. it's added to a queue on the 'master' server, then is eventually 'dispatched'; a container VM is quickly spun up with some technology like docker. we copy in the function's code into the container, execute the function, then stop/deallocate the function.

TODO: copy in the article's picture.

this adds a lot of overhead; while docker lets the linux kernel boot up fairly fast, the *cold start problem* hurts latency. we can mitigate this problem by pre-booting/pre-allocating 'stem cell containers' (vanilla or flavored; if software requires numpy, we could preallocate a stem cell container with numpy preinstalled, wait for a request to come in, then 'differentiate' the container by loading in the code). we can alternately reuse 'warm' containers; if the function is relatively quick, we can just delete its results and reuse the VM for another incoming request.

how do we create workflows with FaaS? **action + trigger model** for FaaS (action = function). a single event can trigger multiple actions for parallel execution, and an action can create an event that triggers *other* actions for serial execution.

there are some other problems with FaaS:

1. debugging. function execution logs were invented so providers could bill accurately, but they can have some debug info as well. it's not great, but it's better than nothing.
2. fixing bottlenecks. we can't run `ps` because we don't even know where the function is executing. we don't really have tools to analyze the functions.
3. no atomicity guarantees. we're running on some server; how do we know when operations are made? how do we thread?
4. at-most-once functions, etc. if we have a function that breaks, can we change it while it's running?

trusted software

on unix, most software isn't trusted. programs can only do what you could do yourself; they run with your permission (so even if bug, damage is limited). some programs are trusted, such as 'login', sudo, or programs with the setuid bit.