CS140 Final Review

Processes and Threads

- A *process* is a single program (e.g. a web browser).
- A *thread* is a single execution context. There can be many threads in a single process (e.g. each tab in a web browser might be a different thread).
- The different threads of a given process share the same virtual memory address space, but have individual registers and program counters.

Process and Threads

Two threading models:

- Kernel level threads: Creating a new thread is a syscall. This syscall is very similar to producing a new process, but it doesn't need to create a new VM address space.
    - This can be slow because syscalls are slow.
- User level threads (green threads): implement threading in user-mode. Use a timer function (supplied by the kernel) to simulate the timer interrupts the kernel uses to preempt threads. Replace all blocking calls with wrappers that switch threads.

Concurrency

- Sequential consistency is the property that the order in which things actually happen is equal to the order in which they are written in your code.
- Often sequential consistency is not guaranteed in order to allow for easier optimization in the hardware or compiler.
- Races include:
  - race conditions: when the timing or ordering of two threads' operations affects a program's correctness.
  - data races: when two threads both access the same memory location simultaneously and one of them is doing a write.

```
/* invariant: n >= 0 */
atomic_int n = 1;
void decrement() {
  if (n <= 0) return;
  --n;
}
```

```
/* touch() is ever run? */
bool dirty_bit = false;
void touch() {
  dirty_bit = true;
}
```

Virtual Memory

- Use Page directories and page tables to allow user processes to believe they have the whole memory to themselves.
- MMU manages page faults in hardware. On every memory access, it decomposes the address into an offset into a page and a virtual page number and looks up the physical page in the page table structure, which is at a location specified by the CR2 register.
- TLB is used to cache recently used MMU activity so that you don't have to perform too many indirections on each memory access.
- Pages are typically kept on swap when not in use.
- Bringing a page in from swap is very slow. If there are too many processes running, then the computer spends all of its time bringing in pages from swap and nothing gets done (thrashing).

Compiling Linking Loading

- Compiler: one C file in, one object files (.o) out.
  - *Preprocessor*: `gcc -E code.c > code.e`
  - *Compiler*: `gcc -S code.c -o code.s`
  - *Assembler*: `gcc -c code.c -o code.o`
- Linker: many object files in, one executable file out.
- Loader: one executable file in, running processes out.
- Compile each source file in a project individually, link them together, then load to run the program.

Compiling v.s. Linking

- Compiler: transform C code into machine instructions, doesn't know addresses of anything, but does know relative offsets of addresses within a segment. (e.g. the virtual address of the start of the function foo is 56 bytes below the address of the start of function bar).
- Linker: Lays out the different compiled segments in memory. This involves coalescing like segments (e.g. put the code segments from foo.c and bar.c together), and choosing virtual address for everything.

Compiling

- Don't need to know details of how compilers work (there's a whole class for that).
- Compilers cannot produce executables because they only look at one file at a time, and files can reference external variables that are not defined in the file.
- "So? why not have the compiler just look at all the files?"
  - Modularity makes life much easier in the long run.

Compiling

- Anatomy of a machine instruction:

  `68 0a 00 00 00`

  The first four bytes are called the *opcode* - it specifies which operation the cpu should perform. In this case, 68 corresponds to PUSH. The rest of the instruction is the argument to the operation, in this case 10.
- Compilers leave in dummy values for variables they don't see in the file:

  `external_function(external_variable)`

  becomes

```
68 00 00 00 00 = push $0
  R_386_32 external_variable
e8 fc ff ff ff = call -4
  R_386_PC32 external_function
```

- The `R_382_32` codes are annotations for the linker - they tell the linker that some external reference needs to be patched, and how to perform the patch given the unknown address.

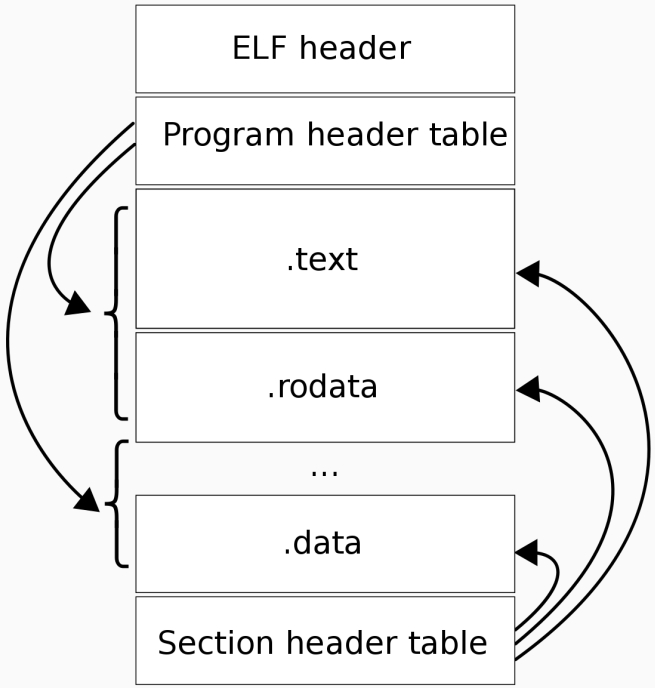
Compiling

- the “call” operation takes a relative offset as an argument (e.g. call -5) would be an infinite loop because the call instruction is 5 words long.
- This way, calls to functions within a given file are correct even if the compiler doesn’t know the absolute virtual address of any value in the file.

Linker: 2 passes

- Pass 1: Decide on actual virtual addresses for all parts of the program. Then, build a table of all symbols shared accross object files and what their virtual addresses are.
- Pass 2: Go back over the executable and perform all the relocations using the table.

ELF executable format



- .text, .data etc are called *sections*. Sections are grouped into *segments*. The program header table tells where the segments are, the section header table tells where the segments are. Sections are useful when the ELF file is describing a shared library to be dynamically linked in. Segments are useful when the loader is actually going to execute the program stored in the file.

Dynamic Linking

- Library functions like printf are common for many programs, so we'd like to share code to conserve memory.
- GOT: Global offset table: a list of addresses of library functions with unspecified values at runtime.
- PLT: procedure linkage table: think of this as a list of "stub version" of library functions like printf that the linker can output. All these functions do is call to an appropriate address in the GOT.

call PLT[5] (stub for printf)
⋮
PLT[5]: call GOT[5] (should be address of printf after dynamic linking)

- dlfixup function is provided by the loader, and is responsible for finding hte library function printf, giving it a virtual memory address, and storing that address in GOT[5].
- Laxy loading: initially GOT[5] is set to be the address of dlfixup, so that dlfixup will only dynamically load library functions the first time they are used.

Interpreter

- Files that do dynamic linking use an *interpreter* program to read their ELF file.
- Think of this as a special process-specified loader. The actual OS loader will read the ELF file, see that the file specifies an interpreter program and then run the interpreter on the ELF file.
- The interpreter lays out the memory for the program by reading the ELF file, making sure to create GOT as appropriate.

Buffer Overflow attack

```
void get_input() {
  char buffer[100];
  gets(buff);
}
```

- Naive programmer writes this program. Attacker runs it and provides an input string that is more than 100 bytes long, which allows the attacker to overwrite the return address of `get_input` on the stack.
- Attacker can overwrite return address to be the address of `buff`, and provide a string that contains arbitrary code, allowing that code to run.
- Attacker can also overwrite the return address to be the address of some library function, and overwrite higher parts of the stack to be malicious arguments to that function.
- Fix first attack by not allowing stack pages to be used as instructions.
- Mitigate second attack by randomizing the addresses of functions so that the attacker doesn't know which return address to provide.

Communicating with Devices

- At a high level, devices are connected to CPU over the *bus*.
- Think of the bus as an airline bag claim belt. The CPU is in the middle and the devices are on the outside. CPU puts information (bags) on the belt and devices check if the bag is for them and pull it off if so.
- Can use `inb`, `outb` functions to communicate with devices over bus.
- Port numbers are like tags that tell devices whether the data is for them or not.

Issues with Ports

- `inb`, `outb` require arguments to be in specified registers, which limits the ability of the compiler to optimize code.
- It's not possible to write user-level device drivers because hardware allows any code that can access ports to also be able to disable interrupts, so a malicious user-level device driver can take over the CPU.

More fancy Communicating with Devices

- Map device registers/memory to physical address: the device itself might have internal memory, so map this device memory to some area in a process's address space. Writes to these addresses will actually go to the device's memory and not to main memory. It's like the CPU is reading over the devices shoulder.
- Direct Memory access: pick an area of memory for the device. The device is allowed to access this area to communicate with the device, a process just writes to the memory. It's like the device is reading over the CPU's shoulder.

Interrupts vs Polling

- How to tell if the device has new data available?
- Option 1: poll the device (just have a loop that keeps checking for new data).
  - Downside is that you are basically busywaiting
- Option 2: allow the device to trigger an interrupt when it has new data.
  - Downside is that interrupts cause context switches, which are slow. If the device has data very frequently, then you spend all your time context switching and you'd have been better off with polling.

SCSI

- SCSI is just a protocol for communicating with hardware (usually disks).
- Each SCSI request has an initiator and a target. Intuitively, the initiator is requesting some action from the target.
- So for example, send a request to read from a disk. The disk is the target, and is given control of the bus. Often a request will say "do some action and then disconnect from the bus so I can issue other requests while you are busy".
- If there's an error, no other commands are executed until the error is read and acted on by a specific REQUEST SENSE command is used to read the error.

Disk Access

- Hard disks are circular platters with data written on them that spin very fast. There is a head that sits on the platter and reads the data that spins by under the head.
- Therefore sequential IO is way faster on disks than random IO.
- OS takes advantage of this when writing data out of buffer cache to disk to try to write blocks in sequential order.
- SCAN procedure: sweep across disk, servicing all requests passed.

Flash Memory

- Flash memory mitigates random access issue of disks, so it's not as important to use SCAN type heuristics.
- Big issue: each address of flash memory stops working after a certain number of writes.
- Flash Translation Layer acts like a kind of virtual memory system for the flash memory. It provides the OS with a view of the flash memory as a ordinary set of "virtual" addresses, but will direct these addresses to actual places on the physical flash memory in order to prevent single physical areas being accessed too frequently (called wear leveling).

Crash-safe file systems

- Computer could shut down at literally any point. Need to make sure that the file system is never corrupted.
- It's ok for writes to a file to not happen, but we want to avoid leaking disk space or having inodes that contain junk sector numbers.

Idea 1: Fix Corruption

- Utility called `fsck` runs at startup (or in the background) and tries to cleanup issues left behind by previous crash.
- Basically scans over the disk and filesystem and makes sure that all the blocks marked unavailable in the free-map are indeed used in the file system (catch leaks), and tries to correct errors in inodes it sees.
- Now the problem is to make sure the system isn't so corrupted that `fsck` can't fix it. This leads to careful ordering of disk writes.

Ordered Updates

- Basic idea is to never have any pointers on disk to uninitialized data, and never have any period of time in which all pointers to some resource are only present in RAM and not on disk.
- The first point protects `fsck` from reading garbage data and therefore doing garbage things. The second point keeps you from totally losing any crucial data.
- Achieve this by enforcing orderings on disk operations. For example, if you make a file, you must create the inode on disk before adding a directory entry for the file.
- Can have leaks: create an inode on disk for a file, but then crash before adding an entry to the directory telling where the file is. `fsck` will run in the background and find that this inode is actually unlinked and so decide that it's actually free.

Soft updates to fix dependencies issues

- With ordered updates you can have some blocks be very slow to write to disk. For example, if you make a bunch of files in a directory, you will take a really long time to write that directory to disk and so you might crash and lose all the files.
- With soft updates, we decouple the choice of which block to write to disk from the dependencies.
- choose any block A in the buffer cache to write to disk. If this block is supposed to be written after some other blocks to preserve ordered updates, just temporarily roll back all changes to block A that required dependencies and then write to disk.
- An issue: Sometimes you need to write a buffer many times with many rollbacks.

Journaling

- Big idea: if we could ensure certain operations were atomic (i.e. couldn't be interrupted by crashes), it would be easy to be crash-resistant.
- Writes are *idempotent*: if you do a write twice it's the same as doing it once.
- Keep a *journal* of actions on disk. Whenever you want to do some atomic writing, first record the writes in the journal. Then do the actual writes. Then record that the writes were completed in the journal.
- Detect crash while recording the writes in the journal because journal entry is incomplete. Just drop these writes - maybe a file isn't created that the user asked to make just before the crash, but that's ok.
- Detect crash during actual writes because the "I'm done" entry isn't in the journal. Recover by just actually performing the write you recorded in the journal. This is ok because writes are idempotent.

Journaling in XFS

- XFS wants to support really large file systems easily.
- In a directory with lots of files, want to look up file locations by name quickly without linear scan.
- In a very large, maybe very sparse file, want to look up sector number efficiently from offset without lots of indirections and file size limits.
- Solution: B+ tree. Think a self-balancing binary tree with a larger branching factor.
- Updating B+ trees is a complicated algorithm, need to make sure the update doesn't leave the state corrupted when you crash.
- Use journaling to make update operations atomic.

Misc File system things

- If the disk gets really full, creating and writing to new files becomes really slow because of fragmentation. Solution: lie to the user. Tell them the disk is only 90% of its actual size. The extra 10% free space makes it very likely it will be easy to allocate new blocks when needed.
- Symbolic links (symlinks) are files that are actually pointers to other files. Think of this as a file that contains the path of another file. You can delete the file or the symbolic link independently (although the symbolic link will be "broken" if you delete the file).
- Hard links are your actual `dir_entry` structs. You make two `dir_entry` structs in two directories point to the same inode. This causes some issues: if you delete one of the hard links to a file, then you need to keep some kind of reference count in the inode to prevent the other hard link from pointing to freed disk space.

Networks

- The OS is responsible for taking information off the network and packaging it in a way that provides a nice abstraction for user programs.
- TCP is basically an algorithm the OS runs to convert unreliable networks that drop packets into simple streams of data that can be consumed by applications.
- The user-level interface provided is called a socket.
- OS provides an abstraction called a *port*, which is basically a way for the user program to provide a "return address' for any information it sends out.
    - For example, firefox opens a socket to www.google.com, while your video game opens a socket to awesomegameserver.net. When google and awesomegameserver send information back, the os needs to know which application should get which packets.

Networks

- Networking protocols are organized in *layers*.
  1. The actual data the application wishes to send is first wrapped in a TCP layer, which adds some metadata needed for keeping track of missed packets.
  2. The TCP packet is then wrapped in an IP packet, which contains metadata needed to route the packet from the destination computer to the target computer in the internet.
  3. The IP packet is wrapped in an ethernet packet, which contains the information needed to physically communicate between two computers that are "adjacent" in the internet, meaning the are connected by a physical wire.

Networks

- The OS needs to manage all these layers, which involves adding various headers at various times to the packet.
- To avoid reallocating memory and copying data when adding/removing headers, use `mbuf` to store packets.
- `mbufs` are basically linked-lists whose items are small 230 byte strings. To prepend to a packet, just add another node to the list at the beginning. To remove the beginning of a string, you can just drop some items at the beginning.
- Sometimes a header is split accross the first two consecutive nodes in the linked list. This is annoying:
  - `m_pullup` function refactors the list so that a given number of bytes is contained in the first node. This allows you to easily access a header by first calling `m_pullup` and then casting a pointer to a header struct pointer.

Permissions

- Give each (user, file) pair a set of permissions (e.g. read, write, execute).
- Unix stores all (user, file) pairs for a given file with that file. When a user accesses a file, the OS checks the pairs associated with the file to make sure the user has permission to do the desired action.
- To avoid having to store a pair for every single user of the system. Unix uses groups congomerate users.
- We don't want users to be able to read the passwords file. So the passwords file only allows root to read it, and the login program runs as root.
- This causes problems when changing passwords, so need "effective user id", a way to allow some programs that were made by root to pretend to be root even when run by a different user.

Security Issues

- TOCTOU (Time of check to time of use) bugs are like race conditions:
  - a root process may be doing some file system maintenance. First it checks if a file should be deleted, then deletes the file.
  - Attacker intervenes between the check and delete to replace directory the file is in with a symlink to a directory containing a critical file of the same name.
- Fix this by either having some way to lock file system resources, or by having OS transactions that make some file system operations atomic.

More Security

- Try to protect classified information: maintain guarantee that any process that reads a classified file cannot subsequently write to an unclassified file.
- Every file has a security level: (c, s). c is a classification level, and s is an identifying set that the resource belongs to.
- We say (c, s) dominates (c', s') if $c > c'$ and $s \supset s'$.
- To prevent leaking classified info, kill any process that attempts to write to a with security level (c', s') if it has already read from a file with security level (c, s) where (c, s) dominates (c', s'). For example, kill any process that tries to write to an unclassified memo after reading a classified intelligence report.
- LOMACS: use this scheme to provent destruction of sensitive files rather than leaking sensitive info: if a process reads or comes from a low-sensitivity area (like the network), then it might be compromised so don't let it write to high-sensitivity files (like the kernel code).
- In practice there are many other channels that make guarantees hard to come by.

Extra Channels

- Exit value: low-clearance level file spawns child. Child reads classified file and becomes high-clearance. Child sets exit value equal to *i*th bit in file and exits. Low-clearance process getds to see the exit value, and just repeats for all bits in the file.
- CPU utilization: high clearance process encodes the sensitive file the "CPU utilization graph" that you see on the activity monitor in your computer: go into a busy loop for a second to encode a 1, sleep and yield cpu for a second for a 0.

Virtual Machines

- Want to run old/insecure software. Maybe want to run many OSs on one computer.
- One solution: write an emulator that literally simulates the hardware. Sometimes you might have to do this (e.g. if you're emulating graphing calculator or a gameboy on your PC).
- In general this is super slow. If the guest OS is supposed to run on the same hardware, perhaps you can actually let it run since most operations aren't sensitive.
- Idea: make the OS run in user mode. Whenever it tries to do something that is really OS-specific, it will trap to a "meta-OS" (the Virtual Machine Monitor), which can inspect the state of memory, find out what the guest OS was trying to do and perform the actions for it.

Virtual Machines

- Guest OSs need to believe they are managing the page tables, which means they need to think they know where things are in physical memory.
- We instead give them a guest physical address, which they think is a physical address but is actually a *virtual* address. The VMM maintains a mapping from guest physical address to host (i.e. real) physical addresses, just like an ordinary OS does virtual memory.
- We translate the guest OS's page table entries (which map from guest virtual address to guest physical addresses) to the real page tables, which map from guest virtual addresses to host physical address.
- VMs are so popular now that modern hardware actually provides help for supporting them.