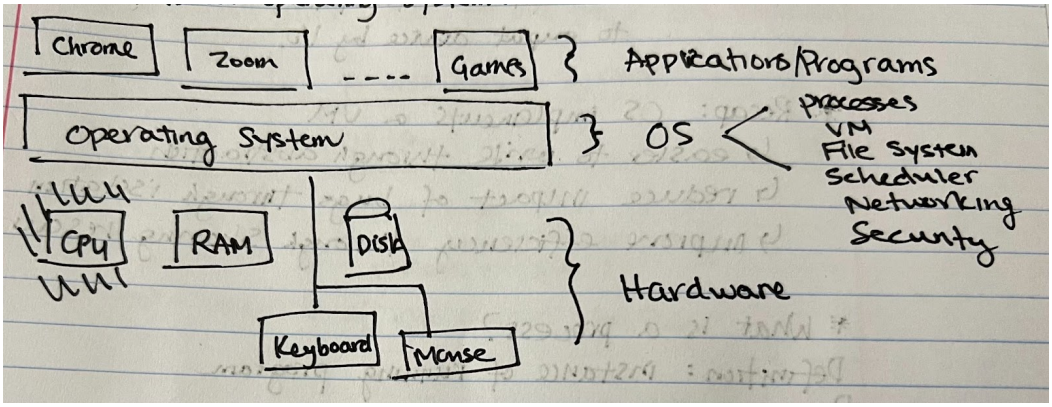
CS 170 Notes

Lecture 1: Introduction

Operating System: software layer that implements a virtual machine that is easier to program than the raw hardware.

- **Hardware Resources:** CPU, memory, disk, I/O devices (nasty to program directly)
- OS includes OS services: processes, virtual memory, file system (file contents, directories and file names), security, networking, inter-process communication, time, terminals, etc.

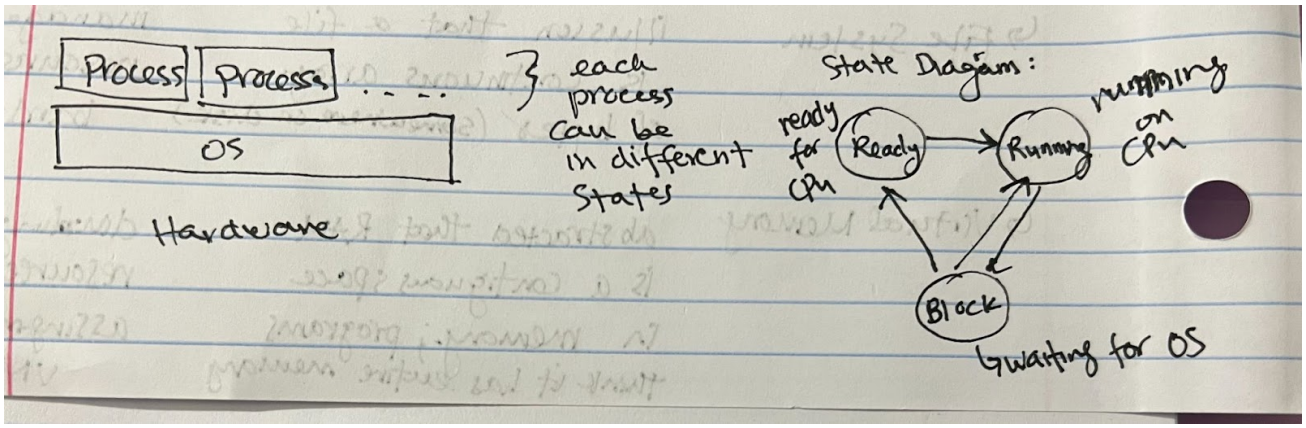


Job of OS: provide services to user-level programs

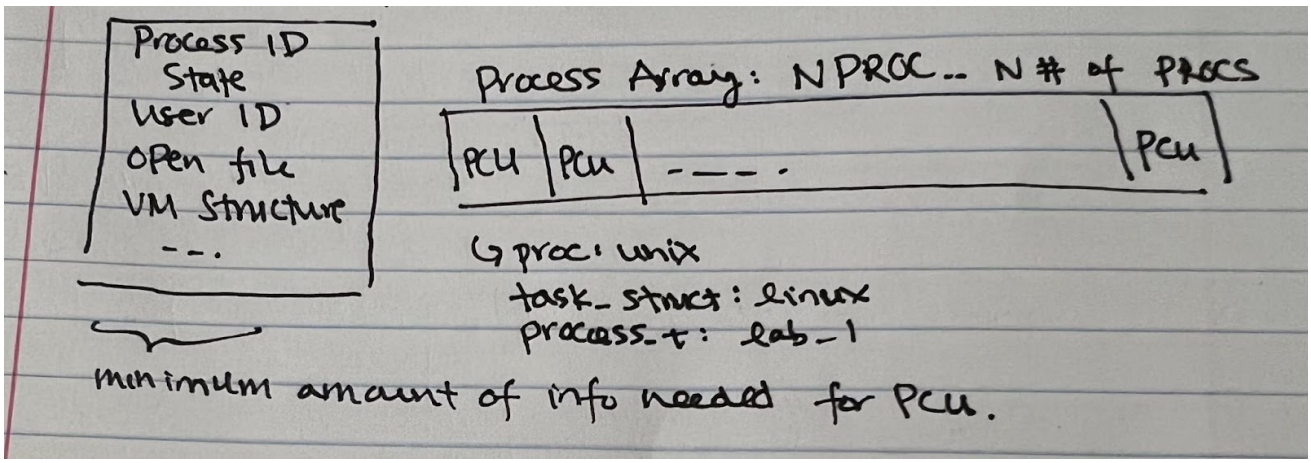
- (1) managing the resources of the machine and protecting via isolation
- (2) abstracting the hardware
- Makes applications easier to write, reduce impact of bugs, machine's resources shared, improving efficiency
- Examples:
    - Scheduling Abstraction: process has the illusion that it is running continuously; it is not
    - Scheduling Isolation: CPU program hogger gets switched in favor of another program
    - File Systems Abstraction: illusion file is continuous array of bytes; it's not; may be split up on disk; abstracting both files and file descriptors
    - File Systems Isolation: user program can't write to a file unless it has permission
    - Memory Abstraction: user program thinks it is reading from 0x1248; it is not; more generally, user program thinks it has a linear, contiguous address space; it does not
    - Memory Isolation: user program can't write to another user's memory
    - Abstracting Hardware: illusion that writing to a device is the same as writing to a file.
    - Managing I/O Resources: what happens if every process tries to write to the screen at once?

Processes: instance of a running program. (i.e., browser, text editor, etc.)

- **Program:** set of instructions. Run a program to get a process; this running program "sees" an abstraction of a virtual machine (virtual memory, virtual CPU, including registers, etc.).
- HUMAN -- writes -> SOURCE CODE -- compiler -> EXECUTABLE -- run -> PROCESS
- Root process created with init. Processes created from forking another process.
- Each process is either in a preemption, ready, running, or blocked state.



- **Process Array:** n number of process control units
- **Process Control Block (PCB):** (Unix = proc, Linux = task_struct, lab1 = process_t)

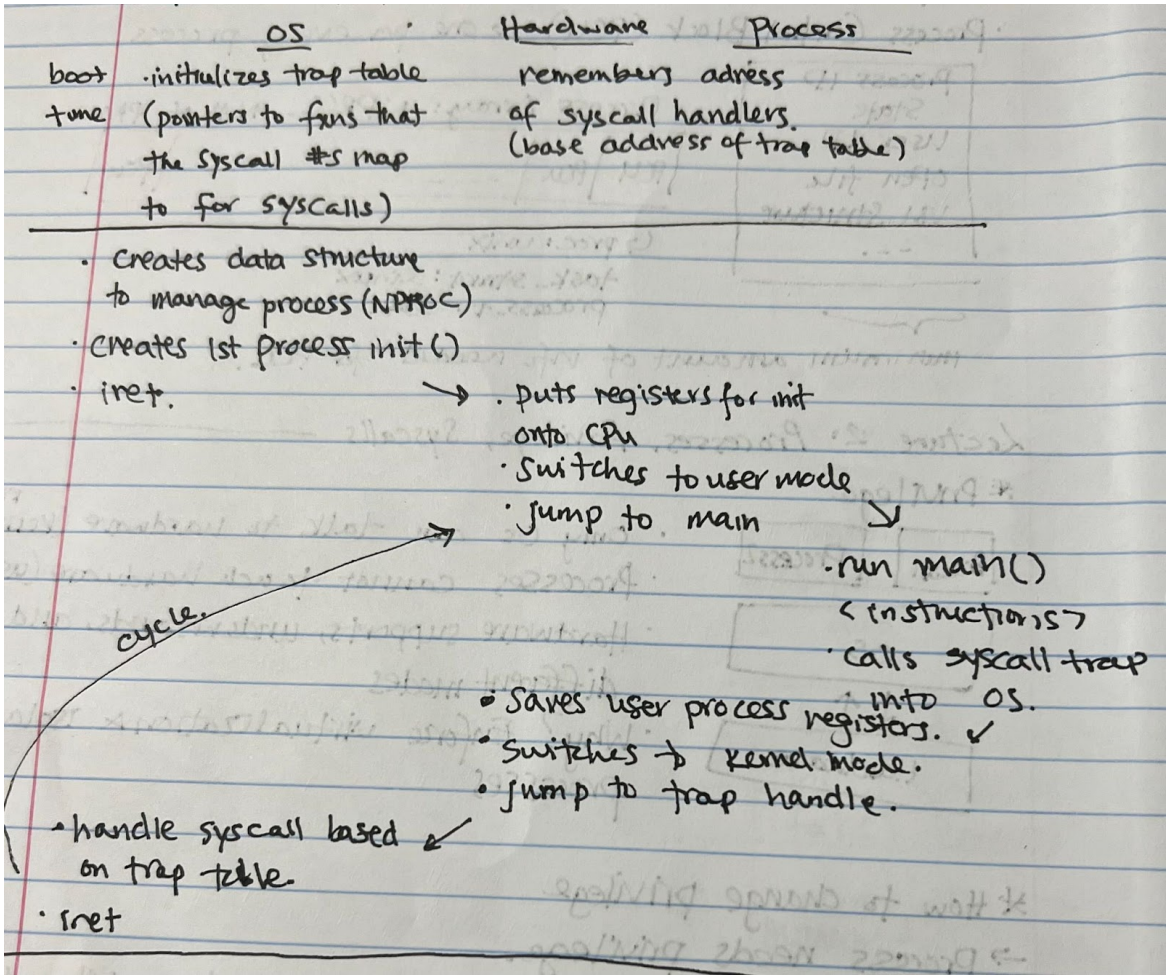


- Each process has at any given time: stack pointer, frame pointer, other registers, state of OS resources, view of memory, which includes: program code (aka "text"), constants, zeroed-out area for variables, stack, heap.

Lecture 2: Processes, Call Stacks, Syscall

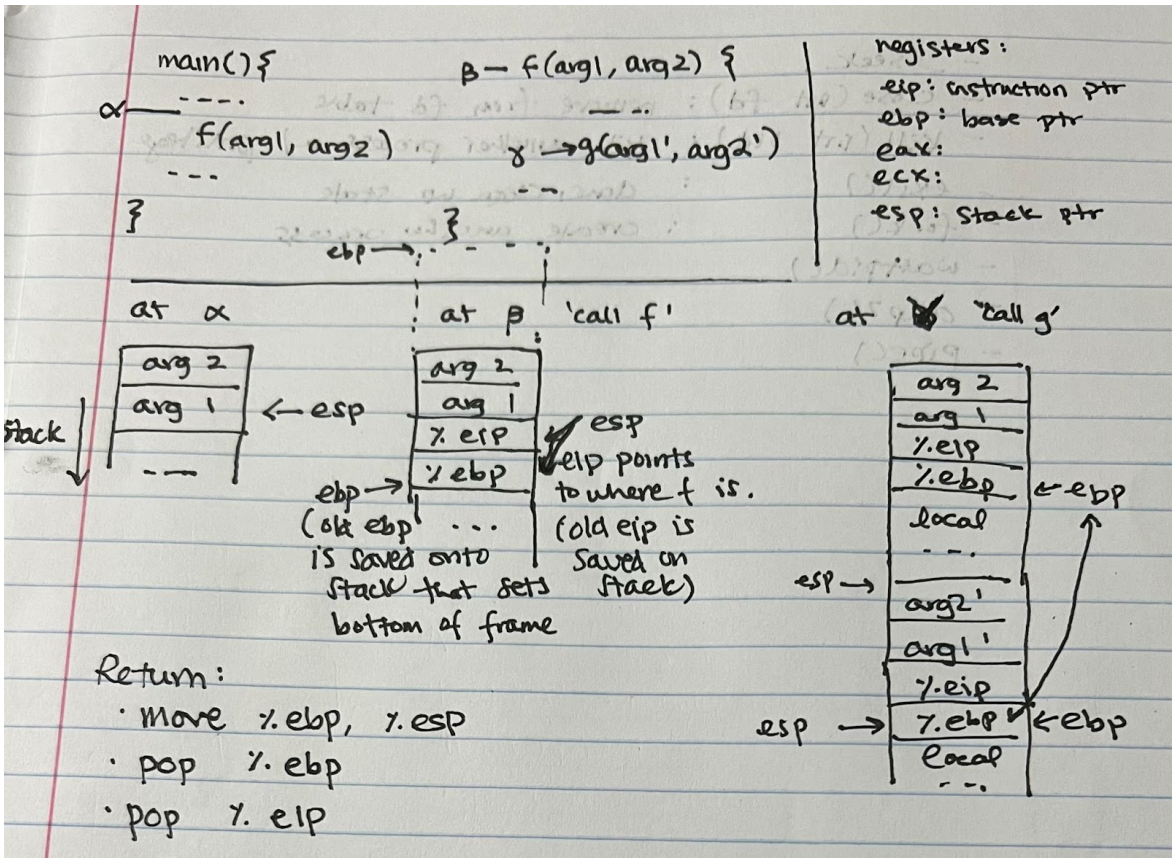
Privileged (Kernel) [Ring 0] vs **Unprivileged** (User) **Mode** [Ring 3]

- OS can mess with the hardware in privileged mode; processes in user mode cannot. Otherwise, processes could run arbitrary instructions, which would break the virtual machine illusion they get.
    - Hardware supports and knows (understands and enforces) privilege modes.
- How do we get into kernel mode?
    - (1) **exception**: user program does something bad, like divide by 0, or invalid memory access.
    - (2) **trap**: user program (process) asks the kernel for help. Use the assembly instruction "int" on the x86.
    - (3) **interrupt**: (i) timer or (ii) device event (I/O completion, keystroke pressed, etc.)
- How do we get back out of kernel mode? "iret" returns to the place where the user program was.
- Process Lifecycle (throughout process, there is illusion that the process is always running)



Process's View of Memory; Stacks

- Think of memory as a contiguous array: [text/code | data | heap --> <--- stack | kernel memory]
- GCC Calling Convention:
    - push: moves stack pointer lower and adds element to stack
    - pop: remove element from stack and moves stack pointer higher
    - call: updates %eip and pushes old %eip on the stack.
    - ret: updates %eip by loading it with a stored stack value.
    - %eip tells the computer where to go next to execute the next command.
    - %ebp contains base pointer (base of stack) and %esp contains stack pointer (growing part of stack).
- Function protocols are determined by compiler conventions.
    - Prolog: push previous base pointer and move stack pointer to base pointer.
    - Epilog: Move base pointer to stack pointer and pop from stack to base pointer and return.
- If I somehow arranged to copy the call stack to another process and said "here's your stack", then that other process would *appear* to return exactly the same as the first one would. (process creation with fork())



System Calls (API for OS), File Descriptors

- fd is a *file descriptor*, an abstraction, provided by the operating system, that represents an open file
- Every process can usually expect to begin life with three file descriptors already open:
    - 0: represents the input to the process (e.g., tied to terminal) [stdin]
    - 1: represents the output [stdout]
    - 2: represents the error output [stderr]

  int fd = open(const char* path, int flags, mode_t mode)
- NOTE: Unix hides for processes the difference between a device and a file; this is a very powerful abstraction.
- Here are some other system calls:

```
--int open(char*, int flags, [, int mode]);
--int read(int fd, void*, int nbytes):
--int write(int fd, void* buf, int nbytes);
--off_t lseek(int fd, off_t pos, int whence)
--int close(int fd);
--int kill(int pid, int signal)
--void exit (int status)
--int fork(void)
--int waitpid(int pid, int* stat, int opt)
--int execve(char* prog, char** argv, char** envp)
--int dup2 (int oldfd, int newfd)
--int pipe(int fds[2])
```

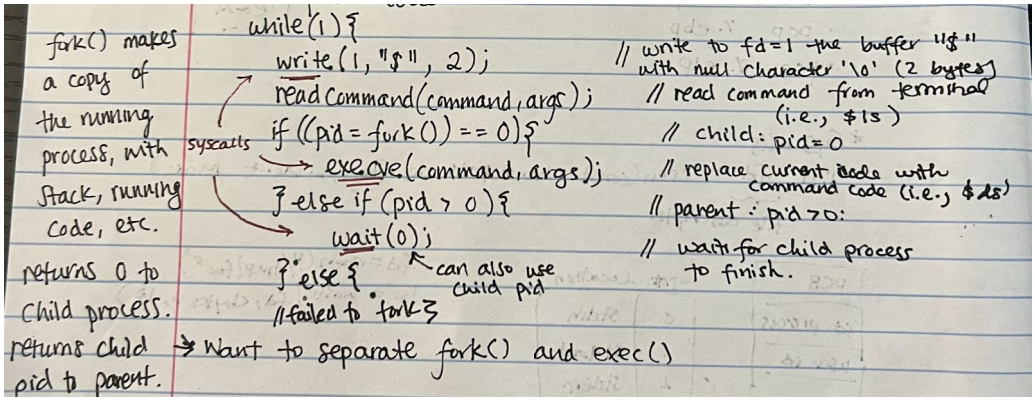
Lectures 3: Syscalls, Shell, Fork/Exec

The whole OS as a unit

1. The machine turns on.
2. Starts executing code from its BIOS, which is a program that lives in the ROM.
    - BIOS = basic input/output system
    - ROM = read-only memory
3. BIOS loads the boot loader from the disk.
4. The boot loader loads the kernel, and jumps to its *entry point* (which is an address that the programmer of the kernel specified). This address shows up in the ELF file for the kernel (ELF = executable and linking format; it's a file format for storing binary executables, among other things).
    - Question: how did the programmer tell the linker where to set the entry point in the ELF file?
    - Answer: see the excerpt "-e multiboot_start" in lab1/GNUmakefile. This argument is passed to the linker (ld). Now, note that multiboot_start is defined in k-int.S, and that entry point quickly calls "start"
5. The kernel starts executing at start().
6. The kernel breathes life into a process, and transfers control to the process (via "iret").
7. Now, behavior is governed by the state diagram.

Shell: a program to create processes; the human's interface to the computer; includes GUIs (graphical user interfaces).

- Examples: terminal, command line, GUI desktop managers
- How does the shell start programs?
    - calls fork(), which creates a copy of the shell.
    - then, the child calls exec(), which loads the new program's instructions into memory and begins executing them. (exec invokes the loader)
    - How can the shell wait for a child process to end? with wait() or waitpid() system calls
    - Pseudocode for basic shell:



- Implementation of $ls

    // read content of file directory

    // write content to stdout

- Redirection:
    - Implementation: removes stdout/stdin from the fdtable and opens redirection location to replace.

        close(1)

        open("tmp1", O_TRUNC | O_CREAT | O_WRONLY, 0666)

    - Pseudocode for shell with output redirection and backgrounding:

```
By output redirection, we mean, for example:
$ ls > list.txt

By backgrounding, we mean, for example:
$ myprog &
$

while (1) {
    write(1, "$ ", 2);
    readcommand(command, args); // parse input
    if ((pid = fork()) == 0) { // child?
        if (output_redirected) {
            close(1);
            open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
        // when command runs, fd 1 will refer to the redirected file
        execve(command, args, 0);
    } else if (pid > 0) { // parent?
        if (foreground_process) {
            wait(0); //wait for child
        }
    } else {
        perror("failed to fork");
    }
}
```

Pipelines: output of first process is sent as input to the next

- Implementation of pipe() system call
    - creates a buffer (space in memory that the OS controls)
    - assigns fdarray[0] as read end of pipe and fdarray[1] as write end of pipe
- How does one of them send its input to the other without rewriting these programs?
    - Left Hand of Pipe: make ptr pointing to stdout point to fdarray[1], and close fdarray[0,1] for cleanup
    - Right Hand of Pipe: make ptr pointing to stdin point to fdarray[0] and close fdarray[0,1] for cleanup
    - Why cleanup?
        - (a) ensure that every process starts with exactly 3 file descriptors
        - (b) ensure that reading from the pipe returns "end of file" after the first command exits. [Cannot exist if the write-end is still not closed on the reading process end.]
- Shell waits for the write end of the pipeline. [Read is a blocking call that doesn't return until EOF is read]
    - If the left-hand process finishes first, great, it exits
    - If the right-hand process has already exited, then the left-hand process gets SIGPIPE, and dies
- Why are pipelines interesting?
    - What if ls had to be able to paginate its input? with pipes, the program doesn't have to get recompiled
    - the program doesn't have to take a file/device/whatever as input; prior to Unix, there was little to no composability. programs had to do everything or use temporary files awkwardly
    - Useful because can combine many basic programs to do powerful things
- Updated Pseudocode with Pipelines:

```
void main_loop() {

    while (1) {
        write(1, "$ ", 2);
        readcommand(command, args); // parse input
        if ((pid = fork()) == 0) { // child?
            if (pipeline_requested) {
            /* NOTE: lab2's logic is different from this */
            handle_pipeline(left_command, right_command)
            } else {
                if (output_redirected) {
                    close(1);
                    open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                }
                exec(command, args, 0);
            }
        } else if (pid > 0) { // parent?
            if (foreground_process) {
            wait(0); // wait for child
            }
        } else {
            perror("failed to fork");
        }
    }
}
```

```
void handle_pipeline(left_command, right_command) {

    int fdarray[2];

    if (pipe(fdarray) < 0) panic ("error");
    if ((pid = fork ()) == 0) { // child (left end of pipe)

        dup2 (fdarray[1], 1); // make fd 1 the same as fdarray[1],
                              // which is the write end of the
                              // pipe. implies close (1).

        close (fdarray[0]);
        close (fdarray[1]);
        parse(command1, args1, left_command);
        exec (command1, args1, 0);

    } else if (pid > 0) { // parent (right end of pipe)

        dup2 (fdarray[0], 0); // make fd 0 the same as fdarray[0],
                            // which is the read end of the pipe.
                            // implies close (0).

        close (fdarray[0]);
        close (fdarray[1]);
        parse(command2, args2, right_command);
        exec (command2, args2, 0);

    } else {
        printf ("Unable to fork\n");
    }
}
```

The power of the fork/exec separation, an innovation from the original Unix

- allows the child to manipulate environment and file descriptor *before* exec, so that the *new* program may in fact encounter a different environment
- To generalize redirections and pipelines, there are lots of things the parent shell might want to manipulate in the child process: file descriptors, environment, resource limits.
- Fork() requires no arguments! Contrast with CreateProcess on Windows (there's also CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, ...):
    - The issue is that any conceivable manipulation of the environment of the new process has to be passed through arguments, instead of via arbitrary code.

```
BOOL CreateProcess(
    name,
    commandline,
    security_attr,
    thr_security_attr,
    inheritance?,
    other flags,
    new_env,
    curr_dir_name,
    .....)
```

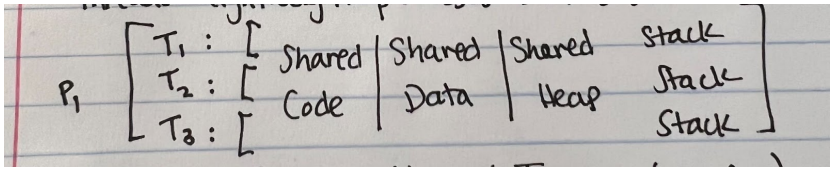
What makes a good abstraction? simple but powerful

- stdin (0), stdout (1), stderr (2); file descriptors; fork/exec() separation; very few mechanisms lead to a lot of possible functionality

Lecture 4: Threads

Threads: lightweight processes that share resources but keep their own stack

- Threads are useful because they are lighter weight, which makes them more efficient than processes
- Implementation: assume for now: abstraction created by OS; preemptively scheduled



- Interface to threads:
  - tid thread_create (void (*fn) (void *), void *); // create a new thread, run fn with arg
  - void thread_exit (); // end of thread call
  - void thread_join (tid thread); // wait for thread 'thread' to exit

Concurrency: stuff happening at the same time

- Sources:
  - Computers have multiple CPUs and common memory, so instructions in multiple threads can happen at the same time!
  - On a single CPU, processes/threads can have their instructions interleaved (helpful to regard the instructions in multiple threads as "happening at the same time").
  - Interrupts (CPU was doing one thing; now it's doing another)
- Why is concurrency hard? Hard to reason about all possible interleavings.

Lecture 5: Race Conditions

Problems with Race Conditions

- Program may work fine most of the time but only occasionally show problems because the instructions of the various threads/processes get interleaved in a non-deterministic order.
- Inserting debugging code may change the timing so that the bug doesn't show up.
- **Sequential consistency:** maintain program order on individual processors ensuring that writes happen to each memory location (viewed separately) in the order that they are issued
- Hardware makes the problem even harder because sequential consistency not always in effect
  - Assume sequential consistency until we explicitly relax it

Critical Sections

- **Critical section:** Regard accesses of shared (i.e., global) variables as being in a critical section; these sections need be protected from concurrent execution
- Critical section solutions need to satisfy three properties
    - 1. **mutual exclusion:** only one thread can be in the critical section at a time, forcing atomicity
    - 2. **progress**: if no threads are executing in the critical section, one of the threads is trying to enter a given critical section will eventually get in
    - 3. **bounded waiting**: once a thread T starts trying to enter the critical section, there is a bound on the number of other threads that may enter the critical section before T enters
    - If no thread can enter C.S., don't have progress; if thread A is waiting to enter C.S. while B repeatedly leaves and re-enters C.S. ad infinitum, don't have bounded waiting
- Protect critical sections with a lock mechanism (lock()/unlock() or enter()/leave() or acquire()/release())
    - Once the thread of execution is executing inside the critical section, no other thread of execution is executing there
- Implementing critical sections
    - "easy" way, assuming a uniprocessor machine:

        enter() --> disable interrupts

        leave() --> re enable interrupts
    - Without interrupts, thread can finish execution in peace; however, this puts too much trust giving the thread privilege to correctly run and finish

Mutexes (for Mutual Exclusion)

- Atomicity is required if you want to reason about code about all possible interleavings
- Atomicity requires mutual exclusion aka a solution to critical sections; mutexes provide that solution
- **Invariants:** code in algorithm assumed to stay the same
    - The meaning of lock.acquire() is that if and only if you get past that line, it's safe to violate the invariants.
    - The meaning of lock.release() is that right before that line, any invariants need to be restored.
- Examples
    - yield( ) can help with performance issues regarding spin locks that take the entire cycle to not achieve progress while trying to acquire a lock
    - We can break a critical section into multiple sections to improve performance.

Lecture 6: Condition Variables

Condition variables: data type to allow wait and signal to sleep and wake threads when waiting for condition

- Motivational Example: producer/consumer queue using mutexes with spin wait is intensive/busy waiting
- It is convenient to break synchronization into two types:
    - **mutual exclusion:** allow only one thread to access a given set of shared state at a time
    - **scheduling constraints:** wait for some other thread to do something (i.e., finish a job, produce work)
- We MUST use "while", not "if". Otherwise, a third thread can invalidate the condition when more than 1 thread is being woken up. Our now-ready thread eventually acquire()s the mutex with no guarantees that the condition it was waiting for is still true.
- cond_wait releases the mutexes and goes into the waiting state in one function call (atomically). Otherwise, it can get stuck waiting when sleep happens after signal if interrupt happens between release and sleep.
- API

```
--void cond_init (Cond *, ...);
    --Initialize

--void cond_wait(Cond *c, Mutex* m);
    --Atomically unlock m and sleep until c signaled
    --Then re-acquire m and resume executing

--void cond_signal(Cond* c);

    --Wake one thread waiting on c

    [in some pthreads implementations, the analogous
    call wakes *at least* one thread waiting on c. Check the
    the documentation (or source code) to be sure of the
    semantics. But, actually, your implementation shouldn't
    change since you need to be prepared to be "woken" at
    any time, not just when another thread calls signal().
    More on this below.]

--void cond_broadcast(Cond* c);

    --Wake all threads waiting on c
```

Monitors: mutex + condition variables

- High-level idea: an object (as in object-oriented systems) in which methods do not execute concurrently; and that has one or more condition variables
- Every method call starts with acquire(&mutex), and ends with release(&mutex)
- Technically, these acquire()/release() are invisible to the programmer because it is the programming language (i.e., the compiler+run-time) that is implementing the monitor
- Synchronization happens with condition variables whose associated mutex is the mutex that protects the method calls

- Rules
    - acquire/release at beginning/end of methods
    - hold lock when doing condition variable operations
    - A thread that is in wait() must be prepared to be restarted at any time, not just when another thread calls "signal()". The implementer of the threads and condition variables package *assumes* that the user of the threads package is doing while(){wait()}.
- Different styles of monitors:
    - Hoare-style: signal() immediately wakes the waiter
    - Hansen-style and what we will use: signal() eventually wakes the waiter. Not an immediate transfer
- Can we replace SIGNAL with BROADCAST, given our monitor semantics? Yes, always.
    - while() condition tests the needed invariant. program doesn't progress pass while() unless the needed invariant is true; result: spurious wake-ups are acceptable, which implies you can always wake up a thread at any moment with no loss of correctness.
    - Though, it may hurt performance to have a bunch of needlessly awake threads contending for a mutex that they will then acquire() and release().
- Can we replace BROADCAST with SIGNAL? Not always.
- Example: memory allocator; when multiple threads are waiting to allocate chunks of memory, when memory is freed, these threads might be able to both function now, whereas signal only wakes one of them.

Lecture 7: Monitor Framework

Framework Steps:

1. Getting Started

a. Identify units of concurrency. Make each thread with a run( ) method or main loop. Write down the actions a thread takes at a high level.

b. Identify shared chunks of state. Make each shared *thing* an object. Identify the methods on those objects, which should be the high-level actions made *by* threads *on* these objects. Plan to have these objects be monitors.

c. Write down the high-level main loop of each thread.

2. Identify Synchronization Constraints.

a. Mutual Exclusion Constraints. (Usually, the mutual exclusion constraint is satisfied by the fact that we're programming with monitors.)
b. Scheduling Constraints ("when does a thread wait")
3. Assign Mutex / Conditional Variables for each Constraint
4. Write Code

Advice

- Highest Priority: Safety First, followed by correctness and race conditions.
- Don't manipulate synchronization variables or shared state variables in the code associated with a thread; do it with the code associated with a shared object.
    - Locks are for synchronizing across multiple threads. Doesn't make sense for one thread to "own" a lock.
    - Encapsulation -- details of synchronization are internal details of a shared object. Caller should not know about these details. "Let the shared objects do the work."
- Why not just hold the lock all the way through "Execute req"? (Answer: the whole point was to expose more concurrency, i.e., to move away from exclusive access.)

Lecture 8: Concurrency Issues

Deadlocks: no progress can be made

- can occur with real resources beyond mutexes and threads
- Conditions: mutual exclusion, hold-and-wait, no preemption, circular wait
- Resolving Deadlock:
    - ignore it: worry about it when it happens (reboot system)
    - detect and recover: debugger-like attachment to keep track of resource allocation
        - For each lock acquired, order with other locks held; If cycle occurs, abort with error; Detects potential deadlocks even if they do not occur
        - Good for development, but not production
    - avoid algorithmically with elegant but impractical banker's algorithm
        - requires every single resource request to go through a single broker
        - requires every thread to state its maximum resource needs up front; if threads are conservative and claim they need huge quantities of resources, the algorithm will reduce concurrency
    - negate one of the four conditions using careful coding

        - mutual exclusion: put a queue in front of resources; virtualize memory
        - preemption: give ability for threads to force release other resource
        - circular wait: done in practice
            - establish an order on all locks and force every thread to acquire its locks in that order
            - can view deadlock as a cycle in the resource acquisition graph; partial order implies no cycles and hence no deadlock
            - hard to represent CVs inside this framework, works best only for locks.
            - compiler can't check at compile time that partial order is being adhered to because calling pattern is impossible to determine without running the program (thanks to function pointers and the halting problem)
            - Picking and obeying the order on *all* locks requires that modules make public their locking behavior, and requires them to know about other modules' locking. This can be painful and error-prone.
    - Static and dynamic detection tools (i..e, valgrind)
        - Disadvantage to dynamic checking: slows program down
        - Disadvantage to static checking: many false alarms (tools says "there is deadlock", but in fact there is none) or else missed problems
- Lesson: can be dangerous to hold locks and resources, in general

Starvation: thread waiting indefinitely (if low priority and/or if resource is contended)

- Solution: temporarily bump lower priority threads with priority from threads waiting for it; disable interrupts; Don't handle it, structure app so only adjacent priority processes/threads share locks

Trade offs:

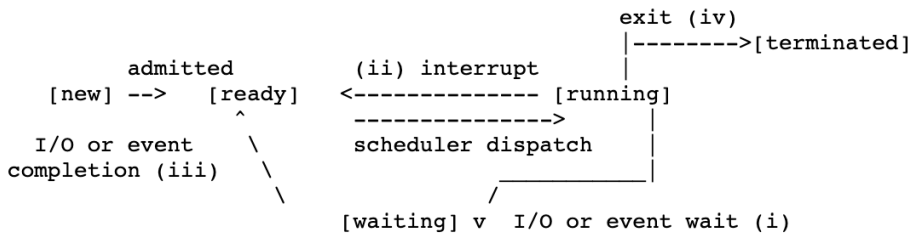
- mutex costs: the raw instructions required to execute "mutex_acquire" going to sleep and waking up implies context switch, which brings a resource cost
- Coarse locks limit available parallelism; the fundamental issue with coarse-grained locking is that only one CPU can execute anywhere in the part of your code protected by a lock. If your critical section code is called a lot, this may reduce the performance of an expensive multiprocessor to that of a single CPU. if this happens inside the kernel, it means that applications will inherit the performance problems from the kernel.
- Finer-grained locking can often lead to better performance; also leads to increased complexity and hence risk of bugs (including deadlock).

Programming Issues:

- The fundamental problem is that the shared memory programming model is hard to use correctly (although mutexes help a great deal).
- Loss of modularity: avoiding deadlock requires understanding how programs call each other.

Lecture 9: Scheduling

Scheduler: operating system has to decide which process (or thread) to run.



- Decisions take place when a process (i) switches from running to waiting state (ii) switches from running to ready state (iii) switches from waiting to ready (iv) exits
    - Preemptive scheduling occurs at all points
    - Non-preemptive scheduling occurs only when processes switch to event wait or terminates

Metrics

- **turnaround time:** time for each process to complete
- response/output time: the time spent waiting for something to happen rather than the time from launch to exit
    - **response time:** time between when job enters system and starts executing following your text
    - **output time:** time from request to first tangible output (e.g., key press to character echo)
- throughput: # of processes that complete per unit time
- fairness: freedom from starvation, all users get equal time on CPU, highest priority jobs get most of CPU
    - often conflicts with efficiency. true in life as well.

Overhead: cost of context switching

- CPU time in kernel: save and restore registers, switch address spaces
- indirect costs: TLB shootdowns, processor cache, OS caches (e.g., buffer caches)
- results in more frequent context switches will lead to worse throughput (higher overhead)

First Come First Serve (FCFS)/First In First Out (FIFO): run each job in order until it's done

- Advantages: simple, no starvation, few context switches

- Disadvantage: short jobs get stuck behind long ones

Shortest Job First (SJF): Schedule the job whose next CPU burst is the shortest

- **Shortest To Completion First (STCF):** preemptive version of SJF; if job arrives that has a shorter time to completion than the remaining time on the current job, immediately preempt CPU to give to new job
- Get short jobs out of the system, big (positive) effect on short jobs, small (negative) effect on large jobs

Round-robin (RR): add preempt timer per time slice or quantum, after time slice, go to the back of the ready queue

- Advantages: fair allocation of CPU across jobs, low average response time when job lengths vary, good for output time if small number of jobs
- Disadvantages, what if jobs are the same length? then, unnecessary context switching
- How to choose the quantum size? want much larger than context switch cost (10 microseconds); majority of bursts should be less than quantum (typical time slice is between 10-100 milliseconds)
    - Too small: spend too much time context switching
    - Too large: response time suffers (extreme case: system reverts to FCFS)

Scheduling with I/O

- Using FIFO can lead to poor disk utilization
- Using RR with a large time slice, only get minimal disk utilization
- Using RR with a small time slice, gets high disk utilization, but lots of preemptions
- STCF, applied to remaining burst time, would give us good disk utilization
    - Advantages: disk utilization, optimal response time, low overhead
    - Disadvantages: starved long jobs, oracle, does not optimize turnaround time
    - Useful as a benchmark for measuring other policies (to approximate the best answer)

Exponential Weighted Moving Average (EWMA): estimate future based on past; reacts to changes, but smoothly

- \tao_{n+1} = \alpha * t_n + (1-\alpha)*\tao_n
- t_n: length of proc's nth CPU burst; \tao_{n+1}: estimate for n+1 burst; choose \alpha, 0 < \alpha <= 1
- upshot: favor jobs that have been using CPU the least amount of time; that ought to approximate STCF

Priority: give every process a number (set by administrator), and give the CPU to the process with the highest priority; can be done preemptively or non-preemptively

- Bad idea to use strict priority because of starvation of low priority tasks.
- Solution to this starvation is to increase a process's priority as it waits.
- Note: SJF is priority scheduling where priority is the predicted next CPU burst time

Lecture 10: More Scheduling Algorithms

Multilevel Feedback Queue (MLFQ)

- multiple queues, each with different priority
- round-robin within each queue (flush a priority level before moving onto the next one).
- feedback: process's priority changes
- Advantages: approximates STCF without oracle
- Disadvantages: can't donate priority, not flexible, bad for estimating priority, can be gamable

Lottery Scheduling: issue lottery ticket to processes and run processes based on winning ticket

- controls long-term average proportion of CPU for each process
- can also group processes hierarchically for control; subdivide lottery tickets
- Advantages: ensures progress in long run (prevents starvation); adding/deleting jobs will affect all jobs proportionally; can transfer tickets between processes
- Disadvantages: unpredictable latency, high variance
- Note difference between donating tickets and donating priority: with donating tix, recipient amasses enough until it runs.
- **Stride Scheduling:** deterministic version of lottery to reduce randomness (similar to Linux)

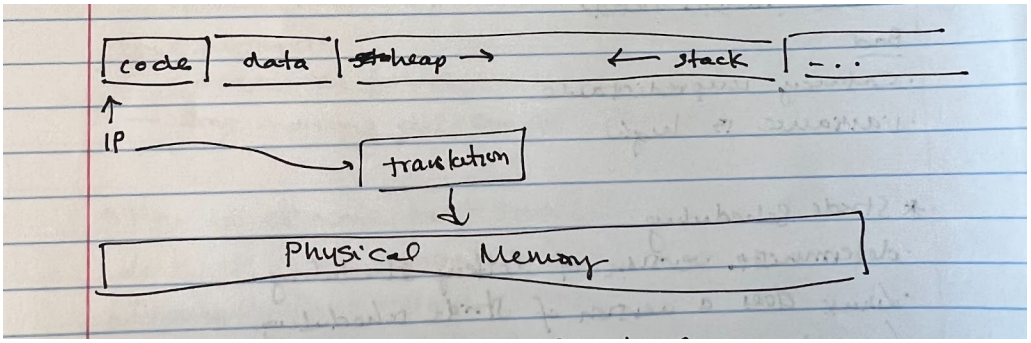
Scheduling Conclusions

- Examples: resources < requests, disk arm, memory; web sites and large-scale networks, real-time systems
- In principle, scheduling decisions shouldn't affect a program's results; rare to calculate the best schedule
    - Cases that can affect correctness: multimedia, web server
- **Mechanism/Policy Split:** *mechanism* allows the OS to switch any time while the *policy* determines when to switch in order to meet whatever goals are desired by the scheduling designer
    - Every mechanism encodes a range of possible policies, constraining possible policies
- Know your goals (and tradeoffs).
- Compare against optimal, even if optimal can't be built; useful benchmark.
- Multiple schedulers interact: mutexes, interrupts, disk, network

Lecture 11: Virtual Memory

Virtual Memory Abstraction: illusion that every process owns the whole (physical, large) memory space.

- [CPU ---> translation box --> physical addresses]



- **Programmability:** program uses contiguous array
    - program *thinks* it has lots of memory, organized in a contiguous space
    - programs can use "easy-to-use" addresses 0x20000; compiler and linker don't have to worry about where the program actually lives in memory when it executes.
    - multiple instances of the same program are loaded, each thinks its using the same memory addresses
- **Protection:** cannot read or write each other's memory
    - prevents corruption
    - don't want spying processes
- **Effective Use of Resources:** programmers don't have to worry that the sum of the memory consumed by all active processes is larger than physical memory.
- **Sharing:** two processes have a different way to refer to the same physical memory cells

Memory Management Unit: OS configured hardware to perform VM translations

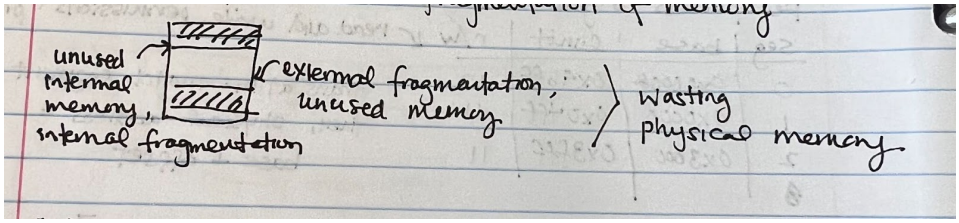
- Why doesn't OS just translate itself? Too slow.
- OS is going to be setting up data structures that the hardware sees; these data structures are *per-process*

Segmentation: memory addresses treated like offsets into a contiguous region.

- Consider 14-bit address: first two bits select are the segment number, next 12 bits give offset
- Advantages: simple, thread-local memory, sandboxing, easy to share memory among processes
- Disadvantages: program may need to know about segments, contiguous bytes required, fragmentation
- Segmentation is old-school and these days mostly an annoyance (but it cannot be turned off on the x86!)

```
                                    virtual                 physical
                                    -------                 ---------
seg     base        limit         rw
-----------------------------------
0       0x4000      0x46ff        10    [0x0000, 0x0700)  -->  [0x4000, 0x4700)
1       0x0000      0x04ff        11    [0x1000, 0x1500)  -->  [0x0000, 0x0500)
2       0x3000      0x3fff        11    [0x2000, 0x3000)  -->  [0x3000, 0x4000)
                                        [0x3000, 0x3fff)  --> not mapped
```

0x0240?	[4240]
0x1108	[0108]
0x265c	[365c]
0x3002	[???]
0x1600	[illegal]

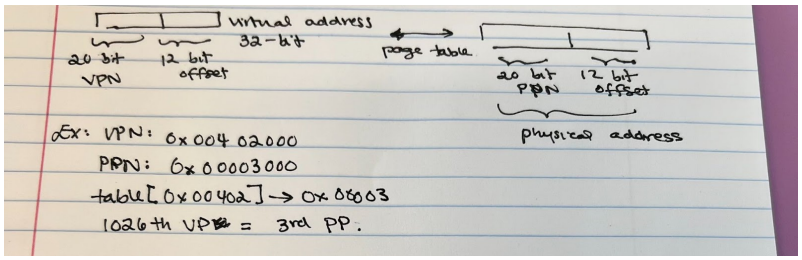


Paging: divide all of memory (physical and virtual) into *fixed-size* pages

- Each process has a separate mapping and each page separately mapped
- We will allow the OS to gain control on certain operations
    - Read-only pages trap to OS on write
    - Invalid pages trap to OS on read or write
    - OS can change mapping and resume application

Page Table:

- assume 32 bit addresses, 4 KB pages
- Naive Solution: page table per process, needs roughly 4MB (2^{20} entries * 32 bits per entry)
- Advantages: fine-grained; eliminates external fragmentation, minimal internal; easier to allocate, free, swap
- Disadvantages: larger data structure, more complex



Lecture 12: Paging

Virtual Memory on x86: has segmentation and paging.

- Cannot turn off segmentation. Instead, set things up so that segmentation has no effect.
- How? Set mapping to be the identity function; make the offset 0 and the limit the maximum.
- **Linear Address**: synonym for "virtual address". (On x86, the segmentation mapping goes from virtual to linear.)

Two Level Mapping Structure

- Divide virtual addresses to 31 ..... 22 [Directory Entry] 21 ..... 12 [Table Entry] 11 ..... 0 [Offset]
- %cr3 is the address of the page directory
- page table location = *(%cr3 + directory entry)
- physical page number = *(table location + table entry)
- actual data = *(physical page number + offset)

[Note that the pointer arithmetic here implicitly multiplies the entries and offset by a factor of 4]

- Entry in page directory and page table: 31.....12 [Entry Value] 11 ..... 0 [Permissions]
    - Note: the entry value, like %cr3, is a physical address in memory
- Each page directory and each page table consumes 4KB of physical memory (1 page) [2^10 entries * 4 bytes]
- Each entry in the page *table* corresponds to 4KB of virtual address space [2^12 possible offsets].
- Each entry in the page *directory* corresponds to 4MB of virtual address space [2^10 possible page tables].

```
uint
translate (uint la, bool user, bool write)
{
  uint pde; /* page directory entry */
  pde = read_mem (%CR3 + 4*(la >> 22));
  access (pde, user, write); /* see function below */
  pte = read_mem ( (pde & 0xfffff000) + 4*((la >> 12) & 0x3ff));
  access (pte, user, write);
  return (pte & 0xfffff000) + (la & 0xfff);
}

// check protection. pxe is a pte or pde.
// user is true if CPL==3.
// write is true if the attempted access was a write.
// PG_P, PG_U, PG_W refer to the bits in the entry above
void
access (uint pxe, bool user, bool write)
{
  if (!(pxe & PG_P)
     => page fault -- page not present
  if (!(pxe & PG_U) && user)
     => page fault -- not access for user

  if (write && !(pxe & PG_W)) {
    if (user)
       => page fault -- not writable
    if (%CR0 & CR0_WP)
       => page fault -- not writable
  }
}
```

Permission Bits: OS is setting them to indicate protection; hardware is enforcing them

- dirty (set by hardware) hasn't been modified?
- accessed (set by hardware) how to prioritize?
- cache disabled (set by OS)
- write through (set by OS) os-performance optimization
- P exists in physical memory?
- user/superuser: who can access?
- read/write: how can it be accessed?

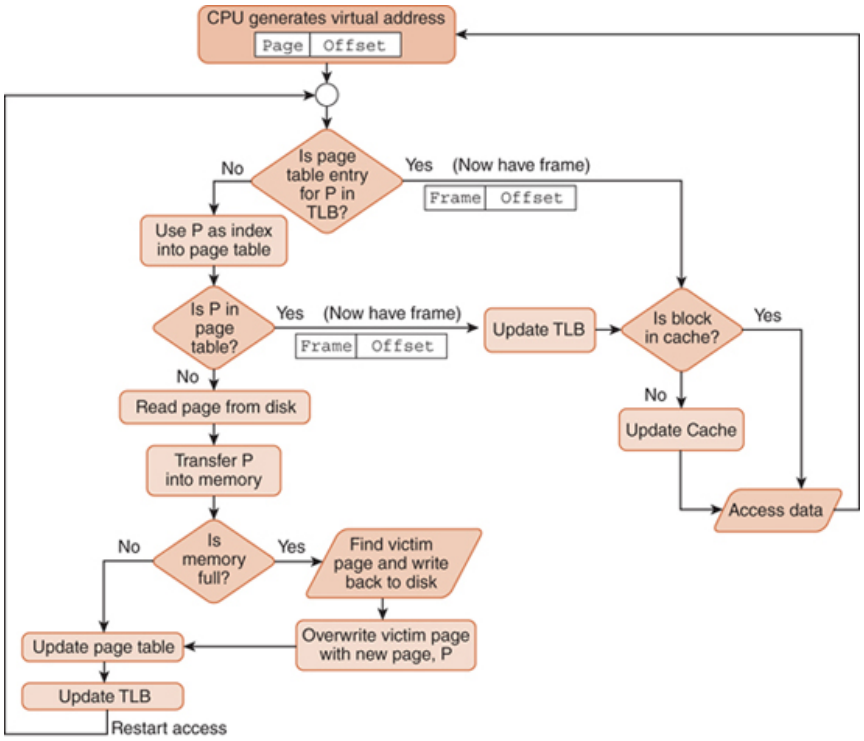
Tradeoffs:

- **Page Size:** large page sizes means wasting actual memory; small page sizes means lots of page table entries

- **Mapping Levels:** more levels means less space spent on page structures when address space is sparse but more memory accesses; fewer levels need to allocate larger page tables, but hardware has fewer accesses

Translation Lookaside Buffers (TLBs): VA -> PA mappings in cache to increase speed

- Hardware-managed in x86; Software-managed in MIPS
- Flushed during context switches (%cr3 load) because each process has its own mapping
- Individual entries flushed on invalid page address
- Hardware populates TLB entries
- TLB miss does not imply page fault; permissions can be different or missing in memory
- Page fault does not imply TLB miss; permissions can be different



Lecture 13: Page Faults

Page Fault: illegal reference due to lack of mapping (present bit) or protection violation (read only); requires OS support

- On x86, the kernel constructs a trap frame and transfers execution to an interrupt or trap handler.
- If protection violation, the OS kills the process.
- If page unmapped, create space in physical memory to swap in page from disk; update page dir/table; resume
    - Kernel need to send a page to disk on 2 conditions: kernel full, selected page is dirty (modified)

Uses of Page Faults:

- Overcommitting physical memory (program has 512 MB of memory, hardware has 256 MB of memory)
    - disk is used to store memory pages

    - Advantage: address space looks huge
    - Disadvantage: accesses to "paged" memory (pages that live on the disk) are slow
- **Distributed Shared Memory:** store memory pages across the network!
    - On a page fault, handler retrieves the needed page from some other machine
- **Copy-on-write:** when copying another process, just copy its page tables, mark the pages as read-only
    - Program semantics aren't violated when programs do reads
    - When a write happens, a page fault results; kernel allocates a new page, copies the memory over, and restarts the user program to do a write
- **Accounting:** good way to sample what percentage of the memory pages are written to in any time slice: mark a fraction of them not present, see how often you get faults

Uses of Paging:

- **Demand paging:** bring program code into memory "lazily"
- Growing the stack (contiguous in virtual space, probably not in physical space)
- BSS page allocation (BSS segment contains the part of the address space with global variables, statically initialized to zero. OS can delay allocating and zeroing a page until the program accesses a variable on the page.)
- Shared text, libraries, memory

Cost of Page Faults:

- **Average Memory Access Time** (AMAT) = memory access time + P(page fault) * page fault time
- Page faults are super-expensive (good thing the machine can do other things during a page fault)
- Concept is broader than OS: need to pay attention to how slow and how common it is.

Lecture 14: Page Replacement Policies

Possible Policies

- In virtual memory, the pages resident in memory are basically a cache to the backing store on the disk.
- **FIFO:** throw out oldest (results in every page spending the same number of references in memory.)
- **MIN/OPT**: throw away the entry that won't be used for the longest time.
- **LRU:** throw out the least recently used (assume future looks like the past)
- **Belady's Anomaly:** increasing cache size can decrease hit rate

- **Stack property:** algorithms like LRU unaffected by Belady's Anomaly; a cache of size N + 1 naturally includes the contents of a cache of size N. Thus, when increasing the cache size, hit rate will either stay the same or improve.

LRU Implementation

- reasonable in web servers that cache pages (or dedicated Web caches; use queue to track least recently accessed and use hashmap to implement the (k,v) lookup
- in OS, LRU itself does not sound great; would be doubling memory traffic
- and in hardware, it's way too much work to timestamp each reference and keep the list ordered

Clock Algorithm

- arrange cache slots in a circle; hand sweeps around, clearing a bit. the bit is set when the page is accessed. just evict a page if the hand points to it when the bit is clear.
- approximates LRU because we're evicting pages that haven't been used in a while....though of course we may not be evicting the *least* recently used one due to the cyclical nature
- **Nth Chance Algorithm:** generalization; don't throw a page out until the hand has been swept by N times.
    - OS keeps counter per page: # sweeps
    - On page fault, OS looks at page pointed to by the hand and checks that page's use bit
        - 1 --> clear use bit and clear counter; 0 --> increment counter
        - if counter < N, keep going; if counter = N, replace the page: it hasn't been used in a while
    - Large N --> better approximation to LRU; Small N --> more efficient.
- Dirty pages are more expensive to evict (need to write); give dirty pages an extra chance before replacing

Fairness: on page swap, is the pool of consideration per process?

- **Global Policies:** more flexible but less fair
- **Local Policies:** less flexible but fairer

Thrashing: Processes require more memory than system has; Specifically, each time a page is brought in, another page, whose contents will soon be referenced, is thrown out

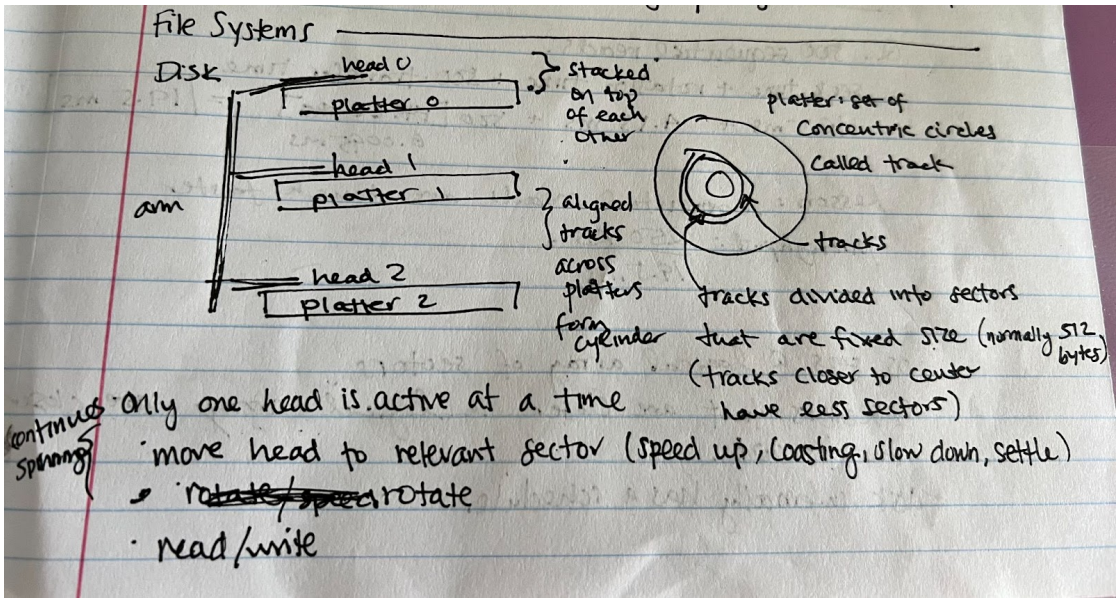
- process doesn't reuse memory (or has no temporal locality) [No fix other than more / restructure memory]
- process reuses memory but the memory that is absorbing most of the accesses doesn't fit [no fix]
- individually, all processes fit, but too much for the system; need to shed load
    - **Working Set:** only run a set of processes s.t. the union of their working sets fit in memory; the pages a processed has touched over some trailing window of time

  - **Page Fault Frequency:** track; if above a threshold, not enough memory; swap out the process
- **Moral:** that if the workload is not cache-friendly, the policy is irrelevant.

Lecture 15: File Systems

Disk Geometry

- **Track:** circle on a platter. each platter is divided into concentric tracks.
- **Sector:** chunk of a track
- **Cylinder:** locus of all tracks of fixed radius on all platters
- **Head:** roughly lined up on a cylinder; generally only one head active at a time
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned
- **Disk Positioning System:** move head to specific track, resist physical shocks, imperfect tracks, etc.



Performance

- **Seek:** consists of up to four phases
  - **Speedup:** accelerate arm to max speed or half way point
  - **Coast:** at max speed (for long seeks)
  - **Slowdown:** stops arm near destination
  - **Settle:** adjusts head to actual desired track

- **Components of Transfer:** rotational delay, seek delay, transfer time.
    - **Rotational Delay:** time for sector to rotate under disk head
    - **Seek:** speedup, coast, slowdown, settle
        - seeking track-to-track: comparatively fast (~1ms); mainly settle time
        - short seeks (200-400 cyl.) dominated by speedup; longer seeks dominated by coast
        - head switches comparable to short seeks
        - settle times takes longer for writes than reads; writes need to be precise and without error
    - **Transfer Time:** time to transfer data to/from disk
- **Takeaway**: sequential reads are MUCH faster than random reads; goal is to perform sequential reads.
    - Disk Cache used for read-ahead. Otherwise, sequential reads would incur whole revolution.
    - Policy Decision: should read-ahead cross tracks? aggressive read ahead forces head-switch.
    - **Write Caching:** if battery backed, data in the buffer can be written over many times before actually being put back to disk. also, many writes can be stored so they can be scheduled more optimally
    - If not battery backed, then policy decisions between disk and host about whether to report data in cache as on disk or not.
    - Can try to order requests (if multiple exists) to minimize seek times; requires disk I/O concurrency; high-performance apps try to maximize I/O concurrency; or avoid I/O except to do write-logging

Disk Scheduling

- **FCFS:** process requests in the order they are received
    - Advantages: easy to implement, good fairness
    - Disadvantages:: cannot exploit request locality, increases average latency/decreases throughput
- **SPTF/SSTF/SSF/SJF:** shortest positioning/seek time first: pick request with shortest seek time
    - Advantages: exploits locality of requests, higher throughput
    - Disadvantages: starvation, oracle
    - **Aged SPTF:** give older requests priority; adjust effective seek time with weighting factor
- **Elevator Scheduling:** SPTF with next seek in same direction; switch direction only if no further requests
    - Advantages: exploits locality, bounded waiting
    - Disadvantages: cylinders in the middle get better service, doesn't fully exploit locality
    - Modification: only sweep in one direction: very commonly used in Unix.

Technology and System Trends

- While seeks and rotational delay are getting little faster, they have not kept up with the huge growth
- Transfer bandwidth has grown about 10x per decade
- Disk Density is growing fast (byte_stored/$) because density is less about the mechanical limitations
- To improve density, need to get the head close to the surface
- Disk accesses a huge system bottleneck and is getting worse.
    - Bandwidth increase lets the system (pre-)fetch large chunks for about the same cost as a small chunk.
    - So trade latency for bandwidth if you can get lots of related stuff at roughly the same time.
    - Cluster the related stuff together on the disk; grab huge chunks of data without incurring a big cost
- Memory size is increasing faster than typical workload size; more workloads fit in file cache: the profile of traffic to the disk has changed to now mostly writes and new data, with logging and journaling viability.

Drivers Interfacing with Disk

- Disk interface presents linear array of sectors, generally 512 bytes, written atomically
- Larger atomic units have to be synthesized by OS; goes for multiple contiguous sectors or even a whole collection of unrelated sectors and OS will make this operation appear atomic
- Disk maps logical sector # to physical sectors
- **Zoning:** puts more sectors on longer tracks
- **Track Skewing:** sector 0 position varies by track, for speed when doing sequential access
- **Sparing:** flawed sectors remapped elsewhere
- The OS does not know the logical to physical sector mapping; highly non-linear relationship
    - Larger logical sector # difference means larger seek
    - OS has no info on rotational positions; need to empirically build table to estimate times

Job of a File System

- provide persistence (durability)
- somehow associate bytes on the disk with names (files)
- somehow associates names with each other (directories)
- Implementation: written onto disk, but may screw up system on error (other implementations as well)

Access Patterns

- **Sequential:** file data processed in sequential order; most common
- **Random Access:** address any block in file directly without passing through
- **Keyed Access:** search for block with particular values

Files

- **User's View:** a bunch of named bytes on the disk
- **File System's View:** collection of disk blocks
- **File System:** map name and offset to disk blocks {file,offset} --> disk address
- **Operations:** create(file), delete(file), read(), write()
    - operations have as few disk accesses as possible and minimal space overhead
    - cache space is never enough; the amount of data that can be retrieved in one fetch is never enough
    - try to keep consistent accesses close in logical array
- RAID supports this by providing an interface
- **Translation Scheme:** file name ----- directory -----> inode; offset --- inode---> disk block address

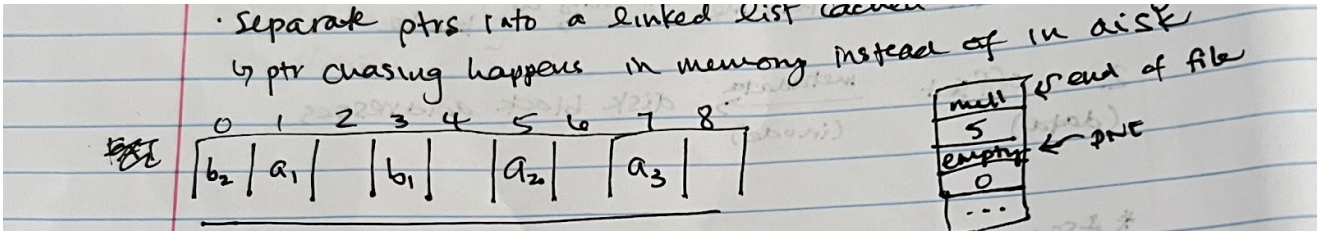
Observations:

- All blocks in file tend to be used together, sequentially
- All files in directory tend to be used together
- All *names* in directory tend to be used together
- Most files are small
- Much of the disk is allocated to large files
- Many of the I/O operations are made to large files
- Want good sequential and good random access

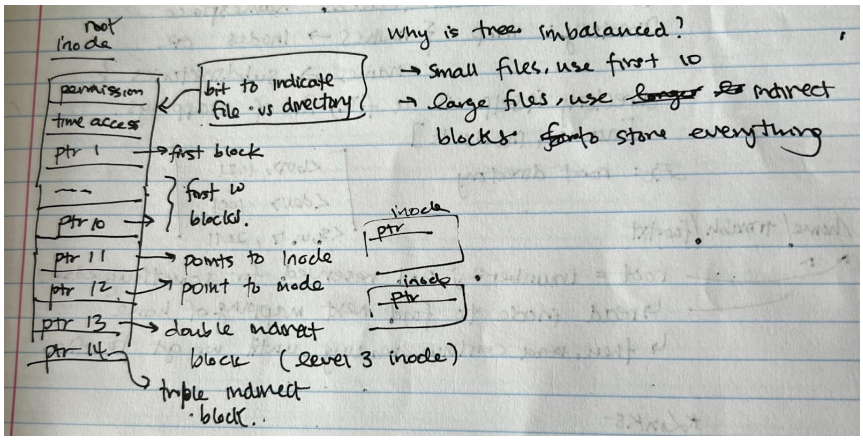
File System Designs:

- **Contiguous Allocation**: when creating a file, make user pre-specify its length, and allocate the space at once
    - Metadata: location and size
    - Advantages: simple, fast access, both sequential and random
    - Disadvantages: fragmentation
- **Linked Files:** keep a linked list of free blocks
    - Metadata: pointer to file's first block, each block holds pointer to next one
    - Advantages: no more fragmentation, sequential access easy
    - Disadvantages: random access is a disaster, pointers take up room; messes up alignment of data
- **File Allocation Table:** keep link structure in memory in fixed-size "FAT"; pointer chasing now happens in RAM
    - Metadata: first block
    - Advantages: no need to maintain separate free list, low space overhead

- Disadvantages: limited maximum size, need multiple copies to guard against bad sectors



- **Indexed Files:** each file has an array holding all of its block pointers, like a page table, so similar issues crop up; allocate this array on file creation; allocate blocks on demand (using free list)
  - Each file has own metadata structure
  - Advantages: sequential and random access are both easy
  - Disadvantages: need to somehow store the array



- **Tradeoff:** large file = unused entries in the block array; large block = huge contiguous disk chunk
  - Use multilevel scheme to not waste disk blocks, but requires extra memory accesses
  - **Imbalance:** optimize for short files. each level of this tree requires a disk seek
  - Advantages: simple, fast access to small files, maximum file length can be enormous
  - Disadvantages: worst case # of accesses, worst case overhead, data get strewn across disk

Inodes: stored in a fixed-size array; size of array fixed when disk is initialized; can't be changed

- multiple inodes in a disk block
- Lives in known location, originally at one side of disk, now lives in pieces (keep metadata close to data)
- The index of an inode in the inode array is called an i-number (the OS refers to files by i-number)
- When a file is opened, the inode brought in memory
- Written back when modified and file closed or time elapses

Lecture 16: Directories

- Approach 0: have users remember where on disk their files are
    - People want human-friendly names
- Approach 1: Single directory for entire system; If one user uses a name, no one else can
    - Put directory at known location on disk; Directory contains <name,inumber> pairs
    - Not scalable, quite messy with unique filenames
- Approach 2: Single directory for each user
    - Still clumsy, but at least multiple users have the same file names
- Approach 3: Hierarchical name spaces; Allow directory to map names to files or other directories
    - File system forms a tree (or graph, if links allowed)
    - Directory itself is a file; file of [name, inode mappings]
        - i-node for directory contains a special flag bit; only special users can write directory files
        - i-number might reference another directory; this neatly turns the FS into a hierarchical tree
        - if you speed up file operations, you also speed up directory operations
    - Start looking at root dir (always inode #2)
    - special names: "/", " . ", " .. "

Links:

- **Hard Link:** multiple dir entries point to same inode; inode contains refcount
    - may create circular references
- **Soft Link:** synonym for a *name*
    - creates a new inode, not just a new directory entry; new inode has "sym link" bit set