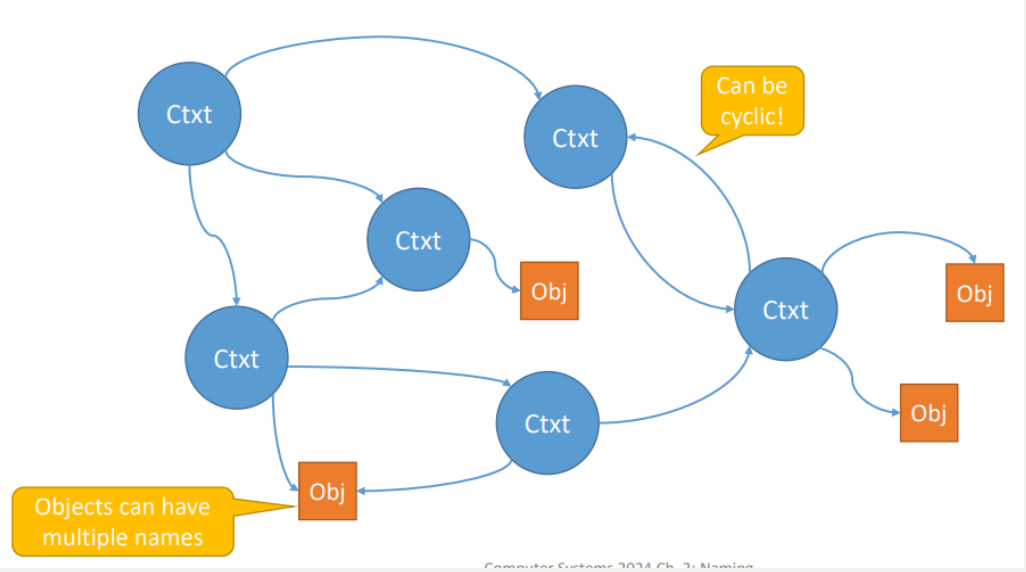
252-0217-00L Computer Systems

Sept. 23rd - Lecture 2: Virtualization and Naming + Classic OS and Kernel

Chapter 2: Virtualization and Naming

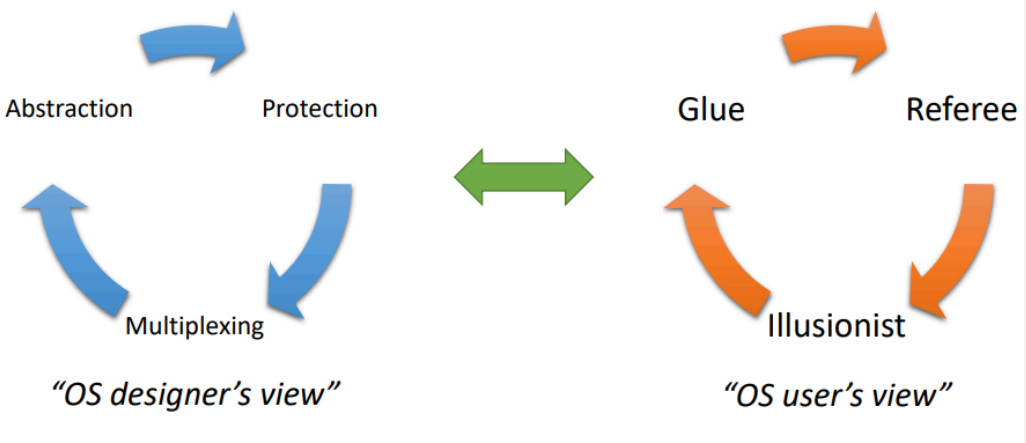
- Naming is a fundamental concept
    - **Names** allow *resources* to be bound at *different times*
    - **Contexts** allow applications to *sandboxed*
        - Set of bindings
    - Replacing **bindings** enables *virtualization* of resources
- Example of names
    - "Roger Wattenhofer"
    - "Pfaeffikon"
    - 192.168.1.1 (IP address - name of networking interface)
    - `int i;` (names a 4-byte location in the memory), `foo` (meta-variable)
- Bindings and contexts
    - Name $\xrightarrow{\text{Binding}}$ Object
    - Context: {Bindings} (also, name-scope)
    - Names are bound to objects; this is always relative to a context
- Name Resolution
    - $(\text{context}, \text{name}) \implies \text{object}$
    - What if the object is, itself, a context?
        - (context, name) → context' (an object)
    - Path names (allows successive resolution of names)
        - $(\text{context}_0, \text{name}_1, \text{name}_2, \ldots) \rightarrow \text{object}$
        - e.g. `/usr/bin/emacs` ↔ `[root, 'usr', 'bin', 'emacs']`

        - e.g. `import email.mime.multipart` ↔ `[PYTHONPATH, 'email', 'mime', 'multipart']`
        - DNS Name: `www.inf.ethz.ch`, Resolution goes from right to left → ['ch', 'ethz', 'inf', 'www']
        - Resolving an email address: `troscoe@inf.ethz.ch`
            - `inf.ethz.ch`: Resolve via DNS to an MX record → Resolve MX machine
            - Resolve `troscoe` to mailbox on server
    - Other resolution algorithms: search paths
        - Used in the Unix shell when resolving the **name of a command** (first search through a PATH directory)
- Naming Networks
    - Objects can have multiple names
    - Can be cyclic
    - 

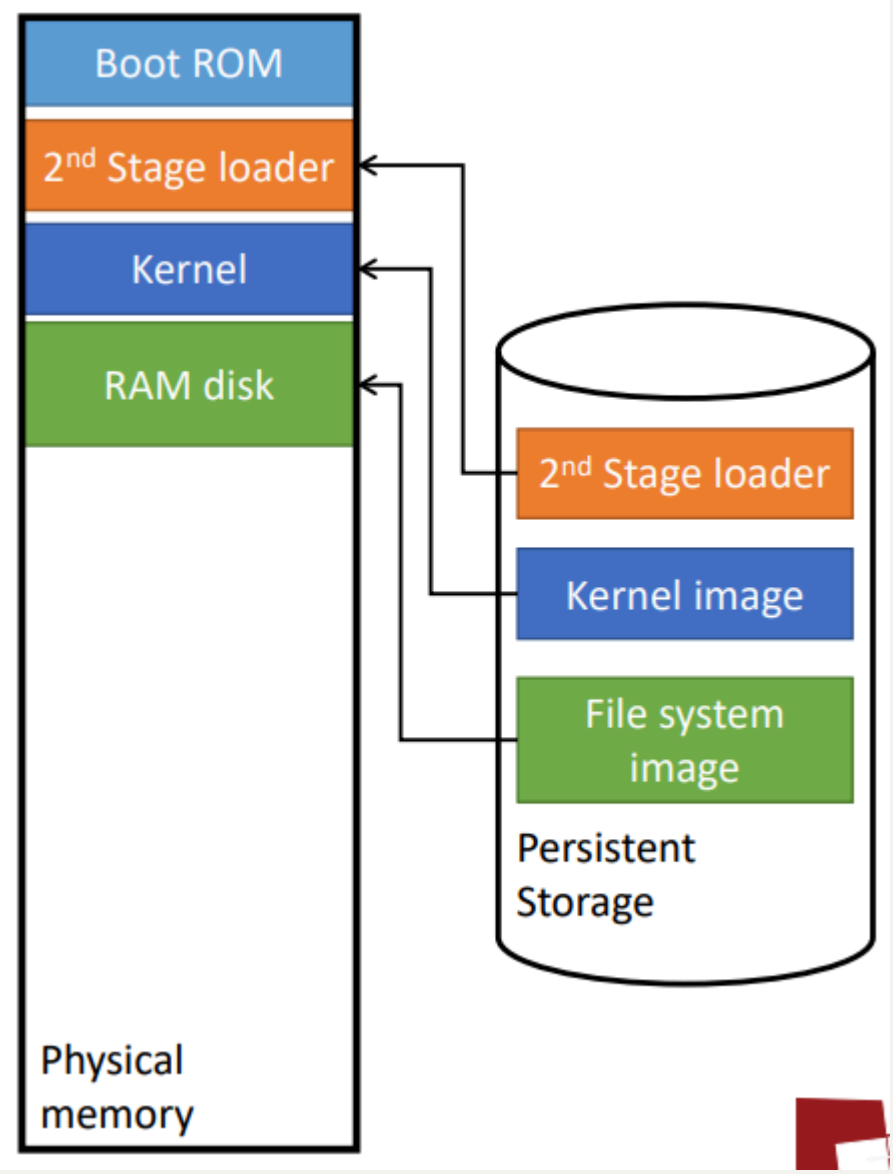
- Homonyms and Synonyms
    - e.g. `192.168.1.1` is home router address; any-cast address (packet arrives in one of them), multi-cast address (packet arrives in all of them)
    - Synonyms: two names bound to the same object (e.g. Links to same file)
        - In different contexts
        - In the same context
    - Homonyms: the same name bound to two different objects
        - In different contexts
        - In the same context?
- Symbolic names

        - `(context, name)` → name′ (another name in the same context)
        - e.g. `ln -s /usr/bin/dc bin/calc`
            - Bind the **string** `/usr/bin/dc` as an object to some name
            - Resolve `bin` to a context; bind some strong to the name `calc` in that context
- "The design of a good naming scheme is a question of taste as well as technical skill"

Chapter 3: Classic OS and Kernels

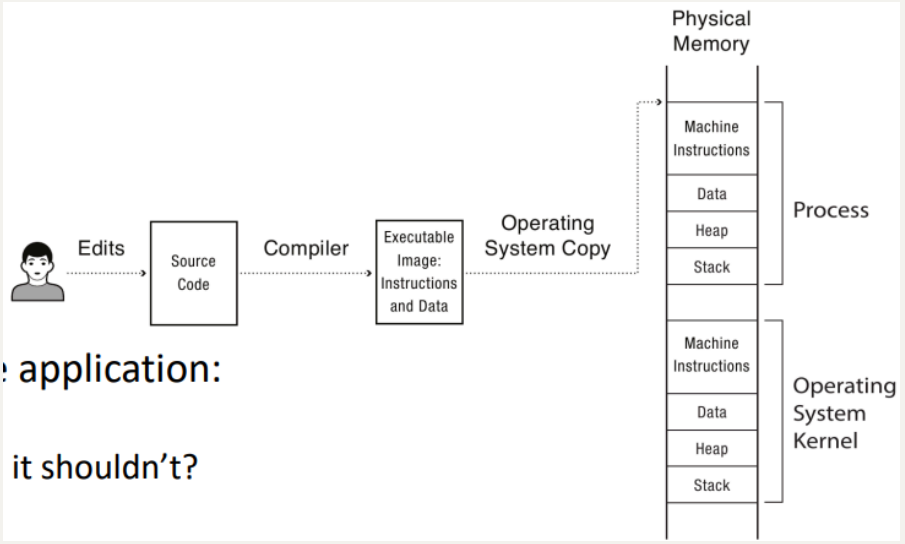
- The functions of the OS
    - 

- Bootstrap (how does the OS kernel actually end up running)
    - OS is a program, kernel is a program
    - How a computer boots
        - CPU starts executing code at fixed address (Boot ROM at this address)
        - Boot ROM code loads 2nd stage boot loader into RAM (Jumps into 2nd stage boot loader code)
        - Boot loader loads into RAM (Kernel, (optionally) initial file system)
        - Jumps to kernel entry point

- 

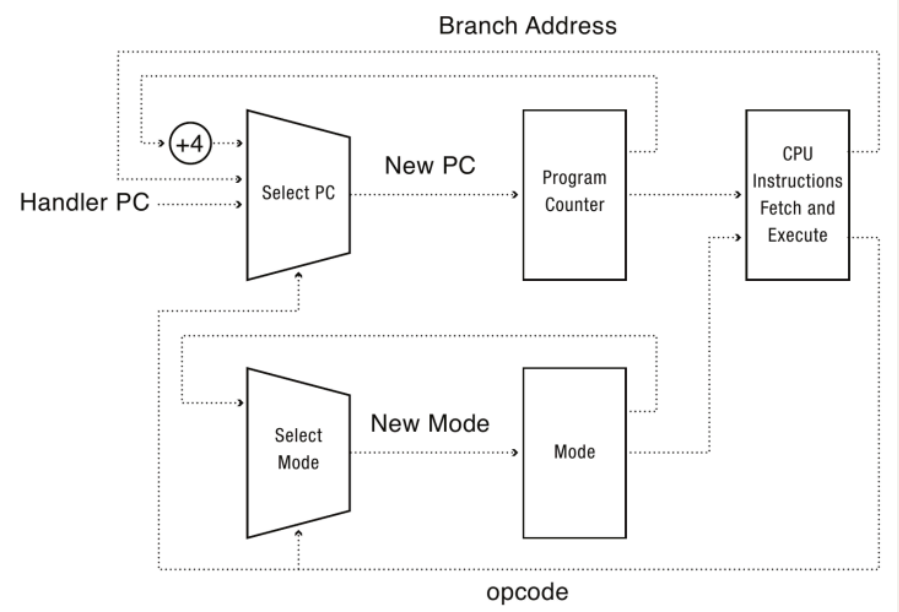
    - At the start of a model computer, DRAM does not exist
  - Does a little bit more at each stage
  - Register remembers the mode it was in before an exception happens, and will return to the same mode after coming back from the exception; when we turn on the computer, we can mimic a situation in which a specific exception happens, and return to the mode we want the register to be in; this allows future exceptions of the same type to return to the original mode properly without excess code, and we can use the same process to start a computer
- The Mode Switch (Protecting the OS from the applications)
  - A problem

- Assume one application

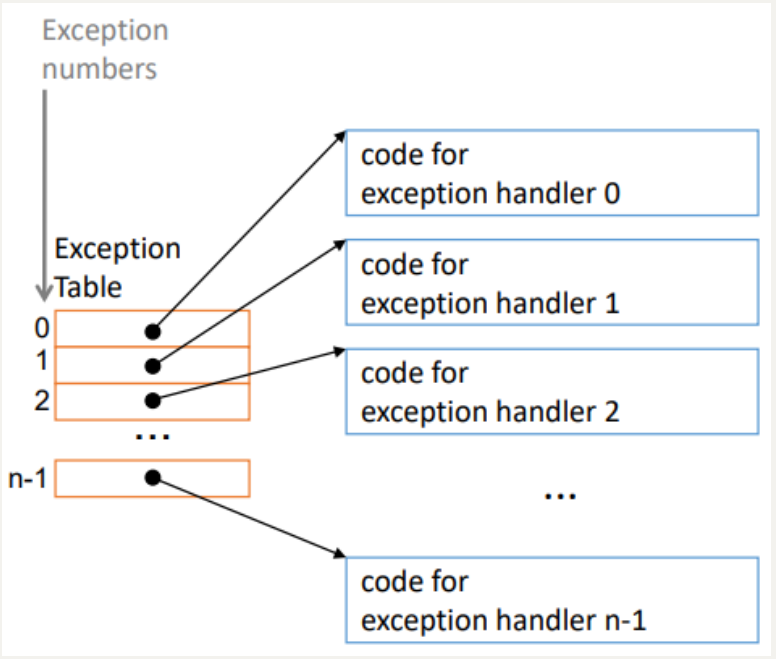


  - How can we prevent the application
    - from corrupting the OS
    - from accessing hardware that it should not
- Part of a more general challenge: Protection
  - How do we execute code with restricted privileges?
    - Either because the code is buggy or if it might be malicious
  - e.g. a script running in a web browser; a program you just downloaded off the Internet etc
- Thought experiment
  - How can we implement execution with limited privilege?
    - Execute each program instruction in a simulator
    - If the instruction is permitted, do the instruction
    - Otherwise, stop the process
    - Basic model in JavaScript and other interpreted languages
  - How do we go faster
    - Run the unprivileged code directly on the CPU
- Hardware Support: Dual-Mode Operation
  - Kernel mode
    - execution with the full privileges of the hardware
    - read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet

        - code must be carefully written
    - User mode
        - Limited privileges
        - Only those granted by the operating system kernel
    - CPU with dual-mode operation



        - Ensures Privilege Mode, never execute User Mode code
- Basic primitive: processor exceptions
    - When an exception occurs
        - Finish executing current instruction
        - Switch mode from user to kernel
        - Look up exception cause in the execution vector table
        - Jump to this address

- 

    - And possibly
        - Save registers (or switch banks)
        - Switch page table (usually not)
- Types of Exceptions
    - A **synchronous** exception occurs as a results of executing an instruction
    - An **asynchronous** exception occurs as a result of events that are external to the processor
    -

Type of exception	Cause	Async/Sync
Interrupt	Signal from I/O device	Async
Trap	Intentional exception	Sync
Fault	Potentially recoverable error	Sync
Abort	Nonrecoverable error	Sync

Sept. 25th - Exercise Session 1

- Naming Basics
    - `Object my_object = new Object()`: name - `my_object`, object - `new Object()`, binding - `=`
    - Context
- Naming Network
- Search Paths
    - Tell the shell where to search for an executable
- The role of the OS

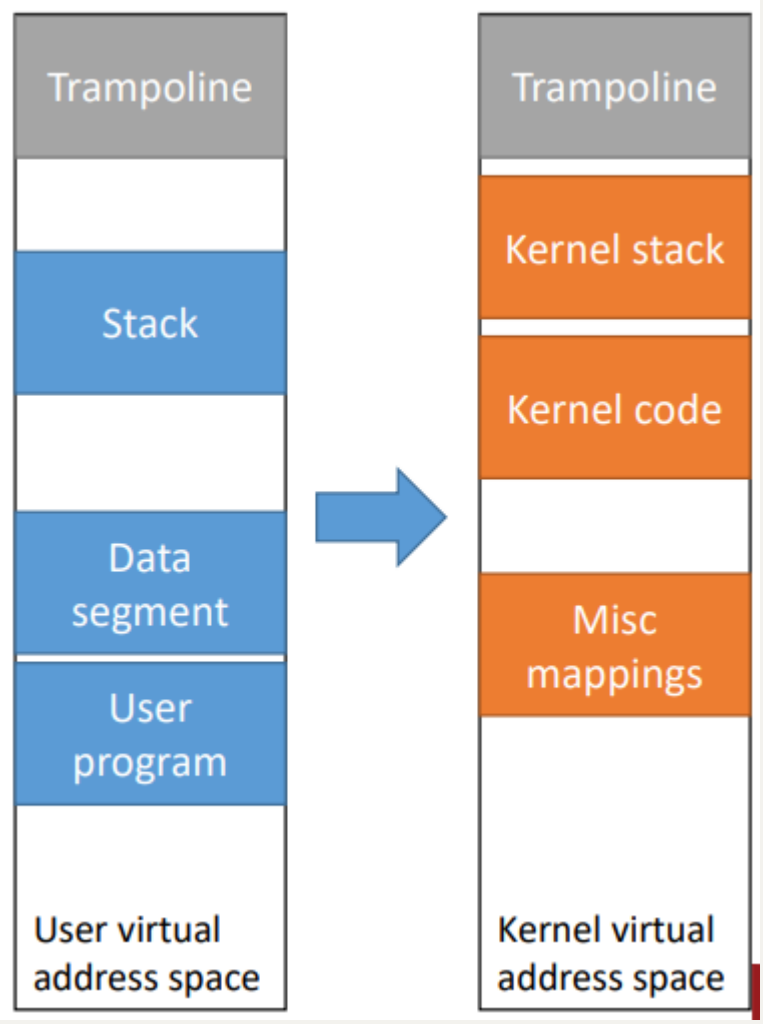
        - Referee
            - Ensure resource sharing / protection / inter-process communication
        - Illusionist
            - provide virtual resources to user-space processes
            - virtual memory
            - shared network interface
        - Glue
            - provide high-level abstraction to user-space application
- General OS Structure
        - Kernel (in Privilege)
            - Special process that runs in privileged mode
            - typically event driven server
        - System Library (in User)
            - Convenience functions
            - System call wrappers
        - Daemon (in User)
            - Process which are part of the OS but live in user-space
            - Provides modularity, fault-tolerance
- Micro- vs. Monolithic Kernel
        - Monolithic OS
            - lots of privileged code
            - services invoked by syscall
            - Faster performance due to direct communication in kernel space
            - Less stable
            - Less secure
            - Less modular
        - Microkernel OS
            - little privileged code
            - services invoked by IPC
            - "horizontal" structure

        - Slower performance due to IPC and context switching
        - More stable, faults in services don't crash the whole system
        - Smaller attack surface, safer to update, more secure
        - Highly modular, easy to update components
- Bootstrapping
    - Power on
    - Load BIOS from ROM to memory
    - BIOS loads boot loader into memory
    - boot loader loads OS (kernel) from disk
    - Transfer control to OS
- Entering and leaving the kernel
    - On start-up
    - Exception occurs (caused by program)
    - Interrupt occurs (caused by "smth" else)
    - upon a System Call
    - User process runs → Execute syscall → Execute kernel mode → Resume process
- Pure name: a pure name encodes no useful information about whatever object it refers to
- Symbolic link can point to a name (file, directory, etc) / another link
- Assignment 1 Overview
    - Compile (and modify) the Linux kernel
    - Build the smallest bootable kernel
    - Optional: Compile the Barrelfish OS
    - Read the instructions carefully

Sept. 27th - Lecture 3: Classic OS and the kernel

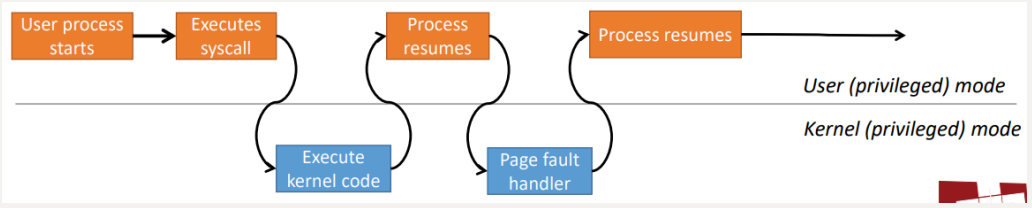
- Exception handling path in xv6
    - From User mode
        - Vector table points to `trampoline.s:uservec`
        - Save trapframe, load kernel more stack pointer and page table base

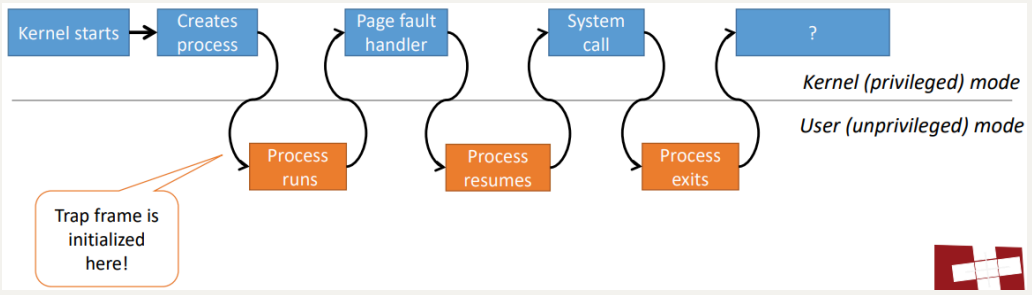
                - First store `a0` register to a register only available in kernel mode `sscratch`
                - Then load an address to `sscratch` (each process has a separate memory area - trapframe, but it's mapped to the same virtual address)
                - Save the user registers in the trapframe
                - Then recover user `a0`
                - Now we can initialize the kernel stack pointer, different from the user stack
                - Then fetch kernel's page table
                - Note: the region of code exists in the same physical address in both kernel address space and user space, so we can use this "trampoline" to enter/exit the kernel
            - Jump to `trap.c:usertrap`
                - First thing we do is to send interrupts and exceptions to `kerneltrap()`, we only handle the user's exception in `usertrap`, others get handled elsewhere
                - Save the user program counter (information used to return to User mode)
                - Figure out what happened for the user space
                    - If the process is not "dead", then first figure out where to jump back after the system call is done, which is the next instruction
                    - Only when faults occur, we want to go back to the original program counter
                    - Disable external device interrupt until relevant information is saved (user PC, kernel stack address etc)
        - From Kernel mode
            - Vector table points to `kernelvec.s:kernelvec`
            - Save registers an stack
            - Jump to `trap.c:kerneltrap`
        - Note: timer interrupts are handled differently (used for process switches)
- Trampolines
    - On RISC-V, mode switch does not change page table

        - Must load new PTBR after mode switch
    - Reloading PTBR must not change the code that is executing
    - Solution: map a page of code at the same address in both page tables
        - This code swaps the page table safely
    - 

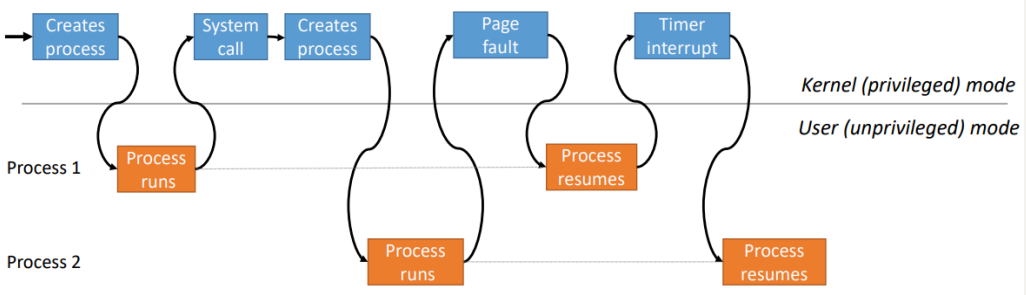
- Mode switch the other way
    - From kernel mode to user mode
        - New process/new thread start (jump to first instruction in program/thread)
        - Return from interrupt, exception, system call (Resume suspended execution)
        - Process/thread context switch (Resume some other progress)
        - User-level upcall (UNIX signal) (Asynchronous notification to user program)
- Returning from Kernel mode in xv6
    - Last line of `trap.c:usertrap()` calls `trap.c:usertrapret()`
        - Initializes the trapframe for next trap

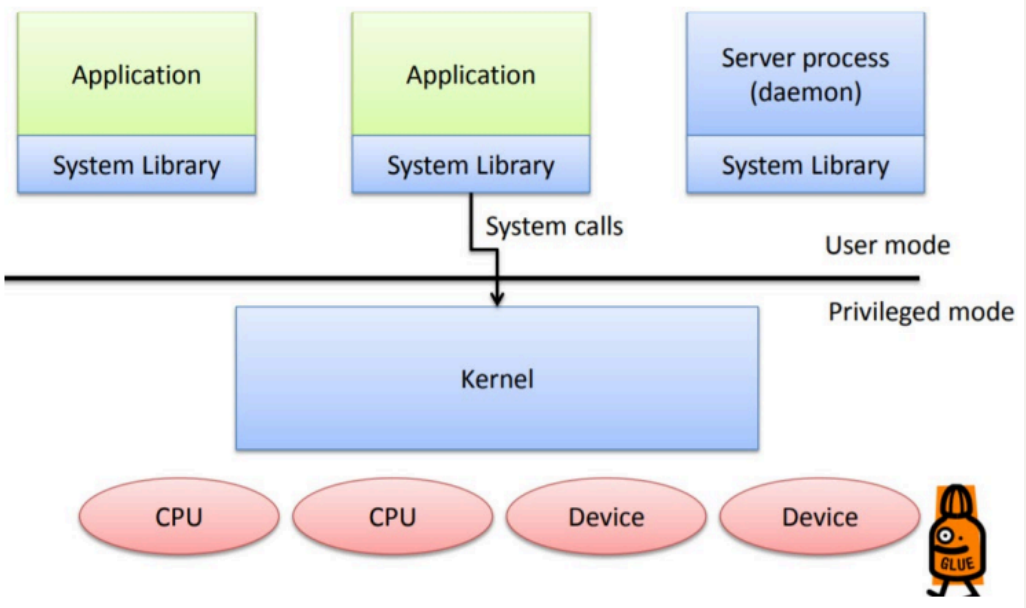
        - First turn off interrupts until we return to User space
        - Then find the address of the trampoline
        - Set up trapframe values that **`uservec`** will need when the process next traps into the kernel
        - Calculate where the trampoline is and...
    - Calls assembly **`trampoline.S:trampoline_userret()`**
- In the trampoline page: **`trampoline_userret()`**
    - Swaps page back to user address space
    - Restore all registers
    - Execute **`sret`** instruction on return to user mode
- From **`trap.c:kerneltrap()`**
    - Much simpler
    - Essentially a local return using the kernel stack

- Conventional perspective
    - 

    - User programs run until the kernel needs to
        - system call, page fault, interrupt
- Alternative perspective:
    - 

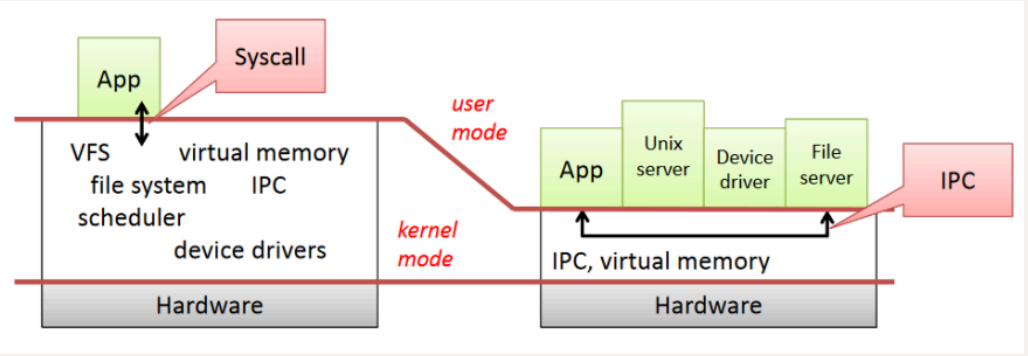
    - Kernel runs, calls sandboxed user applications, then retakes control
- The mode switch is *fundamental* to modern computers!
    - Mode switch enables **virtualization** of the processor
        - Fundamental basis for a function computer
    - Creates **illusion** of multiple computers
        - Each user space context is unaware of the others
    - **Referees** access to the CPU

        - Only one user space context can run at a time
        - User space cannot access kernel mode data or code
    - 

    - Processor
        - A mode switch, a timer interrupt, a memory management → modern OS
- General model of OS structure
    - 

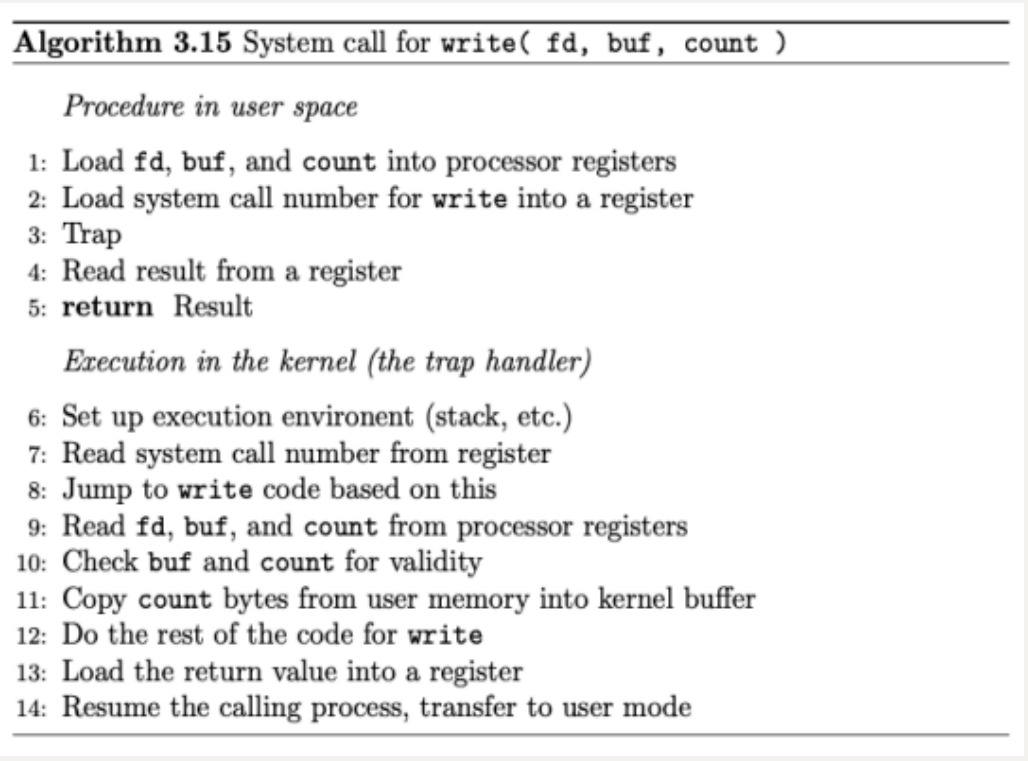
    - Kernel:
        - that code which runs in kernel mode
    - System library
        - convenient interface to kernel
        - useful facilities most programs need
    - Daemon
        - user-space process which provides OS services
        - provides modularity, safety
        - varying levels of privilege / access rights
- Monolithic kernels vs. Microkernels
    - Monolithic OS

        - lots of privileged code
        - services invoked by `syscall`
    - Microkernel OS
        - little privileged code
        - services invoked by IPC
        - "horizontal" structure
    -



System Calls

- Kernel system call handler
    - **Locate** arguments
        - In registers or on user stack
        - **Translate** user addresses into kernel addresses
    - **Copy** arguments
        - From user memory into kernel memory
        - Protect kernel from malicious code evading checks
    - **Validate** arguments
        - Protect kernel from errors in user code
    - **Copy results** back into user memory
        - **Translate** kernel addresses into user address
- xv6: Consider `write()`
    - e.g. `user/printf.c`
    - Assembly stub generated by `user/usys.pl`
    - Loads system call number
    - Executes syscall instruction
    - Trap to `kernel/trap.c:usercall()`

        - Dispatch in `kernel/syscall.c:syscall()`
            - First check whether the syscall number is in range
            - If in range, we check the table of function pointers to figure out what syscall is used
            - store the return of desired syscall to `a0` in trapframe
        - Call `kernel/sysfile.c:sys_write()`
        - Store return value in trapframe
        - Return to User space
        - 

Algorithm 3.15 System call for `write( fd, buf, count )`

Procedure in user space

1: Load `fd`, `buf`, and `count` into processor registers
2: Load system call number for `write` into a register
3: Trap
4: Read result from a register
5: **return** Result

Execution in the kernel (the trap handler)

6: Set up execution environent (stack, etc.)
7: Read system call number from register
8: Jump to `write` code based on this
9: Read `fd`, `buf`, and `count` from processor registers
10: Check `buf` and `count` for validity
11: Copy `count` bytes from user memory into kernel buffer
12: Do the rest of the code for `write`
13: Load the return value into a register
14: Resume the calling process, transfer to user mode

- The kernel is just a program
    - albeit quite a special program
    - can change it
    - remember
        - you don't have the usual libraries
        - even `printf()` is build using the kernel code (`sys_write` etc)

Sept. 30th - Lecture 4: Classical OS and the Kernel + Processes

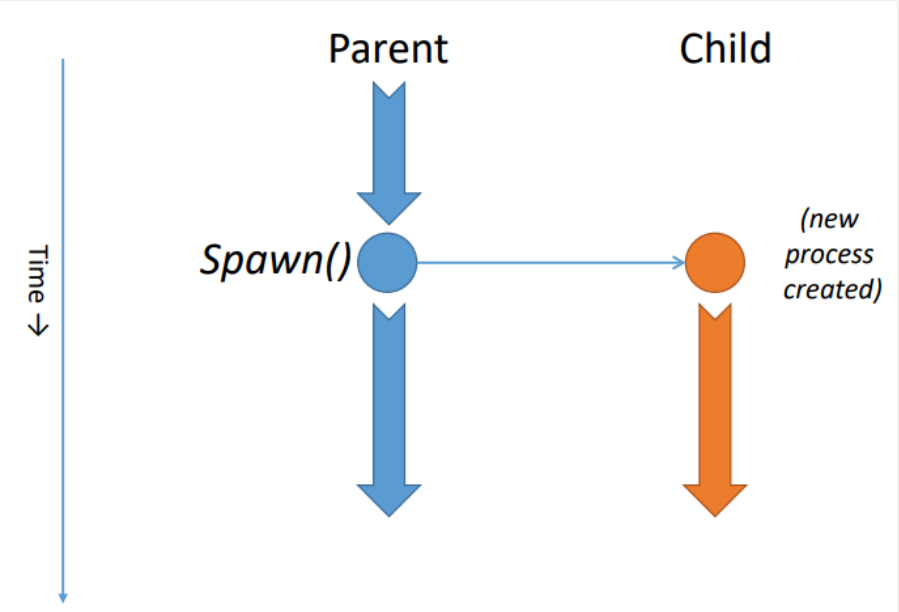
- What if the user program does not cooperate?
    - Hardware timer
        - hardware device that periodically **interrupts** the processor

        - returns control to the kernel handler
        - time of next interrupt set by the kernel
            - not by user code
            - periodic, or repeated one-shot
        - interrupts can be temporarily deferred
            - not by user code
            - interrupt deferral crucial for implementing mutual exclusion
- Interrupt masking
    - interrupt handler runs with interrupts off
        - re-enabled when interrupt completes
    - OS kernel can also turn interrupts off
        - e.g., when determining the next process/thread to run
        - only applies to the current CPU (on a multicore)
    - Used in kernel to help implement atomic operations
- How do we take interrupts safely?
    - Interrupt vector
        - limited number of entry points into kernel
    - On some processors: single instruction atomically changes
        - program counter, stack pointer, address space (page table), processor mode (user to kernel)
    - On RISC-V, more work needed
        - from user mode appears as any other exception
        - from kernel mode, uses the existing kernel stack
- Timer interrupt handling on xv6
    - Processor interval timer interrupt set up in `start.c:timerinit()`
        - Actually causes interrupt in a different mode below the kernel
        - Handler is `kernelvec.S:timervec()`
        - Resets the timer, then raises a regular interrupt
    - Generic device interrupts handled in `trap.c:devintr()`

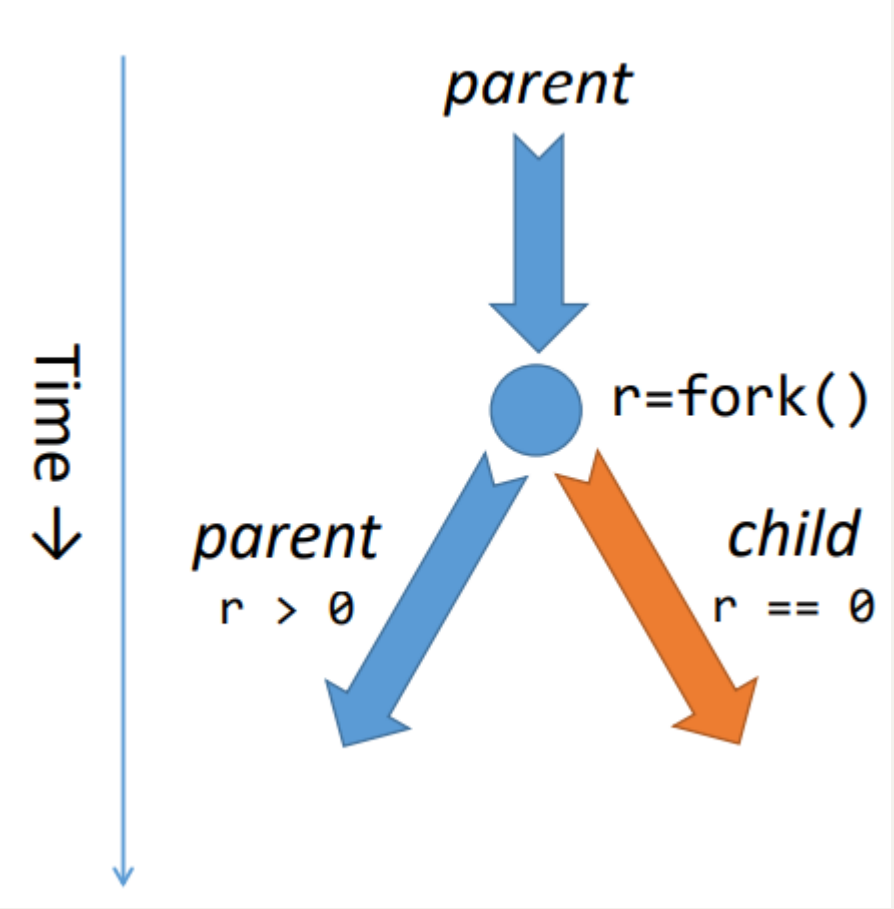
        - Called from both `usertrap()` and `kerneltrap()`
        - Device and interrupt handling (see I/O chapter)
        - Timer interrupt on processor 0 ticks the system clock
        - May cause a context switch
    - Summary
        - A combination of mode switch, page tables, timer interrupts
        - ...suffices to build all OS structures, abstractions, and models above (can completely virtualize the processor cores with **just these**)

Chapter 4: Processes

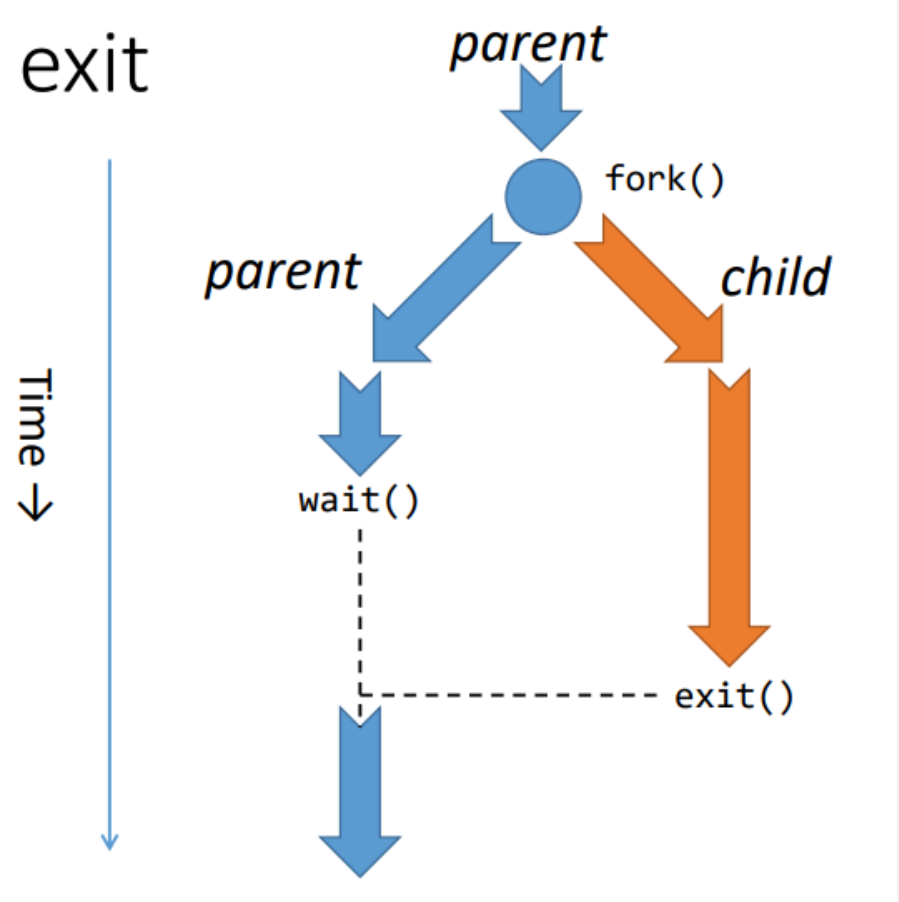
- What is a process?
    - When you run a program, the OS creates a process to execute the program in. A process is an illusion created by the OS.
- What does a process do?
    - Virtualizes the computer → Creates an **execution environment**
- The execution environment - A **virtual machine**
    - One or more **processors** (threads)
    - **RAM** (virtual memory)
    - **Interrupts** (Unix signals)
    - Basic **instruction set** (user space)
    - Additional "**instructions**" (Kernel ABI / system call interface)
- Process abstraction
    - process: an instance of a program, running with limited rights
        - Thread: a sequence of instructions within a process (potentially many threads per process)
        - Program being executed by the thread(s)
        - Name spaces (including virtual memory)
        - Access rights
        - Kernel state: the **process control block** (PCB)
    - A process is therefore both
        - security principal

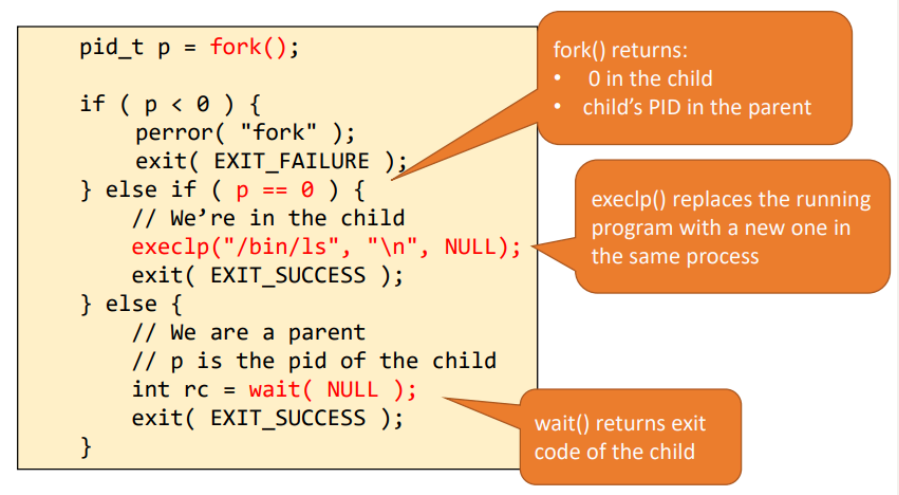
        - resource principal
    - It has a name: the process id (PID)
- Creating a process
    - What's the interface to process creation?
        - Spawn
            - Construct a running process from scratch
        - Fork / exec
            - Create a copy of the calling process
            - Replace the current program with another in the same process
        - Smth else
    - Spawn: `CreateProcess()` in Windows
        - System call to create a new process to run a program
            - create and initialize the process control block in the kernel
            - create and initialize a new address space
            - load the program into the address space
            - copy arguments into memory in the address space
            - initialize the hardware context to start execution at "start"
            - inform the scheduler that the new process is ready to run
        - 

        - It is complex

        - Has to specify **everything** about the new environment
        - many arguments to the call
        - many of these are complex lists and structures themselves
        - most of the time, these arguments are unused or always the same
    - If omit a key element (most arguments are not used most of the time)
        - new process has **insufficient** rights or resources
        - fails to function with security fault
        - safe for the OS
- Creating a process in Unix
    - `pid_t p = fork()`
- Forking a new process
    - 

    - Child is almost exact copy of parent (PID is obviously different)
    - Return value from fork indicates **which process** we are in

        - 0, the child; > 0, the parent, value is child's PID; < 0, error
- Full UNIX process management API
    - `fork()` - system call to create a **copy of the current process** and start it running
    - `exec()` - system call to **change the program being run** by the current process
    - `wait()` - system call to **wait for a process to finish**
    - `signal()` - system call to send a notification to another process
    - 

    - 

- Implementing a Shell

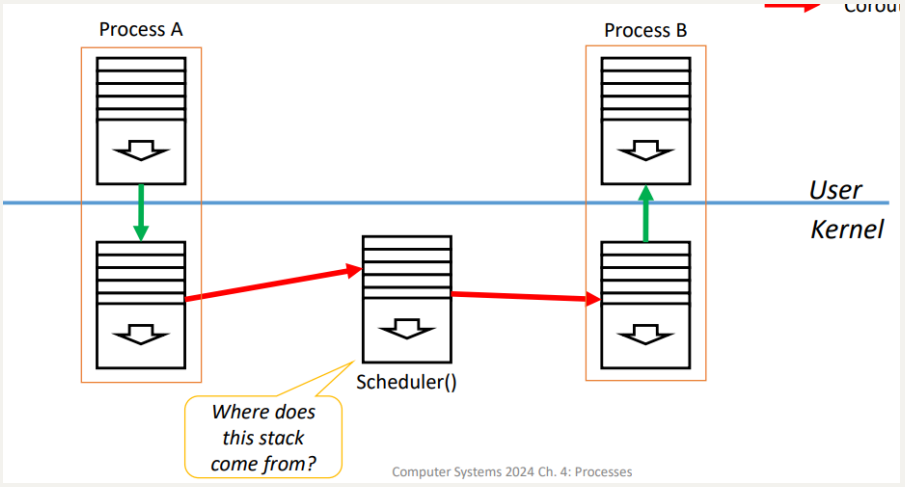
-
```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```


- Comparing fork with spawn
    - Spawn uses a single call
        - all rights a program needs must be granted in one go
        - omitting them will cause an error
    - with fork / exec
        - child revokes rights and access explicitly before `exec()`
        - can use full kernel API to customize the execution environment
        - omitting resources causes security holes
- What does this code print?
    -
```
int child_pid = fork();
if (child_pid == 0) {     // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                  // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```
    - See both lines with number being child process's ID, but we cannot determine the order of output
- Can `fork()` return an error? Why?
    - e.g. cannot create process, no memory
- Can `exec()` return an error? Why?
    - e.g. file name not valid
    - `exec()` can never return a successful result, only returns error; if the result is successful, the process is replaced

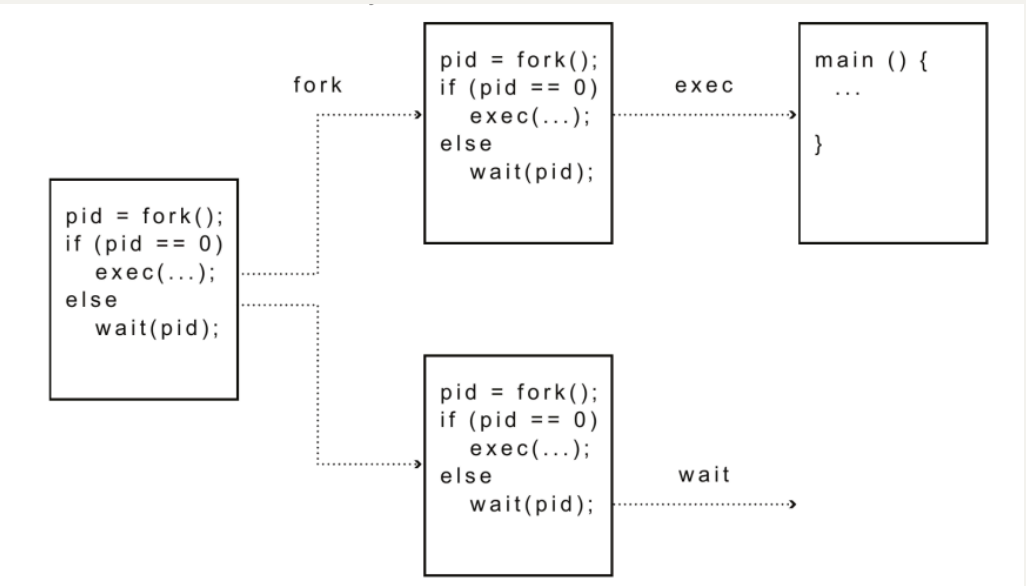
        - Can `wait()` ever return immediately? Why?
            - e.g. child process has already terminated (`wait()` needs to know the child process existed)
- The Process Control Block
    - Main kernel data structure representing a process
        - a struct, basically
    - Has to hold or refer to:
        - page table, trap frame, kernel stack, open files, program name, scheduling state, PID
    - Almost every OS has smth like
        - In Linux, `struct task_struct` (800+ lines long)
        - In xv6, rather short
    - Implementing `fork()` on xv6
        - We saw system calls last time
        - `sysproc.c:sys_fork()` calls `proc.c:fork()` which
            - allocates a new PCB (which will return via `proc.c:forkret()`)
            - copies the page table
            - copies the trap frame
            - duplicates other per-process data structures
            - patches the trap frame so child returns 0
            - sets the child's parent to be the caller
            - makes the new child runnable
        - `exec.c:exec()` is rather more complex, interestingly
- Process context switching
    - When does the kernel switch processes
        - When a process has run for too long (timer interrupt)
        - When a process **blocks** (has issued a system call which cannot complete immediately)
    - Too long

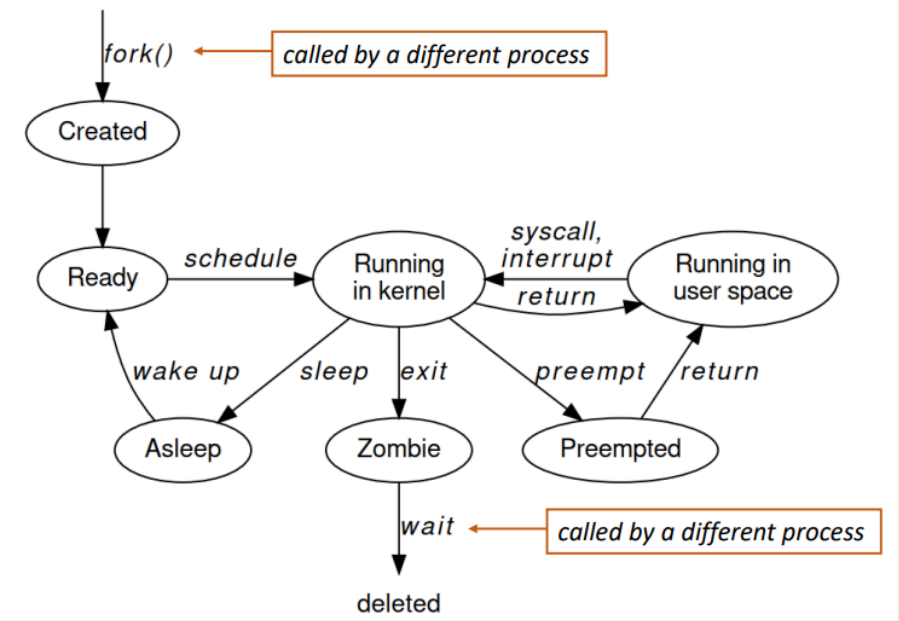
- • Hardware timer interrupt: kernelvec.S:timervec()
    - Runs in RISC-V “machine mode”, below kernel (“supervisor”) mode
    - Resets the timer hardware for the next tick
    - Raises a kernel mode interrupt, which…
  - Calls proc.c:yield() from trap.c:kerneltrap(), which calls proc.c:shed()
  - Key line here is:

    swtch(&p->context, &mycpu()->context);
  - This is a ***coroutine switch*** - see swtch.S:swtch(). But to where?
-

- Blocking
  - Many operations might call `proc.c:sleep()`
    - they already hold a lock
    - lock is released
    - process goes to sleep until smth happens on a "channel" (calls `proc.c:sched()`)
  - What about stacks?
    - 

    - The scheduler stack comes from the stack that boots the machine
- Process hierarchy

- 

- Zombies and Orphans in Unix
    - What if a child process dies, but the parent doesn't call `wait()`
        - Child becomes a Zombie - dead, but still around
        - Needs to continue to exist: preserve exit status
        - Will disappear when `wait()` is called
    - What if a parent dies, but the child doesn't
        - Child is "reparented" to the first process (PID #1, `init`)
- Process Life Cycle
    - 

- The role of the `init` process
    - Process ID 1, `init`, serves an important purpose

```
for(;;) {
    wait(NULL);
}
```

- A null argument to `wait()` → wait for any child to `exit()`

- What about threads?
  - Each xv6 process is single-threaded
    - More full featured UNIX variants separate
      - Process
      - Thread
      - That is: a process may have multiple threads
    - Each tread is part of a process
      - Thread: user stack, kernel stack, registers
      - Process: everything else
    - Not the **only way** to implement threads

Oct. 2nd - Exercise Session 2

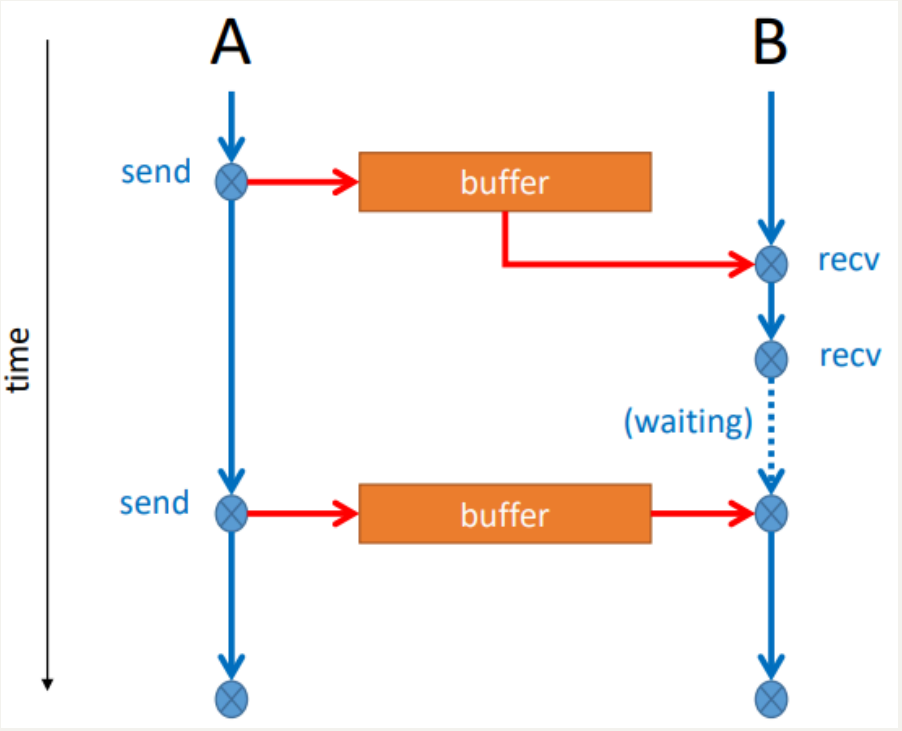
- Process
  - Instance of a program
  - Ingredients of a process
    - Virtual processor (address space, registers, program counter, instruction pointer)
    - Program text (code)
    - Program data (heap, stack)
    - OS stuff (open files, sockets, CPU shares, security rights)
  - Executed in an "execution environment" made up of all the ingredients
  - Run program → OS creates process → virtualizes computer as "execution environment"
  - Identified by PID
  - Complete software of a running computer = running processes + kernel
- Process creation
  - Spawn
    - constructs from scratch

        - complex
            - has to specify everything about the new environments including the command line
            - all rights a program needs must be granted in one go
    - Fork/exec
        - child is almost exact copy of calling parent
        - return value from fork indicated which process we are in
        - can use full kernel API (customize before exec)
- Quiz
    - Difference between a process and a program
        - Process is a running instance of a program
    - How are processes identified
        - By the process ID
    - In what state is a process after it exited
        - Zombie state
    - How does having only one kernel thread simplify the kernel?
        - No locks/synchronization required
    - What is the role of the `init` process?
        - Orphan process reparented to this process

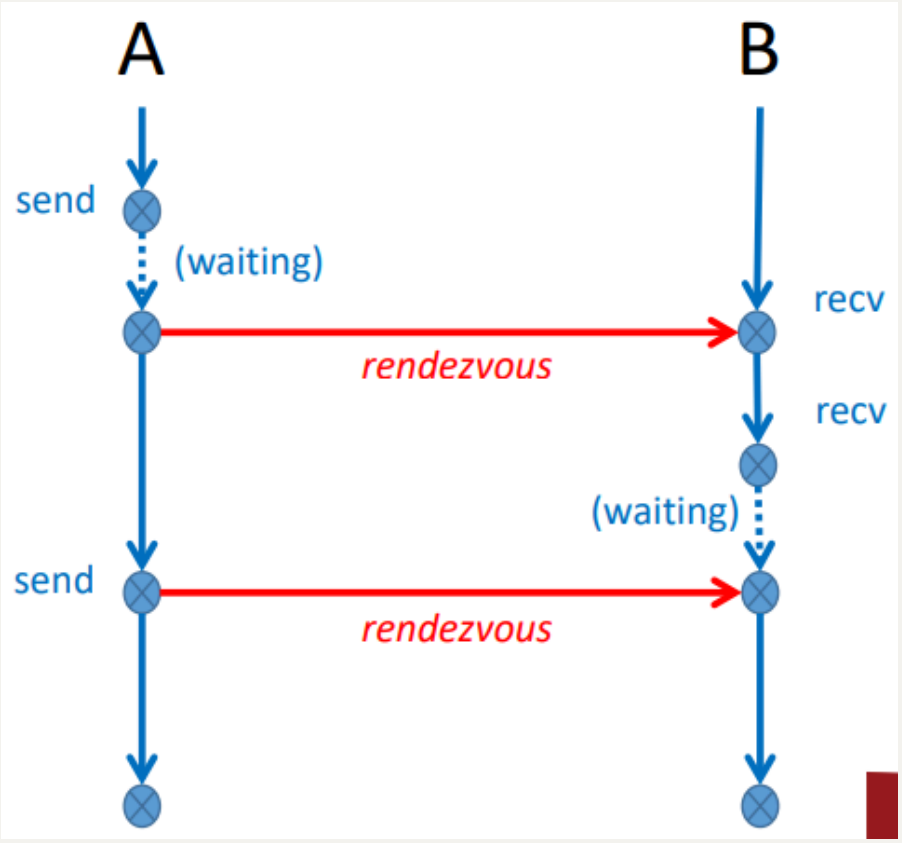
Oct. 4th - Lecture 5: Inter-process Communication

- How can processes communicate?
    - Shared memory
        - hardware mechanisms
            - TAS, CAS, LL/SC
            - Transactional memory
        - spinlocks
            - TAS locks, TATAS locks
            - Competitive spinning
            - Scalable locks
        - programming abstractions

                - Semaphores, mutexes, monitors
        - Message passing
            - Message passing models
                - Async vs. sync
                - Blocking vs. non-blocking
            - Pipes
                - Unnamed and named
            - Upcalls
                - Signals
            - Remote procedure call
                - Client-server
- Duality of messages and shared memory
    - "Any shared-memory system (one based on monitors and condition variables) is equivalent to a non-shared-memory-system (based on messages)"
    - Choice based on performance
- Messaging systems
    - Options:
        - end-points may or may not know each others' names
        - Messages might need to be sent to more than one destination
        - Multiple arriving messages might need to be demultiplexed
        - Can't wait forever for one particular message
    - Seen in networking, parallels between message-passing OS
    - Channels
        - Asynchronous

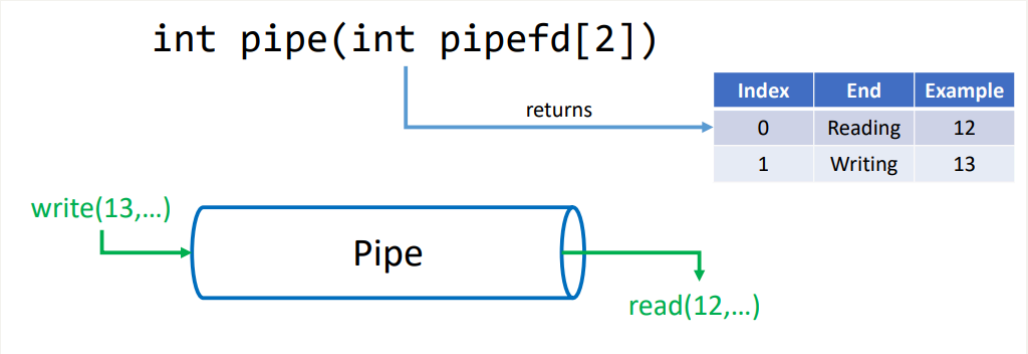


- Synchronous

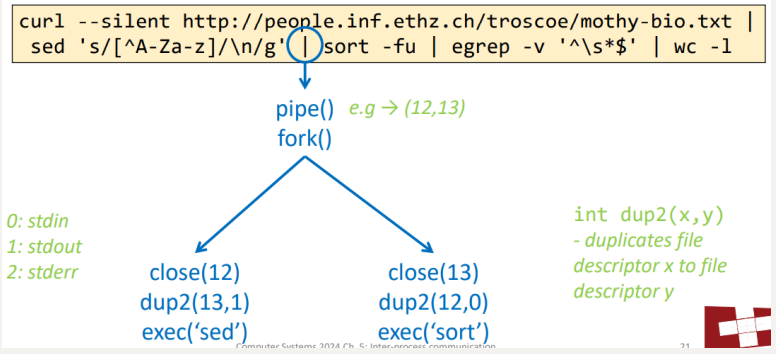


- Blocking / Non-blocking vs. Synchronous / Async
  - If an operation cannot proceed
    - Blocking → block thread until it can
    - Non-blocking → immediately return `error: try again`
  - Property of the API (Blocking and synchrony are independent)

        - (A)synchrony and buffering: property of the **channel**
- Unix Pipes
    - Basic (first) Unix IPC mechanism
    - Unidirectional, buffered inter-process communication channel
    - Creation: `int pipe(int pipefd[2])`
    - Set up pipe between two processes: create the pipe first, then fork

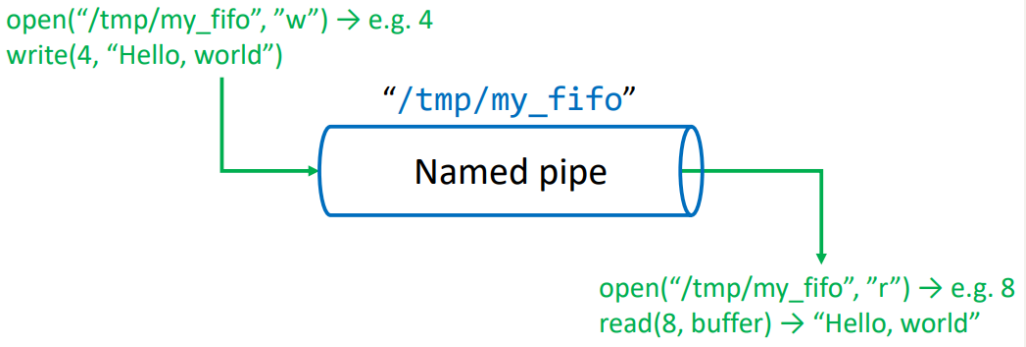


    -
        - Write smth on **file descriptor** 13, and Read smth on **file descriptor** 12
        - Pipe does not provide a way to allow another process get a hold of it
    - Pipe idiom
        - Create pipe, Fork, in child - close the `write` end, read from pipe and write to standard output until EOF, in parent - close the `read` end and write `argv[1]` to pipe
    - Unix shell pipes
        -
```
curl --silent http://people.inf.ethz.ch/troscoe/mothy-bio.txt |
 sed 's/[^A-Za-z]/\n/g' | sort -fu | egrep -v '^\s*$' | wc -l
```
        - Using specialized program in UNIX to complete a computation/calculation
        - What does it do?
            - every time encountering |, it calls `pipe(), fork()`
            -



- Implementing `pipe()` in xv6

    - it's a system call
    - `syscall.c: syscall()` dispatches `sys_file.c: sys_pipe()`
    - `sys_file.c: sys_pipe()` handles file descriptor allocation, calls
    - `pipe.c: pipealloc()` creates the pipe kernel data structure
    - 2 file descriptors, 1 buffer of bytes, counts of sent and received bytes
- Naming pipes
    - Cannot name the pipe itself, and the names of the endpoints of the pipe is local to the process
    - The pipe is copied on `fork()`
    - Make "named pipes" → Special file of type "pipe"
- Named pipes (FIFOs)



    - `"/tmp/my_fifo"` is a common name so that different processes can access the same pipe, common name means this name is shared globally
- Upcalls
    - user-level event delivery
    - Notify user process of some event that **needs to be handled right away**
        - time expiration (real-time user interface, time-slice for user-level thread manager)
        - interrupt delivery for VM player
        - Async I/O completion (async/await)
    - In Unix: **signals**
    - Function calls and System calls are **downcalls**
    - this is kernel calling the user (kernel jumps at upcall handler address)
        - `exec()` is an upcall, it causes process to start running from a specific point

        - an interrupt is an upcall to the kernel (from a hardware to a kernel)
- Unix signals: a simple upcall facility
    - **Asynchronous** notification from the kernel
    - Receiver doesn't wait: signal just happens
    - Each signal type has a **signal number**, interrupt process, and
        - kill it,
        - stop (freeze) it,
        - Do smth else

Name	Description / meaning	Default action
SIGHUP	Hangup / death of controlling process	Terminate process
SIGINT	Interrupt character typed (CTRL-C)	Terminate process
SIGQUIT	Quit character typed (CTRL-\)	Core dump
SIGKILL	`kill -9 <process id>`	**Terminate process**
SIGSEGV	Segfault (invalid memory reference)	Core dump
SIGPIPE	Write on pipe with no reader	Terminate process
SIGALRM	`alarm()` goes off	Terminate process
SIGCHLD	Child process stopped or terminated	Ignored
SIGSTOP	Stop process	Stop
SIGCONT	Continue process	Continue
SIGUSR1,2	User-defined signals	Terminate process

    - `SIGHUP`: hang-up the phone
    - The signals can come from:
        - memory management subsystem
        - IPC system
        - kernel trap handlers
        - TTY Subsystem, etc
    - Sending a signal to a process
        - Unix shell: `kill -HUP 4234`
        - C:

```c
#include <signal.h>
int kill(pid_t pid, int signo);
```

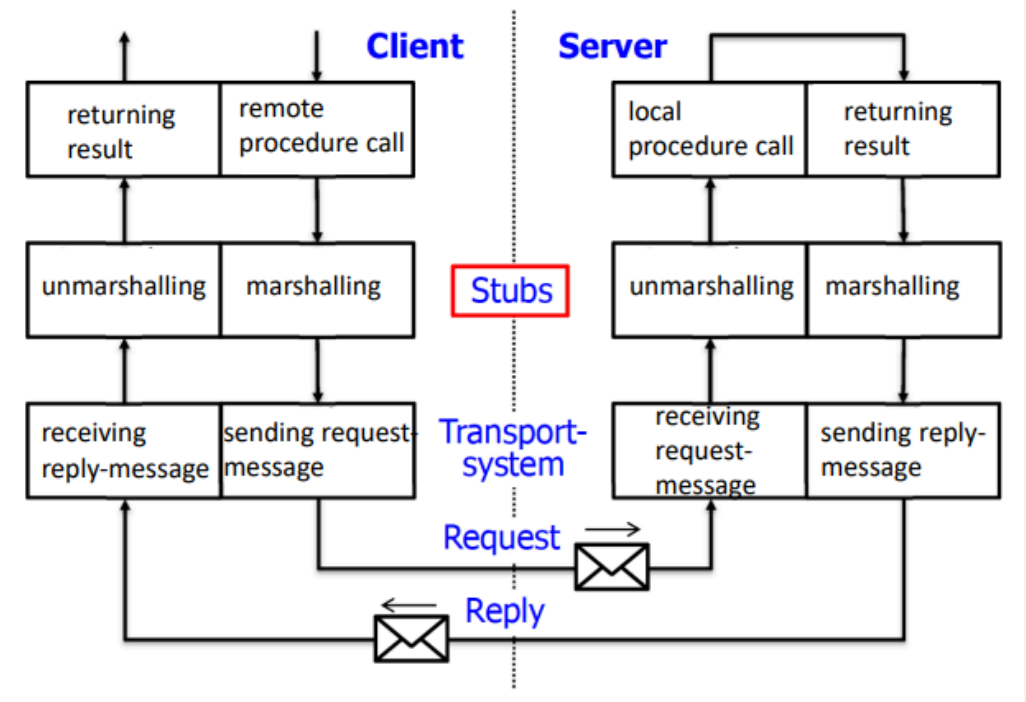
    - Unix signal handlers
        - Change what happens when a signal is delivered

            - default action, ignore signal, call a user-defined function in the process
            - allows signals to be used like "user-space-traps"
- Programming with Signals
    - `void (* signal(int sig, void (*handler) (int))) (int);`
        - A handler looks like `void my_handler(int)`
        - Signal takes 2 arguments
            - an integer
            - a pointer to a handler function
        - returns a pointer to a handler function
    - Unix signal handlers
        - signal handler can be called at any time
        - executes on the current user stack
            - if process is in kernel, may need to retry current system call
            - can also be set to run on a different stack
    - Implications: there is very little that can be safely done in a signal handler
    - Multiple signals
        - multiple signals of the same type, discard all but one
        - if signals of different types are to be delivered, deliver in any order
    - Signals vs. Interrupts
        - signal handlers = interrupt vector
        - signal stacks = interrupt stacks
        - Automatic save/restore registers = transparent resume
        - Signal masking: signals disabled while in signal handler
    - Signals as upcalls
        - Particularly specialized (and complex) form of Upcall
            - Kernel RPC to user process
        - Other OSes use upcalls much more heavily
            - Including Barrelfish

                - “Scheduler Activations”: dispatch every process using an upcall instead of return
- Sockets and Client/Server
    - Sockets: extending IPC to the network
        - as with named pipes, need a name to contact the other side
            - can't name the connection until it exists
            - connection can't be established without naming the other side
            - need name of one party (e.g. server), so the other party (e.g. client) can create the connection
        - (on Monday, TCP socket programming)

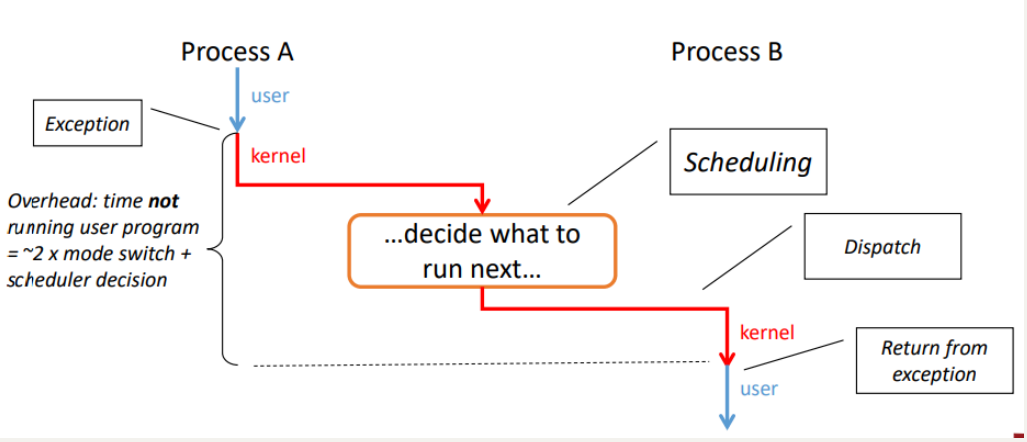
Oct. 7th - Lecture 6: Inter-process Communication (cont.d) & CPU Scheduling

- Sockets and Client/Server
    - TCP socket server (socket is essentially a small integer - file descriptor)
        - Starts up (Internet endpoints are named with IP address + port)
        - Create service on a port
        - Loop:
            - Wait for incoming connection (every TCP connection is uniquely by source IP address, port number and destination IP address, port number)
            - Get next connection (one socket is used for all connections, and another socket is created for each connection)
            - Process request
            - Close connection
    - TCP socket client (only needs one socket)
        - Create a connection endpoint
        - Connect to the server
        - Send the request
        - (Read the response)

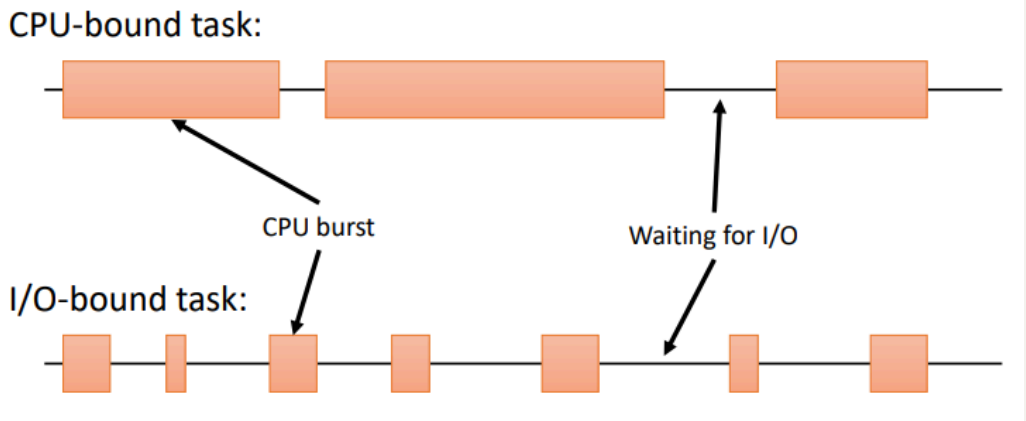
        - Close the connection
- Remote Procedure Call
    - Key idea
        - Client request/response looks like a **procedure call** in the code
        - Server code looks like the implementation of the **same procedure**
    - Stub compiler generates all the rest of the code
        - Marshalling/unmarshalling (serialization/deserialization)
        - Request/response handling
        - Method dispatch
    - Examples
        - XDR/SunRPC, CORBA, Java RMI, Google Protobufs
    - 

- Local Remote Procedure Call
    - Can use RPC locally
        - Define procedural interface in an IDL
        - Compile / link stubs
        - Transparent procedure calls over messages
    - Naive implementation is slow
        - Lots of things (like copying) don't matter with a network, but do matter between local processes

Chapter 6: CPU Scheduling

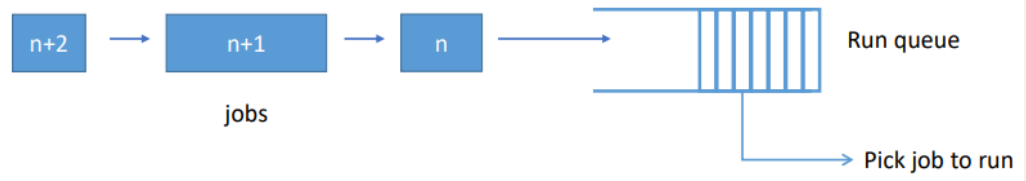
- Scheduling is...
    - deciding how to allocate a single resource among multiple clients in **what order** and for **how long**
    - Usually refers to CPU scheduling
        - will look at selected systems/research
        - OS also schedules other resources (e.g. disk and network IO)
    - CPU scheduling involves deciding
        - which task next on a given CPU
        - for how long should a given task run
        - on which processor should a task run
    - 

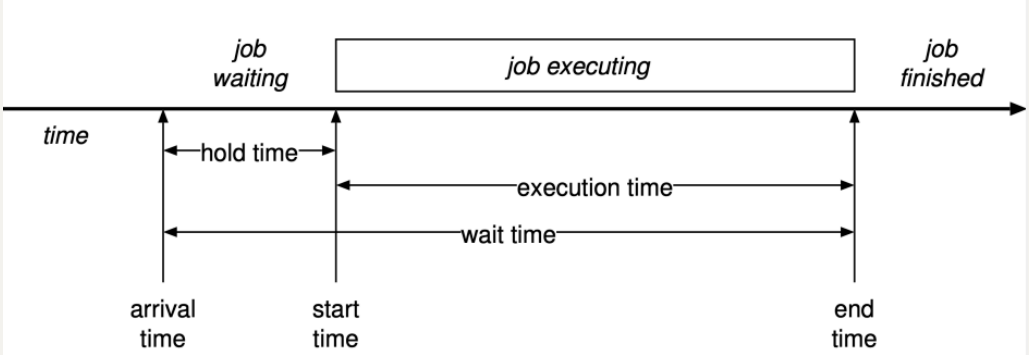
- Optimization
    - Metric to be optimized
        - Fairness (depends on interpretation)
        - Policy (of some kind)
        - Balance (keep everything being used)
        - Increasingly: power (energy usage)
    - usually in contradiction
- Objectives
    - depends on workload: batch jobs, interactive, real-time and multimedia
- Challenge: complexity of scheduling algorithms
    - Scheduler needs CPU to decide what to schedule
        - any time spent in scheduler is "wasted" time

        - want to minimize overhead of decisions to maximize utilization of CPU
        - Low overhead is no good if scheduler picks the "wrong" things to run
    - Trade-off between **scheduler complexity/overhead** and optimality of resulting schedule
    - Increased scheduling frequency (increasing chance of running something different)
    - Leads to higher **context switching** rates (lower throughput)
        - Flush pipeline, reload register state
        - Maybe flush TLB, caches
        - Reduces locality
- Scheduling assumptions and definitions
    - 

    - Simplifying assumptions
        - **One** processor
        - Processor runs at **fixed speed**
            - real time = CPU time
            - Not true in reality for power reasons
            - DVFS: dynamic voltage and frequency scaling
            - In many cases, however, efficiency → run flat-out until idle
        - only consider **work-conserving** scheduling
            - no processor is idle if there is a runnable task
        - the system can always preempt a task
            - rules out some very small embedded systems
            - and hard real-time analysis

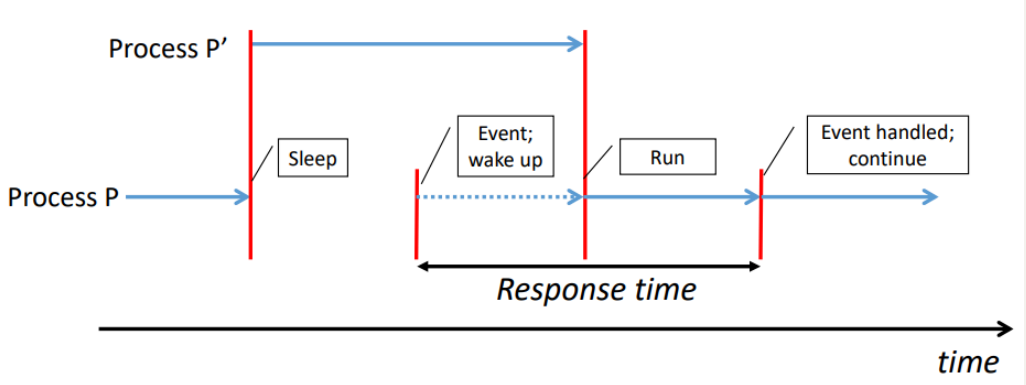
        - and early PC/Mac OSes
    - When to schedule?
        - A running process blocks
        - A blocked process unblocks
        - A running or waiting process **terminates**
        - An **interrupt** occurs
        - Point 2 or 4 can involve **preemption**
    - Preemption
        - Non-preemptive scheduling
            - Require each process to explicitly give up the scheduler
        - Preemptive scheduling
            - processes dispatched and de-scheduled without warning
            - the most common case in most OSes
            - soft-real-time systems are usually preemptive
            - hard-real-time systems are often not
    - Overhead
        - Dispatch latency
            - time taken to dispatch a runnable process
        - Scheduling cost
            - $2\times$ (half context switch) + (scheduling time)
        - Time slice allocated to a process should be significantly more than scheduling overhead
        - Tradeoff: response time vs. scheduling overhead
- Batch-oriented scheduling
    - Batch workloads
        - "run this job to completion, and tell me when you are done"
            - typical mainframe use-case
            - much used in old textbooks, but made a comeback in the last decade with large clusters
        - Goals:
            - throughput (jobs per hour)

        - wait time (time to execution)
        - turnaround time (submission to termination)
        - CPU utilization (don't waste resources)
- 

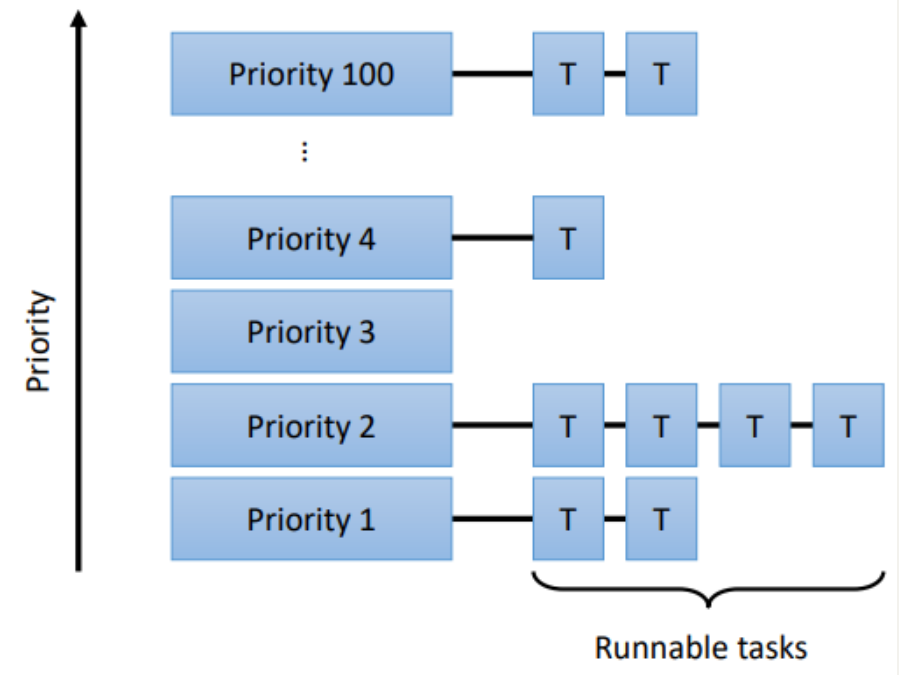
- Batch scheduling
    - Mainframes are 1970
    - Most systems have batch-like background tasks (phones are beginning to act like this as well)
    - CPU bursts can be modelled batch jobs
    - Web services are **request-based**
- 

    - First-come first-served algorithm is the simplest algorithm, but the average hold time can be longer than a re-ordered sequence (re-ordering is unpredictable)
    - Shortest-job first: optimal for minimizing hold time
- Convoy phenomenon
    - Short processes **back up** behind long-running processes
    - well-known and widely-seen problem
        - famously identified in databases with disk I/O
        - simple form of self-synchronization
    - FIFO used for: e.g. `memcached` (cache server for a large variety of Internet companies, like Meta)
- Execution time estimation
    - what is the execution time? mostly impossible to determine
        - for mainframes, could punt to user

        - and charge them more if they are wrong
    - For non-batch workloads, use CPU **burst times**
        - keep exponential average of prior bursts
        - C.f. TCP RTT estimator
        - $\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$
    - or just use application information
        - web pages: size of web page
- SJF & preemption
    - problem: jobs arrive all the time
    - "shortest remaining time next" (now, short jobs may preempt longer jobs already running)
    - Still not an ideal match for dynamic, unpredictable workloads (in particular, interactive ones)
- Scheduling interactive loads
    - Interactive workloads
        - processes, not jobs (can potentially run forever)
        - IO or syncronization
            - blocking, sleep, wake up
        - "Wait for external events, and react before the user gets annoyed"
            - word processing, browsing, fragging
            - common for PCs, phones
        - Goals
            - response time: how quickly does something happen
            - proportionality: some things should be quicker
    - Response time



- Round-robin
    - simplest interactive algorithm: run all runnable tasks for fixed quantum in turn
    - advantages: easy to implement, easy to understand/analyze, higher turnaround time than SJF, but better response
    - disadvantages: rarely what the user want, treats all tasks the same
- Priority
    - assign every task a priority
    - dispatch highest priority runnable task
    - priorities can be dynamically changed
    - schedule processes with same priority using: round-robin, FCFS, etc.
    - priority queues



    - Multi-level queues
        - can schedule different priority levels differently
            - interactive, high-priority: round robin
            - batch, background, low priority: FCFS
        - ideally generalizes to **hierarchical scheduling**
- Starvation
    - strict priority schemes do not guarantee progress for all tasks

- Solution: aging
    - tasks which have waited a long time are gradually increased in priority
    - eventually, any starving task ends up with the highest priority
    - reset priority when quantum is used up
- Multilevel feedback queues
    - penalize CPU-bound tasks to benefit I/O bound tasks
        - reduce priority for processes which consume their entire quantum
        - eventually, re-promote process
        - I/O bound tasks tend to block before using their quantum → remain at high priority
    - generally: any scheduling algorithm can reduce to this (the problem lies within implementation)
    - e.g. Linux o(1) scheduler
        - 140-level Multilevel Feedback Queue
            - 0-99 (high priority)
            - 100-139: user tasks, dynamic
        - Complexity of scheduling is independent of number of tasks
            - two arrays of queues: "runnable" and "waiting"
            - when no more tasks in "runnable", swap arrays

Oct. 9th - Exercise Session 3

- How do processes communicate?
    - Shared memory
        - Semaphores/Locks (synchronization instructions)
        - transactional memory
    - Message passing
        - Async/sync

        - Blocking/non-blocking
        - Pipes
        - Upcalls/Signals
        - RPC
- CPU scheduling
    - Scheduling
        - work is assigned to resources that complete the work (what runs)
    - Dispatch
        - gives control of the CPU to the process selected by the scheduler (actually running it)
    - Preemptive vs. Non-preemptive
        - preemptive: process interrupted without warning
        - non-preemptive: process explicitly has to give up the resource
            - easier, each task can run to completion, less overhead
            - delays for other tasks - starvation, malicious code could keep CPU forever, usually not used
    - Workloads
        - Batch workloads: run this job to completion and tell me when you are done
        - Interactive workloads: wait for external events and react before the user gets annoyed
        - Realtime workloads: this task must complete in less than 50ms; this task must get 10ms CPU every 50ms
        - Hard realtime workloads: ensure the plane's control surface moves correctly in response to the pilot's actions
    - FIFO (batch scheduling)
    - Shortest Job First (SJF, non-preemptive): minimizes waiting time
        - with preemption; shortest remaining time first
    - Round robin (interactive workloads)
        - maintains a queue of jobs and schedules them in order
        - higher turnaround than shortest job first, but better response time

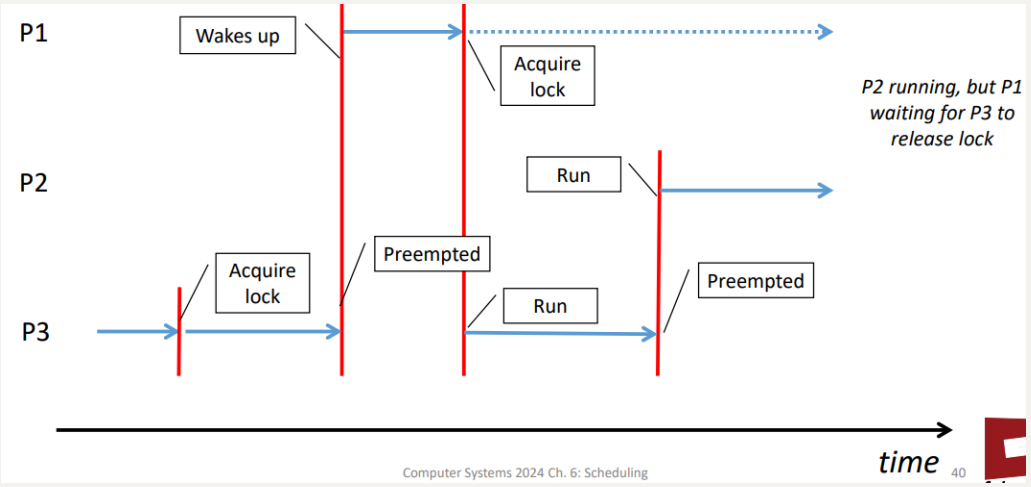
- Priority queues (interactive workloads)
    - can schedule differently on different levels, e.g. round robin and SJF
    - starvation - jobs with low priority never get scheduled; introduce aging to increase priority as time progresses
    - priority inversion (a lower priority process is run when a higher priority process could be run)
        - priority inheritance: when a high-priority job is waiting for a lock from a low-priority job, we can allow low-priority job to "inherit" high-priority job's priority to run to completion
        - priority ceiling
- Multilevel priority queues
    - same idea as priority queues
    - idea: prioritize I/O tasks
        - solution: use priority queue and reduce priority of processes which use their entire quantum
        - I/O bound processes tend to only use part of their quantum, so remain high priority
- Real-time scheduling
    - Hard real-time: correctness depends on the time it needs to execute, deadlines
    - Soft
    - Rate monotonic scheduling
        - schedule periodic tasks by always running task with shortest period first
    - Earliest deadline first

Oct. 11th - Lecture 7: CPU Scheduling & Input/Output

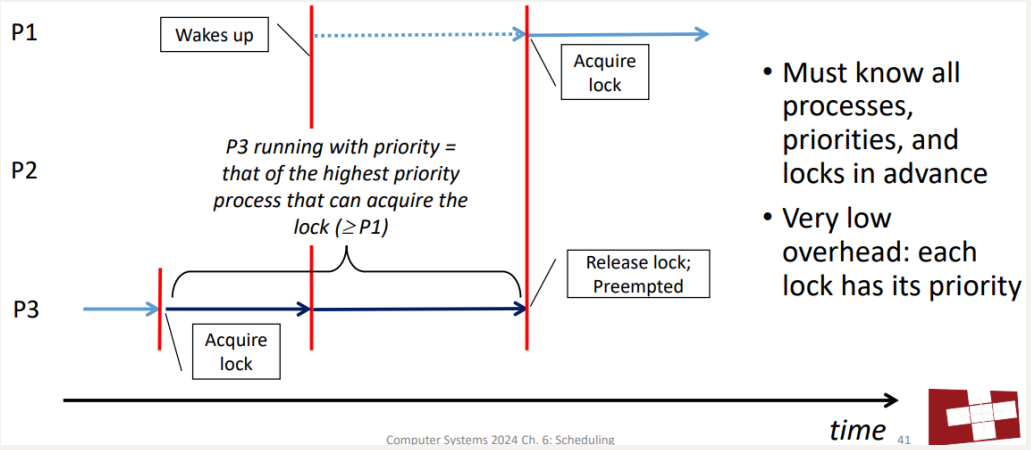
- CPU Scheduling
    - Problems with UNIX scheduling
        - Unix conflates protection domain and resource principal
            - priorities and scheduling decisions are per-process

- may want to allocate resources across processes, or separate resource allocation within a process
    - web server structure
    - if I run more compiler jobs than you, I get more CPU time
- in-kernel processing is accounted to nobody

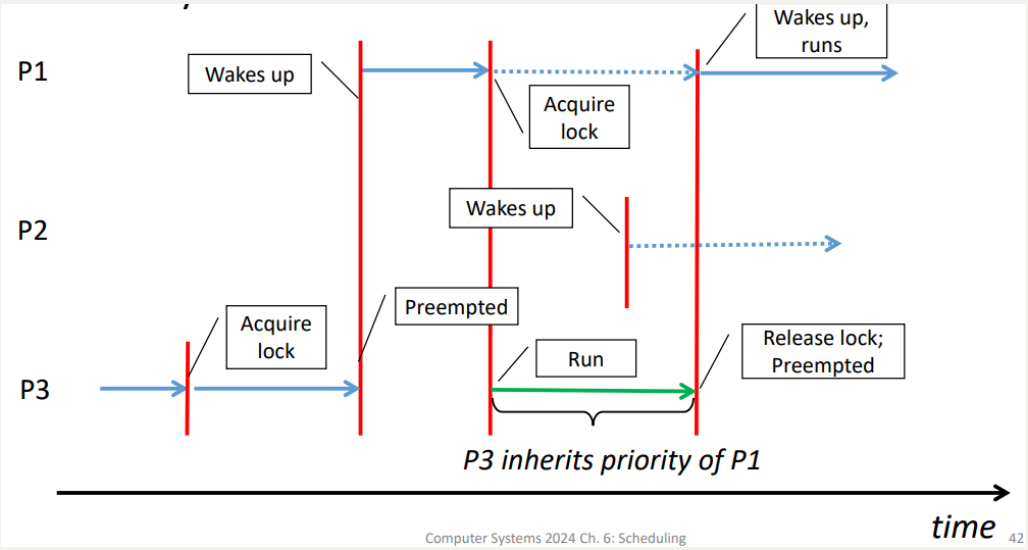
- Priority Inversion: suppose $P_1 > P_2 > P_3$



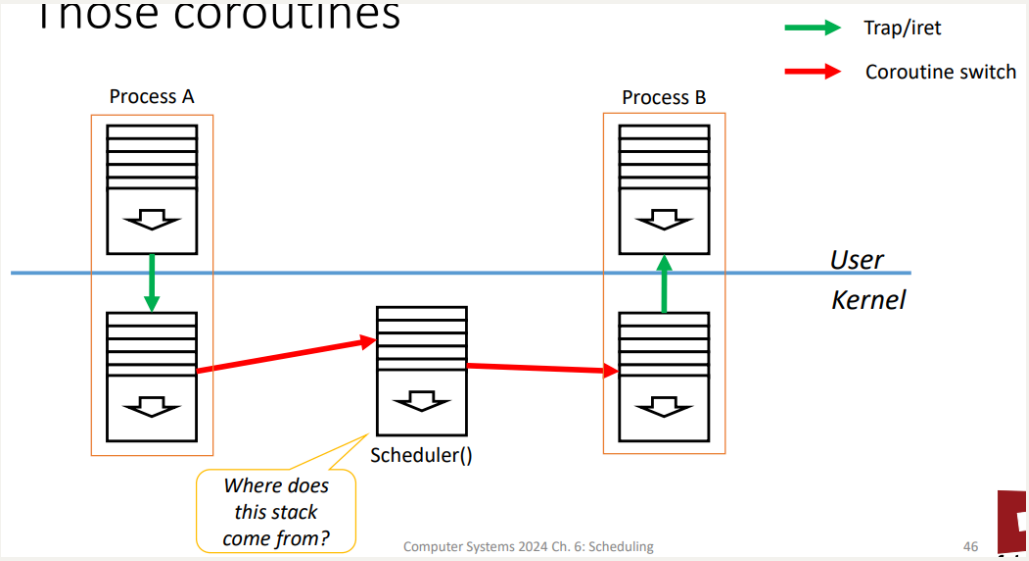
- Priority Ceiling



- Priority Inheritance

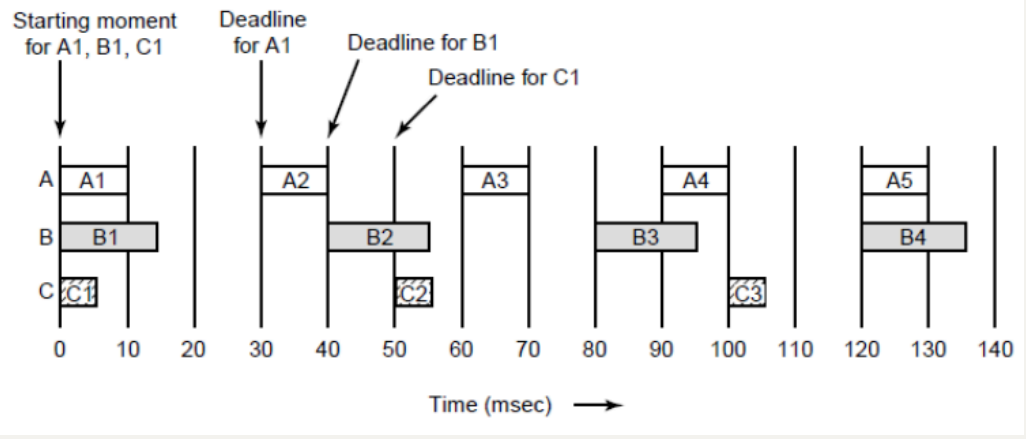


- Resource Containers
    - New OS abstraction for explicit resource management, separate from process structure
    - Operations to create/destroy, manage hierarchy, and associate threads or sockets with containers
    - Independent of scheduling algorithm used
    - All kernel operations and resource usage accounted to a resource container
    - Explicit and fine-grained control over resource usage
    - Protects against some forms of DoS attack
- Implementation of scheduling in xv6
    -
    - Those coroutines



    - The actual scheduling policy
        - pick the first process in state `RUNNABLE`

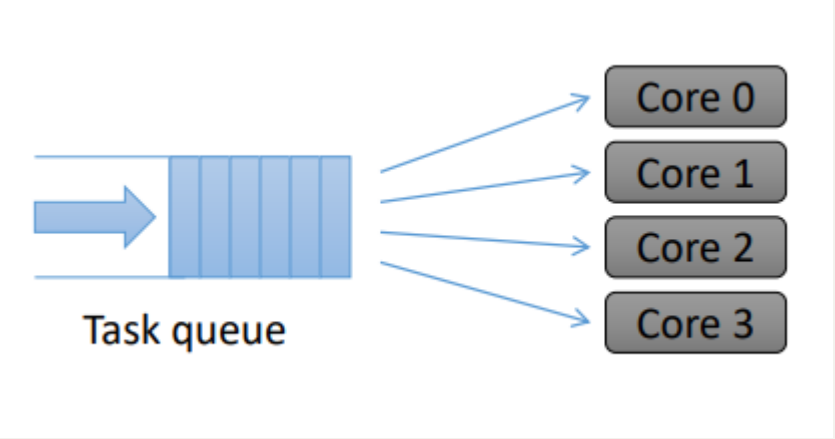
        - From a global queue
        - Shared by all the cores, under a lock
        - xv6 book claims this is "round-robin"
- Real Time
    - Problem: giving real time-based guarantees to tasks
        - tasks can appear at any time
        - tasks can have deadlines
        - execution time is generally known
        - tasks can be periodic or aperiodic
    - Must be possible to reject tasks which are unschedulable, or which would result in no feasible schedule
    - Soft realtime workloads
        - This task must complete in less than 50ms, or this program must get 10ms CPU every 50ms
            - data acquisition, I/O processing
            - multimedia applications (audio and video)
        - Goals: deadlines, guarantees, predictability (real time $\neq$ fast)
    - Hard realtime workloads
        - "Ensure the plane's control surfaces move correctly in response to the pilot's actions"
        - "Fire the spark plugs in the car's engine at the right time"
        - Mission-critical, extremely time-sensitive control applications
    - Challenge of multimedia scheduling

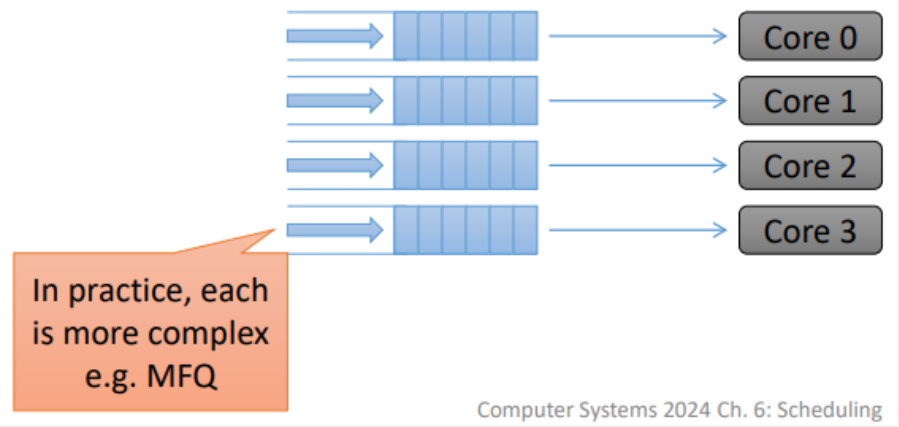


    - Rate-monotonic scheduling
        - good for when you know all the tasks beforehand

        - schedule periodic tasks by always running task with shortest period first (static/offline scheduling algorithm)
        - Suppose: m tasks, C_i is the execution time of ith task, P_i is the period of ith task
        - Then RMS will find a feasible schedule if
            - $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{\frac{1}{m}} - 1)$
    - Earliest Deadline First
        - Schedule task with earliest deadline first
            - dynamic, online
            - tasks don't actually have to be periodic
            - More complex $o(n)$ for scheduling decisions
        - Will find a feasible schedule if

          $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$
        - very handy, assuming zero context switch time
    - Guaranteeing processor rate
        - can use EDF to guarantee a rate of progress for a long-running task
            - break task into periodic jobs, period p and time s
            - a task arrives at start of a period
            - deadline is the end of the period
        - provides a **reservation scheduler** which
            - ensures task gets s seconds of time every p seconds
            - approximates weighted fair queuing
        - algorithm is regularly rediscovered
- Multiprocessor Scheduling
    - Challenge 1: sequential programs on multiprocessors
        - Queuing theory $\rightarrow$ straightforward, although
        - more complex than uniprocessor scheduling, and harder to analyze

- 

- Much harder
    - Overhead of **locking** and **sharing** queue
        - Classic case of scaling bottleneck in OS design
    - Solution: per-processor scheduling queues
    - 

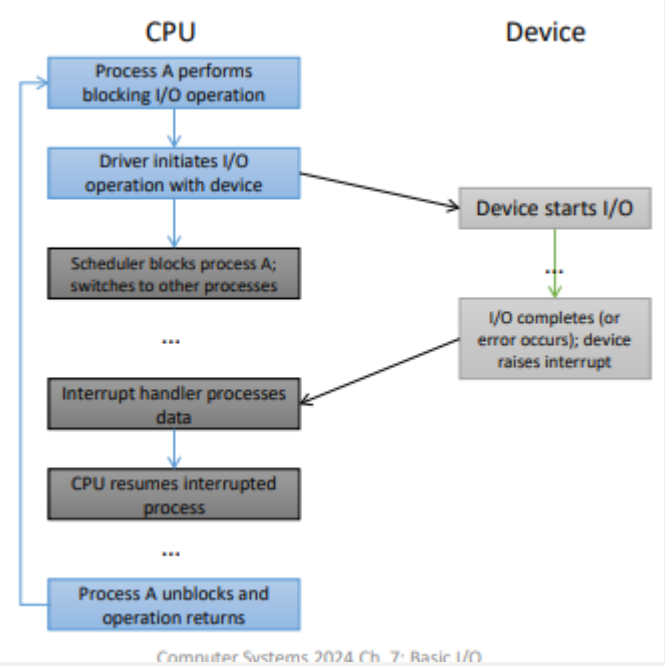
    - Threads allocated arbitrarily to cores
        - tend to move between cores
        - tend to move between caches
        - bad **locality** and hence performance problems
    - Solution: affinity scheduling
        - keep each thread on a core most of the time
        - periodically rebalance across cores
- Challenge 2: parallel applications
    - Global **barriers** in parallel applications → one slow thread has huge effect on performance
    - Multiple threads would benefit from cache sharing
    - Different applications pollute each others' caches
    - Leads to concept of "co-scheduling"
        - try to schedule all threads of an application together

        - Critically dependent on **synchronization** concepts
    - Multicore scheduling
        - multiprocessor scheduling is 2-dimensional
            - when to schedule a task
            - where to schedule on
        - General problem is NP complete
        - Don't want a process holding a lock to sleep
        - Not all cores are equal...
        - Open research problems

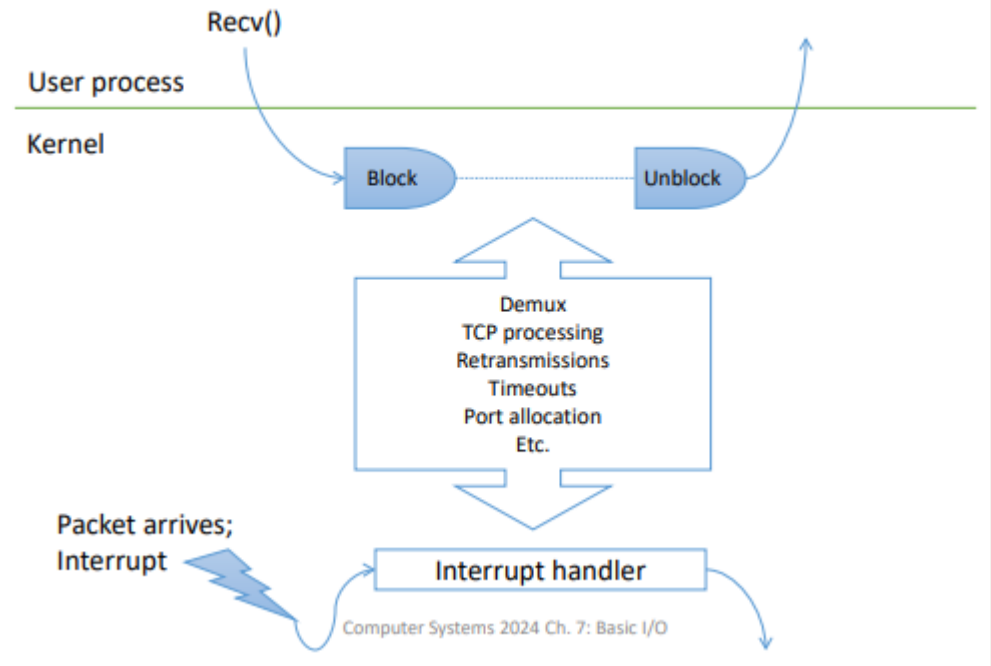
Chapter 7: Basic Input/Output

- Recap: What is a device
    - Piece of hardware visible from software
    - Occupies some location on a bus
    - Set of registers (memory mapped or I/O space)
    - Source of interrupts
    - May initiate Direct Memory Access transfers
- Recap what is a device driver
    - Software object (module, object, process, hunk of code) which abstracts a device
        - Sits between hardware and rest of OS
        - Understands device registers, DMA, interrupts
        - Presents uniform interface to rest of OS
    - Device abstractions ("driver models") vary
        - Unix starts with "block" and "character" devices
- Recap: Registers
- Recap: Interrupts
    - CPU **interrupt-request line** triggered by I/O device
    - **Interrupt handler** receives interrupts
    - **Maskable** to ignore or delay some interrupts
    - Interrupt vector to dispatch interrupt to correct handler

        - based on priority
        - some **non-maskable**
    - Interrupt mechanism also used for exceptions
    - Interrupt-driver I/O Cycle

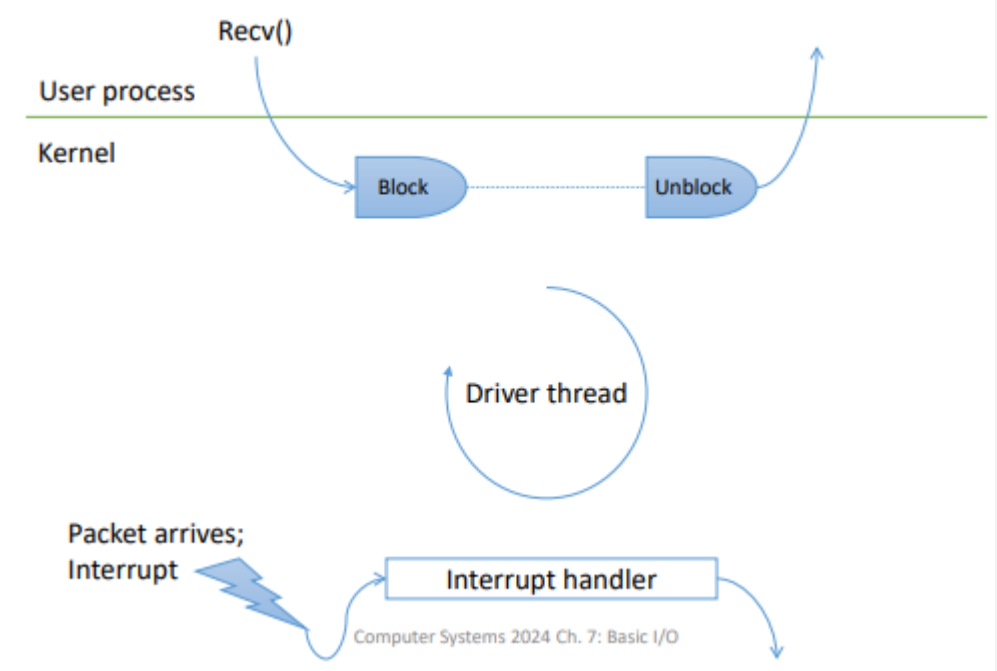


- Interrupts and Device Drivers
    - Device driver structure: the basic problem
        - Hardware is **interrupt driven**
            - system must respond to unpredictable I/O events
        - Applications are **blocking**
            - process is waiting for a specific I/O event to occur
        - Often considerable processing **in between**
            - TCP/IP processing, retries, etc
            - File system processing, blocks, locking, etc.
    - Example: the xv6 console, input/output
    - Example: network receive

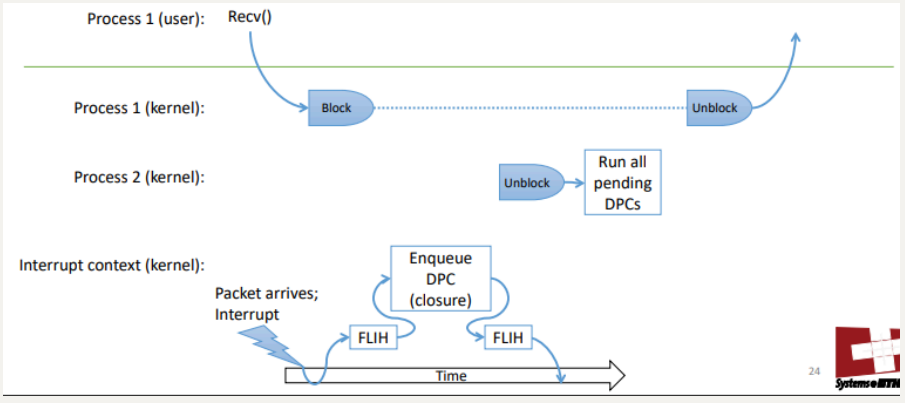


- For interrupt handler:
    - Can't take too long (interrupts disabled?)
    - Can't change much
        - interrupt context, arbitrary system state, can't hold locks
- For the process:
    - process is blocked
    - don't even know it's this process until demux

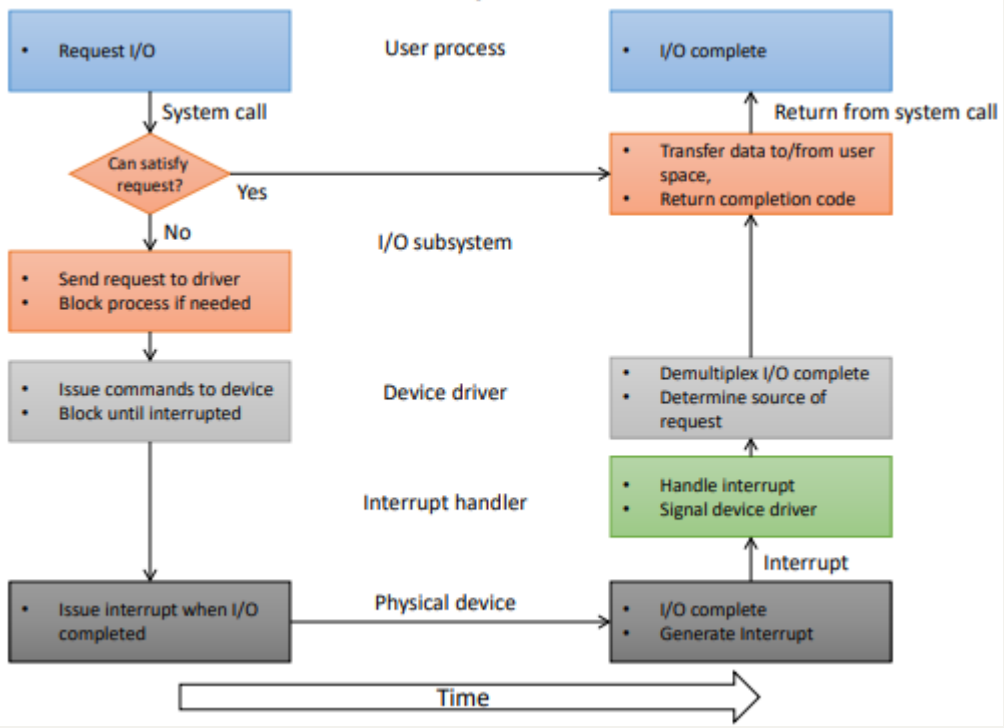
- Solution 1: driver threads



- Interrupt handler:

        - masks interrupt
        - does minimal processing
        - unblocks driver thread
    - Thread:
        - performs all necessary packet processing
        - unblocks user processes
        - unmasks interrupt
    - User process:
        - Per-process handling
        - Copies packet to user space
        - Returns from kernel
- Deferred Procedure Calls
    - Don't create a thread, execute on **next process** to be dispatched (before it leaves the kernel)
    - Solution in most versions of Unix
        - don't need kernel threads
        - saves a context switch
        - can't account processing time to the right process
    - exists a 3rd solution: demux early, run in user space
    - 

    - Terminology
        - Code called from user space - Top Half
        - 1st-level interrupt handler - Bottom Half (Linux: Top Half)
        - 2nd-level interrupt handler - Bottom Half (Linux: Bottom Half)
- Life Cycle of An I/O Request



Oct. 14th - Lecture 8: I/O + Virtual Memory & Demand Paging

- Concurrent Requests
    - Multiple outstanding requests
        - console driver processes each write **sequentially**
            - and each read sequentially, in parallel with writes
        - Not a good match for a disk
            - **High latency** of read/write operations
            - Potential win from **batching**, and/or **reordering**
            - Moving the arm to "seek" data on disk, by reordering tasks, the arm can move less instead of process sequentially
        - Drivers maintain **multiple outstanding requests**
            - Communication model: driver and device exchange messages
    - e.g. xv6 Virtio disk driver
        - disk "hardware" uses 2 shared-memory queues
            - one used for requests to the disk
            - the other used for completion notifications from the disk

        - Need 3 descriptors: location, data, status result
- Device discovery and naming
    - In xv6
        - Devices do not have names
        - Routing requests to devices are mostly hard-coded
            - Including the virtual disk
            - Exception: file access to raw devices (`devsw - file.h, file.c`)
        - Drivers are compiled into the kernel
        - Set of devices fixed at compile time
            - no hotplug
            - need to recompile kernel when you install a new peripheral
    - Naming and discovery
        - What are the devices the OS needs to manage?
            - Discovery (bus enumeration)
            - Hotplug/unplug events
            - Resource allocation (e.g. PCI BAR programming)
        - How to match driver code to devices
            - Driver instance ≠ driver module
            - One driver typically manages many models of device
        - How to name devices inside the kernel
        - How to name devices outside the kernel
    - How devices **were** tracked in the Unix kernel
        - Each driver supports a "class" of device
            - probes for any supported devices
            - hardwired where the hardware is known in advance
            - driver still compiled into the kernel
        - Each "device" is uniquely assigned a triple (these **were** static arrays)
            - Type: **character** or **block** (1-bit)
            - Major number: class, identifies the driver (mostly)

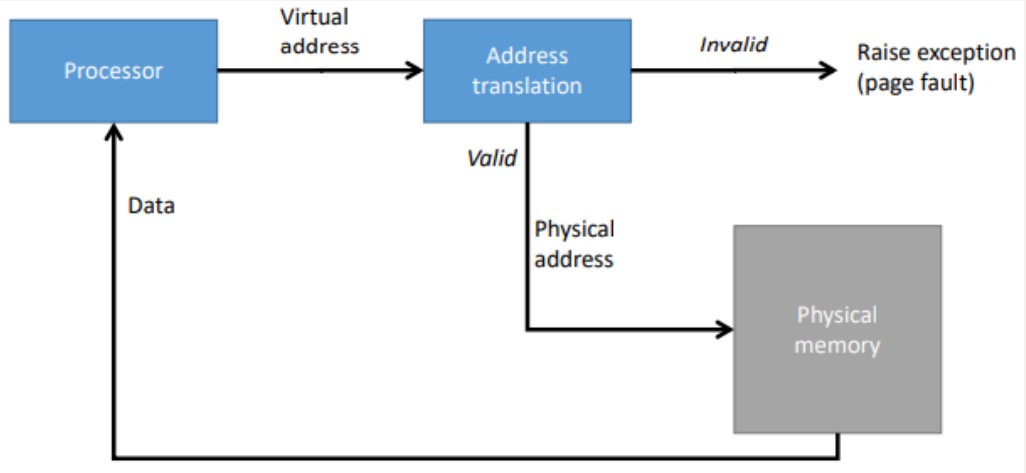
        - Minor number: identifies the **individual device** itself
    - Exception: network devices
- Unix Block Devices
    - Used for "structured I/O"
        - Deal in large "blocks" of data at a time
    - Often seekable, mappable (like files)
        - Often sue Unix' shared buffer cache
    - Mountable
        - file system implemented above block devices
    - examples: Disks, SSD, CD-ROM
- Unix Character Devices
    - Used for "unstructured I/O"
        - byte-stream interface - no block boundaries
        - single character or short strings get/put
        - buffering implemented by libraries
    - examples: keyboards, serial lines, mice
    - distinction with block devices is arbitrary

- Naming devices outside the kernel
    - Device files: special type of file
        - file encode (type, major num, minor num)
        - created manually with `mknod()` system call
    - Devices are traditionally put in `/dev`
    - Pseudo-devices in Unix
        - Devices with no hardware
        - Still have major/minor device numbers
    - Linux device configuration today
        - Fully **dynamic** assignment of major/minor numbers
            - many hardware discovery mechanisms: PCIe, USB, etc
        - Physical hardware configuration published in `/sys`
            - Special fake file system: `sysfs`
            - plug events delivered by a special socket
        - Drivers dynamically loaded as **kernel modules**

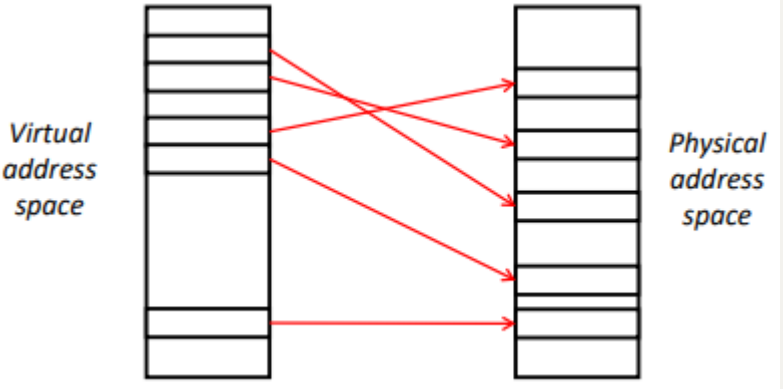
        - Initial list given at boot time
        - User-space daemon can load more if required
    - `/dev` populated dynamically by `udev`
        - User-space daemon which polls `/sys`
- We have not covered
    - Direct memory access
        - descriptor rings
        - memory protection
        - IOMMUs and System MMUs
    - Buffering
        - Moving payload data structure around in memory
    - Enabling high-performance I/O
        - The network is the most important high-speed device today
- We have covered
    - Basic I/O operations
    - Interrupt handling
    - Outstanding requests and concurrency
    - Device naming and device types

Chapter 8, 9: Virtual Memory and Demand Paging

- Virtualizing physical memory
    - Basic address translation



    - Challenges for memory management

    - allocating & freeing physical addresses for applications
    - managing translation of virtual addresses to physical addresses
    - performing access control on memory access
    - 

- Address Translation Uses
    - Process isolation
        - keep a process from touching anyone else's memory, or the kernel's
    - Efficient interprocess communication
        - shared regions of memory between processes
    - Shared code segments
        - e.g. common libraries used by many different programs
    - Program initialization
        - start running a program before it is entirely in memory
    - Dynamic memory allocation
        - allocate and initialize stack/heap pages on demand
    - Cache management
        - Page coloring
    - Program debugging
        - Data breakpoints when address is accessed
    - Zero-copy I/O
        - Directly from I/O device into/out of user memory
    - Memory mapped files
        - Access file data using load/store instructions
    - Demand-paged virtual memory

        - Illusion of near-infinite memory, backed by disk or memory on other machines
    - Checkpointing/restart
        - Transparently save a copy of a process, without stopping the program while the save happens
    - Persistent data structures
        - Implement data structures that can survive system reboots
    - Process migration
        - Transparently move processes between machines
    - Information flow control
        - Track what data is being shared externally
    - Distributed shared memory
        - Illusion of memory that is shared between machines
- Address Translation Goals
    - Memory protection
    - Memory sharing
        - shared libraries, interprocess communication
    - Spare addresses
        - multiple regions of dynamic allocation (heaps/stacks)
    - Efficiency
        - memory placement, runtime lookup, compact translation tables
    - Portability
- Assumed knowledge
    - How a paged MMU works
    - The TLB and TLB fault handling
    - Multi-level page tables
    - Page fault handling
    - Processor caches
- Paging in xv6 (RISC-V 64 bit)
    - 3-level page table

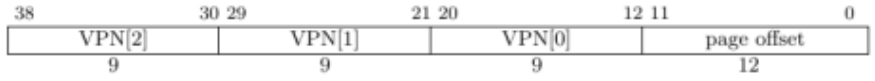


Figure 4.16: Sv39 virtual address.

Figure 4.17: Sv39 physical address.

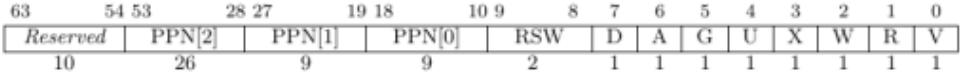
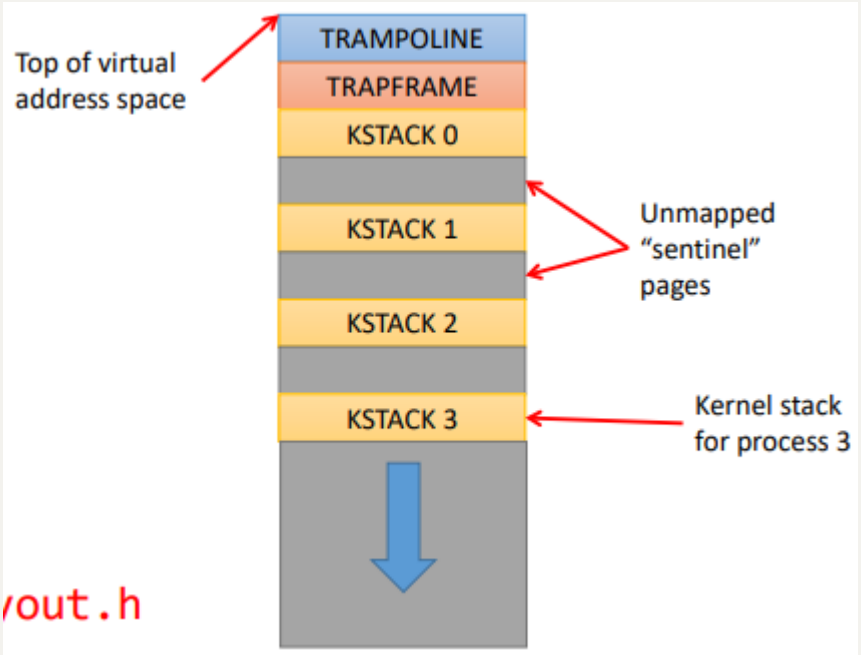
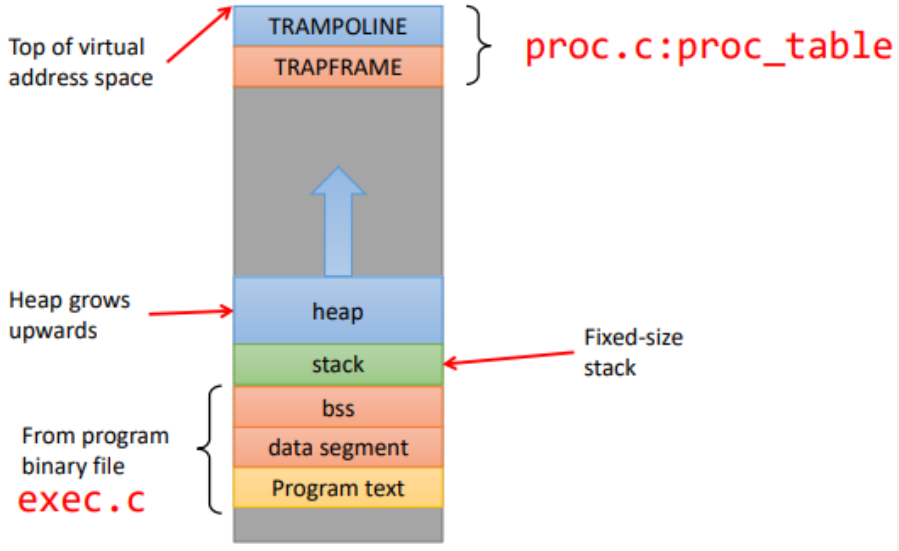


Figure 4.18: Sv39 page table entry.

        - 12-bit: 4 kB per page
        - 9-bit: single page table can be translated in, 512 entries
    - Hardware information in `riscv.h`
        - Page table entry manipulation
    - Mostly in `vm.c`
        - simple, no page faults
        - no separate page table representation
        - no reverse mapping
    - Allocation of physical memory (frames): `kalloc.c`
    - Layout of the address space in `memlayout.h`
    - Interesting bits (`vm.c`)
        - `vm.c: walk()` - simulate the table walker
        - `vm.c: mappages()` - the basic function to create page table entries
        - User and kernel pages have different interfaces
        - `vm.c: uvmalloc()` - implements `sbrk()`
        - `vm.c: copyin()` - copying data between user space and kernel
    - Implementation of `exec()`
        - `exec.c`: probably the most complex use of the `vm.c` interface
- Address space layout
    - Key part of OS design
        - Physical address space
            - Devices, RAM, etc.: mostly determined by hardware

        - Layout of kernel, etc.: in physical RAM, part of OS design
    - User virtual address space
        - Where to text, data, bss, stack, etc. go?
        - Do we need a trampoline?
    - Kernel virtual address space
        - Kernel code and data structures
        - Hardware devices
        - How to access user virtual memory
- xv6 physical address space
    -
```
0x88000000 PHYSTOP  Top of RAM
0x80000000 KERNBASE  Start of RAM, kernel loads here
0x10001000 VIRTIO0  disk controller
0x10000000 UART0  serial controller
0x0c000000 PLIC  (Platform-Level Interrupt Controller)
0x02000000 CLINT  (Core-Local Interrupt Controller)
0x00001000 boot ROM
```
    - see `memlayout.h`, determined partly by QEMU virtual hardware
- xv6 kernel virtual address space (top)
    -


    - "sentinel page" used for stack overflow, triggering page fault
- xv6 user virtual address space

- 

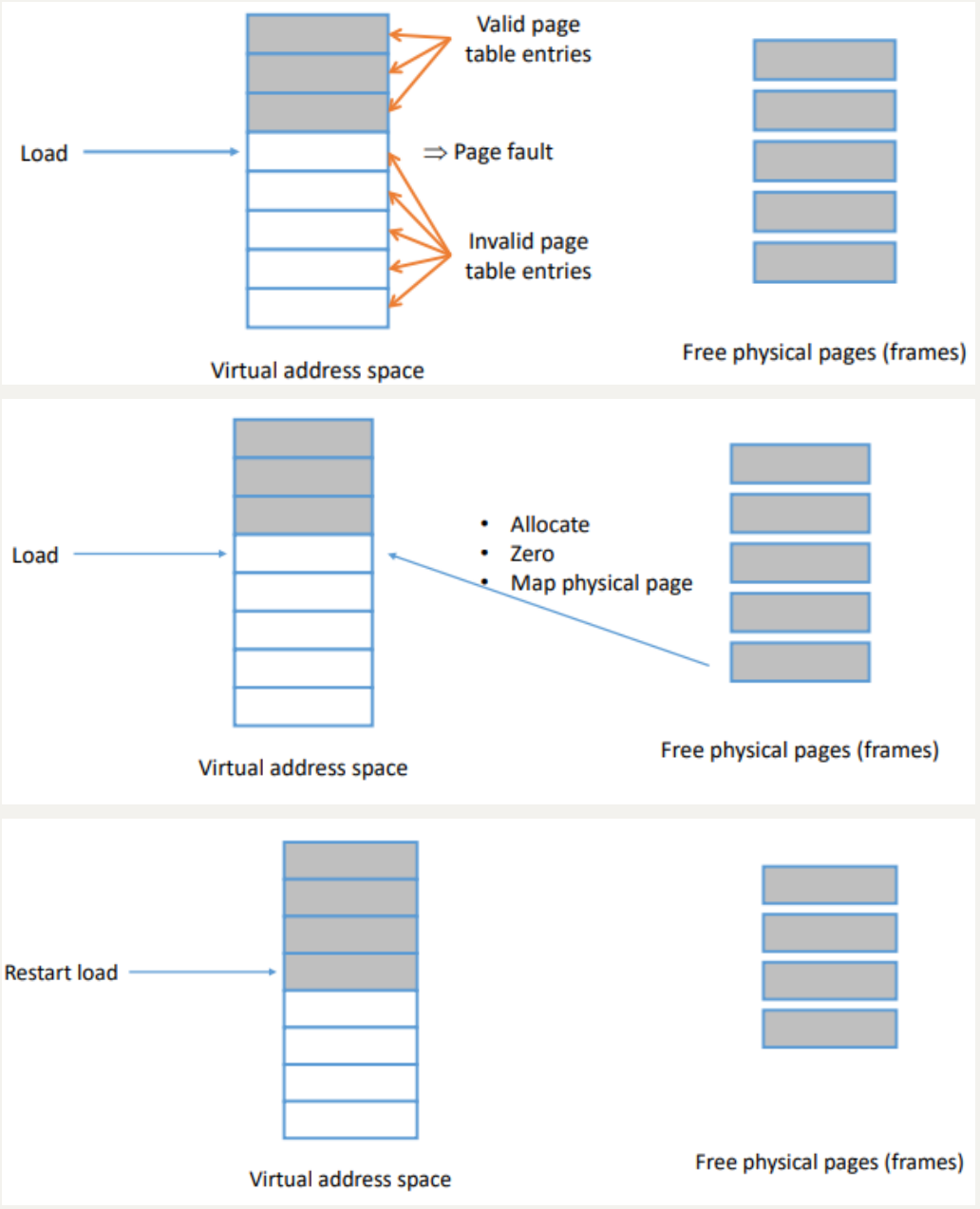
Oct. 16th - Exercise Session 4

- What is a device?
    - Piece of hardware visible from software
    - Occupies some location on a bus
    - Set of registers
    - Source of interrupts
    - etc.
- Programmed I/O vs. Direct Memory Access
    - Device can DMA, no need for CPU
- DMA
    - Avoid PIO for lots of data (requires CPU)
    - Device needs to be able to do DMA (DMA controller)
    - Bypasses CPU to transfer data directly between device and memory
        - Doesn't take up CPU time, only one interrupt per transfer
- I/O protection
    - DMA can be dangerous to normal system operations because they directly access memory
    - need I/O Memory Management Unit
- Evolution of device I/O: PIO with polling → PIO with interrupts → DMA
- Memory Management
    - Allocating and freeing physical address for applications

- Managing the name translation of virtual addresses etc
- Paging
    - physical memory into frames/physical pages (size: power of two)
    - logical memory into (virtual) pages of same size
- Paging Jargon
    - Virtual page numbers
    - Physical frame numbers
    - Page table: VPNs → PFNs (translate via look-up)
- Performance problems
    - Page tables are stored in memory, every virtual page translation needs multiple memory accesses
- Translating with a Translation Lookaside Buffer (TLB)
- Copy-on-Write (COW)
- Caching
    - Write through vs. write back
    - Write allocate
    - Associativity
- Caching + Virtual memory

Oct. 18th - Lecture 9: Memory Management (Redo recording after understanding memory mapping/paging and cache)

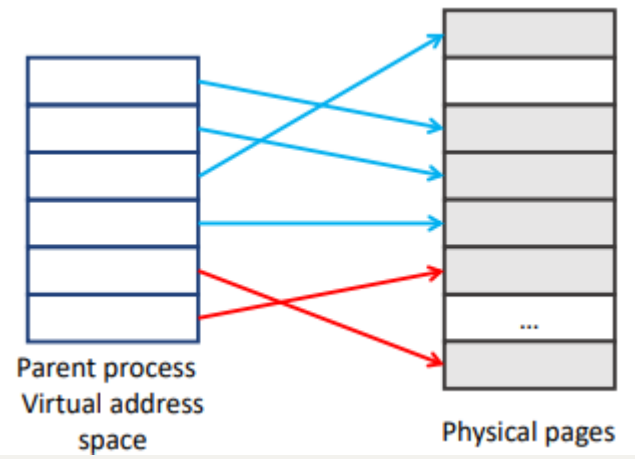
- Fill-on-demand and Copy-on-Write
    - Zero-on-reference
        - How much physical memory is needed for the stack or heap? (Only what is currently in use)
        - When program uses memory beyond end of stack
            - Page fault into OS kernel
            - Kernel allocates some memory (how much?)
            - Zeroes the memory (avoid accidentally leaking information)
            - Modify page table

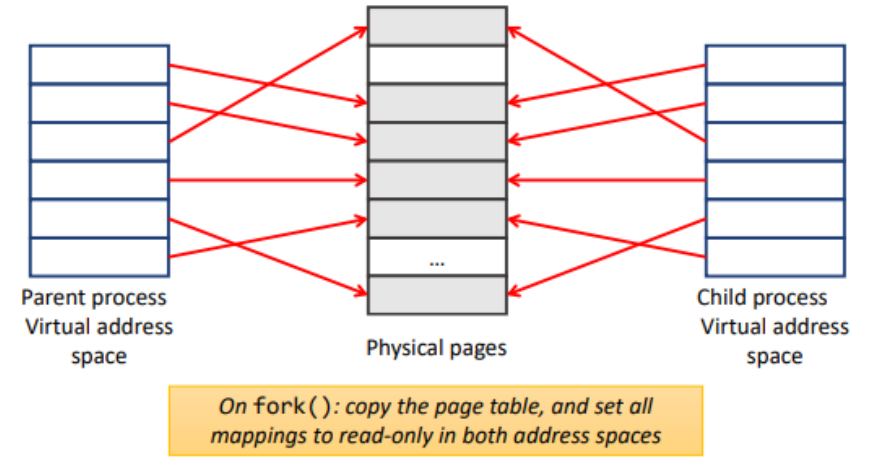
    - Resume process
- 

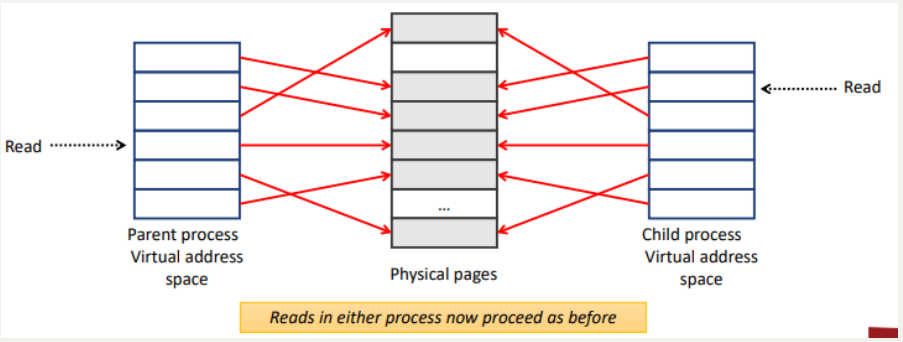
- Fill On Demand (can start running a program before its code is in physical memory)
    - Set all page table entries to valid
    - When a page is referenced for first time (page faults), kernel trap
    - Kernel brings page in from disk
    - Resume execution
    - Remaining pages can be transferred in the background while program is running
- Recall `fork()` copies the entire address space

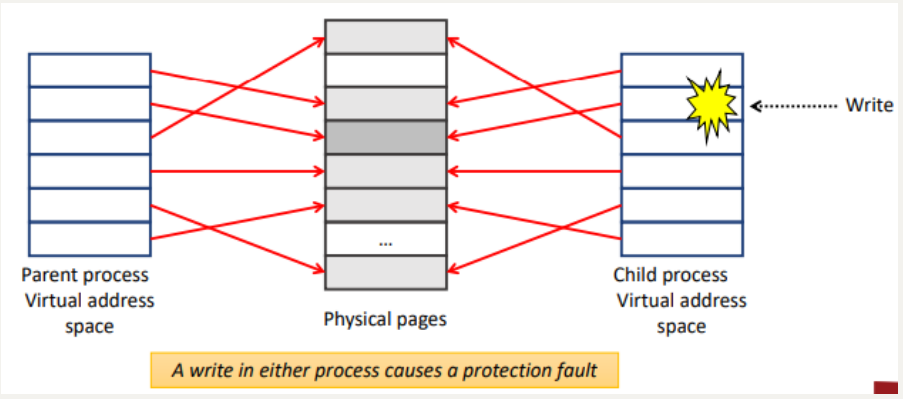
- Can be expensive if `exec()` immediately follows, or most of the child does not change after forking
    - In early UNIX, when changing between processes, we copy all of memory space into disk, and swap the memory of the other process into memory
    - Old fork is just copy original process into disk, and rename the process on memory
- Can use "CoW" to avoid copying until necessary, make `fork()` feasible in performance
- But if we end up mutating most of the pages, CoW can be more expensive than a simple copy

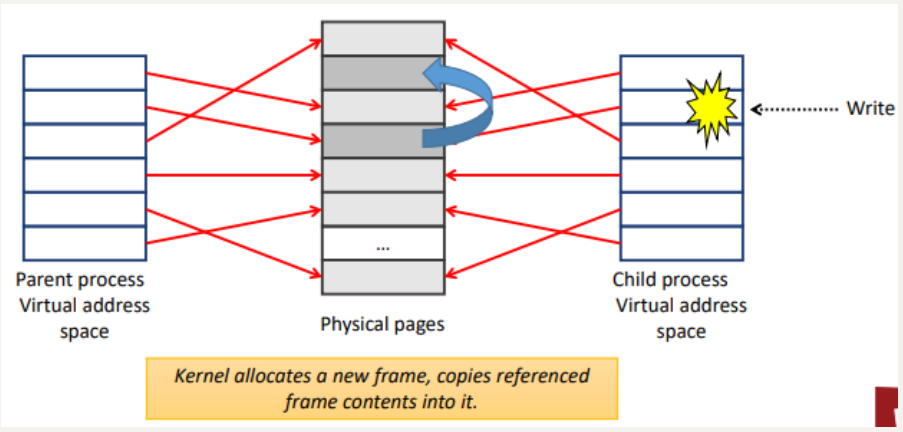
- Copy-on-Write
    - Blue arrows: Read/Write mapping; Red arrows: Read-only mapping

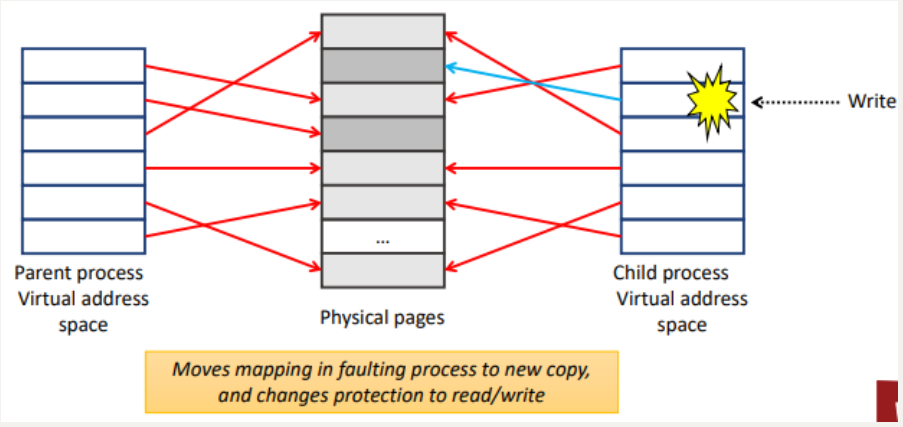


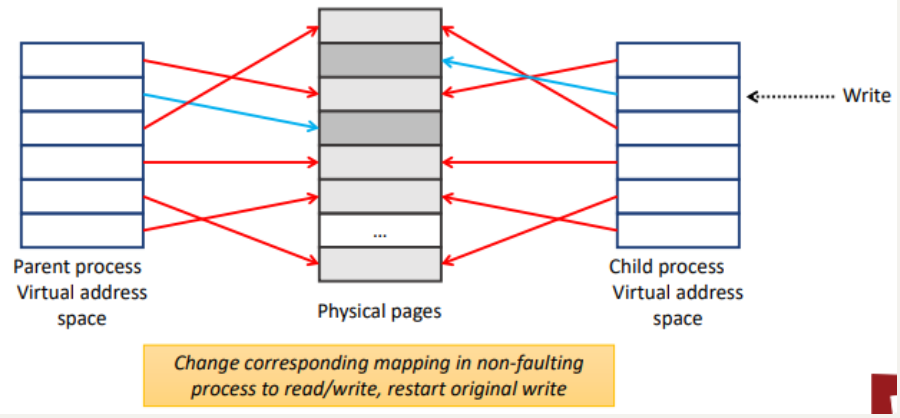



Read
Read
Parent process
Virtual address
space
Physical pages
Child process
Virtual address
space
Reads in either process now proceed as before


Write
Parent process
Virtual address
space
Physical pages
Child process
Virtual address
space
A write in either process causes a protection fault


Write
Parent process
Virtual address
space
Physical pages
Child process
Virtual address
space
Kernel allocates a new frame, copies referenced
frame contents into it.


Write
Parent process
Virtual address
space
Physical pages
Child process
Virtual address
space
Moves mapping in faulting process to new copy,
and changes protection to read/write

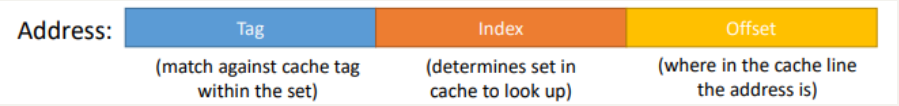


- Frames now belong to more than one process (sharing)
- Read-only mappings might be CoW or not
- Which processes have mappings to this frame? Need PFN → VPN table
- A process use ??? physical memory

- Managing caches and the TLB
  - Context-switching the TLB
    - Cannot leave the TLB as it is, new process mappings are different
    - Can flush TLB → performance hit?
    - Solution: Tagged TLB

      Each TLB entry has a tag (e.g. 6-bit value)

      Processor tag register written by the OS

      Only TLB entries where the tag matches are register are considered valid
    - Only 64 tags: OS must manage dynamic mapping tags ↔ processes
  - Types of caches
    - 

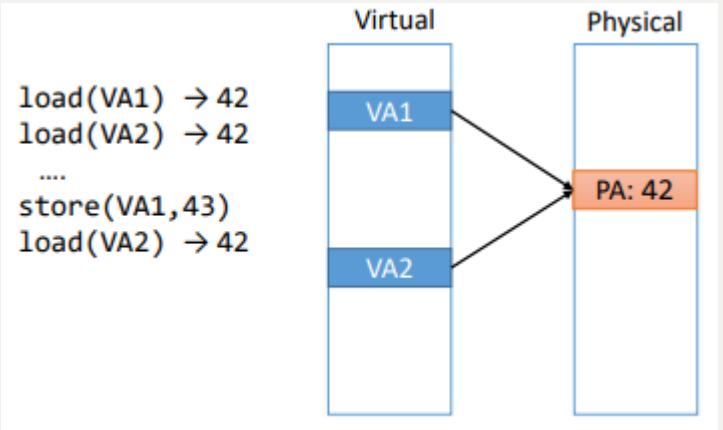
    - Virtually indexed, virtually tagged
      - Simple and fast, but context switch?
    - Physically indexed, physically tagged
      - Can only be accessed after address translation
    - Virtually indexed, physically tagged
      - Overlap cache and TLB lookups

        - Physically indexed, virtually tagged
            - Only known implementation: MIPS R6000
        - Write policies
            - Write through: writes immediately go through cache to main memory
            - Write back: cache lines can be **dirty**, written back later
        - Allocate policies
            - Write allocate: a write miss causes the line to be allocated in the cache
            - Non-write allocate: a write miss bypasses the cache, only a read miss allocates
- Homonyms in virtually-tagged caches
    - Same virtual address maps to multiple physical address spaces (in different processes, may access the wrong data)
    - Solutions

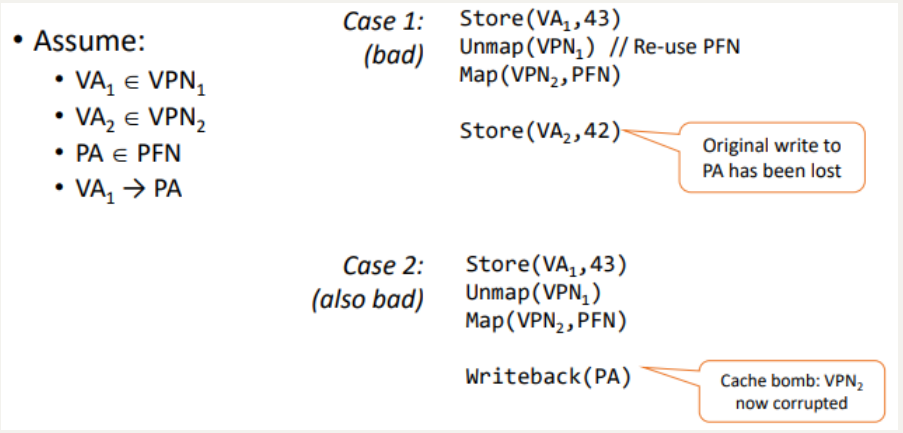
      Flush cache on a context switch

      Ensure disjoint virtual address spaces

      add Address Space Identifier

      Use physical tags
- Synonyms in virtual caches
    - Two virtual addresses map to same physical address
    - Write to one address
    - Inconsistent data in cache at other address
    - 

- Preventing synonyms in virtual caches
    - Physical tagging, ASIDs, cache flushing do not help

        - Hardware synonym detection (expensive)
        - Detect synonyms and ensure all mappings read-only, or only one synonym mapped a time
        - Restrict virtual memory mappings so synonyms map to same cache set (Loading from one synonym evicts the others)
    - Synonym problems in physically-tagged caches
        - Unmap a page with a **dirty** cache line
        - Reuse frame for a different page
        - Write to new page
            - If same cache line: new write overwrites old, original write lost
            - If different cache line: dirty line persists, get written back some time in the future → "cache bomb"
        - 

    - Multiprocessors
        - Caches are usually coherent in multiprocessors: hardware maintains some consistency model between cores
        - Hardware for TLB consistency is very rare
        - Solution

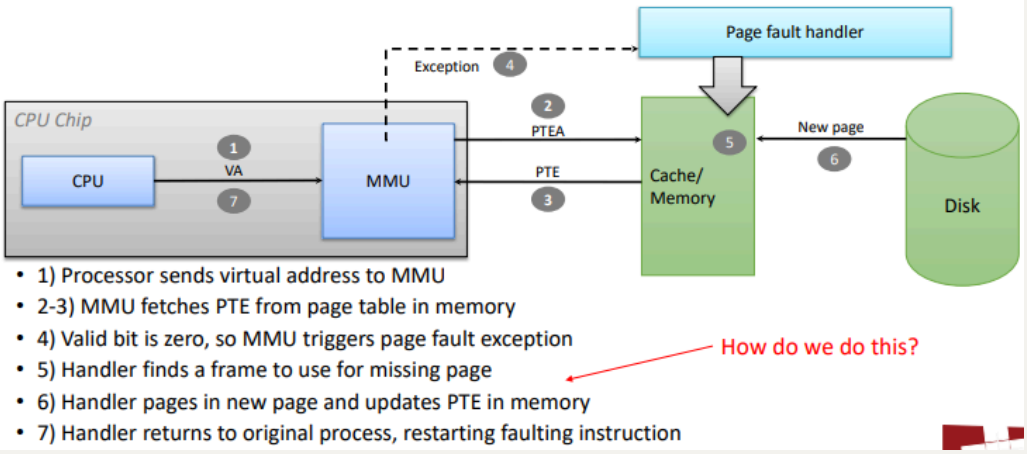
          OS must manage TLB consistency itself

          changing a mapping → ensure no other TLBs in the system have that mapping

          issue IPI (InterProcessor Interrupt)

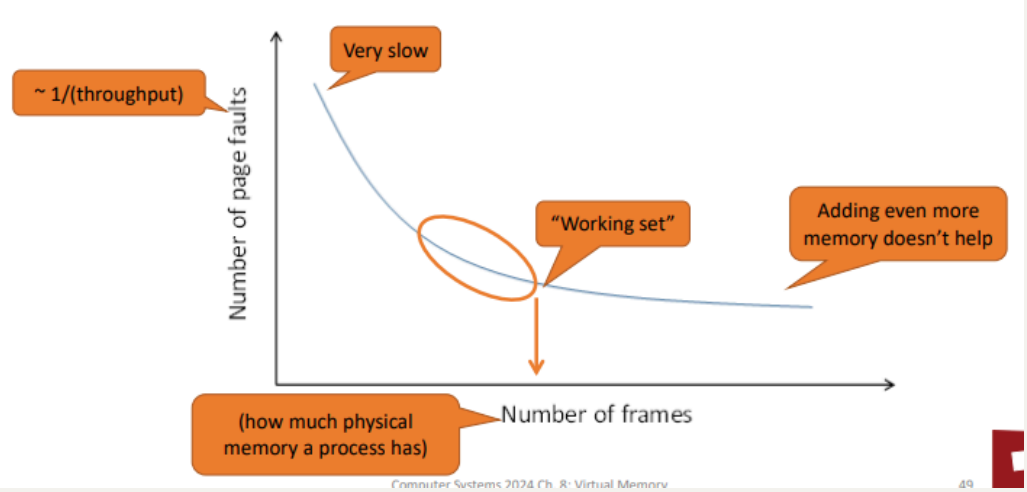
          OS kernel on each core checks for invalidations

          Very slow - for optimization
- Demand Paging
    - Handling a page fault

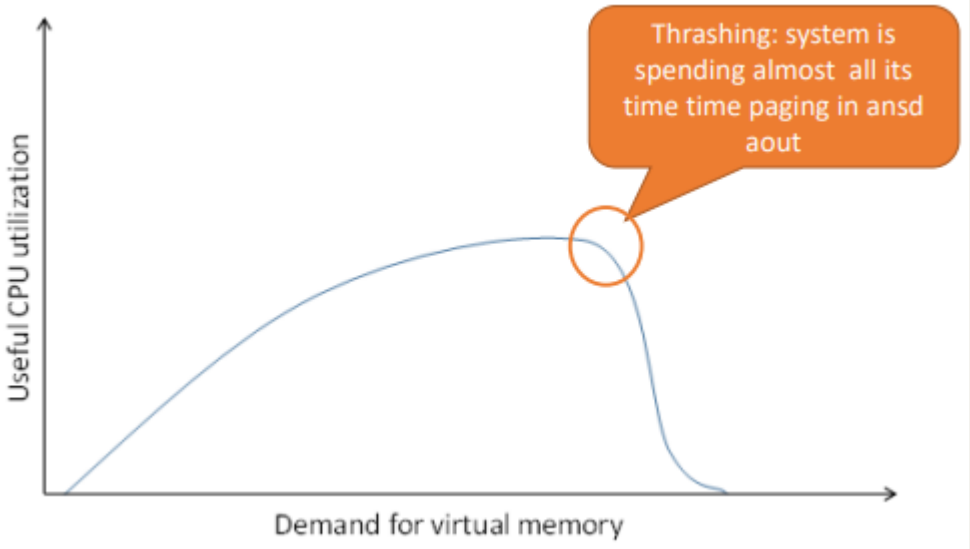


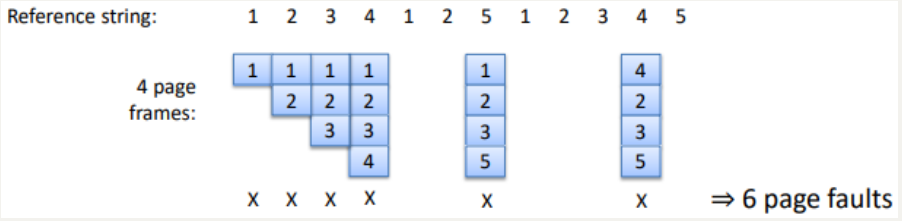
- Page fault rate
    - $0 \leq p \leq 1$, where $p = 0$ - no page faults; $p = 1$ - every access faults
- Effective access time (EAT)
    - $(1 - p) \times l_m + p \times l_f$
    - l_m: latency of memory access

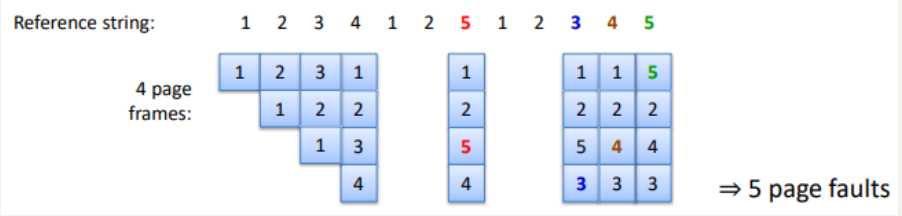
      l_p: latency of page fault handling = page fault overhead + page swap in + page swap out + restart instruction + l_m
- Performance of a paging system



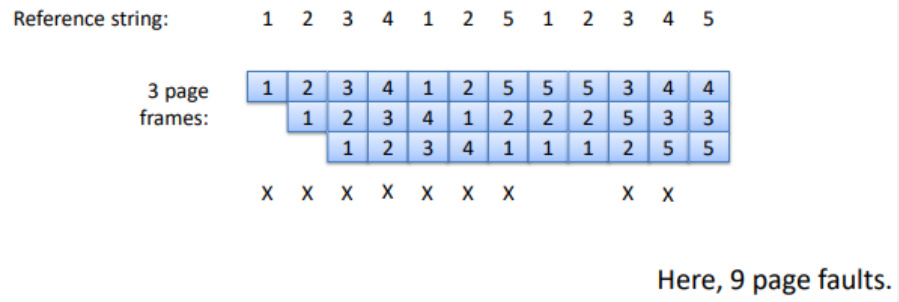
- Thrashing

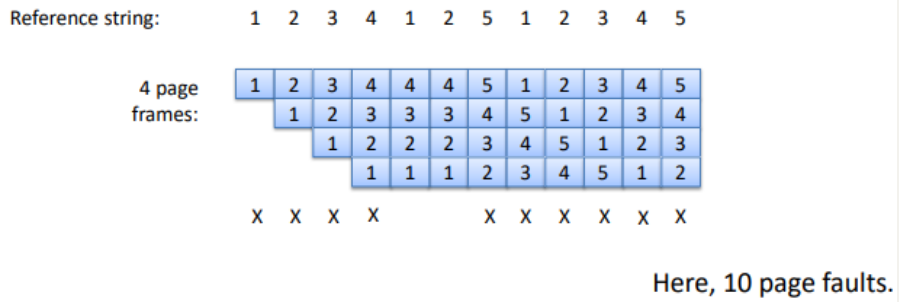


- Page Replacement Policies
    - Page replacement
        - Choose which page to be evicted, how to pick the best page
        - The best page is the page that minimizes EAT for a given workload
        - Relative to a reference string (sequence of page accesses)
    - Optimal algorithm
        - Replace page that **will not be used** for longest period of time
        - Used for measuring how well the algorithm performs
        - 

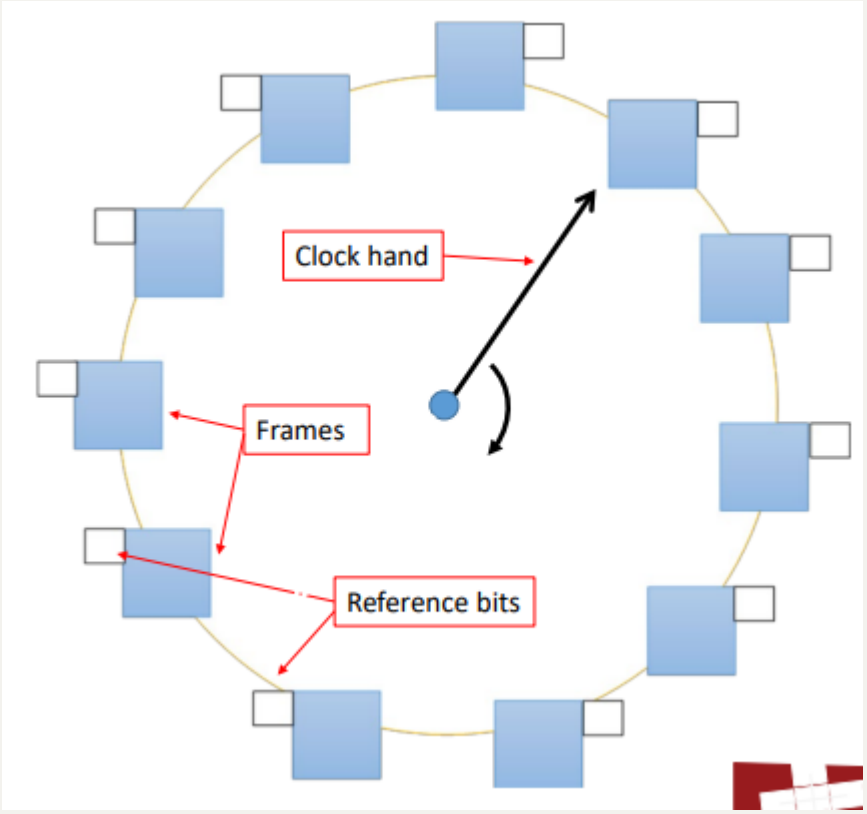
    - Least Recently Used (LRU) algorithm
        - Replace page that **has not been used** for longest period of time
        - Have to track every reference to the page
        - Problem: OS generally does not get informed on every hit
        - Solution: Add flags tat track access and periodically check & clear them
        - 

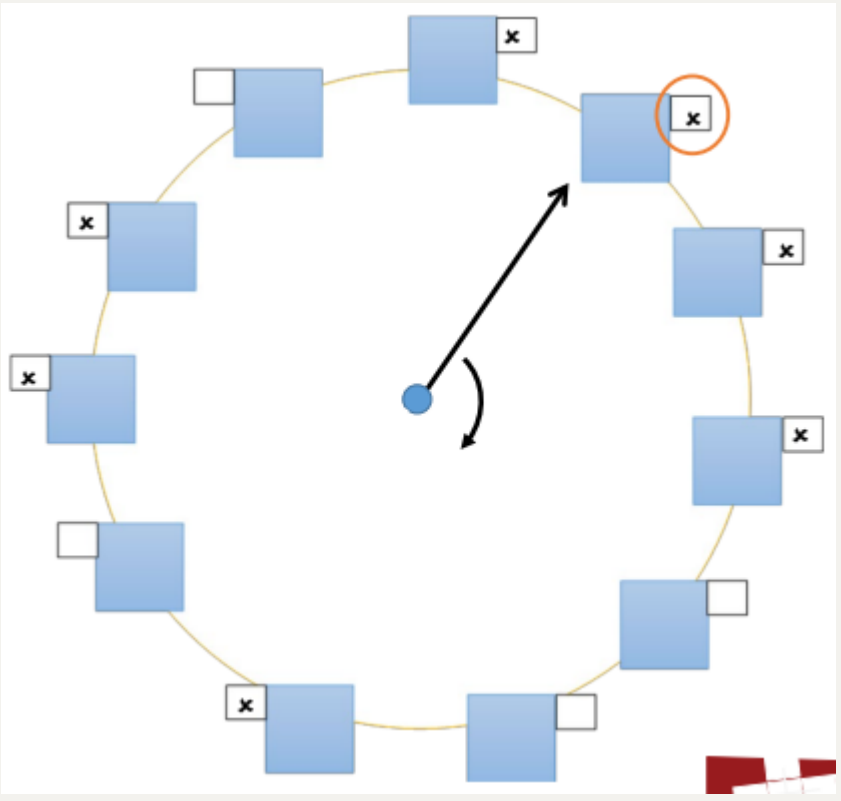
    - FIFO and Bélàdy's anomaly

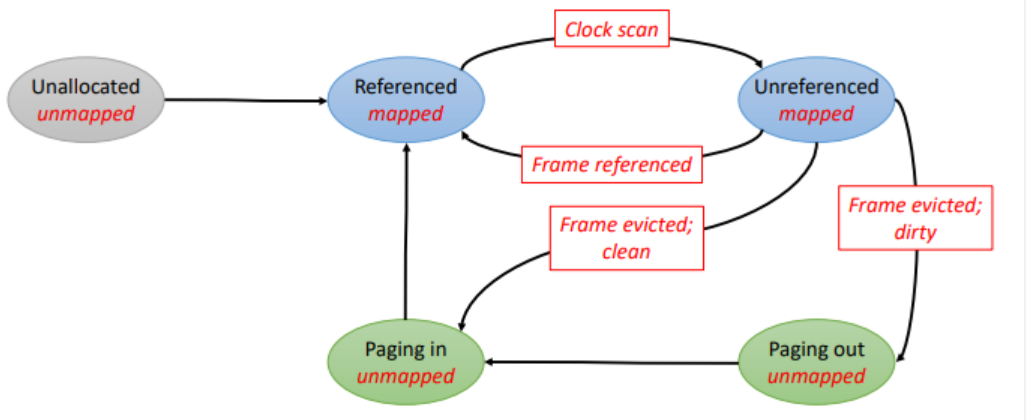


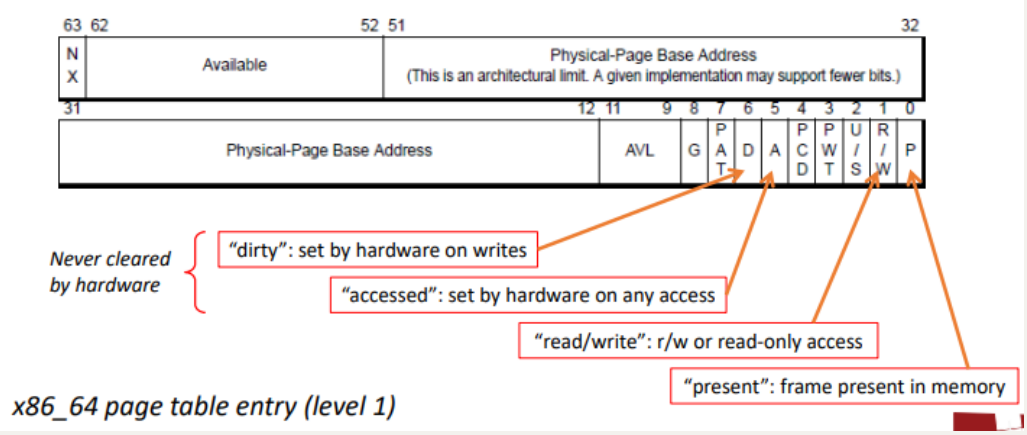


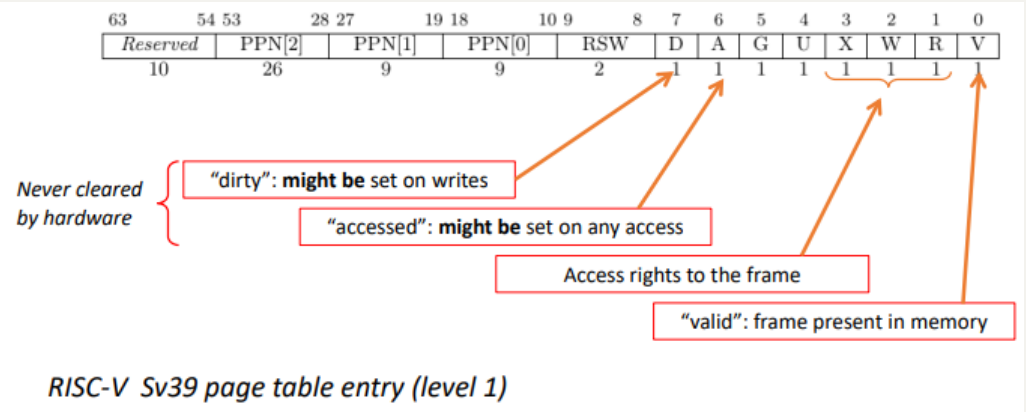
-
    - Adding more frames increases the number of page faults - anomaly
- A better replacement policy?
    - Optimal is theoretically impossible
    - LRU is too expensive (need to record every access)
    - FIFO is poor for most workloads
    - Good compromise: **Clock** (or 2nd chance)
        - Approximates LRU
        - Requires a linear table of all PFNs, with associated "referenced" bits
        - Best visualized as a circular buffer
- Clock algorithm

- 

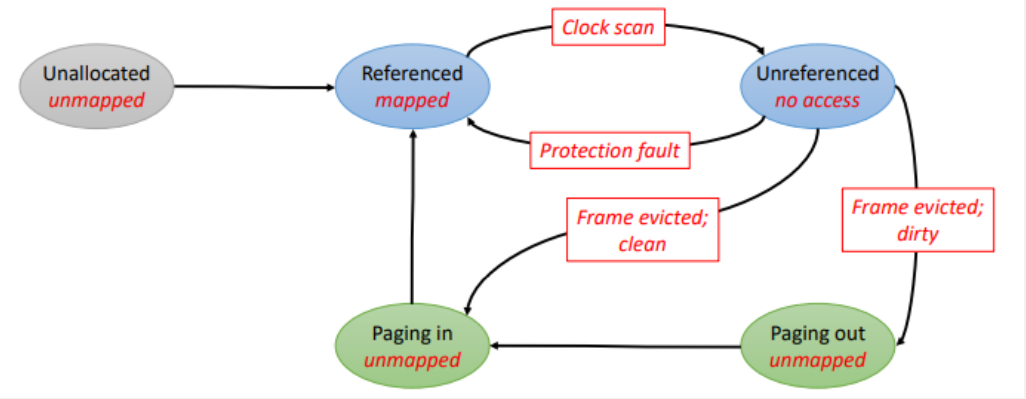
- Assume all frames are already allocated, and no frame marked referenced
- Mark each frame when referenced for the first time
- 
- To replace a frame, look at the "hand"
    - If marked: unmark it, advance hand, repeat
    - If unmarked: this is the frame, allocate and mark it

    - If the clock hand goes all around, and the frame is still not been referenced, then we can page in
- Implement paging: frame states
    - 

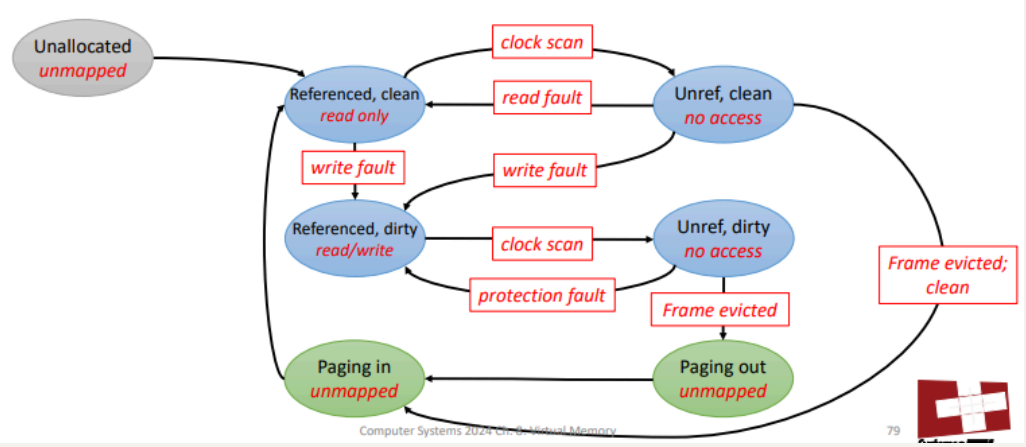
    - Knowing when a page is reference: x86_64
    - 

    - ~: RISC-V
    - 

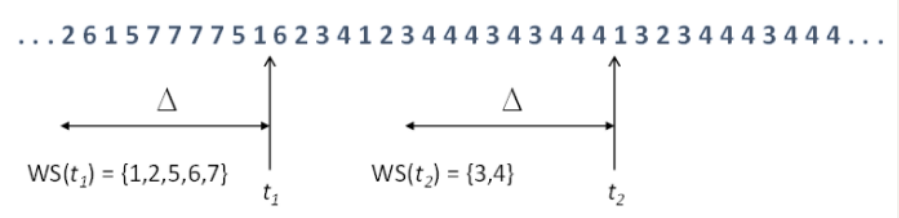
    - Emulating the "accessed" bit



- Emulating the "modified" bit



- Allocating physical memory
  - Allocating frames between processes
    - What process gets the most physical memory? What's fair?
      - Avoid one process "starving" others of memory by faulting a lot
      - Avoid making no progress because it does not have enough memory
    - Example policies
      - Global allocation: evict a page from the set of all frames in the system
      - Local allocation: evict a page from the set used by the faulting process
      - Allocate every process an equal share in the pages
      - Give higher-priority processes more memory
      - Given more memory to processes that make more page faults
  - Defining the working set

        - The working set $W(t, \tau)$ of a process at time t is the set of virtual pages referenced by the process during the previous time interval $(t - \tau, t)$. The **working set size** $w(t, \tau)$ of a process at time t is the $|W(t, \tau)|$
        - Allocating $w(t, \tau)$ pages to a process will prevent it from thrashing, and allow it to make good progress
    - Working set example
    - 

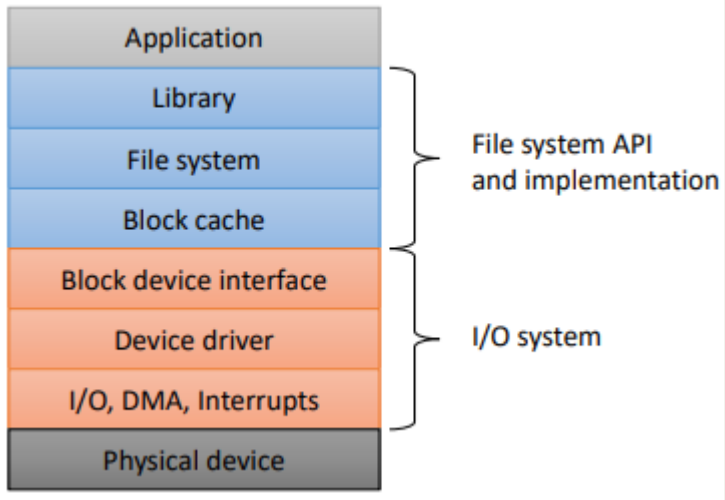
Oct. 21st - Lecture 10: File System Abstractions

- What is the filing sytem?
    - **Virtualizes** the disk
    - Between disk (blocks) and programmer abstractions (files)
    - Combination of multiplexing and emulation
    - Generally part of the core OS
- File system needs to provide:

Goal	Physical characteristic	Design implication
High performance	High cost of I/O access	Organize placement: access data in large, sequential units Use caching to reduce I/O
Named data	Large capacity, persistent across crashes, shared between programs	Support files and directories with meaningful names
Controlled sharing	Device stores many users' data	Include access control metadata with files
Reliable storage	Crashes occur during update	Transactions to make set of updates atomic
	Storage devices fail	Redundancy to detect and correct failures
	Flash memory wears out	Wear-levelling to prolong life

Computer Systems 2024 Ch. 10: File System Abstractions

- File system builds on:



- The File Abstraction
    - What is a file to the filing system?
        - Some data, a size, one/more name(s) for the file, other metadata and attributes, the type of the file, some structure (data organization), where the data is stored on the disk
        - Regular file with data, pipes, devices, soft-link, UNIX main sockets
    - File metadata
        - Metadata: data about an object, not the object itself
        - Names, location on disk, times of creation, last change, last access, ownership, access control rights, file type, file structure, arbitrary descriptive data, **reference counts**, **inode number**, etc.
    - Kinds of files
        - Byte sequence
        - Record sequence
            - Fixed (at creation time) records
            - Mainframes or minicomputer OSes of the 70s/80s
        - Key-based, tree structured
            - e.g. IBM indexed sequential access method
            - Mainframe feature, now superceded by databases, moved into libraries
    - Byte sequence
        - vector of bytes: can be appended to, can be truncated, can be updated in place, typically no "insert"
        - accessed as: sequential files, random access

        - Record sequence
            - vector of **fixed-size records**: actions similar to that of a byte-sequence file
            - Record size (and perhaps format) **fixed at creation time**: read/write/seek operations take records and record offsets instead of byte addresses
- File naming
    - File system operations
        - Naming context → directory
        - Bind a name → `/bin/ln` or `link()` (need an existing name for the object)
        - Remove a binding → `/bin/rm` or `unlink()`
        - Enumerate bindings → `/bin/ls` or ?
    - POSIX directories are (special) files
        - Directories are files → pathnames are sequences of directory lookups

          `drwxrwxrwt`
        - '/' is a special name for the `root` context
        - Lookups start at the root or the Current Working Directory
            - change the CWD with `cd`
            - change the root with `/bin/chroot`
        - Changing the root **sandboxes** the process
            - limits the namespace it can refer to
    - Is a directory really a file?
        - Yes: allocated just like a file on disk, has entries in other directories like a file
        - No: Users cannot arbitrarily read/write to it (which can corrupt file system data structures, bypass security mechanisms); File system provides **special interface** (`opendir, closedir, readdir, seekdir, telldir`, etc)
    - Multiple bindings to a single file cause issues
        - When to **free up space** of a file?
            - **Garbage collection** problem: detect when no references exist

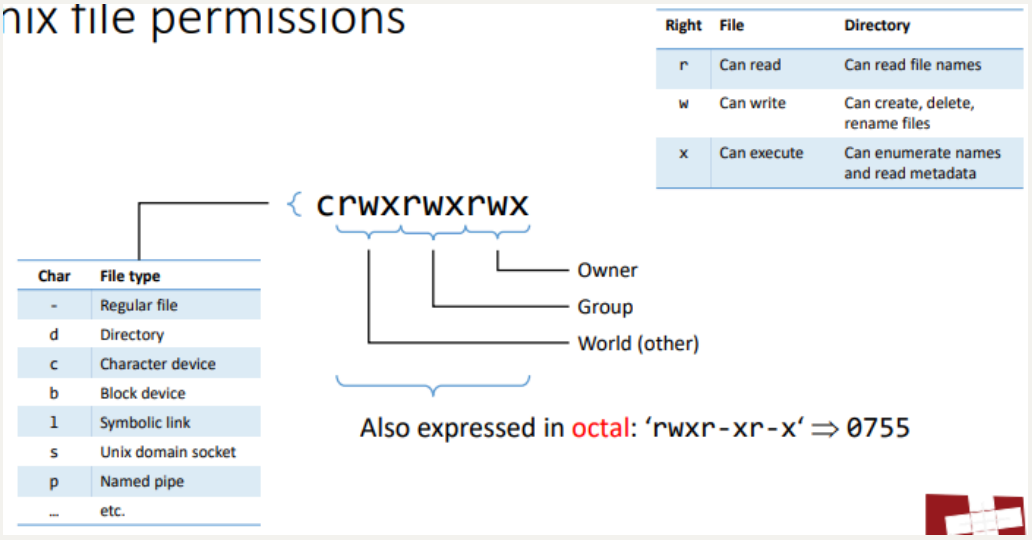
        - **Reference counts**
    - What about cycles? Reference counting do not work on cyclic structures
        - Disallow creating new links to directories
        - Only cycles are hardcoded: `..` and `.`
        - Unix naming graph is therefore **mostly acyclic**
    - N.B. symbolic links are different: they are allowed to "dangle"
- Access Control
    - Protection
        - File owner/creator should be able to control what can be done and by whom
        - Types of access: read, write, execute, append, delete, list
        - (principal, object, operation) $\mapsto$ {True, False}
    - Access control matrix

Principals (e.g. users)

Objects (e.g. files)	A	B	C	D	E	F	G	H	J	...
1	R	RW	R			X	R			
2	RW	RWX		W			RX			
3	RW				RWX					
4	RX	X	X	X						
5	X									
6	R				RWX					
...										

        - Problem: scalably represent this matrix
    - Row-wise: ACLs
        - Access Control Lists: for each right on a file, list the principals
        - Store with the file
        - Advantage: easy to change rights quickly, scales to large numbers of files
        - Disadvantage: does not scale to large numbers of principals
    - POSIX (Unix) access control
        - Simplifies ACLs: each file identifies 3 principals
            - Owner (a single user)
            - Group (a collection of users, defined elsewhere)
            - The world (everyone)

    - For each principal, file defines 3 rights
        - Read (or traverse, if a directory)
        - Write (or create a file, if a directory)
        - Execute (or list, if a directory)
- Unix file permissions



    - The first three should be owner, second is group, last three should be world
- Special Unix permissions: sticky and setuid bits

Right	File	Directory
setuid	File executes with owner's UID	No effect
setgid	File executes with owner's GID	New files inherit GID of the directory
text/sticky	[Program is sticky]	Can't move, rename, or delete files you do not own

```
$ ls -ld /var/tmp /tmp /bin/mount /bin/su
-rwsr-xr-x  1 root root 55528 Mai 30 17:42 /bin/mount
-rwsr-xr-x  1 root root 67816 Mai 30 17:42 /bin/su
drwxrwxrwt 22 root root 12288 Okt 19 16:33 /tmp/
drwxrwxrwt 11 root root  4096 Okt 19 14:20 /var/tmp/
```

- Full ACLs
    - POSIX now supports full ACLs, but rarely used
    - Windows has very powerful ACL support
        - Arbitrary groups as principals
        - Modification rights
        - Delegation rights
        - Used extensively
- Column-wise: Capabilities

        - Each **principal** with a **right** on a **file** holds a **capability** for that right
            - stored with principal, not object (file)
            - cannot be forged or (sometimes) copied
        - Advantage: very flexible, highly scalable in principals; access control resources charged to principal
        - Disadvantage: Revocation - hard to change access rights (need to keep track of who has what capabilities)
- Open Files
    - What can we do with a file contents - given its name
        - `exec()`: replaces the currently running progress with the program
        - `open()`: returns a **file descriptor**
        - This describes an **open file**, this is not a file
    - What is an Open File?
        - Context for file operations: read, write, truncate, seek, tell, mmap, close, etc
        - Can be **shared** between process
            - e.g. after `fork()`
        - Has **state**
            - e.g. a `seek()` pointer
        - A file can have > 1 open files
            - can be opened more than once
        - File descriptor is a **capability**
    - Memory-mapped files
        - Use virtual memory system to **cache** files
            - Map file content into virtual address space
            - Set the **backing store** of region to file
            - Can now access the file using load/store
        - When memory is paged out
            - Updates go back to file instead of swap space
- Concurrency
    - Concurrency

        - Must ensure that, regardless of concurrent access, file system **integrity** is ensured
            - careful design of file system structures
            - internal locking in the file system
            - ordering of writes to disk to provide transactions
        - Provide mechanisms for users to avoid conflicts themselves
            - advisory locks
            - mandatory locks
    - Common locking facilities
        - Type
            - Advisory: separate locking facility
            - Mandatory: write/read operations will fail
        - Granularity
            - Whole-file
            - Byte ranges (or record ranges)
            - Write-protecting executing binaries
    - Compare with databases
        - Databases have way more sophisticated notions of
            - Locking between concurrent users
            - Durability in the event of crashes
        - Records and indexed files have largely disappeared in favor of databases
        - File systems remain much easier to use
            - and much, much faster
            - as long as it does not matter

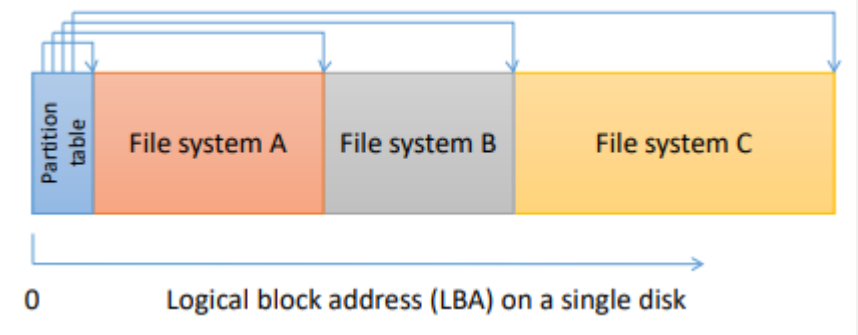
Oct. 23rd - Exercise Session 5

- Demand Paging
    - Idea: bring page into memory only when it's actually touched (lazy); conceptually turn memory into cache ...?
    - Page replacement
        - if there is a free frame just use it

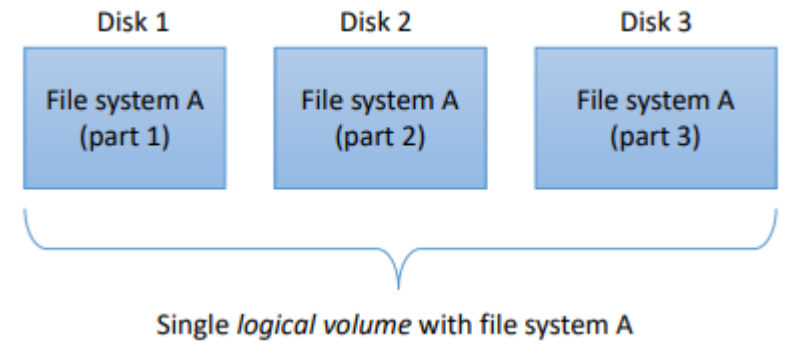
        - or, find "little used" resident page to discard or write to disk
        - Policy defines which page is chosen: optimal, FIFO, 2nd chance etc.
    - Optimal algorithm
        - Replace page that will not be used for longest period of time
        - Used to measure how well other algorithms perform
    - Least Recently Used
        - Replace page that has not been used for longest period of time (have to track every reference to the page)
        - Problem: OS does not get informed on every hit
        - Solution: Add flags that track access and periodically check and clear them
    - FIFO and Bélàdy's anomaly
        - More page frames, more page faults
    - 2nd chance clock
        - Workaround for lack of hardware reference bit
            - mark the virtual page invalid, next time the virtual page is referenced on the OS will take a trap on the reference, and can set the bit in software before marking the page valid and continuing (trap and patch)
        - faster and more scalable than LRU
            - simple data structure, no need to track all accesses
- Allocating Frames between Processes
    - Global physical page allocation
    - Local physical page allocation
        - Choose the per-process set sizes for local allocation
            - Equal
            - Priority-Based
            - ...

Oct. 25th - Lecture 11: File System Implementation

- "Files" vs. "Open Files"
    - Typical operations on files: rename, stat, create, delete, **open** (on-disk data structures)
    - **Open** creates an "open file handle"
        - Different class of object
        - Allows reading and writing of file data
        - In-memory data structures
- The File System vs. File Systems
    - Partitions



        - Multiplex single disk among >1 file systems
        - Contiguous block ranges per FS
        - Virtualizes storage in a different way
    - Logical volumes

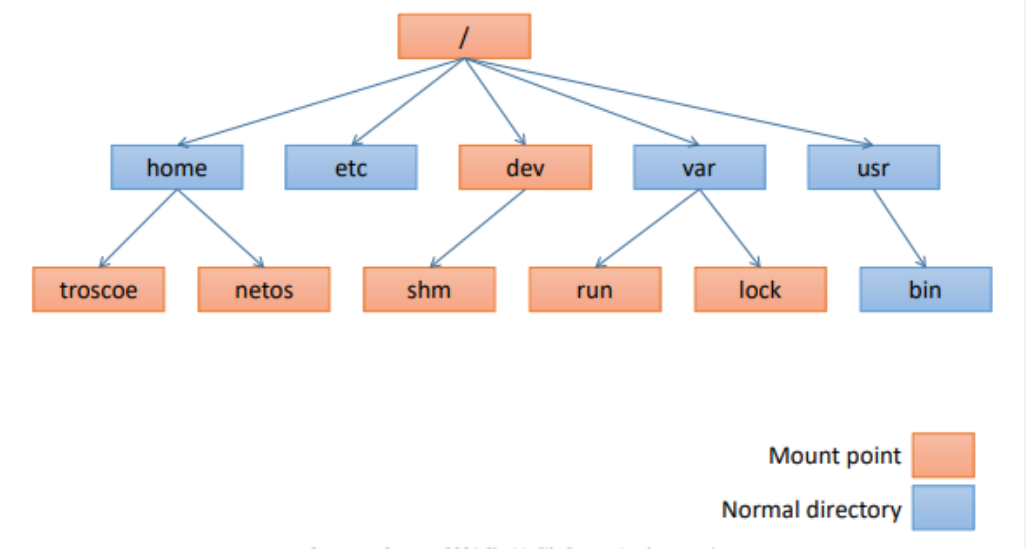


        - **Emulate** 1 virtual disk from >1 physical disk
        - Single file system spanning >1 physical disk
        - Also virtualizes storage
    - Multiple file systems
        - How to name files in multiple file systems?
        - Top-level volume names: Windows C:, D:

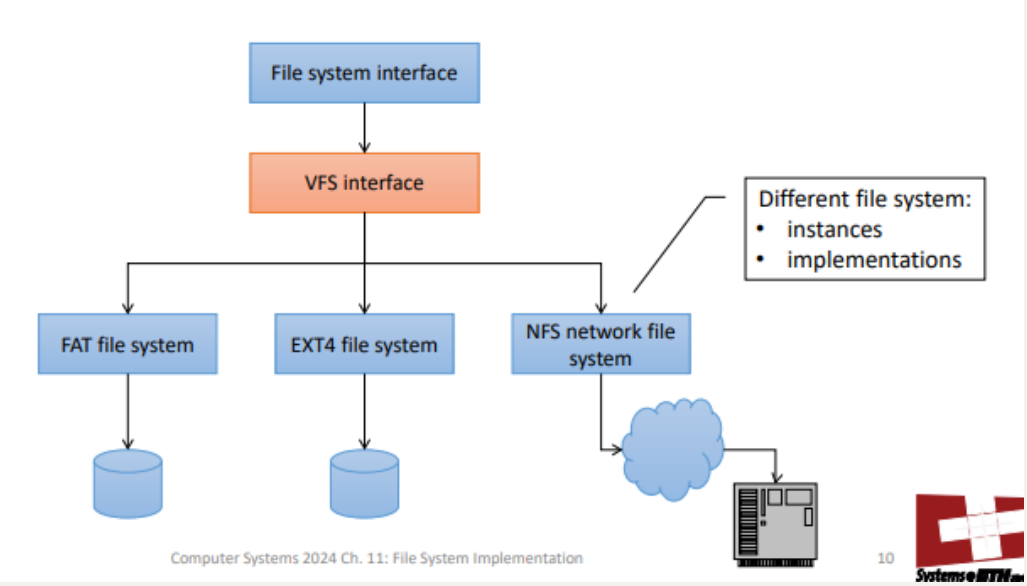
    - Bind "mount points" in name space (Unix, etc)
- Mount points (mount a file system onto another file system)

```
cixous: /home/troscoe> df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1            230693168 168428508  50546048  77% /
none                   1672472       388   1672084   1% /dev
none                   1676688     14568   1662120   1% /dev/shm
none                   1676688       128   1676560   1% /var/run
none                   1676688         0   1676688   0% /var/lock
none                   1676688         0   1676688   0% /lib/init/rw
/dev/sdb1            241264000 193226824  48037176  81% /media/4b5b72a5-a62c-481a-9f2a-f908400131a9
fs.roscoe.inf.ethz.ch:/export/groups/alonso/h1/home/troscoe
                     1043992544 586748832 457243712  57% /home/troscoe
//fs-systems.inf.ethz.ch/teaching
                     1043992532 555417256 488575276  54% /media/teaching
fs.systems.inf.ethz.ch:/export/groups/systems/p1/project/netos
                     1043992544 555417248 488575296  54% /home/netos
cixous: /home/troscoe>
```

File hierarchy with mounts



- Virtual File Systems (VFS)
    - VFS provide an **object-oriented** way of implementing file systems
    - VFS allows the same **system call interface** (the API) to be used for **different types** of file systems
    - The API is to the VFS interface, rather than any specific type of file system



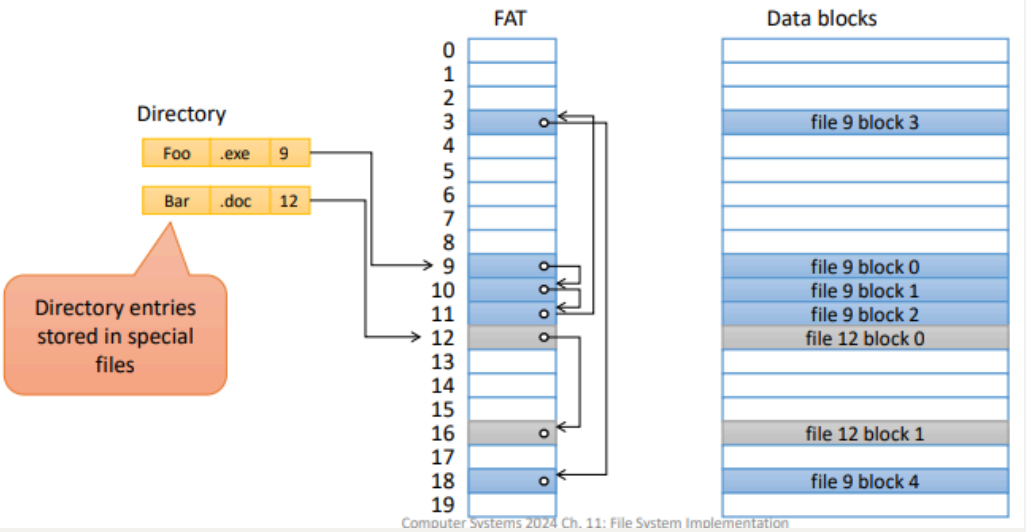
- On-disk data structures
    - Disk addressing
        - Disks have tracks, sectors, spindles etc
            - And bad sector maps
        - More convenient to use **logical block addresses**
            - treat disk as compact linear array of usable blocks
            - Block size typically 512 B
            - Ignore geometry except for performance
        - Also **abstracts** other block storage devices
            - Flash drives (load-levelling)
            - Storage-area Networks (SANs)
            - Virtual disks (RAM, RAID, etc)
    - Implementation aspects
        - Directories and indexes
            - where on the disk is the data for each file
        - Index granularity
            - what is the unit of allocation for files
        - Free space maps
            - how to allocate more sectors on the disk
        - Locality optimizations
            - how to make it go fast in the common case
    - Directory Implementation

- Linear list of (file name, block pointer) pairs
    - simple to program
    - lookup is slow for lots of files (linear scan)
- Hash Table - linear list with closed hashing
    - Fast name lookup
    - collisions
    - fixed size
- B-tree - name index, leaves are block pointers
    - Increasingly common, complex to maintain, but scales well

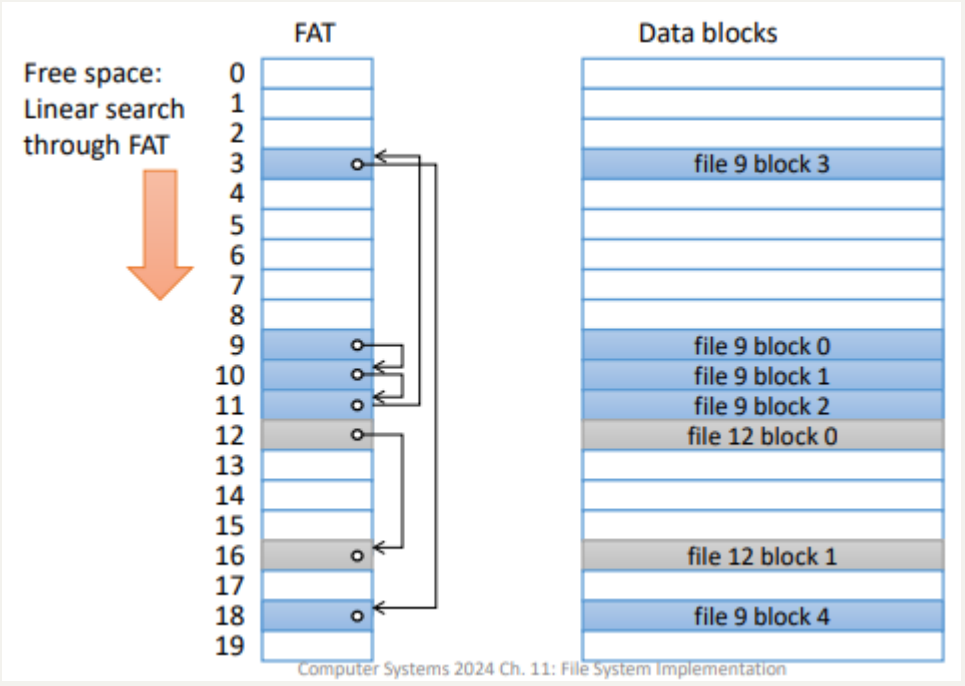
	FAT	FFS	NTFS	ZFS
Index structure	Linked list	Fixed, assymetric tree	Dynamic tree	Dynamic COW tree
Index granularity:	Block	Block	Extent	Block
Free space management	FAT Array	Fixed bitmap	Bitmap in file	Log-structured space map
Locality heuristics	Defragmentation	Block groups, Reserve space	Best fit, Defragmentation	Write anywhere, Block groups

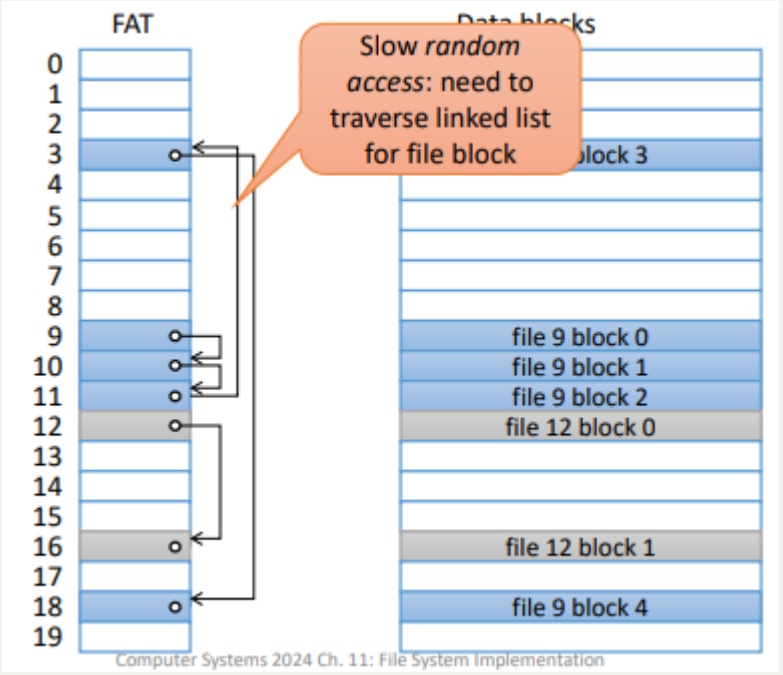
See book for details

- FAT-32
    - FAT background
        - very old - dates back to 1970s
        - No access control
        - Very little metadata
        - Limited volume size
        - No support for hard links
        - but still extensively used: flash devices, cameras, phones
    - On-disk data structures

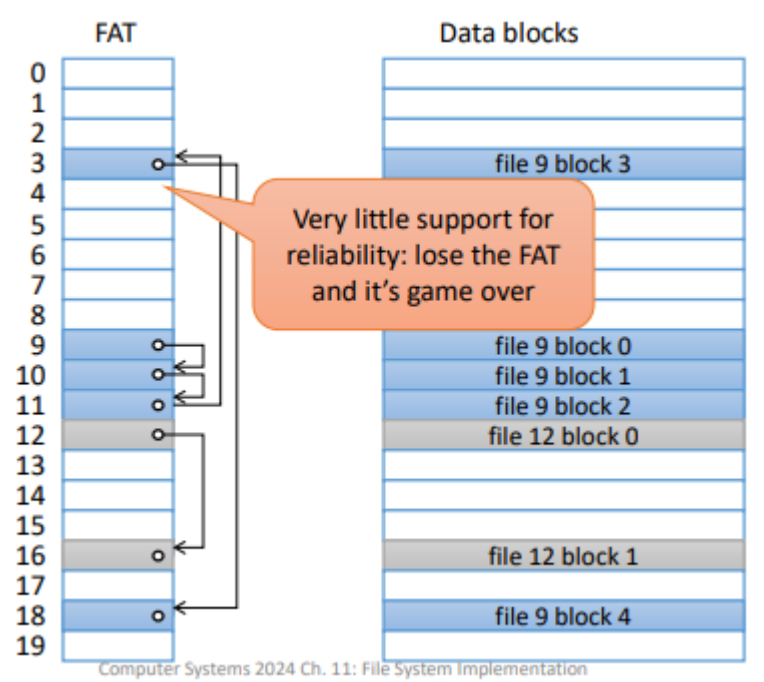


- Block allocation

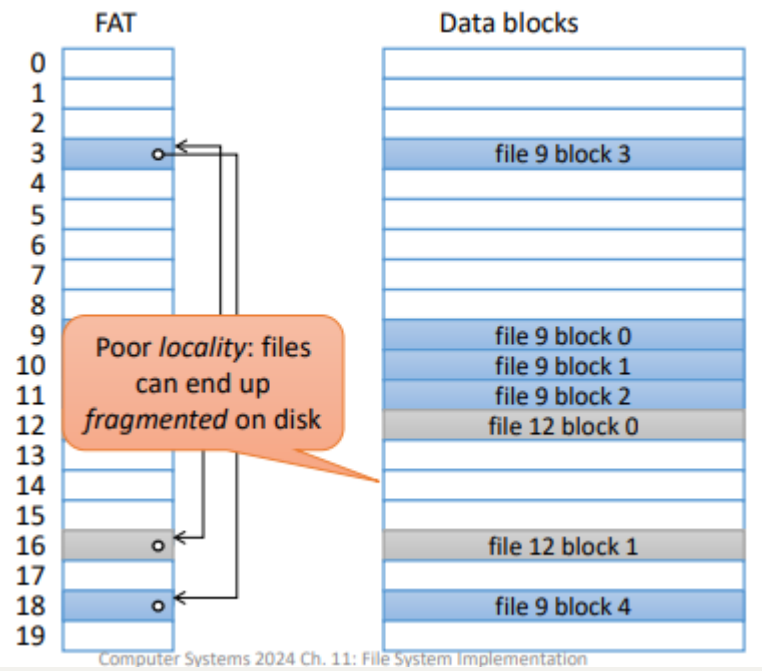




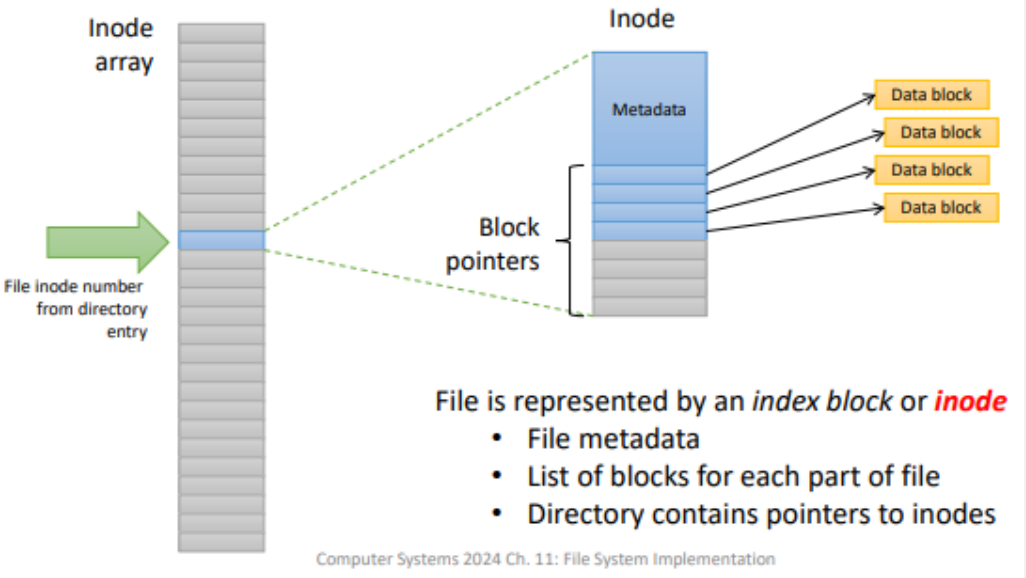
- Durability



- Performance



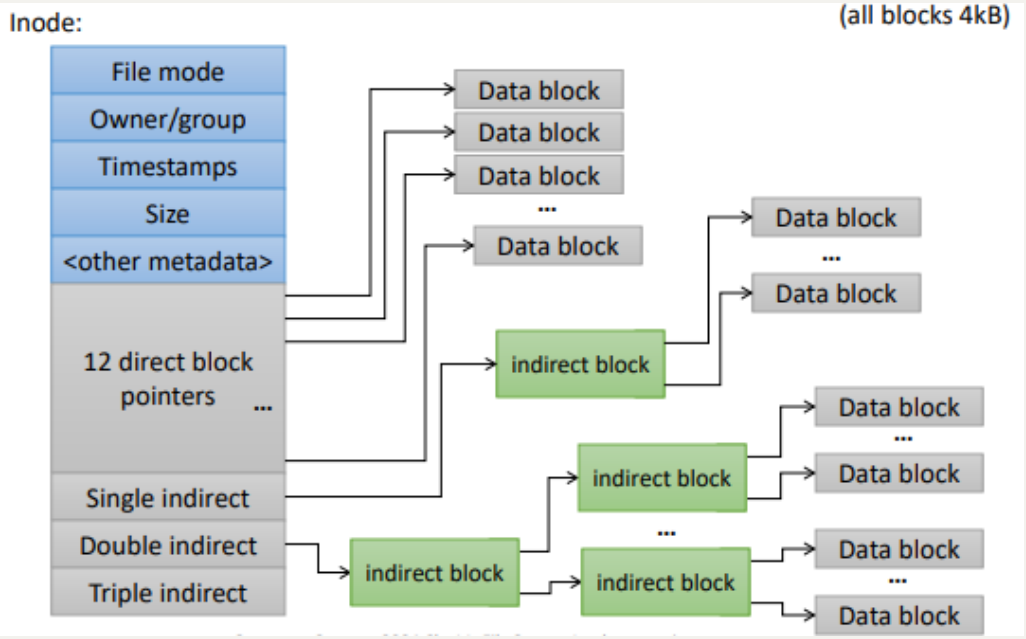
- FFS
  - Unix Fast File System (FFS)
    - First appeared in BSD in the mid 1980's
    - Based on original Unix FS, with performance optimizations
    - Basis for Linux ext{2, 3} file systems
  - FFS uses indexed allocation



- Inode structure in xv6: `kernel/fs.h`

```
#define ROOTINO  1   // root i-number
#define BSIZE 1024  // block size

#define NDIRECT 12

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};

// Inodes per block.
#define IPB           (BSIZE / sizeof(struct dinode))
```


- Inode and file size in FFS
    - Example:
        - inode is 1 block = 4096 B
        - block addresses = 8 B
        - inode metadata = 512 B
    - Hence
        - $\frac{4096-512}{8} = 448$ block pointers
        - $448 \times 4096 = 1792$ kB max. file size
- Unix file system inode format (simplified)

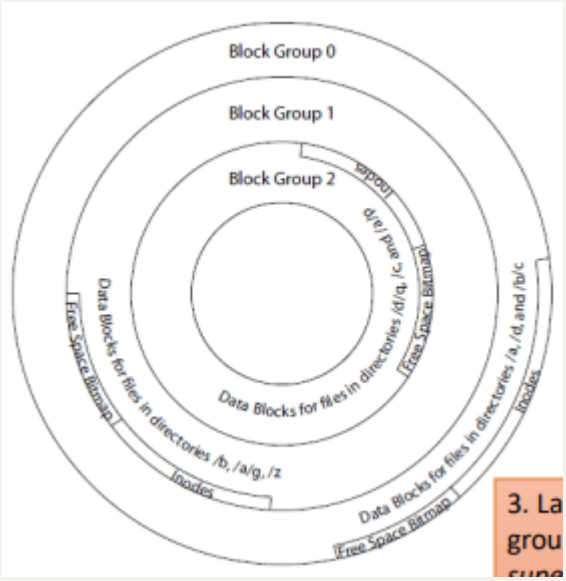


- Very small files: fit data straight into inode in place of pointers
- Very fast random access for files which fit in a single INode
- Very large files: tree makes keeps random access efficient

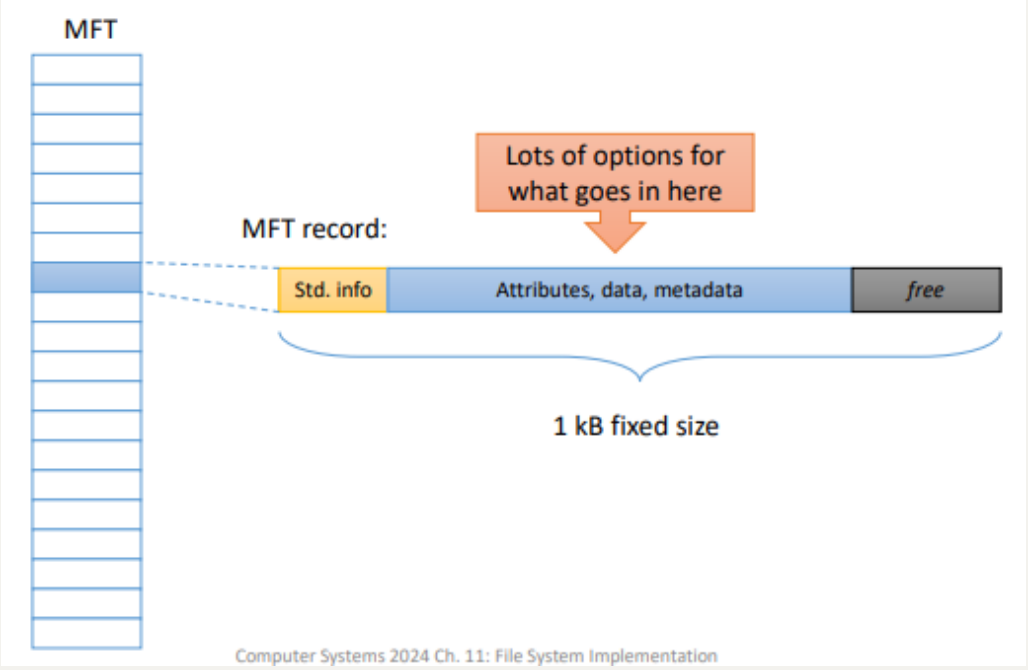
- Free space map
  - FFS uses a simple **bitmap**
    - Initialized when the file system is created
    - one bit per disk (file system) block
  - Allocation is reasonably fast
    - Scan through lots of bits at a time
    - Bitmap can be cached in memory
- The Superblock
  - Where is the inode table?
  - File system **superblock** contains
    - File system **parameters** (size, block size, no. of inodes)
    - Disk address of **free space bitmap**
    - Disk address of **inode map**
  - What if we lose the superblock?
    - Multiple copies spread over the disk at predictable locations
    - Superblock contents rarely change
- Superblock structure in xv6: `kernel/fs.h`

```
// Disk layout:
// [ boot block | super block | log | inode blocks | free bit map | data blocks]
//
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
struct superblock {
  uint magic;        // Must be FSMAGIC
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
#define FSMAGIC 0x10203040
```

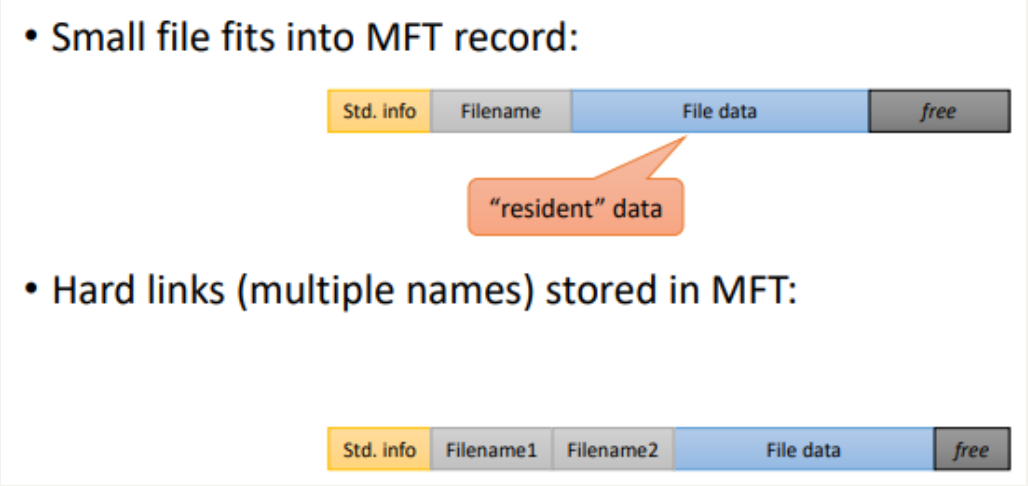

- Block groups
    - Optimize disk performance by keeping together related
        - files, metadata (inodes), free space map, directories
    - Use first-fit allocation within a block group to improve disk locality
    - Layout and block groups defined in the superblock (not shown); replicated several times



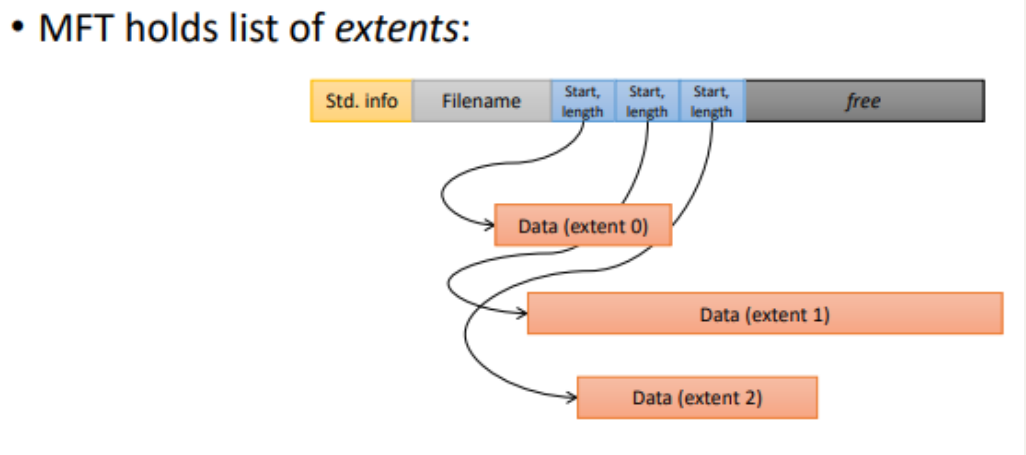
- NTFS
    - Master file table



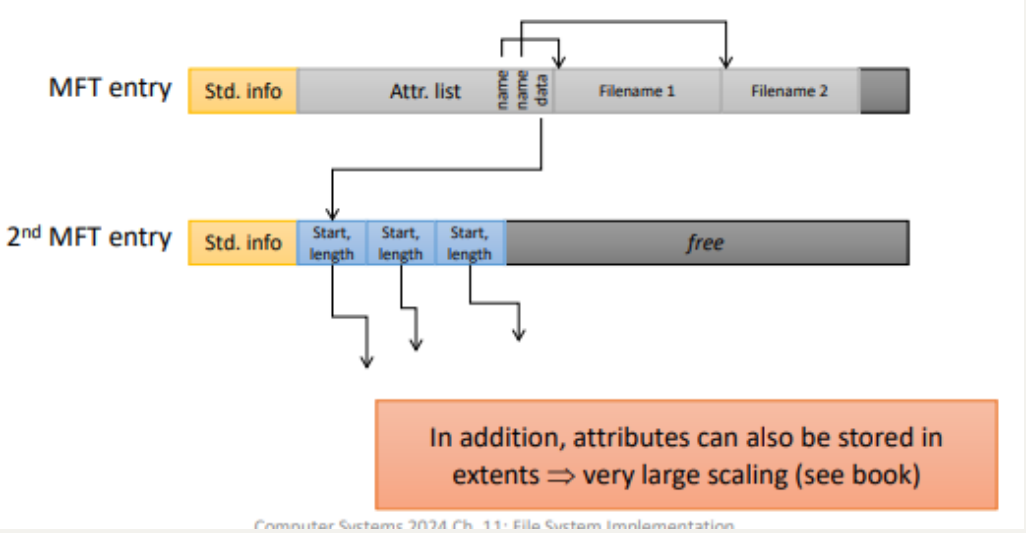
- Windows NTFS
    - All names for 1 file → attribute in the corresponding MFT entry
    - Small files → data is stored as attributes
    - Otherwise, data is stored in extents
    - 1st attribute is attribute list, acts as an index for all attributes
- NTFS small files



- NTFS normal files



- Too many attributes: attribute list holds list of attribute locations



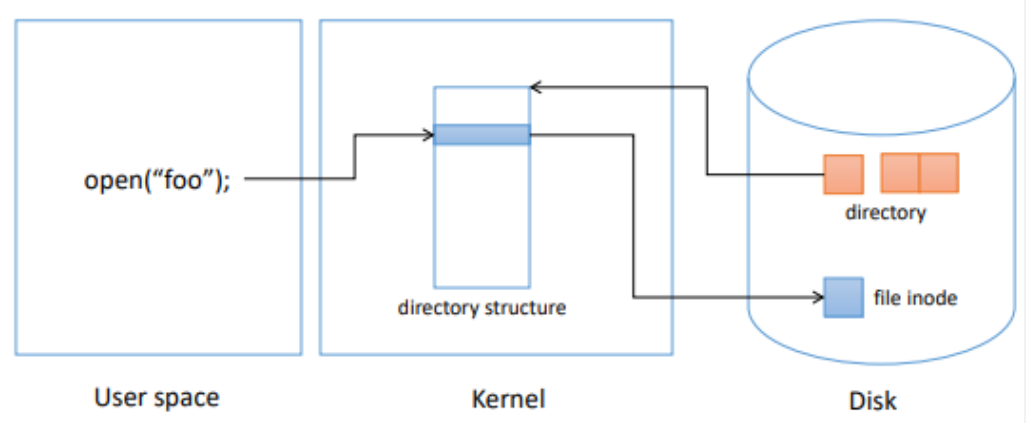
- Metadata files: file system metadata in NTFS is held **in files**

File num.	Name	Description
0	$MFT	Master file table
1	$MFTirr	Copy of first 4 MFT entries
2	$Logfile	Transaction log of FS changes
3	$Volume	Volume information & metadata
4	$AttrDef	Table mapping numeric IDs to attributes
5	.	Root directory
6	$Bitmap	Free space bitmap
7	$Boot	Volume boot record
8	$BadClus	Bad cluster map
9	$Secure	Access control list database
10	$UpCase	Filename mappings to DOS
11	$Extend	Extra file system attributes (e.g. quota)

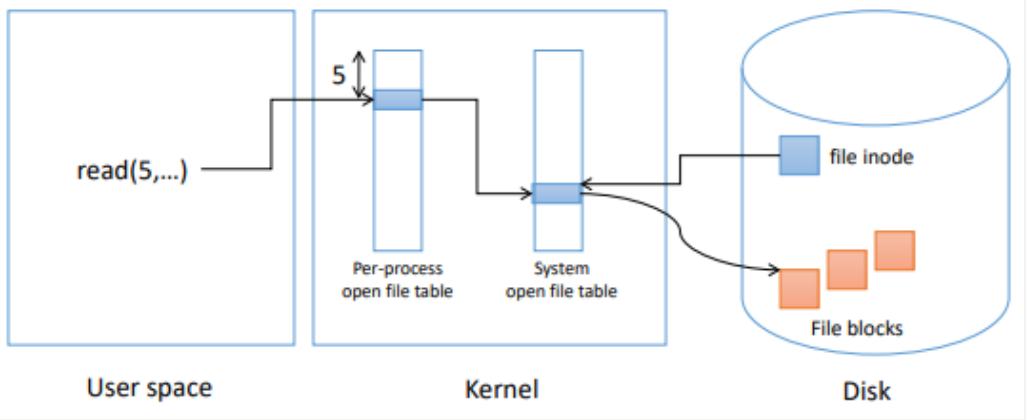
    - Master file table: First sector of volume points to first block of MFT

- In-memory data structures
    - Opening a file

- Directories translated into kernel data structures on demand



- Reading and writing
  - Per-process open file table → index into...
  - System open file table → cache of inodes



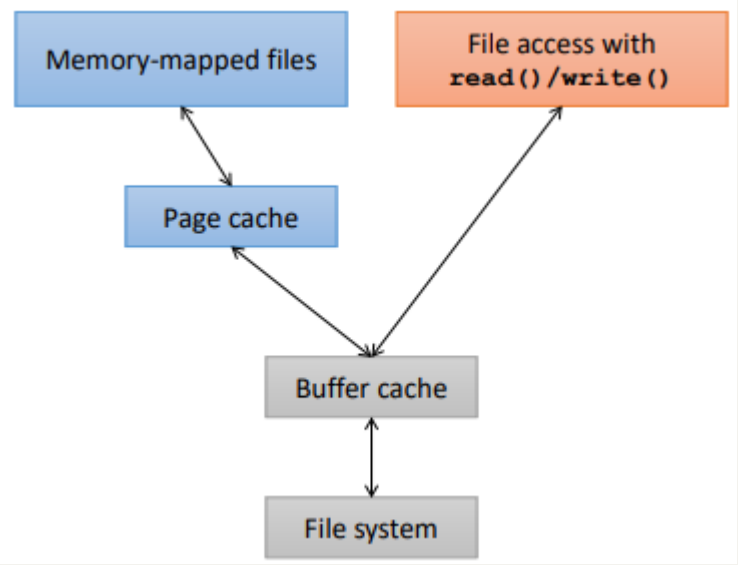
- In-memory inode in xv6: `kernel/file.h`

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

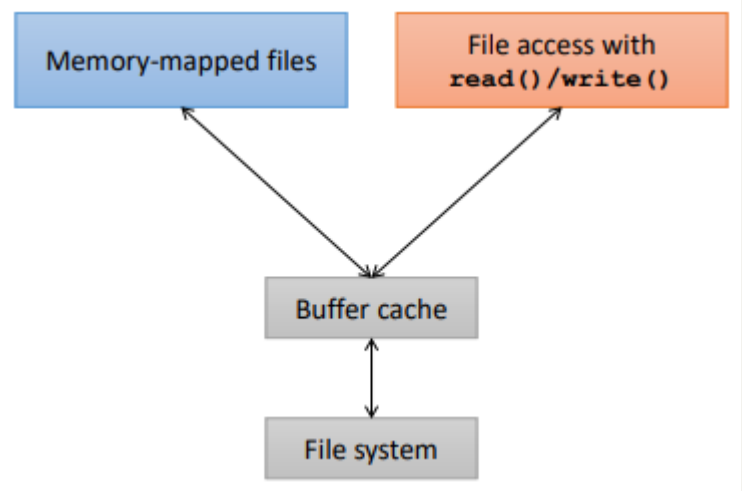

- Open file structure in xv6: `kernel/file.h`

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

- Efficiency and Performance
    - Efficiency dependent on
        - disk allocation and directory algorithms
        - types of data kept in file's directory entry
    - Performance
        - disk cache - separate section of main memory for frequently used blocks
        - free-behind and read-ahead - techniques to optimize sequential access
        - improve PC performance by dedicating section of memory as virtual disk, or RAM disk
- Page Cache
    - a page cache caches pages rather than disk blocks using virtual memory techniques
    - memory-mapped I/O uses a page cache
    - Routine I/O through the file system uses the buffer (disk) cache
    - This leads to the following figure



Unified Buffer Cache

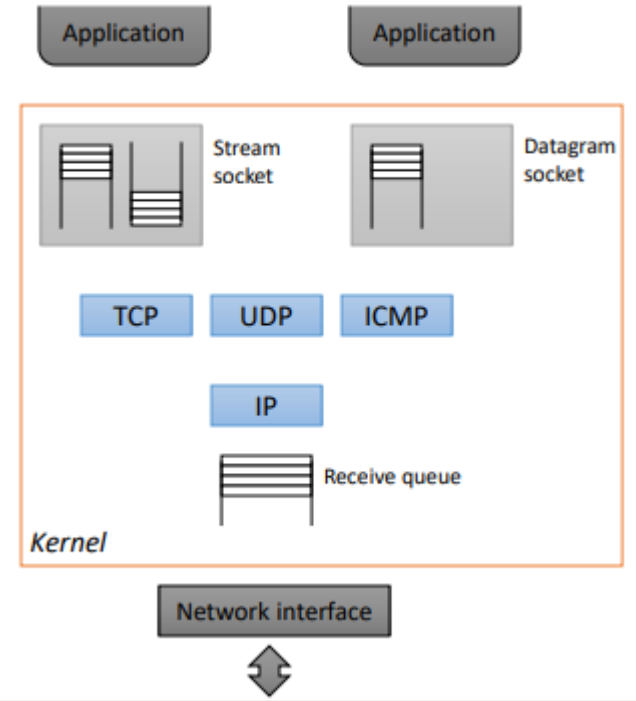


- Recovery
    - Consistency checking - compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
    - Use system programs to **back up** data from disk to another storage device
    - Recover lost file or disk by **restoring** data from backup

Oct. 28th - Lecture 12: Network Stack

- Unix interface to network I/O
    - Data path
        - BSD sockets
        - `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`, etc
    - Part of the **kernel network stack**

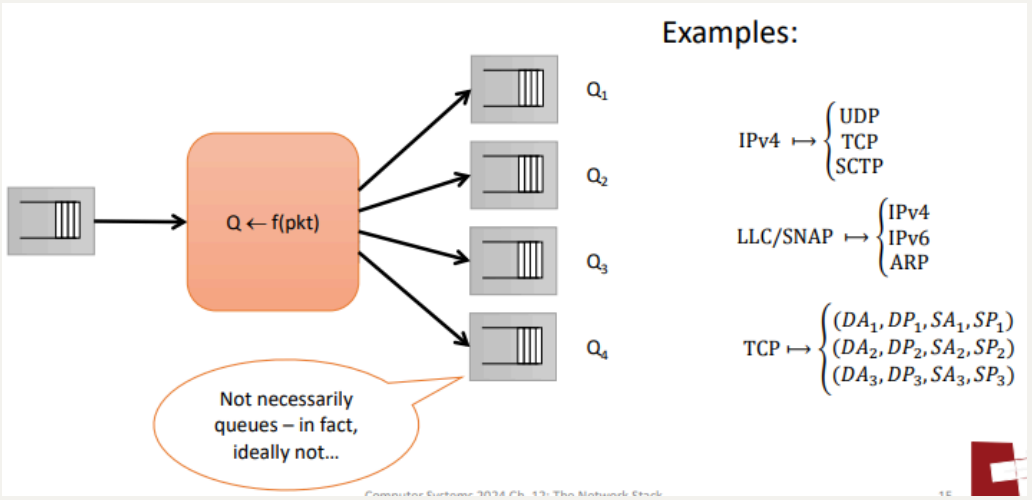
- The Network Stack functionality
    - Packet send/receive
    - Multiplexing/demultiplexing
    - Protocol processing
    - Buffer management
    - Forwarding
    - Routing
    - The most important peripheral?
        - Disks are now on networks
- The Network Stack components
    - Network Interface Adaptors (NICs)
        - Ethernet, 4G/5G adaptors, etc
    - NIC driver bottom half
        - First-level interrupt handler
        - Deferred procedure calls
    - NIC driver top half
        - Kernel threads, abstraction functions
    - Core network stack
        - Protocol processing, user-space copyin/out
    - Application libraries, daemons, utils
- Basic packet transmit and receive
    - Receiving a packet in BSD



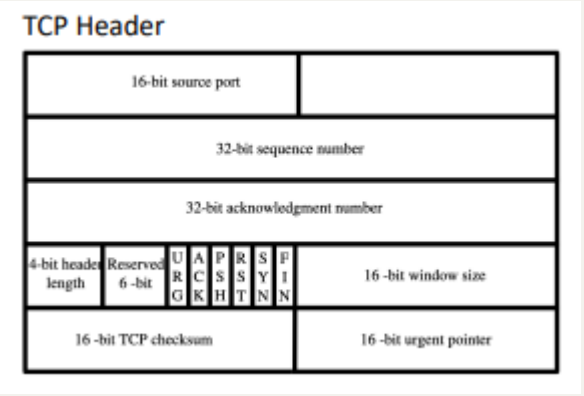
- Interrupt
    - Allocate buffer, enqueue packet, post s/w interrupt
    - Accounting system needed to keep track of metadata for the packets
- S/W interrupt
    - High priority, a process context, defragmentation, TCP processing, enqueue on socket
- Application
    - Copy buffer to user space, Application process context

- Sending a packet in BSD
    - Application
        - Copy from user space to buffer, call TCP code and process, possible enqueue on socket queue
    - S/W interrupt
        - Any process context, remaining TCP processing, IP processing, enqueue on i/f queue
    - Interrupt
        - Send packet, free buffer

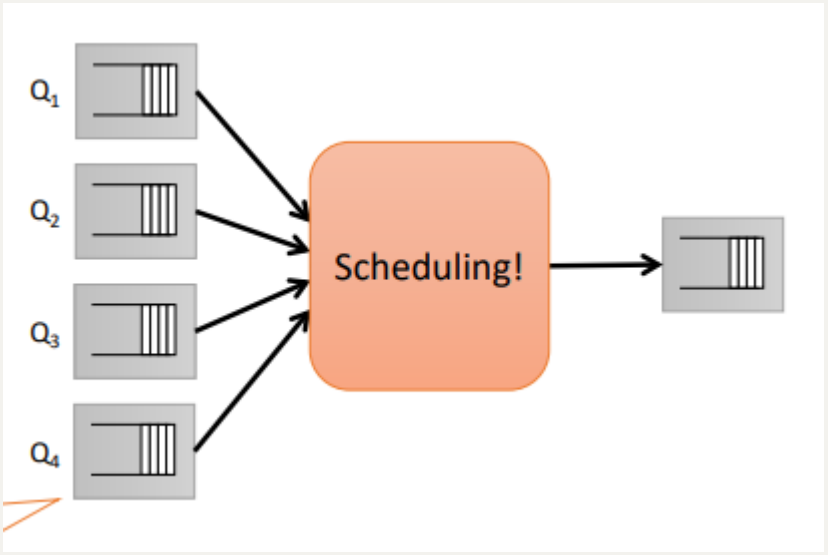
- Multiplexing and demultiplexing
    - Demultiplexing



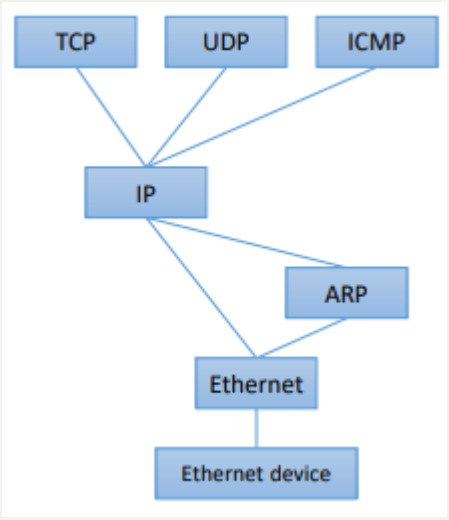
- What's the function f?
    - f : `pkt` $\mapsto$ (`pkt', module`)
    - Function f looks at the packet **headers**
    - **Decapsulates** the packet to "inner" packet
    - **Demultiplexes** to a processing module (Based on packet location in **header space**)
- Header space



    - Set of all possible headers of a packet
    - **Vector space**
        - a packet is a point in this space
        - a dimension is set of values of one header
    - Abstractly
        - Receive: **map** packet's location in header space to a socket
        - Transmit: **calculate** packet's location in header space and set headers accordingly
- Multiplexing

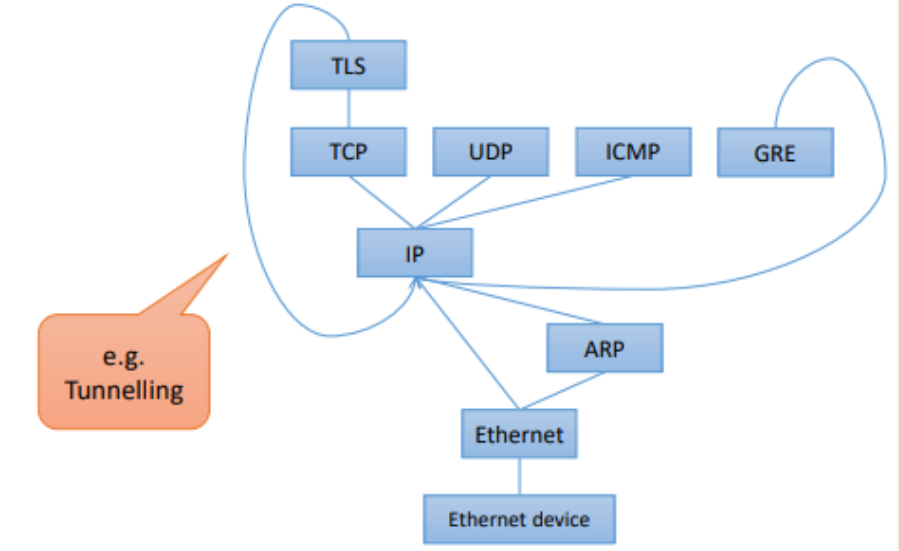


- In-kernel protocol graph

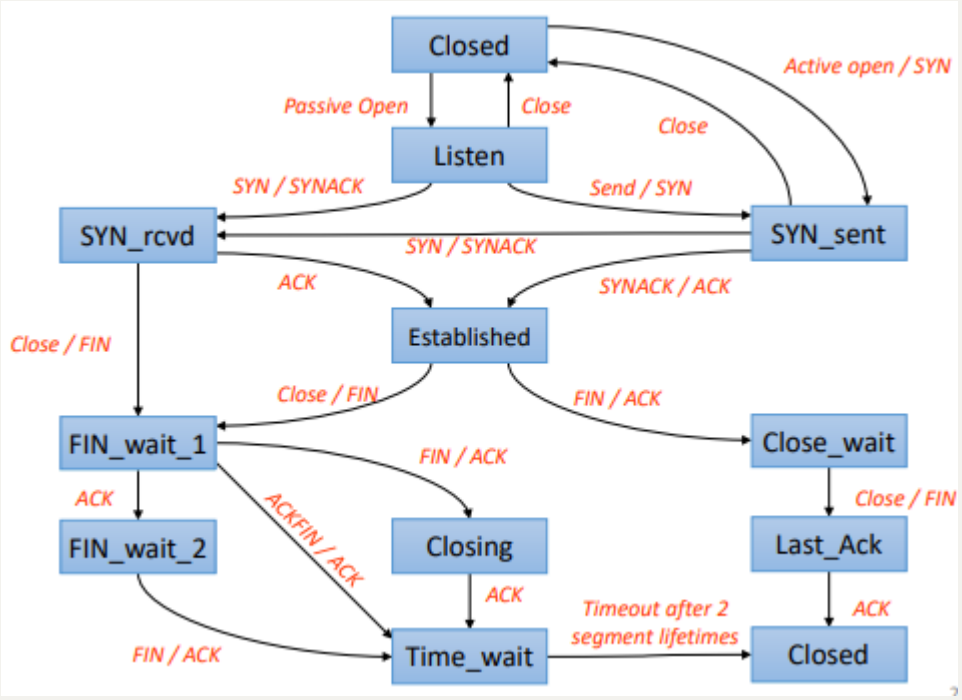


- Example Nodes
    - Demultiplexing
        - Packets based on whether the protocol field is IP
        - UDP packets based on destination IP address and port
        - TCP packets based on four-tuple
    - Processing a single, bidirectional TCP connection
    - Responding to ICMP echo requests
    - Routing IP packets between interfaces
    - Encapsulating messages sent via a UDP socket with the correct UDP header
- In-kernel protocol graph
    - Graph nodes can be

        - Per-protocol (handle all flows): packets are "tagged" with demux tags
        - Per-connection (instantiated dynamically)
    - No need for the implementation to based on this graph
        - several equivalent graphs are possible
    - Optimize for
        - performance isolation
        - CPU load
    - Interfaces can be standard or protocol-specific
    - Can be cyclic



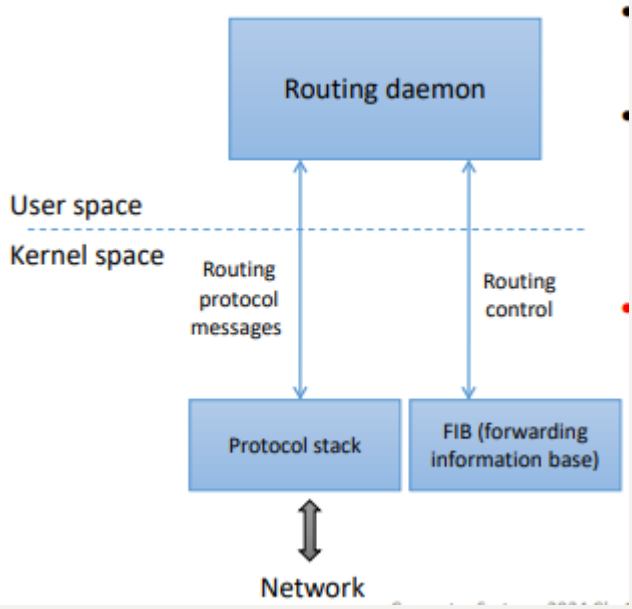
- Protocol processing
    - TCP state machine



    - Needs to handle
        - congestion control state (window, slow start, etc)
        - Flow control window
        - Retransmission timeouts
    - State transitions triggered when
        - User request: send, recv, connect, close
        - Packet arrives
        - Timer expires
    - Actions include
        - set or cancel a timer
        - enqueue a packet on the transmit queue
        - enqueue a packet on the socket receive queue
        - create or destroy a **TCP control block**
- Forwarding

```
Shell
cixous: /home/troscoe> route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         0.0.0.0         0.0.0.0         U     50     0        0 vpn0
default         rtac68u.mothy.o 0.0.0.0         UG    100    0        0 eno1
default         rtac68u.mothy.o 0.0.0.0         UG    600    0        0 wlp3s0
10.6.192.0      0.0.0.0         255.255.224.0   U     50     0        0 vpn0
sslvpn.ethz.ch  rtac68u.mothy.o 255.255.255.255 UGH   100    0        0 eno1
link-local      0.0.0.0         255.255.0.0     U     1000   0        0 wlp3s0
192.168.2.0     0.0.0.0         255.255.255.0   U     100    0        0 eno1
192.168.2.0     0.0.0.0         255.255.255.0   U     600    0        0 wlp3s0
rtac68u.mothy.o 0.0.0.0         255.255.255.255 UH    100    0        0 eno1
cixous: /home/troscoe>
```

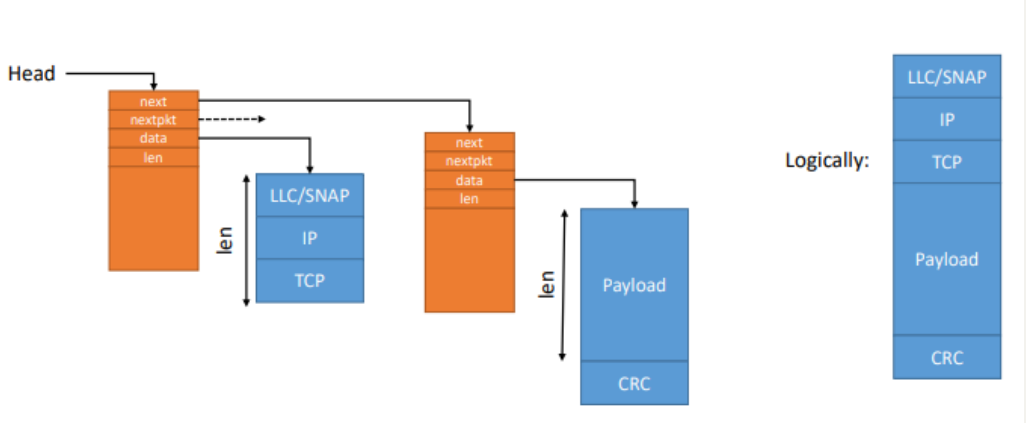
- Routing



        - OS protocol stacks include routing functionality
        - Routing **protocols** typically in a user-space daemon
            - non-critical
            - easier to change
        - **Forwarding** information typically in kernel
            - needs to be fast
            - integrated into protocol stack
- Memory management and buffering
    - Memory management
        - Problem: how to ship packet data around
        - Need a data structure that can
            - easily add, remove headers
            - avoid copying lots of payload
            - uniformly refer to **half-defined packets**
            - **fragment** large datasets into smaller units
        - Solution: data is held in a linked list of "buffer structures"
    - Simplified FreeBSD `mbuf` structure (compare with Linux `sk_buff`)

```
struct mbuf {
  struct mbuf *m_next;      /* next mbuf in chain */
  struct mbuf *m_nextpkt;   /* next packet in list */
  char        *m_data;      /* location of data */
  int32_t      m_len;       /* amount of data in this mbuf */
  uint32_t     m_type:8,    /* type of data in this mbuf */
               m_flags:24;  /* flags; see below */
  char         m_dat[0];
};
```

    - Use of `mbufs` - transmit



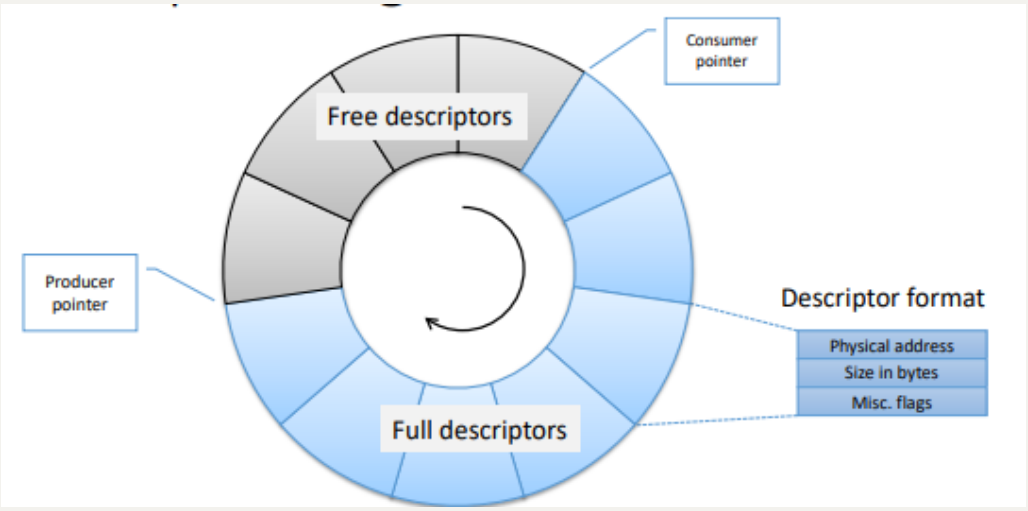
- Getting good performance

- What is performance?
    - Latency: wire ↔ application
    - Jitter: **variation** in latency
    - Throughput: bytes per second, packets per second
    - CPU load: how many **cycles per packet/byte**?
    - Quality of service
        - latency / throughput guarantees to particular applications
        - performance isolation between applications
- Life cycle of an I/O request: unfortunately, this is a bit of a convenient fiction
    - when there is so much data coming in
- Simply impossible to keep up with the network in software when considering 400 Gb/s Ethernet

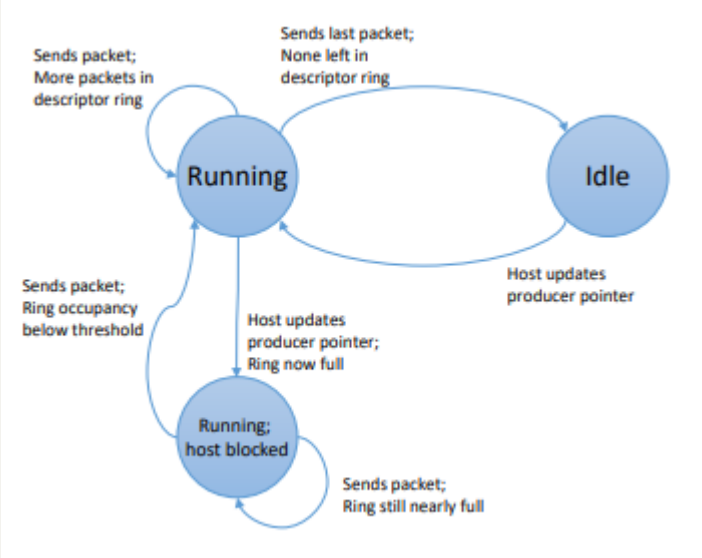
	Hardware platform			
	Automotive SoC	PC	Server	High-end
Line rate (Gb/s)	1	10	40	400
Average pkt size (bytes)	500	500	500	1500
Line rate (packets/second)	200000	2000000	8000000	26666667
Clock frequency (GHz)	1	3	3	3
Available processing (cycles/packet)	5000	1500	375	112.5
Cache miss penalty (cycles)	300	200	100	100
Interrupt overhead (cycles)	100	200	100	100
checksum in assembler (cycles/kB data)	2000	1000	500	500
Data cache misses	8	8	8	24
Per-packet processing overhead (cycles)	1048.56	560.28	316.14	948.42
Cycles available to application	3951.44	939.72	58.86	-835.92
Overhead (percentage)	21%	37%	84%	843%

- Asynchronous buffering
    - Direct Memory Access
        - Avoid programmed IO (PIO) for lots of data
        - Requires DMA controller (generally built-in these days)
        - Bypasses CPU to transfer data directly between I/O device and memory
            - Doesn't take up CPU time
            - Can save memory bandwidth
            - Only one interrupt per transfer
    - I/O protection

    - etc.
- Buffering
    - etc.
- Producer-consumer buffer descriptor rings

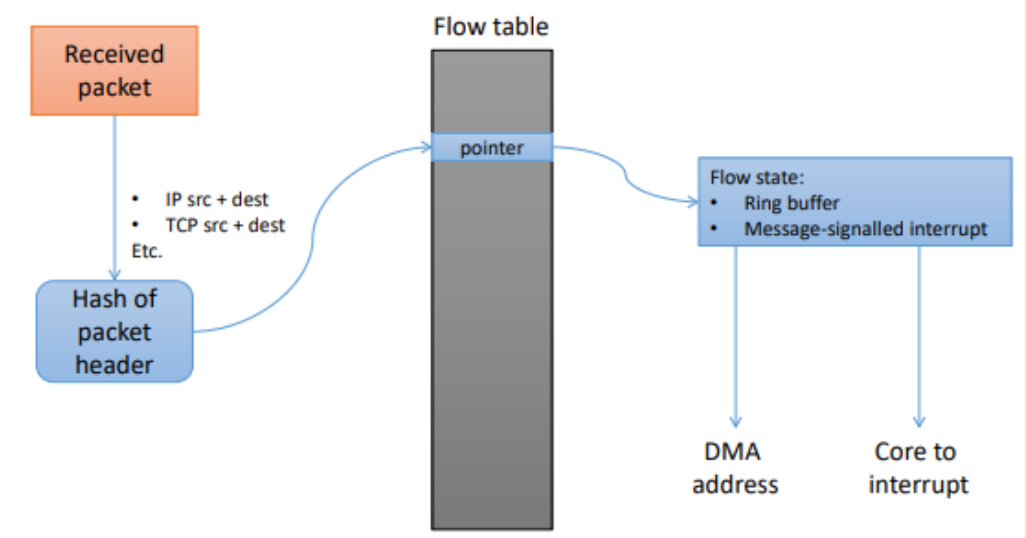


- Buffering for network cards
    - Producer, consumer pointers are NIC registers
    - Transmit path
        - Host updates producer pointer, adds packets to ring
        - Device updates consumer pointer
    - Receive path
        - Host updates consumer pointer, adds empty buffers to ring
        - Device updates producer pointer, fills buffers with received packets
- Example transmit state machine



        - Transmit interrupts
            - Ring empty
                - all packets sent
                - device going idle
            - Ring occupancy drops
                - host can now send again
                - device continues running
        - Buffering summary
            - DMA used twice
                - data transfer, reading and writing descriptors
            - Similar schemes used for any fast DMA device
            - Descriptors send *ownership* of memory regions
            - Flexible - many variations possible
                - host can send lots of regions in advance
                - device might allocate out of regions, send back subsets
                - buffers might be used out-of-order
- Further hardware acceleration
    - What to do?
        - Buffering: transfer lots of packets in a single transaction
        - TCP offload: put TCP processing into hardware on the card
        - Interrupt coalescing / throttling
            - don't interrupt on every packet
            - don't interrupt at all if load is very high
        - more queues: receive-side scaling
            - parallelize: direct interrupts and data to different cores
    - Receive-side scaling
        - Insight
            - too much traffic or one core to handle
            - cores aren't getting any faster, must parallelize across cores
        - Key idea: handle different flows on different cores

        - how to determine flow for each packet?
        - cannot do this on a core
    - Solution: demultiplex on the NIC
        - DMA packets to per-flow buffers / queues
        - send interrupt only to core handling flow



    - Same flow hashed to the same core
- Tip of the iceberg
    - RSS can sometimes balance flows across cores
        - flow steering is more precise
        - neither help with one big flow
    - Transmit scaling requires packet scheduling
    - Multiprocessor scaling
        - protocol processing in software should scale
        - no shared state on the fast path
    - Bottlenecks keep changing
        - connection setup
        - virtual machines

Oct. 30th - Exercise Session 6

- Consider a physical address (segment, address), given (segment, base, length) as logical address
    - We use base + address to get logical address, but address $\leq$ length is required.

- Format of logical address
    - page size → offset bits
    - number of pages → page number bits
    - length/width of the page table (disregarding the access right bits) →
        - number of pages → page table length
        - physical memory space size → number of bits to represent a physical address
        - physical page number (number of bits) = physical address (number of bits) - number of offset bits
- Time of paged memory reference
    - without TLB: time of access page table in memory + time of access the word in memory
    - add TLB, 75% of all page-table references are found in the TLB: average time
        - TLB access + 0.25× (page table access + access the word in memory) + 0.75× access the word in memory
        - 0.25: TLB miss
        - 0.75: in TLB
- File Systems Abstraction
    - Application + File system API & Implementation + I/O system
    - Operations
        - Directory (name space) operations
            - Link (bind a name) → `ln (-s)` existing_filename new_link
            - soft link: path to location; hard link: new name for the same file
            - Unlink (unbind a name) → `unlink/rm` filename
            - Rename → `mv` old_filename new_filename
            - List entries → `ls` directory
    - "Files" vs. "Open files"
        - Typical operation on files: rename, stat, create, delete, **open**
        - **open**: create on-disk data structure, an open file
        - Open files

- `open(filename)` returns open file descriptor (small integer in UNIX)
- open file descriptor identifies open file
- open file provides context

- File metadata
  - data about an object, not the object itself
- Protection
  - (principal, object, operation) → {True, False}
  - types of access: read, write, execute, append, delete, list
- Access control matrix
  - Row-wise: encoding the set of principals that have any rights over the object, together with what those rights are; store with file
    - advantages
      - scales well with many files
      - easily modify access rights of a file
    - disadvantages
      - does not scale to large numbers of principals
  - Column-wise: capabilities, reverse (dis)advantage from row-wise
- Access control in OS
  - Linux: rwx for 3 principals (owner, group, everyone) (9 bits of extra metadata per file)
    - Full ACL implementation exists
  - Windows
    - ACL for all kinds of objects, not just files
    - Supports groups of arbitrary principals. Each group is a principal

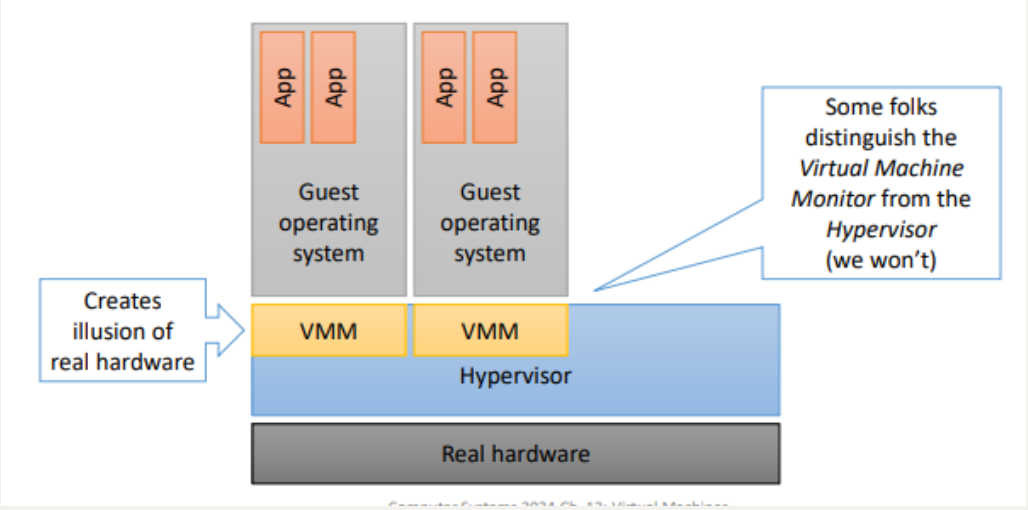
- File Systems Implementation
  - Implementation aspects
    - directories and indices
    - index granularity
    - free space maps
    - locality optimizations

- FAT
- FFS
    - File is represented by an index block or **inode**
        - file metadata
        - list of blocks for each part of file
        - directory contains pointers to inodes
    - Block groups
- NTFS
    - Small file fits into MFT record
    - MFT holds list of extents
        - An extent is a contiguous, variable-sized region of a storage volume, identified by its start LBA and length in blocks
    - Attribute list holds list of attribute locations
    - Metadata files
        - File system metadata in NTFS is held in **files**
        - First sector of NTFS volume points to first block of $MFT
        - Important files: MFT, Logfile, Bitmap, Secure

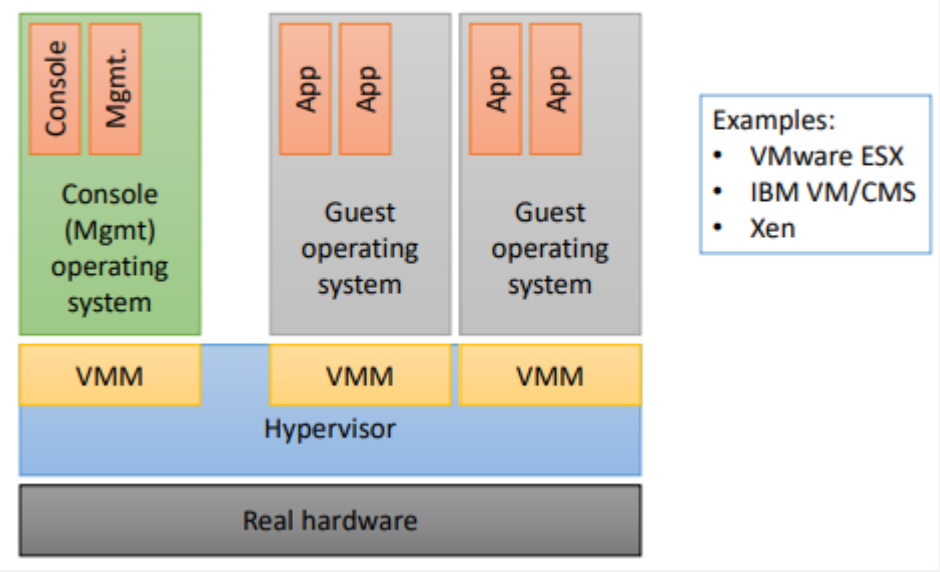
- Quiz
    - FAT still widely used: compatibility and ease of implementation (embedded services)
    - Largest file size with
        - only direct block pointers: 12 * 4KB = 48 KB
        - only direct and single indirect block pointers: 48 KB + 512 * 4 KB
        - only direct and single and double indirect block pointers: 48 KB + 512 * 4 KB + 512^2 * 4 KB
    - Locality matter in a file system because
        - sequential read way faster than random read

Nov. 1st - Lecture 13: Virtual Machines

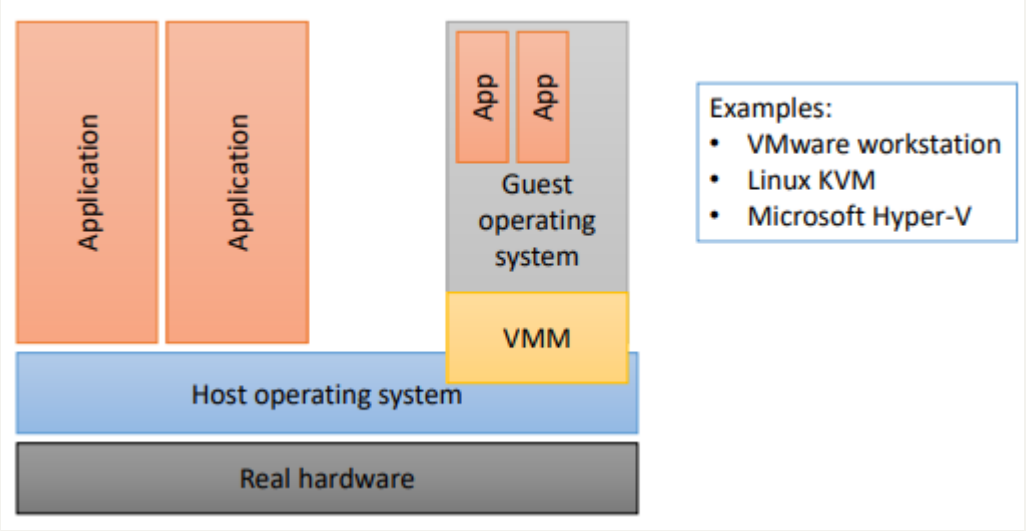
- Virtualizing so far
    - Virtualizing the address space
        - Concept: virtual memory and the virtual address space
        - Mechanism
            - virtual address translation
            - The MMU and TLB hardware
            - Page tables
    - Virtualizing physical RAM
        - Concept: demand paging
        - Mechanism
            - MMU and TLB
            - Invalid / not-present page table entries
            - Page fault exceptions
            - I/O and slower, larger storage
    - Virtualizing persistent storage
        - Concept: files
        - Mechanism
            - file system
            - almost entirely (kernel) software
    - Virtualizing the network
        - Concept: connections, addresses
        - Mechanism
            - sockets
            - protocol layering
            - the protocol stack implementation
    - Virtualizing the user-space CPU ISA
        - Concept: processes / threads
        - Mechanism
            - Processor exceptions and the mode switch
            - Context switching, save / load registers

        - Scheduling
    - Still have more machine parts to virtualize
        - Rest of the CPU, including **kernel mode**
        - The MMU, physical RAM size, devices, the Network Stack
        - Objective: persuade other OSes that they have a real machine to themselves
- Virtual Machine Monitors and Hypervisors
    - 

    - Virtual Machine Monitor
        - Virtualizes an entire (hardware) machine
            - Contrast with OS processes
            - Interface provided is "illusion of real hardware"
            - Applications are therefore complete OS themselves
            - Terminology: **Guest OS**
        - Old idea: IBM VM/CMS (1960s)
    - How does it all work?
        - a hypervisor is basically an OS (with an "unusual API")
        - Basically same functions
            - Multiplexing resources
            - Scheduling, virtual memory, device drivers
        - Differences
            - Creating the illusion of hardware to "applications"
            - Guest OSes are less flexible in resource requirements
    - Type-1 (native) hypervisors



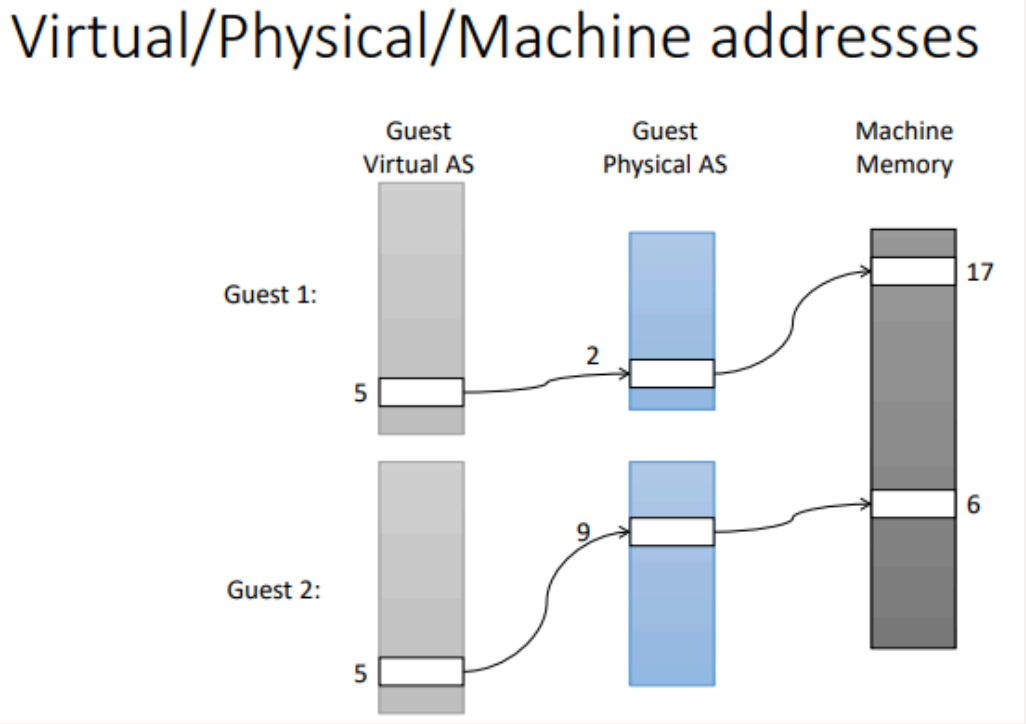
- Type-2 (hosted) hypervisors



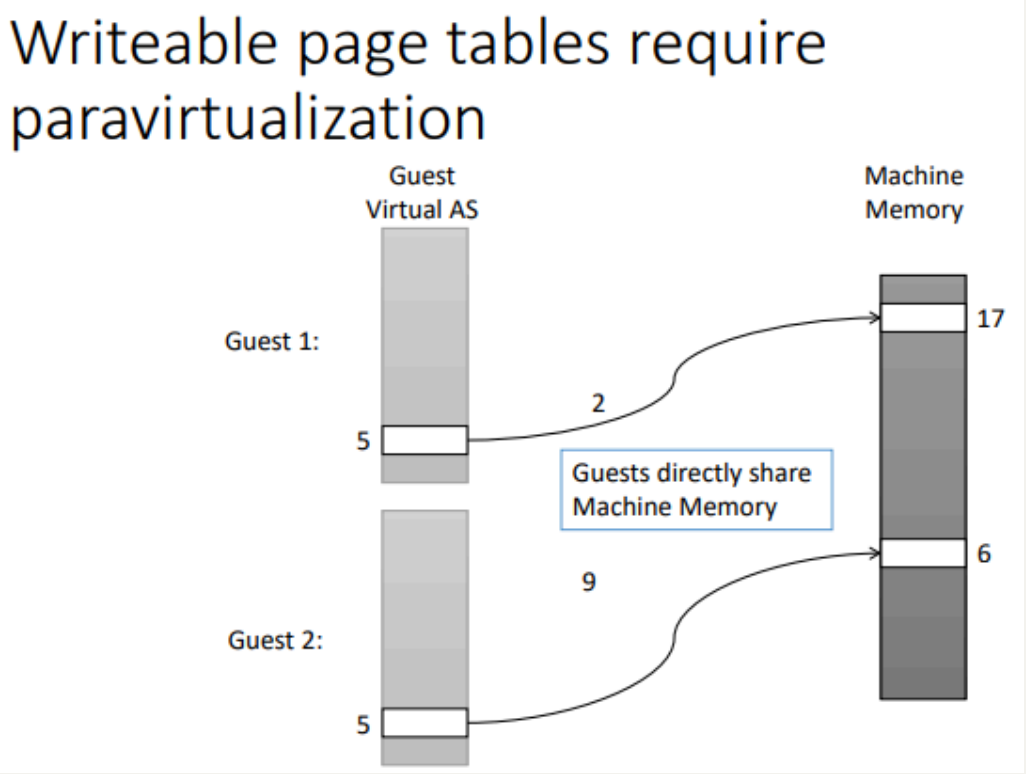
- Why virtual machines?
    - Recall running xv6
        - 2 levels of virtual machine monitor
        - Hyper V virtualizes Linux via WSL
        - Linus virtualizes xv6 by hardware simulation
    - Running multiple OSes on one machine
        - Application compatibility
            - I use Ubuntu for almost everything, but I edit slides in PowerPoint
            - Some people used to compile Barrelfish in a Debian VM over Windows 7 with Hyper-V
        - Backward compatibility
            - Windows 98 virtual machine for playing old computer games

        - OS development
            - Building and testing a new OS without needing to reboot real hardware
            - VMM often gives you more information about faults than real hardware anyway
        - Server consolidation
            - Many applications assume they have the machine to themselves
            - Each machine is mostly idle
            - consolidate servers onto a single physical machine
        - Resource Isolation
            - Surprisingly, modern OSes do not have an abstraction for a single application
            - Performance isolation can be critical in some enterprises
            - Use virtual machines as **resource containers**
        - Cloud computing
            - Selling computing capacity on demand
            - Hypervisors decouple **allocation** of resources (VMs) from provisioning of infrastructure (physical machines)
        - Other cool applications
            - Tracing, debugging, execution replay, lock-step execution, live migration, rollback, speculation
- Virtualizing the CPU
    - We have virtualized the CPU in user-mode (processes etc)
    - Virtualize CPU in kernel-mode?
        - all kernel-mode functionality should appear to work
        - kernel address space
        - memory protection
        - special registers
        - privileged CPU instructions
    - A CPU architecture is **strictly virtualizable** if it can be perfectly emulated over itself, with all non-privileged instructions executed natively
    - Privileged instructions $\rightarrow$ trap

        - kernel-mode emulates instruction
        - guest's kernel mode is actually user mode (or, extra privilege level)
        - Example: IBM S/390, Alpha, PowerPC
- A strictly virtualizable core can execute a complete native Guest OS
    - guest applications run in user mode as before
    - guest kernel works exactly as before
- Problem: x86 is not strictly virtualizable
    - about 20 instructions are sensitive but not privileged
    - mostly segment loads and processor flag manipulation
- Non-virtualizable x86: example
    - `pushf, popf` instructions
        - push/pop condition code register
        - includes interrupt enable flag (`IF`)
    - Unprivileged instructions: fine in user space
        - `IF` is ignored by `popf` in user mode, not in kernel mode
    - VMM cannot determine if Guest OS wants interrupts disabled
        - cannot cause a trap on a privileged `popf`
        - prevents correct functioning of the guest OS
- Solutions
    - Emulation - emulate all kernel-mode code in software
        - very slow - particularly for I/O intensive workloads
    - Paravirtualization - modify Guest OS kernel
        - replace with explicit trap instruction to VMM
        - also called a "HyperCall"
    - Binary rewriting
        - protect kernel instruction pages, trap to VMM on first IFetch
        - Scan page for `popf` instructions and **replace with other code**
        - Restart instruction in Guest OS and continue

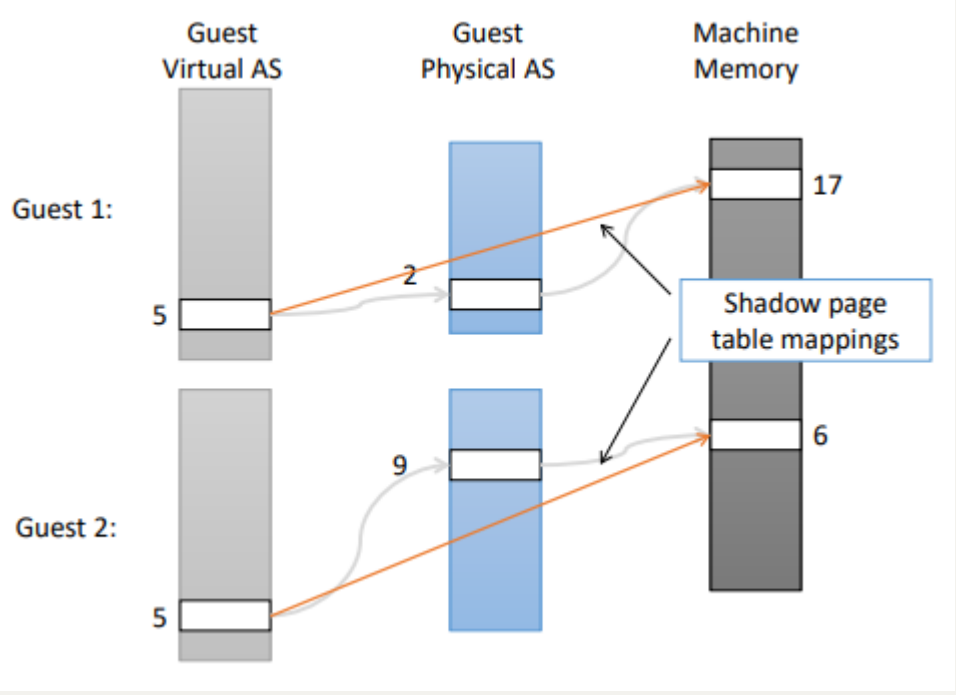
        - Hardware support - Intel VT-x, AMD-V
            - extra processor mode causes `popf` to trap
            - Hypercall instruction traps from Guest kernel to hypervisor
- Virtualizing the MMU
    - Kernel uses MMU to virtualize addresses to user mode
        - hypervisor can virtualize the physical address space to the guest kernel
    - How can the MMU itself be virtualized to the guest kernel?
        - Page tables constructed by the guest kernel
        - Page faults taken in guest user space
        - TLB and cache maintenance by the guest kernel
    - Hypervisor allocates memory to VMs
        - guest assumes control over all physical memory
        - VMM cannot let Guest OS to install mappings
    - Definitions
        - Virtual address: a virtual address in the guest
        - Physical address: as seen by the guest
        - Machine address: real physical address (as seen by hypervisor)
    - 

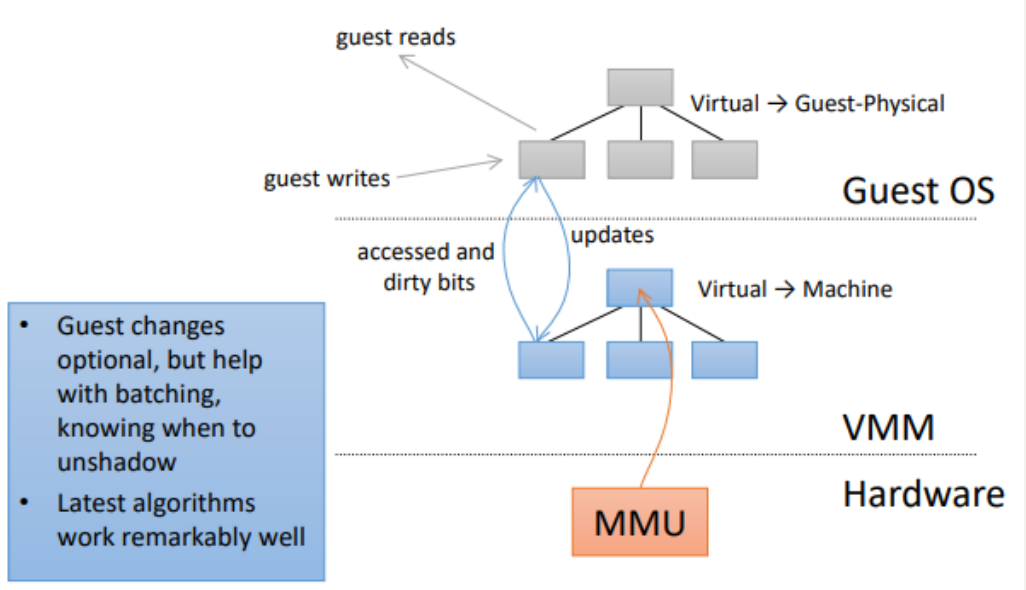
    - MMU virtualization

- Critical for performance, challenging to make fast, especially SMP
    - hot-unplug unnecessary virtual CPUs
    - use multicast TLB flush paravirtualizations etc
- Xen supports 3 MMU virtualization modes
    - direct ("writable") pagetables
    - shadow pagetables
    - hardware assisted paging
- OS paravirtualization compulsory for #1, optional (and very beneficial) for #2 & 3
- 

- Paravirtualization approach
    - Guest OS creates page tables the hardware uses
        - VMM must validate all updates to page tables
        - Requires modifications to Guest OS
    - VMM must check all writes to PTEs
        - write-protect all PTEs to the Guest kernel
        - Add a HyperCall to update PTEs
        - Batch updates to avoid trap overhead
        - OS is now aware of machine addresses
        - Significant overhead

- Paravirtualizing the MMU
    - Guest OSes allocate and manage their own PTs
        - Hypercall to change PT base
    - VMM must **validate** PT updates before use
        - allows incremental updates, avoids revalidation
    - Validation rules applied to each PTE
        - Guest may only map pages it owns
        - Pagetable pages may only be mapped RO
    - VMM traps PTE updates and emulates, or "unhooks" PTE page for bulk updates
- Shadow Page Tables
    - Guest OS sets up its own page tables
        - not used by the hardware
    - VMM maintains shadow page tables
        - Map directly from Guest VAs to Machine Addresses
        - Hardware switched whenever Guest reloads PTPR
    - VMM must keep $V \rightarrow M$ table consistent with Guest $V \rightarrow P$ table and its own $P \rightarrow M$ table
        - VMM write-protects all guest page tables
        - Write $\rightarrow$ trap: apply write to shadow table as well
        - significant overhead

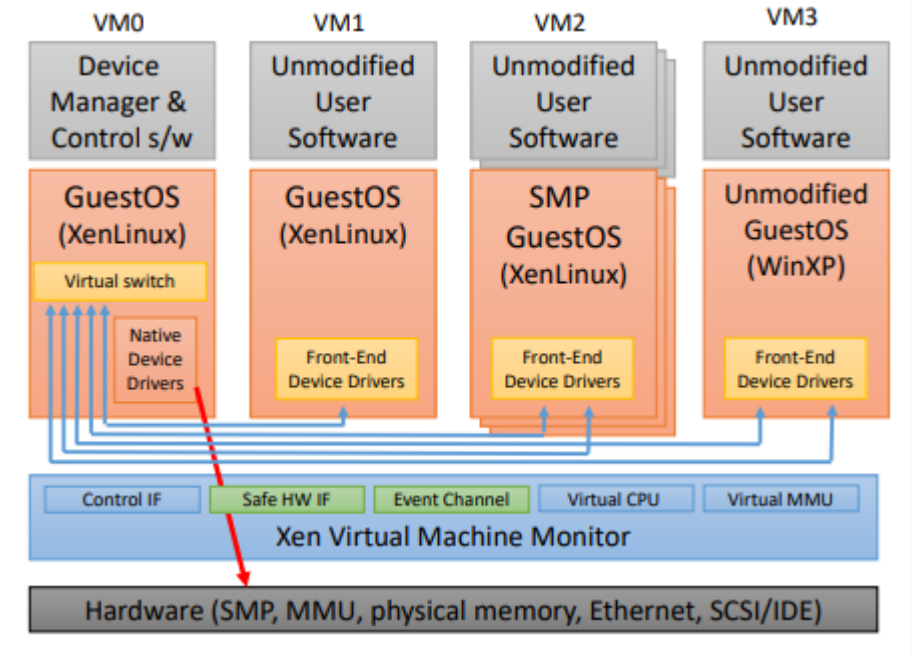




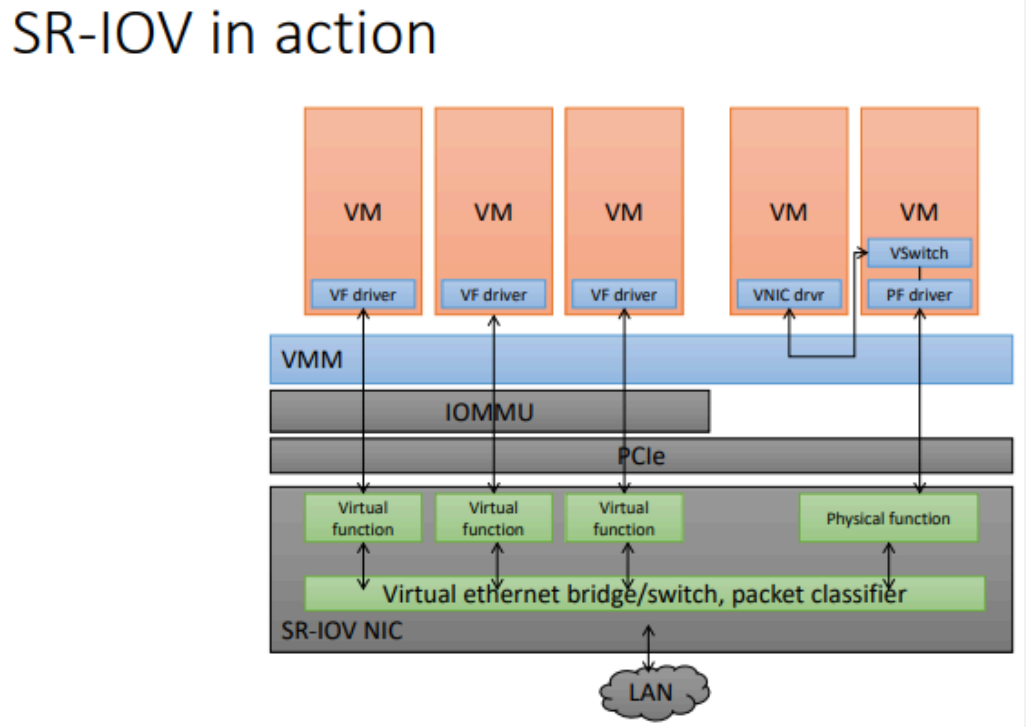
        - write fault, emulate, unhook, first use, re-hook
    - Hardware support
        - "Nested page tables": now in AMD (NPT) and Intel (EPT) hardware
        - Two-level translation of addresses in the MMU
            - hardware knows about: $V \rightarrow P$ tables (in the guest), $P \rightarrow M$ tables (in the Hypervisor)
            - Tagged TLBs to avoid expensive flush on a VM entry/exit
        - Very nice and easy to code to
        - Significant performance overhead (a lot of page walking, TLB miss double the time)
- Virtualizing physical memory, redux
    - Virtualizing guest physical memory
        - with MMU virtualized, the Guest OS can do demand paging
            - Guest OS will assume it has a fixed region of physical RAM
        - How can the Hypervisor
            - overcommit machine memory to virtual machines
            - dynamically change the machine memory a VM has
        - natural solution: Hypervisor does its own demand paging
            - paging guest physical pages into/out of machine frames
    - Problem: double paging

        - Guest writes to page, marked dirty in guest PTE and in machine PTE
        - VMM evicts page, written out to disk (since it's dirty), $P \rightarrow M$ mapping marked invalid / not present
        - Guest OS evicts page, needs to write page to disk (since it's dirty), causes page fault to the Hypervisor
        - Hypervisor pages in the page, fixed up $P \rightarrow M$ mapping, restart Guest OS instruction
        - One (guest) page fault causes 3 paging operations
            - write back dirty physical page to VMM swap device
            - read dirty page back in
            - write back dirty physical page to Guest OS swap device
        - Also causes 1 extra page fault
            - One to the VMM, one to the guest OS
        - Problem: 2 independent paging systems conflicting
        - Solution: persuade the Guest OS to do all the paging the VMM wants
- The Balloon Driver
        - Device driver loaded into the guest kernel
            - limited form of paravirtualization
        - Has private communication channel with the VMM
            - the VMM can ask the balloon driver to do things
        - Can request (physical) memory allocation from the kernel
            - which can cause the kernel to page out other stuff
- Ballooning: taking RAM away from a VM
        - VMM asks balloon driver for memory
        - Balloon driver asks Guest OS kernel for more frames (inflates the balloon)
        - Balloon driver sends physical frame numbers to VMM
        - VMM translates into machine addresses and claims the frames
- Returning RAM to a VM

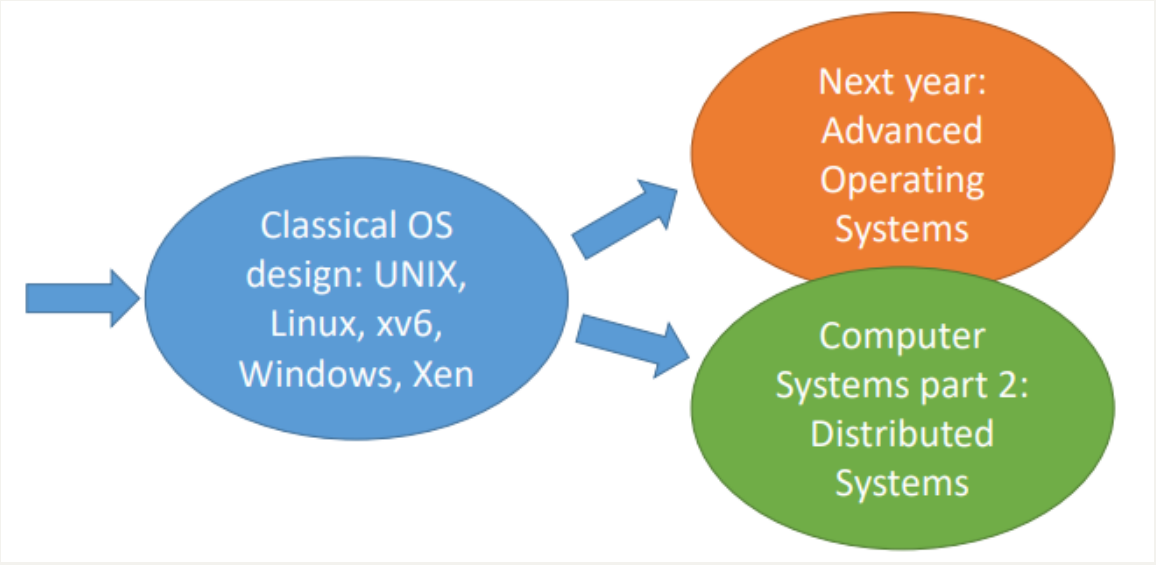
            - VMM converts machine address into a physical address previously allocated by the balloon driver
            - VMM hands PFN to balloon driver
            - Balloon driver frees physical frame back to Guest OS kernel
- Virtualizing devices and the network
    - Virtualize Devices
        - Trap-and-emulate
            - I/O space traps
            - Protect memory and trap
            - "Device model": software model of device in VMM
    - Interrupts → upcalls to Guest OS
        - emulate interrupt controller (APIC) in Guest
        - emulate DMA with copy into Guest PAS
    - Significant performance overhead
    - Paravirtualized devices
        - "Fake" device drivers which communicate efficiently with VMM via Hypercalls
            - Used for block devices like disk controllers
            - Network interfaces
            - "VMware tools" is mostly about these
            - `Virtio` in Linux is one attempt to standardize these
        - dramatically better performance
    - Xen 3.x Architecture



- Where are the real drivers?
    - In the Hypervisor
        - problem: need to rewrite device drivers
    - In the console OS
        - export virtual devices to other VMs
    - In "driver domains"
        - map hardware directly into a "trusted" VM (device passthrough)
        - Run your favorite OS just for the device driver
        - Use IOMMU hardware to protect other memory from driver VM
    - Use "self-virtualizing devices"
- SR-IOV
    - Single-root I/O virtualization
    - Key idea: dynamically create new "PCIe devices"
    - For networking
        - partitions a network card's resources
        - with direct assignment can implement passthrough
- Self-virtualizing devices
    - can dynamically create up to 2048 distinct PCI devices on demand



- Network
    - Virtual network device in the Guest VM
    - Hypervisor implements a "soft switch"
        - Entire virtual IP / Ethernet network on a machine
    - Many different addressing options
        - Separate IP addresses
        - Separate MAC addresses
        - NAT
- Conclusion



Nov. 4th - Lecture 14: Distributed Systems, Introduction

- Abstract a machine to a **node** that provides certain functionality/computation capacity
- Different servers should maintain the same coherent state, such that users interpret the process as one server. Multiple servers are used to increase fault tolerance.
- Consider the rate of failure of servers, the performance/response time of servers
- Having a single serializer acts similarly as a single server.
- Paxos
    - Ticket: can send it to client (who now goes to sleep), the server can just issue a new ticket to another client. When the first client wishes to return the ticket, the server can render it expired.
    - Theorem: if a command c is executed by some servers, all servers eventually execute c
- The Consensus Problem
    - Agreement: some nodes that can exchange messages in the system, we want them to decide on the same thing
    - Validity: If all the inputs are the same, then they must decide on that value; if they decide on some value, then at least one input would have the value they decided on
    - Termination: all nodes must decide on something
- Consensus in $f+1$ rounds: ✓
- Consensus in $< f+1$ rounds: cannot
- Asynchrony: "upon receiving message do..."
- FLP Theorem: Deterministic consensus is impossible in the asynchronous model even for binary inputs and at most one crash failure (what about allow randomizations?)
- Asynchronous communication
    - randomized consensus for $f < \frac{n}{2}$: ✓
    - randomized consensus for $f \geq \frac{n}{2}$

Nov. 6th - Exercise Session 6

- Hard links vs. Symbolic links
    - HL advantage: when delete main file, HL works, SL does not
    - SL advantage: can reference across machines
- Keeping first part of UNIX file in the same disk block as its i-node, advantage?
    - locality, fewer disk accesses for file read
- Single directory, allowing arbitrarily many files with arbitrary long names
    - use file names to simulate directory path
- Sequential files: only allows to read from start to end; Random access files: can read wherever the user wants
    - RA no need, just access the file location desired
- Contiguous allocation of files leads to disk fragmentation (this is internal fragmentation)
    - Internal fragmentation: within an allocated area of memory
    - External fragmentation: between allocated area of memory
- Virtual file system layer allows an OS to support multiple types of file systems
    - VFS translates generic calls from OS to the matching call in the right file system by providing functin calls
- Recap (Network Stack)
    - Network Stack Functionality
        - encapsulation, decapsulation
        - multiplexing, demultiplexing
    - TCP State
        - TCP has state in constrast to UDP
        - need to store this state per TCP connection
        - State processing often performed in kernel
    - Data movement inside the network stack
        - problem: while processing the data, we don't want to copy it over all the time
        - we need DS to easily store and remove data
    - `MBuf`

        - mbuf chain represents a single contiguous packet, using non-contiguous area of memory, singly-linked of mbufs chained with `m_next` field
        - Moreover, packets themselves can be hooked together in lists or queues using the `m_nextpkt` field
    - Hardware accelerated networking
    - Receive Side Scaling: uses many CPU to balance packet across CPUs; based on Hashing a combination packet header fields
        - Flow steering: driver can configure flow table to reduce cases where single flow uses up all the bandwidth
    - TCP chimney offload
    - Remote Direct Memory Access
        - allow different machines to write to each other's memory without involving the OS
- Is the TCP protocol state processing done in userspace? Why or why not?
    - in kernel, if in userspace, unprivileged action can abuse the system
- The network stack in an OS only ever sends packets when explicitly requested by a user space application. (False, the TCP protocol requires sending packets that the user will never see)
- What is the trouble this causes when implementing a TCP protocol in the top half?
    - This would require the application to be scheduled at all kind of events not scheduled by the user program
- How does `m_next` field help to reduce data copies?
    - Adding headers can be done simply by adding an additional mbuf struct, and setting `m_next` to our payload
- Recap (Virtualization)
    - Hypervisors
        - Type-1: runs "on the metal", it functions as an OS kernel
        - Type-2: runs on top of/as a part of, a conventional OS like Linux or Windows
    - Tradeoffs
        - performance, flexibility (type-2), security, compatibility, type-1 for data centers, type-2 for consumer use cases
    - Server consolidation

- What needs virtualization: CPU, MMU, Physical memory, etc
- CPU
    - Full virtualization: VM simulates hardware, not aware that it is virtualized, support in hardware
    - Virtualize kernel space: trap and emulate - enough if strictly virtualizable
    - If not strictly virtualizable: software emulator; paravirtualization; binary rewriting; virtualization extensions
- MMU
    - Direct page tables, shadow page tables
    - Nested paging
    - Virtualizing RAM: Ballooning
- Device Virtualization
    - Device model used to emulate a hardware device inside a VM
    - paravirtualized device
        - driver's aware it's in a VM
    - Driver domain

Nov. 4th & 8th - Chapter 14, 15, and 16: Introduction to Distributed System, Fault Tolerance & Paxos, Consensus

Nov. 11th & 15th - Chapter 17, 18

Nov. 13th - Exercise Session 7

- Recap
    - Introduction
        - Node: single actor in a distributed system, can be both client or server
        - Challenges: messages can get lost; nodes may crash; messages can have varying delays
        - First Goal: State Replication

        - All servers execute the **same commands** in the **same order**
    - First approaches: server sends acknowledgement message
        - Reasonable with one client
        - Inconsistent state with multiple clients and servers
    - Serializer - all commands go through one node which orders them
        - Single point of failure
    - Two-phase protocol: ask for locks, execute once acquired all locks
        - breaks down if we even have just one node failure
        - deadlocks
- Fault Tolerance and Paxos
    - Paxos - Main ideas
        - Tickets: weak lock; can be overwritten by later tickets; reissuable; expiration
        - Require majority: ensures only single command gets accepted
        - Servers inform clients about their stored command: client can switch to supporting this command
    - Paxos - steps
        - clients ask for a specific ticket t
        - server only issues ticket t if t is the highest ticket requested so far
        - if client receives majority of tickets, it proposes a command
        - when a server received a proposal, and the ticket of the client is still valid, the server stores the command and notifies the client
        - if a majority of servers store the command, the client notifies all servers to execute the command
- Consensus
    - Agreement: All (correct) nodes decide on the same value
    - Termination: all (correct) nodes terminate (violated by Paxos)

- Validity: the decision value is the input value of at least one node
- Impossibility: consensus cannot be solved deterministically in the asynchronous model
- Synchronous model
    - nodes operate in synchronous rounds
    - in each round a given node can
        - send a message, receive messages, do some local computation
    - Runtime is the number of rounds from start to finish of execution in the worst case
    - Termination: runs for exactly $f + 1$ rounds
    - Validity: selection from set of broadcast input values
    - Agreement: at least one round without failures guarantees finding local min of remaining nodes
    - Lower bound: any deterministic consensus algorithm in the synchronous model has a runtime of at least $f + 1$ rounds for any $f \leq n - 2$ even under a relaxed validity constraint
- Randomized Consensus
    - easy cases: all inputs are equal (all 0 or 1), almost all input values equal
    - otherwise: choose a **random** value locally $\rightarrow$ expected time $O(2^n)$ until all agree
- Ben-Or: Consensus Proof
- Shared coin