- Case studies
- Interrupt handling
    - An external device signals the interrupt controller. It then prioritizes signals (depends on architecture) and also handles masking (if blocked or not by controller in bitmask)
    - The interrupt controller notifies the CPU, preempting any user process
    - The CPU halts normal execution
        i. and jumps to a predefined interrupt handler address
    - Disables further interrupts to prevent nested itnerrupts
        i. Usually interrupt flag register or signal to contorller
    - Context switch
        i. cpu saves registers, pc, processor status flags on stack
            1. Processor status flags are for executing instructions, like carry or zero flag
                a. Arithmetic Logic Unit (ALU):
                    i. Performs arithmetic (add, subtract, multiply, divide)
                    ii. Performs logical operations (AND, OR, XOR, NOT)
                    iii. Sets flags based on operation results
        ii. Switches to kernel
    - Reads interrupt descriptor table to get correct handler
        i. Used to be interrupt vector table, but only gave addresses and not metadata like privilege level/flags
    -
    - Later after: Cpu Writes to specific registers in the interrupt controller or sends end of interrupt signal depending on architecture to acknowledge interrupt's end
    - Cpu restores pc, flags, registers; jumps back to where was executing
    -

2. **High-Level Handler (C code)**
    - First runs the "top half" - critical, time-sensitive work
    - Schedules the "bottom half" for later execution
    - Returns control to allow CPU to handle other interrupts
3. **Bottom Half Mechanisms**
    - Types:
        i. Softirq
            1. Use if timing is critical, networks/block devices
            2. Runs in interrupt context (interrupt handler)
            3. Is fastest so is best for many threads
            4. Static so must plan ahead
            5. Low overhead so can scale well. Requires locks still for shared data
            6. Defined at compile time; fixed number of types
            7. Handled by kernel

        ii. Tasklet
            1. Dynamically created (during runtime, not compile) and easy to use
            2. Cannot be interrupted
            3. Runs in interrupt context (interrupt handler)
            4. Handled by kernel
        iii. Work queue
            1. Use for if timing is not critical
            2. Runs in process context (when process is in control)
                a. Can sleep, allocate memory, wait for io
            3. Can be preempted/scheduled unlike other 2
            4. Can sleep, unlike other 2
            5. Handled by kernel threads
    ○
●
● signal handling
● Signals are generated from hardware (ctrl c sigint), software (segfault), ipc, hanging up
● Sending special cases
    ○ Hardware - hardware interrupt, sent to cpu by hardware. If needed, Kernel takes over and kernel maps hardware conditions to signals (like sigsegv, sigint). Then kernel sends to process that created it
    ○ Terminal - terminal driver's line discipline detects special character (ctrl c for example). Based on mode, either converts it to special character or to signal. If signal, then calls kernel to do this: Kernel identifies fg process group for that terminal, then sends signal to all of them.
    ○ Timer based - setitimer, hardware creates timer interrupt once expires, kernel handles this then gives sigalarm to process
    ○ Ipc - kill() system call used to send signal
        ■ Int signal num is parameter
        ■ Sigusr1, sigusr2 are signals that can be defined by user with
    ○ Hangup - sent for daemon to restart, or for terminal related process to terminate since terminal dead. Nohup command & to prevent this or disown for bckgroudn command
● Sending ipc
    ○ Use syscall kill(pid, signal_num)
    ○ Kernel verifies sender has permissions to send
    ○ Locates task struct by pid and sets the receiver's pending signal bitmap to indicate it has a signal
    ○ Kernel might wake up process
● Receiving
    ○
    ○ Kernel changes process 's signal bitmap pending bit is changed so that it has signal
    ○ When user process does syscall or is scheduled, it has kernel handles the signal
        ■ Kernel has Sigkill immediately kills it

    - Can have special handler
    -
- Raw vs cooked mode, changes with termios, terminal io interface (config)
    - Cooked
        - Input is buffered until press enter
        - User can edit line
        - Ctrl c creates signal, etc
        - echo
    - Raw
        - Keystrokes immediately available to app
        - No line editing
        - Ctrl c is character code 3, not signal
        - No echo
- ## Process lifecycle
- Fork + exec
    - Slab allocator
- Is new / ready
- Wait in cpu queue, then running
- If block on io, then in waiting state. Once get io, then ready again and waiting in cpu queue
- Prempted, then is ready and waiting in cpu queue
- If terminated, then is in exit state
    - Doesn't disappear until acknolwedged by parent that it is dead
    - This makes it a zombie process if it is not acknowledged
-
- 7 state version adds 2 more states:
    - Suspend ready - was ready but then memory had to be swapped out to disk. Now it is this state until memory is swapped back
    - Suspend wait - is same as suspend ready, but for processes that were in waiting state when this happened. Once done waiting due to io return, then becomes suspend ready

- Then talk about schedulers:
- Long term - what processes are to run and when (cron, systemd)
- Medium term - what processes are in memory vs swa
- Short term - cpu scheduling; which use the cpu
-
- ## Io schedulers

Noop (No Operation)

- Extremely basic - just a FIFO queue with minimal request merging
- No actual scheduling or reordering of requests
- Best for SSDs or environments where overhead matters more than optimization

- Low CPU overhead but minimal I/O optimization

Deadline

- Balances fairness with performance
- Uses three queues: read FIFO, write FIFO, and sorted (elevator) queue
- Enforces maximum wait time for requests
- Good for real-time systems but can cause starvation when prioritizing deadline requests
- Prevents indefinite postponement of requests

Anticipatory

- Similar to Deadline but adds a waiting period after reads
- Anticipates that more reads might come from the same process/location
- Reduces disk seeking by waiting briefly for nearby requests
- Good for systems with heavy read workloads
- Trades some latency for better throughput

CFQ (Completely Fair Queuing)

- Allocates I/O time slices to processes based on their priority
- Uses a round-robin approach for fairness
- Provides per-process queues
- Can prioritize certain processes based on niceness values
- Good general-purpose scheduler for desktop systems

Advanced Multi-Queue Schedulers

These newer schedulers address the single-lock bottleneck of older schedulers:

BFQ (Budget Fair Queuing)

- Creates a queue for each process with a budget
- Controls bandwidth and time allocation
- Supports cgroups for better resource control
- Extends budgets when no other processes need I/O
- Good for mixed workloads with varying priorities

MQ-Deadline (Multi-Queue Deadline)

- Modern version of Deadline for multi-queue devices
- Assigns deadlines to requests to prevent starvation
- Prioritizes reads over writes
- Excellent for database and enterprise storage systems
- Better parallelism than single-queue deadline

Kyber

- Focus on meeting latency targets rather than fairness
- Dynamically adjusts queue depths based on observed latency
- Only two tunable parameters: target read latency and target write latency
- Self-tuning for optimal performance
- Ideal for low-latency SSD workloads and modern NVMe devices
- Trades some throughput for predictable latency when needed

-
- Mode switch (for interrupts, signals, syscalls)
- User program loads syscall number into rax register
- Arguments placed into registers (RDI, RSI, RDX, R10, R8, R9)
-
- Cpu does this: User rip/program counter is stored in rcx and rflags in r11
    - Depends on architecture, most save to kernel stack
    - Rflags are different flags like direction flag, carry, zero, interrupt, i/o privilege, etc.
- Trap instruction like Syscall / sysenter / int 0x80 is called, depending on architecture
    - Trap into kernel mode in a controlled way that's why it's called that
- Gets kernel code and stack segment locations from msr
- Execution jumps to handler stored in lstar msr
    - Most linux stores entry_syscall_64 function address there
        - This is entry point for kernel
- Entry_syscall_64 switches to kernel stack and saves user registers into kernel stack. Sets up kernel environment (may rearrange registers as needed, handle errors, set up systracing, etc.)
- Setup
    - Privilege ring changes from ring 3 to 0. This info is in cs register current privilege level field
    - User registers stored in kernel stack
- Validates call is legitimate + reads syscall table to find syscall handler
- Kernel jumps to handler address
- Kernel does work
- Kernel stores answer in rax register
- If field in task's signal structure becomes pending during htis, kernel sets up user to handle the signal after switch back
    - Changes rip to signal handler
        - Sets up signal trampoline in vDSO or stack frame that will call rt_sigreturn. signal handler runs, then calls signal trampoline as return address
            - Vdso is virtual dynamic shared object. Is small library that allows kernel to map memory into all userspace; allows for user and

            kernel to share memory, often so that kernel can execute syscalls in user space by having user do it
        - Signal trampoline includes sigreturn
            - Sigreturn is syscall but operates in user context
            - Does this cuz user doesnt have specific security permissions to edit cpu registers and abstraction cuz doesn tknow how to
                - Also atomicity (no interrupt) and update signal mask
- Sysretq restores user context
- Cpu returns to ring 3 privilege, restoring rip from rcx
- Execution resumes for user
-
- Context switch - is process switch; terminology is quite loose. Some resources say it includes mode switches + interrupts (since this needs to be in kernel + change out for ISR) while some say it doesn't and it depends on OS
- Scheduling queues updated, task struct state updated, monitoring stats/memory updated
- Current t ask struct swapped out for new one and update state

- Saving the entire state of the currently running process (register states) into task struct, aka process control block
    - Memory mapping, file descriptor, signal handlers are all ptrs in task struct already but these are needed for future
- Update mmu to point to table for new process; usually done by updating cr3 to be that
- Loading the state of the new process to be executed
- Updating various system tables and data structures to reflect the change
- Potentially flushing the translation lookaside buffer (TLB)
- Network send, receive
- Socket creation
    - **Socket creation** is initiated by the user application through a system call like `socket()`. The application specifies the **protocol family** (e.g., AF_INET for IPv4, AF_INET6 for IPv6), the **socket type** (e.g., SOCK_STREAM for TCP, SOCK_DGRAM for UDP), and the **protocol** (e.g., IPPROTO_TCP).

    - After the socket is created, it can be bound to a local port using `bind()`, and for listening to incoming connections, the `listen()` and `accept()` system calls are used (for TCP).
    -
- Network interface card receives packet, uses direct memory access to put it into memory (no cpu involvement). It is stored in nic rx ring buffer
- Link layer - nic checks mac address + header to decide if wants to accept packet
- If using napi (new api, no particular name), driver could do polling or interrupt so that kernel can get packets

- Otherwise, hardware interrupt to cpu
- Cpu looks up in irq interrupt handler and runs it. Disables interrupt to avoid more interruptions. Shcedules bottom half
- Softirq runs when kernel has time. Removes packet from nic rx buffer and puts it into appropriate socket buffer in kernel
- processes packets b4 put into buffer
    - Network layer - checks if packet is for computer; otherwise forward. Ip header processed for ttl, checksum verify, packet should be fragmented?. Ip header removed
    - Kernel then extracts transport layer header (tcp header, etc.)
        - More checksum + fragmentation piecing together
        - Tcp retransmissions / state management
    - Kernel then put it in socket buffer
- Delivered with socket api (recv system call)
    - Copy from kernel space to user space

- Send
- **User Application Call**
    - Application calls `send()`
    - Data is copied from user space to kernel space (first copy)
- **switch in kernel - Protocol Stack Processing**
    - Data goes through socket layer
    - Transport layer (TCP/UDP) adds headers, handles segmentation
    - Network layer (IP) adds routing information and headers
- **Queuing and Network Device Layer**
    - Packet is placed in the kernel's transmission queue
        - Several layers, like tcp write/sync queues
        - Qdisc for traffic control, shaping, etc.
    - Network subsystem schedules packets for transmission
- **Driver Interaction**
    - Kernel notifies device driver
    - Driver prepares data for hardware (second copy)
    - Data is copied to NIC's transmit buffer/rings
- **Hardware Transmission**
    - NIC processes packets from its TX buffer, turning them into frames
    - Interrupts CPU when transmission complete or for buffer updates

Optimized/Accelerated Paths

Kernel Bypass with DPDK

1. **Direct Memory Access**

    - Application writes directly to memory regions mapped to NIC TX rings
    - No syscalls or context switching involved
    - No kernel copies required
2. **Polling Mode**
    - Application actively polls device status rather than using interrupts
    - Dedicated CPU cores manage the NIC
    - Direct manipulation of hardware queues
3. **Hardware Transmission**
    - NIC transmits packets directly from user-allocated memory

Zero-Copy Optimizations

Even within the kernel path, there are optimizations like:

- `sendfile()` and similar syscalls that avoid user-to-kernel copies
- TCP Segmentation Offload (TSO) that lets hardware handle segmentation
- Scatter-gather I/O that allows transmitting from multiple memory regions
-
- Mmap
- Mmap called by user; is to put file into memory so is file backed. Anonymous is not file backed
    - Mmap signature: starting address, length, flags, file descriptor of desired file (or -1 for anon), offset for file
- Kernel takes over and reserves memory space in virtual memory for file. Entries marked as invalid/not present, along with permission bits
- Tlb flushed to avoid old mappings
- Pages may be larger than standard to avoid filling up tlb too fast. Mmap flag can be map_hugetlb
- Demand paging - when those parts of files are called in memory, then page fault exception. Kernel catches. Loads requested part of file in, but only into page cache, not actual physical memory of process. Mapped in vmem of process, though
    - Read ahead may occur depending on OS and its detection of sequential reads, where read additional parts of file before is requested to reduce io
- Cached in page cache
- If mmapped page is written to, then is marked as dirty. Msync write back directly; otherwise is written back during unmapping
- Deallocation
- All are removed from page table, tlb invalid only for mmap (mappings not removed for others)
- Stack -> ptr just moves
- Dynamic -> return memory to libc after free() but not to kernel
- Mmap - removed from page table, written back to disk if dirty, kernel may get back right away via page alloc, thus tlb invalidated

- ## Memory pressure
- In order:
- Reaping - slab reclaim
        - Reclaim unused kernel objects in slab caches
    - File-backed page not needed, dropped (clean dropped)
        - No longer held by process/is written back already (clean), or was never written back
    - Kswapd - More needed: dirty/needed pages are moved to swap space, increasing disk io
        - Since is background task, wakes up at thresholds: High threshold: no action, low: runs in background to gradually free memory, min: foreground + aggressively frees memory
        - Can turn on direct reclaim to just run in foreground when memory is critically low
    - Forces dirty pages to be written back (even more disk io)
    - Oom killer - kills lower priority + high memory usage + higher oom score (based on past two items) processes first
    -
    - other
    - Reference Counting
        - Used for objects like inodes, dentries, and file handles.
        - When the reference count drops to zero, the object is freed.
    - Page cache + process clean up (memory, file descriptors)
- ## Reading file
- Process calls syscall read() or fread on file, context switch to kernel
- Looks up file in global file table and verifies permissions. Creates entry in global file table if not already present. This points to inode
    - Global so that way can point to same inode if already open (saves lookup)
- Creates entry in process file table that points to global file table entry
- Kernel creates kernel buffer space if it doesn't exist already
    - Does so with slab allocator for smaller buffers + buffer head. Page allocator for bigger
- Check if is page cache. If so return that.
- Translating file path to inode via vfs
    - Break down into components (separated by /), with first one being /
    - Traverses directory tree by starting at / dentry, then looking up next component in that dentry, repeat
        - If dentry not in cache, must read it from disk. Otherwise is in kernel space memory dentry cache
    - Until we find the file
- So in kernel memory we have inode cache in slab allocator
    - Check this; if file inode is in there, we use it
    - Otherwise, fetch inode from disk
- Then use inode pointer to load actual file

    - Some filesystems map logical to physical
- File system converts file offset to logical blocks
    - Logical blocks are abstraction of block to represent consistent data size despite underlying hardware
- Volume manager and RAID layer determines which disk(s) these blocks live on
    - Different translations to disks based on striping, mirroring, parity, along with how disks are aggregated into giant disk pool
- I/O scheduler optimizes and merges requests. Knows who dirtied which page based on page metadata (address space struct in linux)
    - Io scheduler merges adjacent read requests (front and back merging), reorders for efficiency for elevator if hdd, prioritizes based on process priority
        - Front - if x is right before y, merge there
        - Back - opposite of front
        - Elevator efficiency - grouped based on close together in hdd
    - Noop: Simple FIFO with adjacent request merging, minimal overhead
    - Deadline: Uses elevator algorithm with read/write FIFOs and deadlines to prevent starvation
    - Anticipatory: Extends deadline with delay after reads to aggregate sequential operations
    - CFQ: Allocates I/O time slices based on process priority (like round robin)
    - Multi-queue Schedulers:
        - One queue for cpu core; enables parallelism for submitting io. Tends to be paired with bfq
    - BFQ: Budget Fair Queueing - creates per-process queues with budgets, supports cgroups
    - Kyber: Dynamically adjusts queue lengths to meet target latencies (good for SSDs)
    - MQ-Deadline: Deadline scheduler adapted for multi-queue SSDs
- Block device driver prepares commands for the specific hardware interface and sends logical block addresses to the disk controller
- Disk controller translates logical blocks to physical locations (sectors, pages, etc.)
- Perform read
- Interrupt, and kernel picks up data
- Blocks are coalesced/assembled back into the complete data
- Load into kernel buffer, kernel puts it into user buffer. Update page cache
- Inode has
- File permissions, timestamp, owner, pointers to data blocks on disk, file type, size
- But no file name
- Servers
- Multi-path Considerations:
-
- In SAN environments, multiple physical paths may exist between server and storage
- Provides redundancy and parallelism
- May require special drivers (multipathing) to manage load balancing and failover

- Ssd vs hdd

Block Device Driver

- **Translation Layers**:
    - For HDDs: Translates logical block to physical cylinder/head/sector
    - For SSDs: The FTL (Flash Translation Layer) maps logical sectors to physical pages
        - FTL maintains mapping tables (physical:logical)
        - This allows wear leveling and bad block management
        - Pages (4-16KB) are the smallest readable unit
        - Blocks (128-256 pages) are the smallest erasable unit
- **SSD-Specific Handling**:
    - SSDs optimize for 4KB reads, 1MB writes
    - Write operations require:
        - Copy valid data elsewhere
        - Erase block (slow operation)
        - Write back with modified valid data
    - TRIM commands inform SSD which blocks are truly free
        - Prevents write amplification and premature aging
        - Not enabled by default due to compatibility issues
- **Interface Types**:
    - SATA: Consumer-grade, up to 6 Gbps
    - SAS: Enterprise-grade, up to 22.5 Gbps
    - NVMe: Direct PCIe connection, much faster (31.5 GB/s with PCIe 4.0)
        - Supports multiple queues (up to 64,000 commands per queue)
        - Lower latency (~20 microseconds)
-
- ### Writing a file
- Write system call - fd, write buffer, size
- Kernel switch
- Write to page cache (alloc if not there)
- Once flushed, file system translates logical to physical. Logical since file system considers all disks as one whole disk, and we need to actually find the physical location in which disk, as well as take into account raid calculations
- Stored in buffer cache, which is another cache to prevent too many writes to disk. Here congregate any repetitive changes made to the same blocks
    - Buffer head is in buffer cache and keeps track of if is dirty, etc. + cached block meta data
- Then, put in ios cheduler. Io scheduler merges front/back. Io scheduler can be noop, deadline, kyber, mq-deadline, bfq, etc.
    - Flushing can be done in the following ways: write through (write through disk + in cache), write back (lazy full or when expire/requested)

- Then send to disk. Disk depending on sdd or hhd will deal with level wear or disk proximity. Disk proximity dealt with elevator priority
- Usually all writes are asynch unless system call is synch
- Kernel returns back to user
- Fg bg
- When processes run, they have process group, and this associates them in foreground or background
    - Pgid. maintained in task struct
    - Foreground -receives input from keyboard, direct access to terminal io, shell waits
- Background - shell does not show its output or wait for it.
    - Can show output if specified by unsetting TOSTOP flag; foreground has process in control + input/output in shell
- After a process is paused with sigstp or sigstop, fg or bg can be used to run in foreground or background
    - The shell calls tcsetpgrp(STDIN_FILENO, pgid) to move the process to the foreground, ghancing process group id in task struct
    - Shell keeps track of foreground process group with Terminal Group ID (TPGID). Whichever process group id this is equal to is the process in the foreground.
        - Foreground process has terminal translating special keys like ctrl c and echoing output if canonical (Cooked mode)
        - Shell updates its internal job table to mark this job as foreground
    - Sends SIGCONT if the process was stopped.
        - The shell waits for the process to complete or stop using waitpid() with the WUNTRACED option
    - While waiting, the shell itself is not part of the foreground process group, so it doesn't receive terminal-related signals
    - **File descriptors stay the same**
    - Sigttin sent to background processes when try to refad ainput from terminal. Sent by kernel when background process calls read() on fd 0 / stdin
    - Sigttou sent to background processes when try to output to terminal. Can ignore this (like vmstat can) Sent by kernel when background process calls write() on fd 1 / stdout
- fg command execution:
    - Sends SIGCONT to target process group to resume execution
    - Calls tcsetpgrp() system call to change the terminal's foreground PGID
    - Shell updates its internal job table to mark this job as foreground
    - Shell waits for the process group to complete or stop using waitpid()
- bg command execution:
    - Sends SIGCONT to target process group to resume execution
    - Does NOT change terminal's foreground process group
    - Shell updates its internal job table to mark this job as background
    - Shell continues to accept commands without waiting

- ## Piping command
- Shell tokenizes input
- Shell forks child processes for cmd1, cmd2
- Shell creates pipe between cmd1 and cmd2
    - pipe() system call - makes unidirectional communication with read end (fd[0]) and write end (fd[1])
    - Kernel creates fixed size buffer (usually 16 kb) for each pipe, such that any process can write/read from it
    - Requests so from page allocator if is big; otherwise slab allocator
- dup2() is used to redirect input/output of each process. Cmd1's output is write end of pipe (fd 1) and cmd2's input is read end of pipe (fd 0)
- 1st cmd calls write() (to it sstdoutput, which was changed) and 2nd cmd calls read. 2nd cmd is ut to sleep in wait queue for pipe until 1st cmd closes its write. Once cmd 1 writes into buffer, closes. Now kernel knows to kernel copy into cmd 2's user space the data from the buffer and wake cmd 2 up. Cmd 2 resumes read(), which returns the data or 0 if there's nothing in the buffer.
    - Read has file descriptor, user buffer space (for putting any data), size of read/write
- ## Vmstat output
- first row is avg since boot; rest is interval of 1 sec. can specify it too
- Everything is in kb or kb/sec
- Cpu
    - Sys, us, id, st
        - High sys - high time spent in kernel: many context switches, itnerrupts, syscalls, lock contention
        - High Us - cpu bound
        - Low id - too much cpu time since not enough time spent idling
            - How is it measured: `
        - High St - too much vm cpu time. Can have virtual machine and that uses up resources
    - R - runnable (run queue); if massivley over num of cpus, then bad.
- Mem
    - Si, so - swap in, swap out. High swap out is not bad. Is part of overcommitting for VM. but swap in is bad since means have disk latency
    - Swapd - amount of swap memory used. Higher tends to be bad, but high si is where latency kicks in
    - Buffer, free, cache
        - High cache buffer is good. Lower = maybe freeing up due to memory pressure
        - Low free can be bad but is not always. Is definitely bad if high swapd
- Io
    - Wa, bi, bo
        - High wa time - io bound task

        - Bi - reads, bo - writes. Can show how often read/write. Not always bad if high
    - B - number of processes waiting for io (uninterruptible)?
    - Gpt ver

Alright, if I were in an interview and got that, I'd probably answer like this:

Interviewer: Can you walk me through `vmstat`? What are the columns, rows, and how would you interpret it in practice? Not just what each column is, but what it *means* when troubleshooting.

Explain in these segments: r/cs (cpu sat), b/wa (potential disk sat), bi/bo (cache inefficiency/high disk usage), swpd/si/so (memory saturation potentially if high si), buffer/cache/free (memory contention), us (cpu usage), sys/in (syscalls/io interrupts), st (vm)

---

1. Procs

- **r**: Number of running or ready processes (ready queues across all cores). Subtract cores from r to find how many in total queue (use /proc/sched_debug to see how split up)

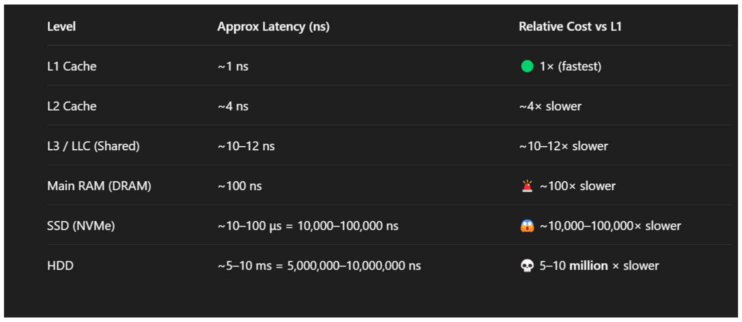
    - If `r` is consistently higher than the number of CPU cores, that's a red flag for CPU saturation or contention.
        - If it's just bigger for a bit, can be many small processes that dont' ake up much cpu and wil finish quickly once they get their time slice
        - Or processes that just finished waiting for io getting back in and waiting to finish up
            - In both cases, queue should shrink fast
    - R provides insight into cpu saturation across system. r is the length of the run queue, which is the sum of the lengths of the ready queues across all cores, and the number of running processes. we have ready queues because we often have multiprocessing, meaning that there are mtultiple processes running on a OS for efficiency and operational purposes. often, there are more processes than cpu cores. thus, we require contet switching to change which process or thread is using the cpu currently to ensure all fo them can use the cpu fairly, such as with cfs. this provides illusion of parallelism due to rapid switching, as given each small time interval, processes are able to have a proportion of that time be cpu time and thus be perceived as running. we can use r for looking at cpu saturation. it is a little tricky since we need to know number of logical cpu cores, not cores

due to hyper threading existing. we can look at lgical with lscpu or nproc. if n is consistently massively over number of logical cpu cores, there is saturation, meaning there are too many processes waiting for cpu, or there is some scheduling inefficiency causing high wait for cpu. this causes processes that depend on cpu to be much slower since they don't get the runtime as often as they should. however, r can be briefly over cores if we have short cpu processes or processes returning from io that need some acknowledgement. in these cases, r will decrease quickly and it is nothing to worry about unless it is a latency sensitive or real time system

- **b**: Processes blocked in uninterruptible on I/O. does not include network io outside of nfs
  - High b means I/O bottlenecks, possibly slow disk, nfs, or overloaded I/O subsystem.
    - Usually: synchronous reads/writes, page fault read from disk, locks, file system journaling
  - Synchronous reads, writes are processes blocked on io usually
  - Connect to wa: this shows number of processes blocked, while wa shows how much cpu time is lost to waiting

---

2. Memory

- **swpd**: Amount of virtual memory used (swap).
  - Non-zero isn't always bad, but if it grows steadily or is large, you might be swapping heavily, which usually kills performance.
  - Talk about si and so: swapd only bad if si is high; means high read/write from disk when could have had memory read/write
  - Read from didsk is much slower
    - 

Level	Approx Latency (ns)	Relative Cost vs L1
L1 Cache	~1 ns	🟢 1× (fastest)
L2 Cache	~4 ns	~4× slower
L3 / LLC (Shared)	~10–12 ns	~10–12× slower
Main RAM (DRAM)	~100 ns	🚨 ~100× slower
SSD (NVMe)	~10–100 µs = 10,000–100,000 ns	😱 ~10,000–100,000× slower
HDD	~5–10 ms = 5,000,000–10,000,000 ns	💀 5–10 **million** × slower

    - Ssd is 100-1000 times slower. Hdd is 50,000-100000 slower

- **free**: Free physical memory.

  - Low free memory is normal on Linux due to caching, but near-zero could be concerning *if* coupled with high swap usage.

- **buff** / **cache**: Buffers (for block devices) and page cache (file system cache).

  - Buffer is memory used to store writes before write to disk
  - Cache is memory used to store frequently read items from disk
  - Healthy systems will use most free memory for cache to facilitate read and writes from disk, which mentioned earlier, is much slower. A sudden drop may indicate memory pressure.
    - Since is in mem, isn't durable, which s why need occasional flush
  - Use buffer to aggregate writes to make them as sequential and less random s possible
    - Short random bad because:
      - Ssd parallelism works better without large sequential
        - Writes are in blocks
        - Reads are in pages (smaller == waste bandwidth). Also larger == distribute across channels/dies/planes better, increasing parallelism
          - Also larger == can use prefetch algos + less controller lookups (in batches, consumer less cycles for commands, utilizing cycle bandwidth better)
      - Hdd has seek + rotational latency due to physical heads and disk rotation to find block + sector
      - Also prefetching
  - Cache the same but for reads

---

3. Swap

- **si** / **so**: Swap-in / Swap-out.

  - If you constantly see high so (swap-out), the system is under memory pressure and is actively evicting pages to swap.

  - Occasional swap activity can be OK, but sustained swapping usually means you need more RAM or need to optimize memory usage.

---

4. IO

- **bi** / **bo**: Blocks in / blocks out (disk reads / writes).
  - High `bo` could mean heavy write activity, common during backups, logging bursts, or database activity.
  - High **bi** could be I/O-heavy reads like database queries or applications doing large reads.
  - If `b` in Procs is also high when `bi`/`bo` are high, likely I/O is becoming a bottleneck.
  - Does not include network io
  - Could indicate bad caching, even if cache is high
    - Check cache:
    - Check io wait time (io top)

---

5. System

- **in**: Interrupts per second.
  - High interrupts could be due to heavy I/O, network traffic, or misconfigured devices.
- **cs**: Context switches per second.
  - Elevated `cs` can happen with high process counts/multitasking, inefficient scheduling, or lock contention.

---

6. CPU

- **us**: User CPU time.
  - Indicates time spent in user-space (apps).

    - 70-80% isn't automatically bad, but you want to make sure it's aligned with expected workload.

- **sy**: System (kernel) CPU time.
    - If `sy` is higher than ~20-30%, it may mean kernel overhead is too high — for example, too many syscalls, bad drivers, or inefficient I/O.
- **id**: Idle time.
    - If this is near 0%, you're maxing out your CPU. not waiting for io, not working
- **wa**: I/O wait time.
    - High `wa` means the CPU is sitting idle, waiting on I/O, which correlates with high `b` and high `bi/bo`. Does not include network io
    - Shows percentage of cpu time lost spent idle due to waiting for io; shows inefficiencies (less cpu time and thus power to do work)
- **st**: Stolen time (if running in a VM).
    - Time stolen by the hypervisor. High values may mean the host is oversubscribed.

Interpreting Rows

Each row is just a snapshot of those values taken at the interval you specified, e.g., `vmstat 1` gives you 1-second samples.

Red flags:

- `r > CPU count` = CPU bound
- High `b` = I/O bottleneck
- High `si` / `so` = Memory pressure
- High `wa` = I/O causing stalls
- Low `id` + high `us` or `sy` = CPU saturated

- High `st` = Problems on the virtualization layer

How I Use It Practically

In real troubleshooting, I correlate:

- If I see high `r` and high `us/sy`, it's a CPU issue.
- If `b` and `wa` are high, it's likely a disk or network bottleneck.
- If swap is in heavy use, I know I'm under memory pressure.
- If context switches are skyrocketing, maybe too many small threads/processes are fighting for CPU.

Then I'd usually follow up by checking:

- `top`, `iostat`, `pidstat`, `sar` to zoom into specific processes, disk stats, or CPU usage.

Optional if interviewer is interested: Also, `vmstat -s` gives you cumulative counters, which is helpful to detect trends instead of just looking at the short-term snapshots.

Would you like me also to tell you how I'd combine `vmstat` with other tools in real incidents? It often helps make the answer feel more "battle-tested."

If you want, I can show you how to make it even more interview-polished depending on your style (casual, confident, system engineer vibe, etc.). Just say *yes*, and I'll tailor it.

  -
- Context switches
- High sys - inefficient io
  - Inefficient io usually means too many small random io

        - Random io bad since ssd's nand flash chips's parallel channels handle best when long sequential, and hdd for each search needs to move seek head and rotate disk to sector
    - High sys and high cs - likely bad scheduling, swapping too much for io
    - High sys and si/so, meaning inefficient kernel buffer management, (random, non aligned, non buffered io)
- High b, wa
    - Means high sync writes/reads wait
    - However, asynch can implicitly wait on page cache to be flushed (during reads or writes), but this technically doesn't increase wa, b
- ## Wal
- Write to log before write to disk. Sequential, so faster
- Ensures less bottleneck in disk (like less random access)
    - Thus can batch
- durability
-
- ## Page faults
- Mmu checks tlb. does page walk in process's page table
    - Start at root table. Each level tends to be X hexadecimal digits, use these to find next node in curr page table. This leads to next table until we get to leaf node, which has dirty bit, permissions, and physical number
    - Add this to virtual page number and we get physical frame address
        - If node doesn't exist or leaf node doesn't exist, page fault
- Page fault occurs -> pc jumps to page_fault handler - do_page_fault() is called to handle page fault error thrown
    - Retrieves faulting address from the CR2 register (on x86).
    - Also handles cow
- This happens When stack grows, or part of file or heap not in memory (in disk or swap)
    - Heap must go through libc allocator though
    - Inaccessible to program: Seg fault. Nonexistent physical (bad translation, bad alignment (4 byte int needs to start at address divisible by 4): bus
- Kernel checks if it is in within process's memory bounds
    - Get task_struct's mm
    - This is mm_struct, and then look through red black tree of vm_area_struct objects
    - Do binary search to Look for vm_area_struct where its start <= faulting address <= end
    - Then checks permission flags
    - If not matching vm_area_Struct, then throw sigsegv
- Minor page fault - is valid mapping (like it's in page cache), no disk
- Major page fault - not in memory at all; is in disk
- If is minor page fault, meaning it is valid but not actually allocated, such as lazy allocation of valid stack address, then we contact global page allocator
    - Page allocator gets page from free lists

    - Zeroes it out
    - Page table updated with this page
- Major - Fetch from disk due to swap or loading file:
    - Usually due to mmap, swap, loading exe instructions
- May not need to fetch from disk if is in page cache
- If need to fetch from disk, then higher latency. After fetch, stored in page cache
- Process's page table is updated so points properly to thath physical page
- ## Stack memory
- Is anonymous, meaning is memory not backed by file
-
- When process accesses unmapped stack address, triggers page fault
    - Each thread has its own stack
- do_page_fault() and do_anonymous_paging() runs
    - Kernel funcs
- expand_stack() validates stack growth (if is just below stk ptr). If go over soft limit, then get warning depending on system
    - Configured with ulimit -s
- Hard limit is impossible to go over. Sigsegv if go over
- Otherwise, after validation: Physical page then allocated and table entry created. Set as present, writable, user accessible. Page reference count increased by 1
- Mm_struct for pcoess updated
    - Stack_start, stack_vm (number of pages allocated), vma (virtual memory area / address range) updated
-
- Stack memory can be swapped during memory pressure
-
- `how deallocate; how affect free in vmstat vs __
- ## Dynamic memory
- Each user process has libc allocator instance
- Libc allocator is software that manages heap for process
- Process creates dynamic memory with malloc and frees it with free (glibc functions)
- Libc has bins of different sizes as a sort of bucket sort, but with each bucket representing ranges. Within each, it has doubly linked lists of free blocks
    - Uses algorithms
        - Segregated - looks for close to what need in a bin of select size(s)
        - First fit - first that meets need in free list is given. Can lead to internal frag due to being bigger than need
        - Best fit - scans entier free list and gives as close to what need as possible. Takes longer
- When malloc is called, gets free block of size x in its free lists (bins)
- If can't service, then send to page allocator via brk, mmap system calls
    - brk/sbrk (for smaller allocations to extend the heap)
    - mmap/munmap (for larger allocations, typically >128KB)
    - Page allocator is in kernel space

    - It allocates pages, not blocks
    - Memory divided up into segments: each segment has its own free lists or varying order. Uses free_area struct with head of free list + how many free blocks in list
        - Ex: dma, dma 32 (32 bit devices), normal, highmem, movable (migratable)
    - Uses buddy system
        - Memory can be broken up into powers of 2 (4 kb, 8 kb, etc.)
            - Reduces memory fragm
        - If block of desired size is available, use it. Otherwise, break up existing memory in half until get block of desired size. This is given to requester
        - Remaining buddy blocks are put in free lists of varying order. Order corresponds to size
        - When block is freed: checks if block's buddy is free too
        - If so, merge into bigger block. Repeat recursively
            - Buddy - same size + physical addresses only differ by 1 specific bit (torwards right)
        - This allows for quick merging instead of linear scans
- Kernel then reserves physical pages with lower addresses (user space) and maps it to virtual addresses of process in its page table.
    - Due to modern lazy loading, physical page corresponding to physical address doesn't usually exist until called later, which then it will be populated with 0's
        - But the page table entry still exists before populating; it is just marked as reserved
- Libc then breaks up these pages into blocks according to need
    - Bins are ranges: smaller ones are exact. Larger are ranges
- Libc returns pointer to new memory to program (is after metadata stored for memory)
    - Typically first 8-16 bytes are metadata - size, prev size
- ## Ls -l
- Shell tokenizes input
- If any *
    - Then terminal does globbing, which is looking for all files in given dir with getdents. See for each file if match pattern, then adding them as arguments to cmd if so
- Checks if is built in; if it is, then terminal runs it (no new process needed)
- otherwise:
- Terminal does system call fork(), fork returns 0 to child and parent returns child pid if no error
    - Copies registers, file descriptors, cwd, environment variables, signal handling settings, resource limits, process group, user/group id, umask (file creation perm mark), current root directory
    - Creates task struct from slab allocator
    - Fork uses COW optimization to share memory at first, and when written to, it will make copy
        - Memory includes virtual memory (stack/heap/vars)

        - Is default optimization
        - Memory is marked as read only by setting write-protect bit (if try to read, then cow page fault)
    - Passses copies of file descriptors, permissions/limits, cwd
- Exec called in child
    - Usually execvp to look for cmd binary not execv (need full path)
    - Checks $path for directories (separated by :) that hold cmd binaries
    - Check those and see if cmd is found in there. Get binary
    - replace its memory segment but preserving file descriptors
        - Linux does lazy loading, where elf has headers and entry point. Must load entry point, and then headers tells about which segments (critical) should be loaded immediately (offset + size), and to what virtual address, and how much to allocate, alignment
            - Elf has data, unint data, code, read only data
- Ls runs, calling stat(), open(), getdents(), close() as needed (when pass in dir args) to read directory contents
    - No -a -> filters out those that start with .
    - .. -> open() reoslves this by looking at cwd and just going back one by looking at inode and then its parent dir
    - Stat usually read from proc any stats/info
- Calls exit()
    - Flushes buffer (buffer io) and does other cleanup
    - Flushes Buffers: Any data still in the user-space buffers is flushed to the filesystem (for instance, through fsync() or msync() if needed). This is handled by the kernel.
    - File Descriptors Close: All file descriptors opened by the process are closed, releasing the resources. This ensures that any files, sockets, or pipes the process opened are cleaned up.
    - Memory Release: The kernel deallocates memory resources used by the process. This includes both user space memory (heap, stack, and memory-mapped files) and kernel resources like page tables.
    - Exit Status: The process provides an exit status (e.g., exit(0) for normal exit or exit(1) for an error). This status will be stored in the kernel for the parent process to collect.
- Writes to terminal via same fd (fd 1)
- Sigchld given and parent acknowledges with wait
-
- Typical cleanup
- Flush buffer, clear dynamic memory/temp files, release locks/conns
- Lsof ver
    - Calls open(), getdents() (directory entries), fstat (for info on file descriptors), readlink (to get info on symbolic links)
    - /proc/[pid]/fd/, /proc/[pid]/fdinfo/ for file descriptor info
    - For mount / group owner: /proc/mounts, /etc/passwd, /etc/group

- How it runs at assembly
    - Machine code is translated into instructions (assembly)
    - Calls system calls; parameters passed into registers of cpu for syscall to use
        - Caller vs callee saved registers
            - Caller: caller saves registers if it wants to preserve them
            - Callee: the child function saves registers and restores them once modifying them
        - Overall best to use unused registers. Goal is to minimize loading/storing in registers (graph color problem, making sure as few registers are used). This is a compiler optimization
            - Nodes are variables. Edges mean x and y have lifetime overlap
            - Want to have as few colors but ensure those connected adjacently are different colors (different registers if at same time)
    - On modern x86-64, the syscall instruction is used
        - Older x86 systems used int 0x80 (interrupt 0x80)
    - Finishes running
    - Registers now how return values. Registers are loaded from snapshot
    - Following the specific calling convention for the architecture (e.g., System V ABI for Linux)
    - Understanding stack alignment requirements and stack pointer adjustments
-
-
- **Namespace memberships**
-
- Misc trivia
- Page table entry bits
- Permissions, valid, accessed
- Making active inactive
- Lru scanning daemon runs every so often, setting accessed bit to 0
- Then adds it to active page if 1 and then inactive page list if 0. then it repeats this every so often, and it shows which pages are active within the most recent time period
- If not accessed after again, will remain as 0
- But when mmu accesses it, accessed bit becomes 1
- Getting memory back

1. Order of Page Reclamation (Hierarchy)

When memory pressure increases, **Linux tries to reclaim memory in the most efficient way first** before resorting to swapping. The order is:

Step 1: Reclaimable Kernel Memory (Slab Cache)

- **Reclaimable memory** includes:
    - **Slab objects** (dentries, inodes, kernel structures).
    - Some kernel caches that **can be freed** if necessary.
- These are **freed first** because they don't require disk I/O.
- Inactive first, then active

Reclaimable user - Step 2: Drop Clean File-Backed Pages (Inactive File Pages, also unused pages recently)

- If a file-backed page is **clean (not modified since it was loaded from disk)**, it can be **immediately discarded**.
- These pages can be **reloaded from disk later** if needed.
- This includes:
    - Cached files
    - Read-only binaries
    - Shared libraries
- **This is the easiest and cheapest reclaim step.**

Reclaimable user - Step 3: Write Back Dirty File-Backed Pages

- If the file-backed page is **dirty (modified)**, it **must be written back to disk** before being freed.
- The **writeback daemons (pdflush, dirty_background_ratio settings)** handle this.
- Once written, the page becomes **clean** and can be dropped like in Step 2.

Step 4: Swap Out Inactive Anonymous Pages

- If **file-backed pages aren't enough**, Linux **starts swapping out anonymous memory**.

- These pages **cannot be dropped** because they don't exist on disk already.
- Instead, they are **written to swap space** before being freed.
- These pages include:
    - Heap memory (`malloc` allocations).
    - Stack memory (thread/process stacks).
    - Other dynamically allocated memory.
    - Can only be nonreclaimable user (not mlock'd)
        - Kernel no

Step 5: Swap Out Active Anonymous Pages

- If **inactive pages aren't enough**, Linux starts scanning **active anonymous pages**.
- This is the **last resort before OOM killer**.

Step 6: OOM Killer (Worst Case)

- If all the above fails, the **Out of Memory (OOM) Killer** selects processes to terminate.
-

- Reclaimable vs non, inactive vs active
- Active - memory used by process
- Inactive - memory not used by process recently. Is not necessarily free. Free means can be used immediately. Inactive might have to be reclaimed
    - When first allocated, is inactive
    - When page fault and filled (or just filled), then becomes active
    - If hasn't been used for a while, becomes inactive
- Free - completely unused
- Both are reclaiamble, but inactive more likely to be reclaimed
- Reclaimable - able to be used for something else and freed. Usually buffer, cache, file-backed data.
    - Does not include anon mapping, ipc shared memory, kernel memory, locked memory
- Nonrelciamble - opposite of reclaimable
    - Kernel nonreclaimable cannot be swapped out, but most user can unless locked with mlock()

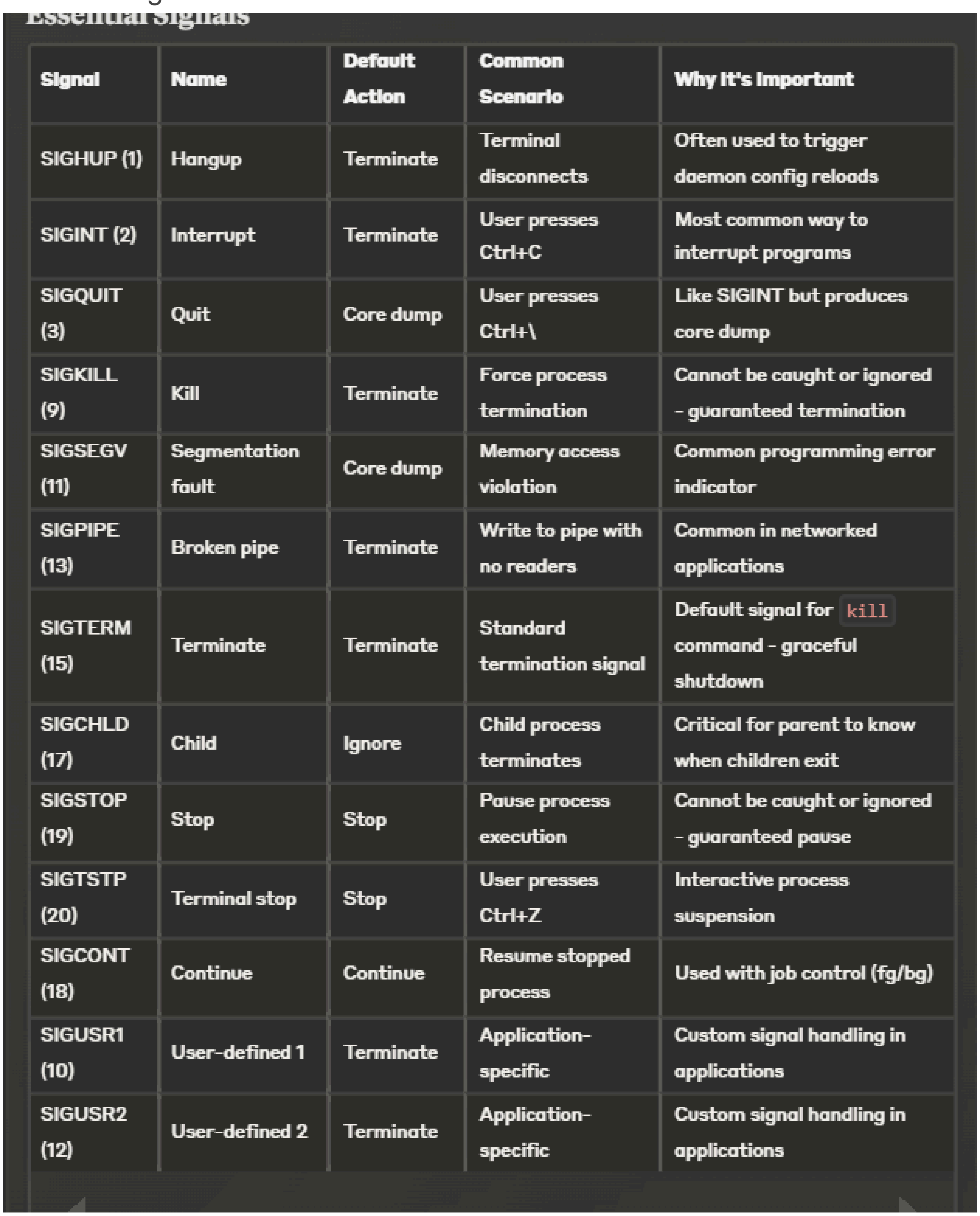
- Explain registers
- General purpose registers - stores data, addresses, return values / calc results
- Vector registers - holds multiple data elements at once
- Segment registers - defines segments in memory, like code, data, stack segmetns when all processes used to share memory
    - Mostly vestigal now
    - But this includes specific registers related to current cpu states
        - Cs register - for privilege level
        - Gs / fs register - points to thread info
- Special registers - registers for rip or status flags (rflags)
- Msr - model specific registers. There are typically hundreds of these. Need higher privileges to access these
    - Used for power management, perf, thermal monitoring, configs
- Control registers - configures cpu operating modes and behavior
- Debug registers - for debugging
- X86_64

In a modern x86_64 processor (found in most PCs and servers):

- 16 general-purpose 64-bit registers (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15)
- 16 128/256/512-bit vector registers (XMM/YMM/ZMM0-15), extendable to 32 registers in newer CPUs
- 8 segment registers (CS, DS, ES, FS, GS, SS, TR, LDTR)
- Special registers: RIP (instruction pointer), RFLAGS (flags)
- Control registers (CR0-CR4, CR8)
- Debug registers (DR0-DR7)
- Hundreds of MSRs (Model Specific Registers) for various CPU features
- Control registers
- **CR0**: Controls operating mode (protected mode, paging, etc.)
- **CR2**: Contains the page fault linear address
- **CR3**: Points to the page directory (memory translation tables)
- **CR4**: Controls advanced features (PAE, VMX, SMAP, etc.)
- **CR8**: Controls task priority register (interrupt handling)
-
- Kill zombie process
- Bad because takes of space in process table. Does not take up memory though
    - If process table is too big can't create new entries
- Find it - ps aux | grep Z
- Ask parent to send wait to child
    - kill -SIGCHLD <PPID>
    -
- Restart parent process
    - kill -HUP <PPID>

    -
- Kill parent process
    - sudo kill -9 <PPID>
- ## Adjusting swap frequency
- Adjust swappiness
    - sudo sysctl vm.swappiness=<value>
    - 0 is not swappy
    - 100 swaps a lot
- This is temporary
- To see current default, look at proc/sys/vm/swappiness
- /etc/sysctl.conf to edit permanently
- Cgroup - by editing cgroup files, can change swapiness for a control group
    - echo 10 > /sys/fs/cgroup/memory/<cgroup_name>/memory.swappiness

- Common signals



Essential Signals

Signal	Name	Default Action	Common Scenario	Why It's Important
SIGHUP (1)	Hangup	Terminate	Terminal disconnects	Often used to trigger daemon config reloads
SIGINT (2)	Interrupt	Terminate	User presses Ctrl+C	Most common way to interrupt programs
SIGQUIT (3)	Quit	Core dump	User presses Ctrl+\	Like SIGINT but produces core dump
SIGKILL (9)	Kill	Terminate	Force process termination	Cannot be caught or ignored - guaranteed termination
SIGSEGV (11)	Segmentation fault	Core dump	Memory access violation	Common programming error indicator
SIGPIPE (13)	Broken pipe	Terminate	Write to pipe with no readers	Common in networked applications
SIGTERM (15)	Terminate	Terminate	Standard termination signal	Default signal for `kill` command - graceful shutdown
SIGCHLD (17)	Child	Ignore	Child process terminates	Critical for parent to know when children exit
SIGSTOP (19)	Stop	Stop	Pause process execution	Cannot be caught or ignored - guaranteed pause
SIGTSTP (20)	Terminal stop	Stop	User presses Ctrl+Z	Interactive process suspension
SIGCONT (18)	Continue	Continue	Resume stopped process	Used with job control (fg/bg)
SIGUSR1 (10)	User-defined 1	Terminate	Application-specific	Custom signal handling in applications
SIGUSR2 (12)	User-defined 2	Terminate	Application-specific	Custom signal handling in applications

-

- Common interrupts

Common Interrupts in Operating Systems

Here's a breakdown of important interrupts for OS interview preparation:

Hardware Interrupts

Interrupt Type	Source	Handling Mechanism	Interview Relevance
Timer Interrupt	Programmable Interval Timer (PIT)	Triggers scheduler, updates system time	Preemptive multitasking, process scheduling
Keyboard Interrupt	Keyboard Controller	Buffers keystrokes, wakes processes waiting for input	I/O handling, event-driven programming
Disk I/O Interrupt	Disk Controller	Signals completion of read/write operations	Asynchronous I/O, DMA operations
Network Interrupt	Network Interface Card	Indicates packet arrival/transmission complete	Network stack performance, interrupt coalescing
USB/Device Interrupt	Peripheral controllers	Signals device state changes or data availability	Hotplug capabilities, device management
Error Interrupts	Memory controller, CPU	Reports hardware errors (e.g., memory parity)	Reliability, error recovery mechanisms

Software Interrupts (Traps/Exceptions)

Interrupt Type	Cause	Handling Mechanism	Interview Relevance
System Call	`syscall`/`int` instruction	Transfers control to kernel mode	User/kernel space transition, privilege levels
Page Fault	Memory access to unmapped address	Maps memory, swaps pages, or signals SIGSEGV	Virtual memory, demand paging, COW
Divide by Zero	Division with zero divisor	Raises exception, typically signals SIGFPE	Exception handling, process signals
Breakpoint	Debug instruction	Transfers control to debugger	Debugging mechanisms, ptrace
Invalid Opcode	Executing undefined instruction	Typically signals SIGILL	Instruction set validation, emulation
Protection Fault	Privilege violation	Signals SIGSEGV or terminates process	Memory protection, security mechanisms

- Common syscalls

Linux System Calls Reference

dup()

- **Parameters**: None
- **Returns**: The new file descriptor
- **Purpose**: Creates a copy of an existing file descriptor, using the lowest available number
- **Error Handling**: Returns -1 on error, check errno for details

dup2(old_fd, new_fd)

- **Parameters**:
    - old_fd: The original file descriptor to duplicate
    - new_fd: The specific file descriptor number you want to use for the duplicate
- **Purpose**: Creates a copy of old_fd but lets you choose which number to use (new_fd)
- **Behavior**: If new_fd is already open, it closes it first
- **Error Handling**: Returns -1 on error, check errno for details

fork()

- **Parameters**: None
- **Returns**:
    - In parent process: Child's PID (positive number)
    - In child process: 0
- **Purpose**: Creates a new process by duplicating the current one
- **Error Handling**: Returns -1 on error, check errno for details

exec family (execl, execv, execle, execve, execlp, execvp)

- **Parameters**: Varies by variant, but generally:
    - Path to the program to execute
    - Arguments to pass to the program
    - Environment variables (in some variants)
- **Purpose**: Replaces the current process image with a new one
- **Error Handling**: Returns only if an error occurs (returns -1), check errno for details

pipe(fd[2])

- **Parameters**: An array of two integers that will hold the file descriptors
- **Returns**: 0 on success
- **Purpose**: Creates a unidirectional data channel for interprocess communication
    - fd[0] is for reading from the pipe
    - fd[1] is for writing to the pipe
- **Error Handling**: Returns -1 on error, check errno for details

wait(status)

- **Parameters**:
    - status: Pointer to store the exit status of the child

- **Returns**: PID of the terminated child
- **Purpose**: Waits for any child process to terminate
- **Error Handling**: Returns -1 on error, check errno for details

waitpid(pid, status, options)

- **Parameters**:
    - pid: Specific child PID to wait for (or special values for groups)
    - status: Pointer to store exit status
    - options: Control behavior (e.g., WNOHANG to not block)
- **Purpose**: Waits for a specific child process to change state
- **Error Handling**: Returns -1 on error, check errno for details

kill(pid, signal)

- **Parameters**:
    - pid: Process ID to send signal to
    - signal: Signal number to send (e.g., SIGTERM, SIGKILL)
- **Purpose**: Sends a signal to a process or group of processes
- **Error Handling**: Returns -1 on error, check errno for details

getdents(fd, dirp, count)

- **Parameters**:
    - fd: File descriptor of an open directory
    - dirp: Buffer to store directory entries
    - count: Size of the buffer
- **Purpose**: Reads directory entries from a directory file descriptor
- **Error Handling**: Returns number of bytes read or -1 on error

chmod(path, mode)

- **Parameters**:
    - path: Path to the file
    - mode: New permissions (e.g., 0644)
- **Purpose**: Changes file permissions
- **Error Handling**: Returns -1 on error, check errno for details

getpid()

- **Parameters**: None

- **Returns**: Current process ID
- **Purpose**: Gets the process ID of the calling process
- **Error Handling**: Always succeeds

clone(fn, stack, flags, arg, ...)

- **Parameters**:
    - fn: Function to be executed in new thread/process
    - stack: Memory area for new stack
    - flags: Controls behavior (process vs thread, sharing resources)
    - arg: Argument passed to the function
- **Purpose**: Creates a new thread or process with more control than fork()
- **Error Handling**: Returns thread ID or -1 on error

mmap(addr, length, prot, flags, fd, offset)

- **Parameters**:
    - addr: Suggested starting address (usually NULL to let system decide)
    - length: Length of the mapping in bytes
    - prot: Memory protection flags (READ, WRITE, EXEC)
    - flags: Controls mapping type and behavior
    - fd: File descriptor (-1 for anonymous mapping)
    - offset: Offset in the file
- **Purpose**: Maps files or devices into memory or creates anonymous memory
- **Error Handling**: Returns MAP_FAILED on error, check errno for details

sbrk(increment)

- **Parameters**:
    - increment: Bytes to increase/decrease program break by (can be negative)
- **Returns**: Previous program break address
- **Purpose**: Adjusts process data segment size (grows/shrinks heap)
- **Error Handling**: Returns (void *)-1 on error, check errno for details

brk(addr)

- **Parameters**:
    - addr: New program break address (end of data segment)
- **Purpose**: Sets the exact end address of the process data segment
- **Error Handling**: Returns 0 on success, -1 on error

open(pathname, flags, mode)

- **Parameters**:
    - pathname: Path to the file
    - flags: How to open (O_RDONLY, O_WRONLY, O_CREAT, etc.)
    - mode: Permissions if creating (e.g., 0644) - only needed with O_CREAT
- **Purpose**: Opens a file and returns a file descriptor
- **Error Handling**: Returns -1 on error, check errno for details

read(fd, buf, count)

- **Parameters**:
    - fd: File descriptor to read from
    - buf: Buffer to store read data
    - count: Maximum bytes to read
- **Purpose**: Reads data from a file descriptor into a buffer
- **Error Handling**: Returns number of bytes read or -1 on error

write(fd, buf, count)

- **Parameters**:
    - fd: File descriptor to write to
    - buf: Buffer containing data to write
    - count: Number of bytes to write
- **Purpose**: Writes data from a buffer to a file descriptor
- **Error Handling**: Returns number of bytes written or -1 on error

close(fd)

- **Parameters**:
    - fd: File descriptor to close
- **Purpose**: Closes an open file descriptor, freeing resources
- **Error Handling**: Returns -1 on error, check errno for details

lseek(fd, offset, whence)

- **Parameters**:
    - fd: File descriptor
    - offset: Number of bytes to move
    - whence: Starting position (SEEK_SET, SEEK_CUR, SEEK_END)
- **Purpose**: Repositions the file offset within a file

- **Error Handling**: Returns new offset or -1 on error

stat(pathname, statbuf)

- **Parameters**:
  - pathname: Path to file to examine
  - statbuf: Pointer to struct to store file information
- **Purpose**: Gets file status information (permissions, size, timestamps)
- **Error Handling**: Returns 0 on success, -1 on error, check errno for details
-
- Attacks
- Buffer overflow attack
- Overflow a buffer, like rx buffer, so that whatever overflows modifies memory next to it. This can be kernel or user space memory, and if attacker knows that memory is there, then can plan ahead and write exactly what they want there to change the OS in their favor
- Lib attack
- If buffer is/near on stack, can modify return address
- This return address tends to be near the local variables/buffer
- Attacker would do this:
  - Current function is vulnerable. Does not handle input correctly. Could limit it with get to fix, etc.
  - Find the address of the `system()` function in the `libc` library.
  - Find the address of `/bin/sh` in the process's memory (this might be stored in a static buffer or the environment).
  - Use a buffer overflow to overwrite the return address of the current function to the address of `system()`.
    - Or could jump to any other page on stack and user can inject code ther with buffer overflow
  - Want to run system so can call other shell cmds with access that the vulnerable function has
  - Provide the address of `/bin/sh` as the argument to `system()`.
  - When the function returns, it jumps to `system()` and executes `/bin/sh`, giving the attacker a shell.
    - Attacker could also continue and overwrite the return address of system() to call other desired functions if system is not implemented right
- Ways to fix
  - Use memory safety languages like java or c++
  - Compiler flags like stack canaries - random value on stack right before return address, and if this is modified, then buffer overflow is being attempted. Abort function
  - Just check input size and use limits with func like get

    - Aslr
    - Nx stack - memory pages on stack are not executable so user can't inject code and have it be run
- Fork bomb
- Repeatedly call fork to create ne processes; new processes created exponentially
- Classic ver: :(){ :|:& };:
    - Defines function : that calls itself in background and that function does the same thing
- This quickly consumes cpu, memory, PIDs/table entries
- Prevents legit processes from running
- Requires hard boot to restart
- Defend against this with cgroups, user/process limits (ulimit)
- ## Buffer vs cache in linux
- Buffer vs cache - buffer is for storing data in memory before transferring from location x to y (mem to io, vise veersA). Also used for batching and when producer/consumers have different processing times. fifo
    - Cache is to remember readf rom disk in memory to not read from disk again. Usually lru
    - Buffers used for block transfer while cache used for files
- Before linux 2.4, page cache and buffer cache were different. Buffer cache used to cache blocks at block layer after they were flushed from page cache. This meant that there was data duplicated twice: page cache and buffer cache. Now writes just go through the page cache, and are flushed to disk
    - Buffer cache exists to store metadata and raw block io
- Caches in vmstat
    - Page cache
    - Kernel data structure caches
- Buffers in vmstat
    - Raw block
    - Buffer heads (metadata) for those blocks
        - Journaling buffers (like in ext4)
- Buffer cache and page cache are unified now after 2.4
- https://www.quora.com/What-is-the-major-difference-between-the-buffer-cache-and-the-page-cache-Why-were-they-separate-entities-in-older-kernels-Why-were-they-merged-later-on
- https://stackoverflow.com/questions/3192579/buffer-and-cache-difference?noredirect=1&lq=1
- ## Free vs vmstat
- Free has available, which includes reclaimable, while free's/vmstat's free doesn't
    - This is more accurate
- Free's cache, buffer includes slab allocator's caches and kernel structures (inodes, etc.) while vmstat's doesn't
- Free has used, which is total - cache - free - buffer

- ## Swapped cache
- Is optimization to make swap reads faster if it was recent
    - Decreases under extreme memory pressure; isn't freed up immediately like other caches might under memory pressure
- Memory swapped out and is in disk -> store copy in swap cache -> if accessed again recently page fault, go to swap cache. Saves trip to disk.-> remove from swap space/cache, put into ram
- Check size in vmstat -s
- ## Types of ipc
- Regular pipe
    - Unidirectional, related processes (shares ancestor), not persistent (dies after no one uses it)
- Sockets
    - Bidirectional, unrelated processes
- Named pipes
    - unidirectional, unrelated processes, persistent
    - Requires more coordination; needs at least one reader
- Message queues
    - Unidirectional, unrelated processes, persistent, dumps messages in pool that is read eventually
    - Asynch
    - More overhead to manage
- Shared memory
    - Avoids overhead to copy data over to buffers and context switches to write to it (unlike with buffers)
    - Butr equires synchornization
        - Can save race condition by sharing memory for files, library

- Af_unix - domain sockets. Are basically bidirectional, supports unrelated processes. Like named pipe but created as socket and is bidirectional. Can send file descriptors.
        - More overhead than shared memory
        - Need socketpair to connect to
    - Socketpair
        - Bidirectional, related processes, local
        - Not persistent
        - Is a connection between domain sockets
- ## User process segment memory - low to high
- text/code
- Initialized
- Unitialized (.bss)
- Heap (up)
- Stack (down)
- Mmap files and shared libraries

- Kernel
- Page guards for stack overflow
- ## Security
- ### Kaslr
- Kernel address layout space randomization
    - Helps randomize kernel memory to prevent attackers from overwriting buffers, etc. to overwrite a specific part of memory
- The base address where the kernel is loaded.
- The addresses of functions, data structures, and certain kernel modules.
-
- ### Aslr
- User address space layout randomization
- User stack, heap, shared libraries, memory mapped regions are randomized
- ### Cgroup vs etc/security/limit.conf vs ulimit
- For all, if go over memory limit -> killed. Anything else -> get error (files, cpu, etc.)
    - Cgroup - can set hard vs soft limits. If go over hard for memory, oom killer
    - Cpu and io usually throttled
- Cgroup - is hierarchical group system to assign limits where children groups have same properties as parent. Thus, any process spawned in a group inherits properties
    - sudo cgcreate -g memory:/my_cgroup
        - Create group under specific resource subsystem
    - sudo cgset -r memory.limit_in_bytes=1048576000 /my_cgroup
        - Set memory limit
    - sudo cgclassify -g memory:/my_cgroup 12345
        - Add process to group
    - cat /sys/fs/cgroup/memory/my_cgroup/memory.usage_in_bytes
        - Check memory used by cgroup
    - If goes over limit, oom killer sets in
        - Set this with
            - sudo cgset -r memory.oom_control=1 /my_cgroup
    -
    -
- etc/security/limit.conf
    - Can set soft, hard limits for user for specific types
    - bob   soft   nofile   1024 # Soft limit for open files
    - bob   hard   nofile   4096 # Hard limit for open files
    - bob   soft   nproc   100 # Soft limit for user processes
    - bob   hard   nproc   200 # Hard limit for user processes
    -
    - Can also edit etc/security/limit.d/USER.conf for limit file per user
- Systemd for systemd initiated processes
    - In unit file of service
- Sysctl for kernel limits (like max pid or swappiness)

    - kernel.pid_max
- Ulimit
    - Limits processes spawned form curr shell
        - -f: Maximum file size
        -
        -
        - -m: Maximum physical memory (memory usage)
        -
        - -s: Maximum stack size
        -
        - -u: Maximum number of processes
        -
        - -v: Maximum virtual memory (address space)
        - -l: Maximum locked-in memory
        -
        - -c: Core dump size
        -
        - -n: Maximum number of open file descriptors
        -
        - -i: Maximum number of signals
        -
        - -x: Maximum number of file locks
        -
        - -t: Maximum execution time for a process
        -
        - -a: Show all limits for the current shell session
- Acl, Namespaces, and cgroups
- Acl - change permissions for specific users for specific files (FILE PERMS)
- Namespaces - separate resource scope (mounts, processes, etc.)
- Cgroups - groups to aggregate users to change limits for

Here’s a quick cheat sheet on **ACLs**, **Namespaces**, and **Cgroups** tailored for OS interviews:

ACL (Access Control Lists)

- **What it is**: A mechanism for defining more fine-grained permissions for files and directories compared to traditional Unix file permissions.

- **Key Concepts**:

    - ACLs allow multiple users or groups to have different permissions (read, write, execute) on a file/directory.

    - **Access control entries (ACEs)**: Each entry specifies the permissions for a particular user or group.
    - ACLs are stored in extended file attributes.
- **Commands**:
    - **View ACL**: `getfacl <file>`
    - **Set ACL**: `setfacl -m u:<user>:<permissions> <file>`
    - **Set default ACL**: `setfacl -d -m u:<user>:<permissions> <directory>`
    - **Remove ACL**: `setfacl -x u:<user> <file>`
- **Use Cases**:
    - Shared file systems, where different users need varied permissions.
    - Granular permissions management in multi-user environments.

Namespaces (Linux)

Namespaces provide process isolation by creating separate instances of global resources such as networking, process IDs, and filesystems.

A namespace is a scope used to isolate system resources. It provides a restricted view of certain resources such that a process can only see and interact with what's inside that namespace. They provide isolation for containers so that way it's like they have their own OS, although they still share kernel and hardware. containers are apps packaged with their dependencies that can be sent anywhere and run

- **Types of Namespaces**:
    - **PID Namespace**: Isolates process IDs, allowing processes in different namespaces to have the same PID.
    - **Mount Namespace**: Isolates filesystem mounts, enabling different views of the filesystem.
    - **UTS Namespace**: Isolates hostname and domain name.
    - **IPC Namespace**: Isolates inter-process communication resources (e.g., message queues, semaphores).
    - **Network Namespace**: Isolates network interfaces, IP addresses, routing tables, etc.
    - **User Namespace**: Isolates user and group IDs, allowing a process to have different user IDs inside a namespace.
    - **Cgroup Namespace**: Isolates cgroup hierarchies, making each namespace have its own view of cgroups.

- **Commands**:

- **Create new namespace** (e.g., PID namespace):
  unshare --pid --fork --mount-proc bash
    ○
- **View namespaces** (e.g., using `lsns` command):
  lsns
    ○
- **Use Cases**:

    - Containers: Namespaces provide isolation for processes running in containers.
    - Process isolation: Running a process with isolated resources (networking, filesystem, etc.) without affecting the host system.

Cgroups (Control Groups)

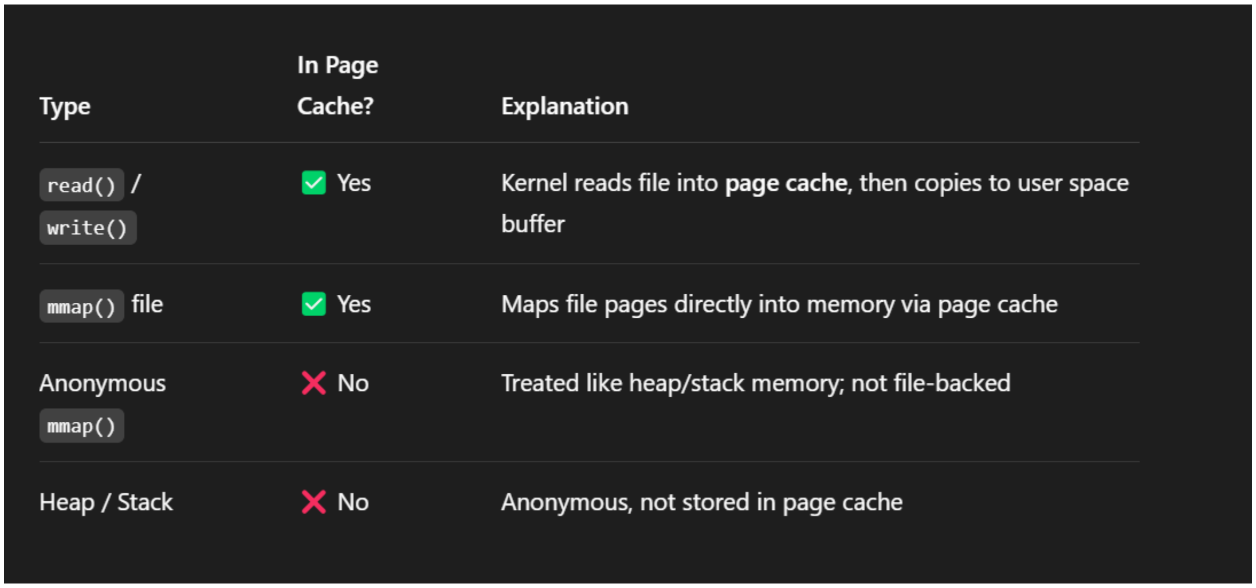
Cgroups allow you to allocate resources (CPU, memory, I/O) among groups of processes, which is useful for limiting and prioritizing resource usage.

- **Key Concepts**:

    - **Hierarchies**: Cgroups organize processes into hierarchies for resource management.
    - **Controllers**: The mechanisms that manage resources in cgroups. Key controllers include:
        - **cpu**: Limits CPU usage.
        - **memory**: Limits memory usage.
        - **blkio**: Limits block device I/O.
        - **cpuset**: Assigns CPUs and memory nodes.
        - **devices**: Controls access to devices.
- **Commands**:

- **Create cgroup**:
  mkdir /sys/fs/cgroup/cpu/my_cgroup
    ○
- **Set CPU limit for a cgroup**:
  echo 50000 > /sys/fs/cgroup/cpu/my_cgroup/cpu.cfs_quota_us
    ○
- **Add process to cgroup**:
  echo <pid> > /sys/fs/cgroup/cpu/my_cgroup/cgroup.procs
    ○

- **Set memory limit for a cgroup**:
  echo 100M > /sys/fs/cgroup/memory/my_cgroup/memory.limit_in_bytes
    -
- **Use Cases**:
    - **Containers**: Cgroups are crucial for limiting resources (CPU, memory, etc.) for containerized applications.
    - **System performance tuning**: Cgroups allow you to isolate and limit resource consumption of specific applications or services.
    - **Resource allocation for multi-tenant systems**: Ensuring fairness and avoiding resource hogging in multi-user systems.
-
- ## If have time - nfs
- `
- ## What uses page cache



Type	In Page Cache?	Explanation
`read()` / `write()`	✅ Yes	Kernel reads file into **page cache**, then copies to user space buffer
`mmap()` file	✅ Yes	Maps file pages directly into memory via page cache
Anonymous `mmap()`	❌ No	Treated like heap/stack memory; not file-backed
Heap / Stack	❌ No	Anonymous, not stored in page cache

-
- Stack and heap give zeroed page (and mapping) from p allocator - minor page fault
- Mmap / files do demand paging (only map but upon page fault load into page cache) - major page fault
- ## Mapping fd's
- Dup2's replace new fd with copy of given old fd
- Dup2 copies old into new fd
- Dup2 overwrites new if exists
- Dup copies old into lowest avail fd
- 1> is output (overwrite)
- > is input
- 2> is err
- >> is append

- ## Load avg + idle calc
- load avg: decaying exp average of unint/runnable/runn processes: avenrun[i] = avenrun[i] * exp + n * (1 - exp)
- The kernel increments a counter (e.g., cpuacct.usage_idle) whenever the CPU is in the idle loop (i.e., it’s doing nothing).
  - This is stored as a cumulative number of jiffies since boot.
  - %idl = idle jiffies / total jiffies
- # Short misc
- ## Interrupt masking
- Happens at two levels:
  - Io apic in redirection table
  - Local apic in local vector table
- Mask based on priority number
  - Technically kernel can do so with interrupt flags in cpu too
  - Task priority register also masks too (lower not looked at)

The **I/O APIC** has a **Redirection Table**

- Each IRQ entry has a **mask bit**
  - If set → interrupt **will not be delivered**
  - Used to mask **individual hardware IRQs**

The **Local APIC** also has masking ability via **Local Vector Table (LVT)** entries

- You can mask timers, LINT0/LINT1, etc
-
- ## Irq priority
- Devices (like drivers) are hard coded to be connected to interrupt controller with irq line
  - Old school (PIC era): Each device got a physical IRQ line (a wire) connected to a Programmable Interrupt Controller (PIC).
  - Modern systems (APIC, MSI, MSI-X): Devices now signal interrupts by performing a memory write, not by toggling a wire.called message signaled interrupt (msi)
    - Into apic memory associated with device, and what it writes is interrupt vector number
- Interrupt flow: io apic -> local apic -> kernel
  - Io apic - global apic. Has redirection table, one entry per irq line: interrupt number, delivery mode, destination, mask bit
  - Io apic decides which cpu to send to. Based on one of these:

        - Hard coded (certain irqs to x cpus)
        - Least busy
        - Logical groups
- Pic - programmable interrupt controller; apic is just advanced pic
    - Picks priority bby masking same level or lower (preventing itself + lower from interrupting). Linux uses tpr task priority register to hold what value to be less than to have priority to interrupt curr interrupt (higher priority)
        - 256 bit register - interrupt request register (irr), where each bit represents a given interrupt (tehre's 256 total). This is the queue
        - In service register - same thing as irr but instead of queue to be serviced, these are in service
    - Also picks specific hardirqs first, like network; priorities are hard coded for irq lines. Just starts lower then goes up and picks if not in mask or isr
        - Usually just goes through irr in order from 0 to 256 and sees if interrupt is queued
    - Each interrupt creates new stack frame on interrupt stack
- In other os, higher interrupt vector numbers == higher priority; hard coded beforehand
    - Uses priority queue (possibly red black) + preemption of lesser priority
    - Or hard coded into priority registers
- Apic assigns interrupts to cpus
    - Based on numa (some may be closer to requried mem), pinning (cpu affinity), balance based on cpu load (distribute evenly)
        - Irqbalance Daemon does this
- Linux doesn't use this. It uses soft/tasklets vs hard vs workqueues
    - Hard is top half; must be handled immediately
    - Softirq is deferred scheduled by hard, run later. In interrupt context
        - Softirqs have their own priority
    - Tasklet less priority than softirq
    - Work queues are the least priority. In user context; can read/write files and then be interrupted
- Kworker thread (not daemon, which are user processes in bkgrnd) handles soft/work queue in background
- Interrupt controller then assigns interrupt of highest priority to cpu, what vector number to use, edge or level trigger type
- Eoi given by os to apic when done with a specific interrupt
- ## How strace works
- Strace is parent; child is cmd u are trying to trace
- Strace intercepts all syscalls (receives notif). Records sys call, argos. Also return values
- ## Vm vs container vs lightweight vm
- Vm has stronger isolation; is different os
    - Hyperviser supervises. Vm specific optimizations exist
- Namespace shares system os
- Namespace faster/less overhead, smaller + easier to port

- Namespace + containers allow for easy development + deployment + including dependencies
- Lightweight vms are in the middle bc bare min simul of vm
- ## Making swap file
- sudo fallocate -l 2G /swapfile
    - Or dd sudo dd if=/dev/zero of=/swapfile bs=1M count=2048
- Set correct permissions (chmod 600)
- Sudo mkswap - format it for swap space
- Sudo swapon - enable it as swap file
- Sudo swapoff - turn it off
- ## Swap partition vs file
- Partition - space in disk reserved for swap
    - Faster; no file system overhead
    - Since isn't file, avoids fragmentation issues
        - Files can be split up once they grow larger in disk, causing external fragmentation
    - Must repartion disk to change size
    - Can only be used for swap; some space could go to waste
- File - file for swap
    - Easier to set up
    - Can adjust size or add more
    - Is slower
    - Fragementaiton due to file system
    - Comeptes with rest of disk
- Can put both on different disks
- ## Clearing cache
- `/proc/sys/vm/drop_caches` (clears all cache).
- `vmtouch -e` (evicts specific files from cache).
- ## Locking memory

Making Memory Unevictable

- **mlock(), mlock2(), mlockall()** prevent kernel from evicting specific pages.
- ## Journal for filesystem backup
- If filesystem uses journaling, writes what change ill happen beore file system is modified
- Then wehn complete, journal marked as complete
- During crash, instead of full restore with fschk, can just replay latest completed actions and discard any incompleted actions
- Cons:
    - Extra writes, disk space
- Alternative: COW
    - Instead of editing, make copy, edit copy, then delete old

        - Atomically editing to avoid corruption
    - Btrfs, zfs

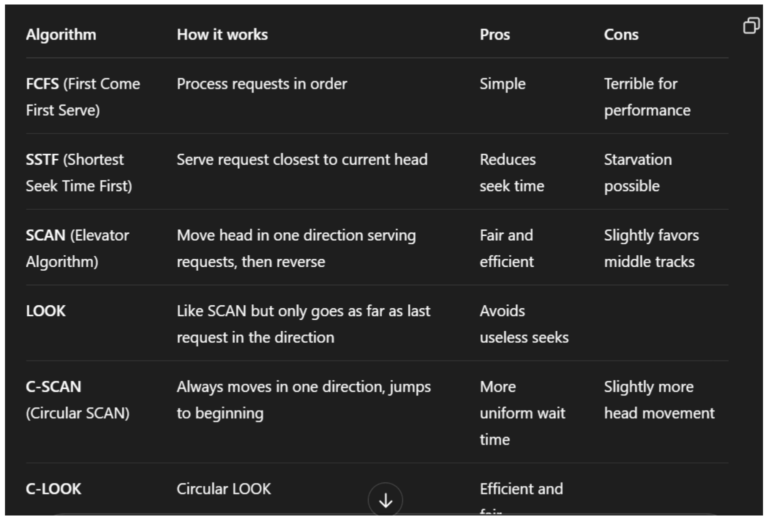
- ## High io vs high throughput
- High io can mean high operations persecond – either lots of reads/writes that are small or one large one being broken up
    - Means disk is being used a lot
- May not be bad unless is increasing latency, decreasing reads or write amount, increasing queue
    - Usually higher iops is shown with higher latency though
        - Higher iops is associated with random, not sequential
        - Lower iops is associated with sequential
-

- ## Ttl traceroute
- traceroute measures how many hops from routers it takes to get to destinatino by using icmp echo requests. ip header has tll, and each intermediaryy router subtracts ttl by one until it becomes 0. if it is not destination, will send icmp timeout back. then source will send another but with ttl + 1. it will repeat this until it doesn't get timeout so that way it finds out the exact number of routers traveled traditionally it is done with icmp, linux does udp on high ports, then tcp to avoid firewalls
    - Sends icmp timeout if ttl 0
    - If dest then sends icmp request reply

- ## Ssd vs hdd random access
- Both are slower than ram since can't just lookup location
- Ssd is better since parallel channels dies planes allow access in paralllel. Hdd requires heads to move to seek and disk rotation, which is physically slower
    - Parallelism is better with sequential (most likely cuz can split up)
- Ssd has wear leveling, which means needs ftl, to find out physical location of logical block. Also writing to electrical charges + fixing errors is much slower

- ## Ssd vs hdd
- Ssd - wear leveling + erase in blocks/write to empy pages (leads to garbage collection, FTL + movement), faster
    - Wear leveling means have x number of writes per cell. This is due to electrons needing to move through oxide tunnel for storing states, which wears down oxide tunnel
    - Can read by page. Can only erase by blocks, not pages. Can only write to empty pages. To overwrite, must erase block. Thus can only really write in blocks
        - Read the entire block
        - Modify the data in controller memory
        - Erase the entire block
        - Write back all the pages with the modified data
    - Has pages and blocks (pages smaller)
        - Pages are 4kb-16kb; blocks are 128-156 pages (256kb-1mb)

            - Ssd blocks are bigger than hhd blocks
    - Need to handle garbage collection, too, unlike HHD
        - Need this to delete in pages (which can't do; can only delete on blocks)
        - Basically deleted page is marked as stale. OS might send TRIM cmd signal or this to happen more efficiently
        - Block is mix of invalid/valid. Once reaches threshold, valid moved to new location, ftl mapping updated, but don't move deleted. Now this block is free
    - Requires device mapping (flash translation layer); allows for data movement due to wear leveling
        - Logical blocka ddresses to nand flash levels
        - Dynamic + changing due to wear leveling, garbage collection
    - Enables parallelism with channel,dies,planes
        - channel - data paths connecting controller to nand flash packages
        - Die - nand flash chips within a package
        - Planes - subdivided units that can act independently in die
        - Basically, data is sent through multiple channels, and these are assigned to dies, then planes. More is sent through channel and more assigned to empty dies/planes
            - Basically limited bandwidth by channel for sending, but work can be spread across many dies/planes after, causing for massive parallelism
    - Ssd handles wear leveling by:
        - Handling write amplification, which is having more writes than user requested due to moving data around. This is caused by having to erase blocks to write to empty pages
            - Wear assignment - write to newer blocks; move heavily accessed data to newe blocks occasionally
            - Rewrites - marks for deletion thenw rites elsewehre
        - Trim cmd to mark for deletion (othewise just stays in disk) / Garbage collection (prevents overwriting)
- Hhd - has elevator time to select disk, slower
    - Disks are in rows. Hhd handles seek time by:
        - Disk scheduling algorithms, oftentimes merging adjacent requests + adding all requests that need same seek area
        - A cylinder consists of all rows that are at the same distance from the center, across all disks
        - Seeker moves across all disks, going from center, like multi layered record player)
            - Like comb; each read/write seeker is tooth
        - Rotation + seek to find proper block
            - Disks are called platters
    - Has sectors and blocks (sectors smaller)
        - 512 b or 4 kb

        - Blocks are 4-64 kb
    - Can directly overwrite any area
    - Platters are the disks
    - 

Algorithm	How it works	Pros	Cons
FCFS (First Come First Serve)	Process requests in order	Simple	Terrible for performance
SSTF (Shortest Seek Time First)	Serve request closest to current head	Reduces seek time	Starvation possible
SCAN (Elevator Algorithm)	Move head in one direction serving requests, then reverse	Fair and efficient	Slightly favors middle tracks
LOOK	Like SCAN but only goes as far as last request in the direction	Avoids useless seeks	
C-SCAN (Circular SCAN)	Always moves in one direction, jumps to beginning	More uniform wait time	Slightly more head movement
C-LOOK	Circular LOOK	Efficient and	

    - Fifo, sjf, Scan, look, c-series
- Posix vs flock (posix better)
- Posix can lock bytes while flock is whole
- Posix (fcntl) associated with pid, inode instead of file descriptor unlike flock
- Posix can detect deadlock
- Disk info

Below is a concise summary covering the key concepts:

File System & Storage Units

- **Logical Blocks:**
    - Abstraction used by the file system to represent disk space.
    - Smaller than a page and mapped to physical blocks with an offset (e.g., logical block 1 might map to physical block 123).
- **Physical Blocks:**
    - Chunks of disk space (often comprising 128–256 pages or more).
    - The smallest unit the OS deals with at the storage device level on SSDs and HDDs.
- **Sectors:**
    - Traditional smallest unit on disk, originally 512 bytes; modern drives use 4 KB sectors.

    - Disk controllers translate logical block addresses to these physical sectors.

Mapping, Translation, and Controller Functions

- **Logical-to-Physical Mapping:**
    - Managed by the disk controller using a mapping table.
    - Enables **wear leveling** (spreading writes evenly), fault tolerance (remapping damaged areas), and abstraction (allowing physical reordering without changing logical addresses).
- **HDD Considerations:**
    - **Throughput Formula:** Sectors per track × sector size × (rpm/60).
    - **Performance:** Best when reads are sequential to minimize seek times.
    - **Sector Zoning & Elevator Seeking:** Outer tracks hold more sectors; I/O requests are reordered and merged to minimize mechanical delays.
    - **Error Correction:** ECC exists per sector to maintain data integrity.
    - **SMR Drives:** Use narrow tracks for high density; best for reads but require rewriting adjacent tracks on writes.
- **SSD Considerations:**
    - **NAND Flash Memory:** Most SSDs use NAND flash (often with DRAM) with multiple channels for parallelism.
    - **Flash Translation Layer (FTL):** Maps logical block addresses to physical NAND pages or blocks.
    - **Page vs. Block:**
        - Data is read/written per page (typically 4 KB–64 KB).
        - Erases occur in larger blocks (256–512 KB), meaning writes often involve copying valid data, erasing, and rewriting the entire block.
    - **Performance Optimizations:**
        - SSDs are optimized for 4 KB reads and larger, batched writes (e.g., 1 MB) to reduce internal fragmentation.
        - **TRIM commands** inform the SSD which blocks are free, helping reduce write amplification.
        - **Overprovisioning** provides extra space to delay garbage collection.
    - **Cell Types:**
        - SLC offers best endurance (~1M cycles), MLC is common (1,000–10,000 cycles), while TLC and QLC offer higher density but lower endurance.
    - **Aging and Latency:**
        - As NAND wears out, latency can spike (due to ECC and retries).
        - Fragmentation of the FTL's block map may increase garbage collection overhead and latency.

This summary outlines how file systems use logical blocks to abstract storage, how disk controllers map these to physical sectors/blocks, and contrasts HDD and SSD characteristics along with techniques (like wear leveling, TRIM, and overprovisioning) that optimize performance and lifespan.

-
- Kuber
- so pods are just instances and nodes are just servers. containers are just dependencies plus my app. controller used to control scaling (remove/add pods)