Compiled from OS Notes 1, 2

- Used these sources:
    - Kerrisk, Michael - The Linux programming interface a Linux und UNIX system programming handbook (2018, No Starch Press)
    - UNIX and Linux System Administration Handbook, 4th Edition
    - Linux Kernel Development 3rd Edition - Love - 2010
    - Andrew S. Tanenbaum - Modern Operating Systems
    - Partially: [Addison-Wesley Professional Computing Series] W. Richard Stevens, Stephen A. Rago - Advanced Programming in the UNIX Environment
    - Brendan Gregg - Systems Performance_ Enterprise and the Cloud (2020, Pearson)
    - https://pages.cs.wisc.edu/~remzi/OSTEP/

Os

General walkthrough

- Where to find logs
    - sys/var/logs for general
    - Dmesg | tail -n 50 - 50 recent kernel logs
    - Auth: /var/log/auth.log
    - Journalctl -b (systemd startup issues)
        - Journactl can be used to find any logs (kernel, systemd, etc.)
        - --since allows to find since dateX
        - -u to search for keywords
        - -xe for most recent
- Bootup
    - Turn on, load bios, which initializes hardware/motherboard and testing of it
    - Load mbr from disk; first 512 bytes have bootloader code. Next 64 bytes are partition table and last 8 bytes are magic number (signature)
    - Load bootloader, which loads kernel
    - Kernel initilaizes memory management/its space, drivers, mounts, and systemd (init, pid 1)
    - Init system then starts networking, daemons, services, login prompts
        - Used to be called rdinit
- Linux scheduler:
    - Mid term - swapping, paging, killer
        - Is not a literal scheduler bit instead collection of mechanisms to limit and swap
        - Includes swapping, oomkiller, kdswap, cgroup memory limit
    - Long term - which jobs run (systemd, slurm, ulimit)
        - Is a literal scheduler for jobs to run at x time
        - Includes cron, systemd (long background) slurm, job limits (cgroup, user)
    - Short term - cpu allocation (cfs)

        - Is a literal scheduler for cpu. Linux uses cfs
    - Cfs - completely fair scheduler
        - Is a mix of least time used and priority so no starvation/lack of priority
        - Red black tree to store vruntime of active processes contending for cpu
        - Curr run: get leftmost node to get smallest vruntime
        - Then when another vruntime becomes smaller (leftmost is smaller than curr) then we preempt curr and let that one run
        - But what is vruntime? Vruntime is time run, where increments are weighted inversely by priority. This means higher priority (lower nice value), increment/time used in a given cycle is exponentially shrunken by priority
        - Not optimal for real time due to trying to balance time used + priority (priority not completely respected). Prioritizes fairness over efficiency, preempting cpu cound processes, making it take longer
    - Short term Types
        - First come first serve (no priority), not preemptive
        - Round robin (no priority)
        - Shortest job first (starves long jobs), not preemptive sometimes
        - Priority (starves low priority)
        - Multi level queue - have labels for priority and this puts each into a queue
            - Then each queue has its own scheduling (like real time, fifo, priority depending on needs)
            - Usually highest level has highest priority so those are run first
            - Good since adds more flexibility for scheduling (more types). Takes into account priority
            - However, it allows for starvation. Also more complex
        -
- Locking
    - Prevents race conditions (depending on write order, 2nd or 1st overwrites other deepening on latency) and reading after write and reading before write when it was supposed to be after
- Cpu architecture types
    - Smp - multiple cpu's share one mem. Needs synchroinization but now have failure tolerance + parallelism
    - Numa - non-uniform memory access - memory is partitioned. There are local ones to a cpu, and this is faster. Further ones are remote and take longer
- Cpu rings (kernel vs user)
    - Ring 0 (Kernel Mode): Direct hardware access.
    - Ring 3 (User Mode): Restricted; uses syscalls to request privileged operations.
    - Uncommon now:
        - ring 1 (Driver Mode):
        -
        - Often used by device drivers.

        - Provides some level of isolation from Ring 0, allowing the OS to run critical components (such as drivers) with elevated privileges while preventing them from having full access to the entire system.
        - Not widely used in modern operating systems, as many drivers run in Ring 0 for performance reasons, though this can pose security risks.
        - Ring 2 (Intermediate Mode):
        -
        - Used for less privileged operations than Ring 1, like device drivers or other low-level functions.
        - This ring offers more protection compared to Ring 0 but less than Ring 3.
        - Similar to Ring 1, it's not commonly used in most modern operating systems.
- Kernel ring buffer - temporary buffer that allows you to have last x logs
    - Does not contribute to buffer in vmstat
    - Kernel uses printk to print to buffer
    - Buffer is circular, with 2 pointers: read and write
    - Look at it with dmesg, or journalctl if you're using systemd for logging
    - Can write it to disk manually by redirecting dmesg > file.log
- Cpu state
    - Program Counter (PC): Tracks the current instruction being executed.
    - Registers: Includes general-purpose registers (like RAX, RBX in x86), stack pointer, and frame pointer.
        - Ultra fast memory but very expensive
    - Stack Pointer (SP): Points to the top of the stack.
    - Instruction Register: Holds the instruction currently being executed.
    - Processor Status Registers (Flags): Contains condition flags (e.g., zero flag, carry flag).
    - MMU State (if applicable): Page table base register, segment registers (on older architectures).
    - All this is put into pcb upon context switch.
- ram/Memory - volatile fast read/write compared to disk
    - Bits in page - permission, dirty bit (used to know if need to write back when evict), page number
    - Mmu - hardware that sits between cpu and physical memory and translates virtual addresses to physical ones
    - Memory protection - each process has own page table and thus virtual memory. Also mmu handles translation, so process can never get memory outside of its own.
    - Swaps
        - Memory pressure -> memory is running out
        - If memory falls below certain threshold, Kswapd swaps out least recently used memory to swap space in disk so it can be deleted in mem
            - For clean it just deletes. Dirty it swaps

                - This is better than flushing cuz if we have pressure we need to free up memory quickly
                - Therefore, writing may take too long depending on the file. It would need to figure out where it goes
                - This is just like a quick eviction policy into a known area, being the swap sapce
                - Allows for overcommitment and juggling more memory than have in ram
            - Swap files
                - Easier to change size of instead of partitions
                - If need to change size of partition, we got to repartition whole disk
                - Also can be smaller. Also can be easier to move/backup
                - However, they can cause fragmentation and wear on ssd's since partitions are automatically aligned and configured
                    - Aligned as in aligned with ssd erase blocks
            - Changing swap
                - Swappiness value - higher means swap more. Default is 60. Start swapping when memory is 60% full
                - cat /proc/sys/vm/swappiness
                - sudo sysctl vm.swappiness=10
                -
    - If free reaches 0, oom killer activates - just kills processes
        - Kills highest badness score, which is determined by priority (inverse. Root not likely), memory usage (directly proportional), manual
    - How kernel gives memory to a process
        - Process requests virtual memory size and kernel gives root table first, and this is split up into multiple layers, which is created as needed
            - However, number of layers is set and based on page size, original VM size
                - Number of pages = original VM size / page size
                - Num of entries per table = page size / Entry size
                - Page table size = page
                - Total entries = vm size / entry size
                - Num of levels is by choice, but we want to maximize num of levels such that required number of tables per table is not a fraction
                - Num of tables per level = total entries / num of entries per table

- V = number of bits in the virtual address
- O = number of offset bits ($= \log_2(\text{Page Size})$)
- T = number of bits used per level in the page table
- L = number of levels
- Then you want:

$$V - O = L \times T$$

    - Alt:
        - Can grow later
    - Dirty pages - pages that have been written to and need to be flushed back to disk to update disk. Done due to memory pressure, file closes, periodic
    - Page frame vs page - frame is physical; page is virtual. Page points to frame. Both tend to be same size
- Virtual vs physical memory
    - Virtual is set of fake addresses hat can eventually map to physical address that is given to a process. Allows for abstraction so process thinks it has contiguous memory meanwhile memory is spread out. Also prevents processes from reading each other's memory since they dont' know real address and kernel handles mapping
    - Physical - literal memory in ram
- Virtual memory and aslr
    - Address space layout randomization
    - Randomizes starting address of sections of a process's memory, like heap, stack
    - This way heap isn't always first or second. This makes exploits unreliable.
        - Ex: buffer overflow. It's when data is bigger than buffer. It causes data to be written beyond buffer to mess up other data. Random sections make this harder to know what you're messing up
- Virtual table
    - Resolution
        - Resolution: check tlb. If not there, then do page table walk. Go to path that builds up virtual address that need
        - Then look at permission bits and physical frame number. This is like the having a neighborhood street/block in the real world, and the virtual number is like the address. We know relative to the start of the block that we need the nth address
            - Less overhead since if we had full address that would overlap with virtual number (like less significant digits within range of block start to size of block start would be redundant)
        - Ram physical address = offset aka virtual page number offset (remaining after translation) + physical frame number
        - Check page cache. If in there, return. Otherwise, load from disk
        - Go to physical memory and get that if it was loaded in. but, if not loaded in (like had to create subtable/it just was not loaded), then we have page fault
        - Then we read from disk (slow)

        - Then we store in tlb and page cache. Both typically use lru
    - Page Cache - caches recently used pages
    - Usual page size - 4096 bytes in Linux
    - Permissions - each page has permission bits, which affects what process can do in physical memory (read, write, exe)
    - Multi vs single table
        - Single means we have VN : PFN. bad but simpler to implement. Bad because if table is large, need to allocate space for every single address in there
        - Multiple levels. Each non-bottom layer points to another table's physical address. With each one we get we go to that table and look. is better since if virtual memory is large, we just have root table and that's it. We then allocate as needed, allowing only a subset of virtual addresses to exist in table, decreasing overhead. May be slower since have to walk more to translate
            - Each level tends to be ~2 hexadecimal digits, and first level is first 2, 2nd level is 2nd two, etc. we use the nth 2 digits at the nth level to find where to go next (either a new table or actual physical address)
            - Lowest level still has frame number
- Each process has: stack (grows down), heap (Grows up), elf/code, bss (uninit), data, mmap, kernel sapce
    - Elf vs exe:
        - Elf is on linux, exe is windows
        - Elf has data, unint data, code, read only data
        - Is just binary to execute
            - Has been compiled from source code into object files, then linked into one with dynamic libraries and references set in assembly then binary. Code also adjusted since addresses have shifted after combining object files
- File load in chunks
    - File descriptor is start of file in memory so don't have to load all in. kernel keeps track of open files in table
        - This goes in open file
        - File descriptor is unique per process
            - Same file cuz same inode
        - File table exists for each process and system wide. System wide is for synchronization (regional locks)
    - Read in chunks with read function in high level language
        - This causes any next read to be loaded from disk (it knows it isn't in memory)
    - Mmap - map part of file into memory. Load on demand
    - Can load whole like this:
        - with open("file.txt", "r") as f:

        - data = f.read() # Loads the whole file into RAM
- Spooling
    - Is basically buffer with queue and scheduling. Helps with high load
    - Often found in pritners since have concurrent usage or prints
- Disk - nonvolatile storage. Consists of hdd (cheaper longer durability slower with actual disks and slow seek time) and sdd (opposite of hdd)
    - Buffer - stores io writes to disk/network to allow for batch processing so less trips to disk made
        - Minimizes overhead of system call/context switching
        - Mnimizes io wait time for a specific process/network round trip
- Io
    - Buffering
        - Happens in kernel space. Instead of immediately being written back to disk, file/mem saved in page cache
        - Process still waits until what it wrote is flushed (synchronous)
        - Flush so writes in batches/chunks (fsync)
            - Due to memory pressure
            - Periodic flush
            - File closes
        - Still have to wait if flushing and page cache is “under construction”
        - More efficient due to batch processing, but can be lost due to crashes
    - Direct - process requests immediate writeback instead of batch in buffer
    - Asynch
        - Kernel uses separate thread to perform io task, then sends signal/callback indicating is done
        - Process can continue
        - Call using libaio
        - May have data discrepancies if file read from was not written back to yet
    - Scheduling of io (not for disk) - BUFfER DOES NOT MERGE IO
        - Cfq - old. Allocates time slices and rotates between processes for i/o
        - Budget fair queueing - just gives budget of io to process. Larger budgets prioritized.
            - Best for overall/interactive
        - Multiqueue-deadline - used for RTOS
            - Best for ssd, predictable response times
            - Just is queue; preempt as needed
        - Noop - merge write requests (unlike buffer) then send to disk
            - Raid, ssd with hardware level scheduling
        - Kyber - low latency, high throughput. Pririoitizes reads and merges requests. Random reads prioritized due to doing well in ssd.
            - Nvme ssds, low latency
    - Scheduling within disk
        - First Come First Served (FCFS): Serves requests in the order they arrive.

        - Shortest Seek Time First (SSTF): Chooses the request closest to the current head position, minimizing seek time.
        - SCAN (Elevator Algorithm): Moves the disk arm towards one end and serves all requests in that direction before reversing direction.
        - C-SCAN (Circular SCAN): Similar to SCAN but once the end is reached, it moves back to the other end without serving requests in reverse order.
        - LOOK: Similar to SCAN, but the arm only goes as far as the last request in the current direction.
        - C-LOOK: Similar to C-SCAN, but the arm only goes as far as the last request in the current direction.
        - Weighted Shortest Seek Time First (WSS): Variation of SSTF with weights assigned to requests to prioritize certain requests.
    - Perf for disk
        - Seek Time: Time taken for the disk arm to move to the correct track.
        - Rotational Latency: Time taken for the disk to rotate to the correct position.
        - Transfer Time: Time taken to actually transfer data once the head is in position.
- File system and how it works
    - Each directory is a table of file name to inodes
        - Inode has file permissions, owner, size, time stamp, but no file name
        - Then each inode points to the file
    - Hard link - another entry in table tha tpoints to file. If all hard links die, file dies. Otherwise no die
    - Symbolic link - inode that has file path of what it is shortcut to. If file dies, this link becomes broken
    - Write ahead logging - Journaling
        - Keep log of data cahnges so can roll back changes to avoid corruption after crash
            - Helpful to recover buffer too
        - Ext4, XFS, and ReiserFS
    - Shadow paging
        - Updates are written to copy of page, then page table points there instead
        - Inefficient since need to keep on cleaning up stale pages and setting pointers
        - BETTER:
            - Cow in file systems -
                - Btrfs, ZFS, APFS
    - Mount
        - Entry point where existing file system will be attached
        - Use mount -t TYPE_FILESYSTEM DEVICE MOUNT_POINT
        - This is basically justa dding directory entry to attach point, and directory entry represents file system root dir
    - Ext4 details

        - Has journaling
        - Large file support
        - Extents for large files
        - Compatible with ext2,3
    - xfs details
        - Journaling
        - Doesn't allocate disk until write
        - Concurrent operations + large file system oriented
    - Btrfs
        - Snapshots (read only)
        - Checksumming to prevent corruption
        - Built in raid controllers and customizable mountable partitions within
- Networks
    - Networking stack (which part is in the kernel, which part is handled by userspace libraries, common syscalls, sockfs)
        -
    - Assign dynamic port for client connection
    - When packets come in, write to circular buffer using Direct memory access (meaning no cpu needed to do so). Circular buffer is to prevent overflow to maliciously overwrite other mem. Then, either throttled or system polls or packets queued for system to eventually pick up interrupt in slower fashion (since normal speed too fast). Sometimes coalesce the data packets based on time/number (just batches)
    - Then driver then kernel handles it and sees what server port it came from (tells protocol), finds corresponding socket. Since ips are unique, will map to specific process
    - Copies to process socket buffer. Process notified. Process can use it
- Fork and exec impl
    - Fork - parent is cloned exactly, with pcb, memory, file descriptors, etc. to make child. But child gets new pid
        - Copy on write is used - only make full copy if child changes it. Otherwise, share
    - Exec - replaces process image with new one, usually from binary. New file descriptors, etc.
        - Loads from elf (executable and linkable) a format type and replaces code, heap, stack and sets up arg/env variable passed into exec
- Kernel
    - Context switching
        - System call -> read ivt and pick corresponding interrupt function -> cpu saves user registers and pc and stack pointer and memory in pcb -> kernel executes func -> swap back/load
        - Between processes:
        - Same but if voluntary, no interrupt. Preempt -> timer interrupt

    - Handling interrupts - kernel catches, look up in ivt the interrupt to get relevant function, run that
    - Kernel vs user space
        - Kernel space is used by kernel for system calls/interrupts and keeping track of processes. Has higher privilege. Is separate from user (both cannot access each other)
            - Kernel space consists
        - User space is used by user processes
    - Main responsibilities of the kernel and init (remember to ask the questions, e.g., why do we need a kernel?)
    - Kernel space
        - Idt / ivt - interrupt descriptor table, maps interrupts to functions
        - Page table
        - Kernel stack per process
        - Device driver
        - Io buffers/caches
        - Compiled kernel code and data structures
        - Process table with pcb
- **System calls**
    - **Know key signals such as SIGTERM, SIGSTP, SIGCHLD, SIGKILL, SIGSEG, signal handlers, signal masking, default handlers, tracing signals)**
      w the 12 key system calls, how they are initiated, CPU protection rings, mode switch)
        -
    - **Interrupts (what events cause interrupts, interrupt context vs. thread context, how interrupts are executed, top half and bottom half, and a bit about interrupt masking)**
        - **Interrupt event examples**
            - Software errors
            - Hardware events
            - Network events
    - Interrupt controller
        - This assigns priorities to interrupts (can customize)
        - Hardware is built in to choose priorities, likely fifo if custom interrupt
    - **Interrupt masking**
        - Helps prioritize higher priority interrupts by disabling temporarily if lower priority than others
        - Also while interrupt is running, more of its source is disabled to prevent spam
        - In cpu there is special register called interrupt mask register
            - Setting a bit there disables the interrput that corresponds to that bit
            - Unsetting bit re enables interrupt
    - **Top half**

        - Is just the interrupt function. It is the time critical that is immediately done when interrupt occurs. It schedules the bottom half
            - It is the function passed into request_irq
        - Needs to tell hardware that interrupt received, save registers/pc/flags before context switch, sometimes disable further interrupts, then do specified interrupt work
    - **Bottom half**
        - Interrupt does important interrupt function related stuff then adds to backlog to do rest of it later
        - Interrupt context - can't sleep; not associated with process. Non preemptible. No user space access. Limited stack.
        - Process context - associated with process; can sleep
        - No longer exist: task queue, bh
            - Task queue used to be like work queue but allowed synchro between tasks
        - Types:
            - Softirq
                - Use if timing is critical, networks/block devices
                - Runs in interrupt context (interrupt handler)
                - Is fastest so is best for many threads
                - Static so must plan ahead
                - Low overhead so can scale well. Requires locks still for shared data
                - Handled by kernel
            - Tasklet
                - Dynamically created and easy to use
                - Cannot be interrupted
                - Runs in interrupt context (interrupt handler)
                - Handled by kernel
            - Work queue
                - Use for if timing is not critical
                - Runs in process context (when process is in control)
                    - Can sleep, allocate memory, wait for io
                - Can be preempted/scheduled unlike other 2
                - Can sleep, unlike other 2
                - Handled by kernel threads
- Security
    - How kernel prevents user from accessing kernel space directly
        - Page tables, randomized memory, permission bits in page, interrupts / system calls only, separate kernel / user space
    - System ahrdening - updating system and fixing vunerabilities (bugs)/exploits (loopholes)
- Ipc

    - Pipes are unidirectional while message queues are bi. Pipe is continuous stream so no fixed size while message queue has max size and discrete size
        - Pipes can block if nothing in there and tyring to read or full and try to write while message queue is just mailbox (dump and leave). Pipe like phone call almost
    - Pipes - created with pipe() syscall; usually anon between parent/child (related). They are effectively kernel buffers. One process writes to buffer. Other reads from it.
    - named pipes. Communication between unrelated processes. Same as pipe otherwise. Represented by file in /tmp/myfifo unlike regular pipe.
    - shared memory between processes. fastest. Created with syscall shmget(). Usually semaphores or mutex to prevent race conditions.
        - Is mapped to process vm with mmap. Has own section besides heap
    - message queues - msgget() system call, and messages are sent or received using msgsnd() and msgrcv(). Is FIFO with priority.
    - Sockets - network comms. Can be local but is like pipe but with more flexibility. Has buffer as well
- Shutdown
    - Shutdown cmd / systmctl poweroff
        - Systemd activates shutdown.target
    - Send sigterm to all processes; after timeout, then sigkill
        - Flushes all pending writes
    - Syncs io buffers and unmounts file system
    - Acpi (Advanced configuration and power interface) power off sequence
        - Motherboard shutdown, bios executes power off sequence
        - S0 (Working State) – System is fully on.
        - S1-S3 (Sleep States) – CPU is stopped, but RAM stays powered.
        - S4 (Hibernate) – RAM is saved to disk, and the system powers off.
        - S5 (Soft Off) – The system is completely off, requiring power-on to restart
- Misc
- Cpu affinity
    - Bind process / thread to a specific cpu
    - More likely that cpu's cache (l1, l2, l3. L3 is slowest) won't be invalidated since doesn't have to start on another cpu
    - Less overhead of moving task between cpu (loading states)
    - Better for virtual machines so that way only drains resources from one (heavily does so)
    - Bad because: less flexible for load balancing between cpus,
- Deadlock - cycle in dependency graph. When process A needs resource Y but process B has that, but process B needs resource X, which process A has
- Deadlock prevention - no holding. Have premption. You probably want mutual exclusion still to avoid race condition
    - Deadlock requirements: mutaul exclusion, process can hold and wait, no preemption of resources. Then, circular wait.

    - Linux does: avoiding nested locks, lockless most of the time (favor atomic), priority inheritance (inherit higher priority process dependency's priority if holding onto what need (atomic). This allows lower priority task to run more and eventually release lock). timeout of holding lock.only get lock if can get all locks at once.
    - Others have detection algorithms
- Locks (prevents race conditions)
    - How to choose locks: spin lock has low overhead but wastes lots of cpu when spinning for long time
        - Therefore, spin lock for short time
        - Mutex for long time and with sleep (but has higher overhead)
        - Semaphore for synchronization/not mutex but control max X using resource y
    - Upgrade lock -> read lock to write lock when needed (allows others to read while upgradee is still holding to read not write yet but once writ eit is like a normal write. Can only have one upgrade lock at a time. No other writes)
    - Read copy update lock - when update, make copy and update that one. Once there are no readers, then set og as copy
    - Spin lock - lock that while waiting you sleep check loop. Bad cuz consumes cpu
    - Mutex - mutual exclusion. Regular lock so only one can look at critical section at a time. Usually has a queue to obtain (sleeps and is blocked. Context switch occurs)
        - Read write locks so read lock can still read at same tiem but write lock cannot read at same time
        - Higher overhead due to queue and context switch to put to sleep (block) and then wake up
        - Front of queue is woken up
    - Critical section is section of code where race conditions can occur and thus a lock is needed to go there
    - LOCK CONTENTION IS CPU BOUND. High cpu usage means lock contention
    - There are read and write versions of semaphores and spin locks. Can have many reads but only one write holder
- Semaphores - less than x can use resource. controls num of resource and their access to not overload database or disk. Has counter that decreases by one every time one process or thread uses its resource. If over 0, can continue to let more in. has queue too. Binary semaphore is just mutex
    - Gives synchro too so that way you can control if group x goes before group y (size n). Can do grouping by prioritizing threads/processes (sorting)
        - Use counting semaphore to figure out how many threads have finished via signal
            - Need x signals (let's say 6, 3 per group and 2 groups. 3 per group due to having 3 workers in each)
        - Once syncrhonize, continue
    - Mutex semaphores can just make queue

    - Wait (p op) - decrements counter
    - Signal (v op) - increments counter
- Barriers - functions called that require all operations to finish before reach that function
    - Ex: all memory writes/reads to finish (smp mb)
    - Ex 2: all threads to finish (pthread barrier)
- Process vs thread
    - Process is instance of program and has more overhead, no shared memory
    - Thread is smallest running part of process and shares memory so faster, less overhead than process
    - In linux, they usethe same task struct
    - Coroutine vs serspace vs lightweight threads
        - Coroutines focus on cooperative multitasking within a single thread and are often used for I/O-bound tasks where tasks yield control during waits (e.g., I/O operations).
        - User Space Threads are managed by libraries outside the OS and can be either cooperative or preemptive, typically used for lightweight concurrency without OS overhead.
        - Lightweight Threads refer to a thread model with minimal overhead, either in user space or kernel space, and are often used to manage many concurrent tasks efficiently.
- Types of caching
    - Cache is set size of quick access items (due to being in volatile expensive quick memory). Once reaches max size, some policy is used to determine eviction
    - Eviction policies
        - Lru, fifo, mfu, random
    - New write policies
        - When about to write new stuff to disk, write to cache too instead of just disk
        - Skip cache then just write to disk
    - Write policies
        - Write through - write to disk
        - Writeback - cache acts as buffer, then when evicted, flush back
- Paging vs segmentation
    - Paging allows for less fragmeentation
    - Segmentation has better security since segments are pre-defined
    -
    - Segmentation has variable size of blocks between physical/virtual (not really virtual so it's called logical) called segments unlike paging
    - Segmentation is contiguous within segment
    - However, segments are not contiguous
    - You can think of segmentation as having table that tells you where to find your relevant blocks. However, because of this, was harder to fill smaller gaps, thus creating external fragmentation

    - Segment table - segment base (start of physical address), segment limit (size) to give physical address range
    - To translate logical address, add segment base to offset. Verify is within segment limit bounds. If out of bounds, is seg fault
    - Segments have stack, heap, code (read only)
- How background process started
    - The fork() call is used to create a separate process.
    - The setsid() call is used to detach the process from the parent (normally a shell).
    - The file mask should be reset. The reason for this is because we want to create new files with the mask that is needed for the child process.
    - The current directory should be changed to something benign. We may not want the child to be in the same pwd as the parent.
    - The standard files (stdin,stdout and stderr) need to be reopened.
- All system calls
    - fork() - Creates a new process.
    - execve() - Replaces process image.
    - wait() - Waits for child process to exit.
    - exit() - Terminates process.
    - open() - Opens a file.
    - close() - Closes a file descriptor.
    - read() - Reads data from a file.
    - write() - Writes data to a file.
    - mmap() - Maps files into memory.
    - brk() - Adjusts heap memory.
    - ioctl() - Device-specific control.
    - kill() - Sends a signal.

Virtual File System (VFS)

The **Virtual File System (VFS)** is an abstraction layer in the operating system kernel that allows different file systems (such as ext4, NTFS, or NFS) to be accessed in a uniform way. VFS abstracts the differences between different file system types, providing a standard interface for file management while supporting various underlying file systems.

Key Concepts of VFS and Related Topics:

1. **Pseudo-File Systems**: These are file systems that don't represent real disk storage but provide access to other resources in a "file-like" way.

    - **/proc**: A virtual file system that exposes process information and kernel parameters. For example, `/proc/cpuinfo` contains CPU details, and `/proc/[pid]` contains information about running processes.
    - **/sys**: Exposes kernel settings and provides interfaces to change system parameters.

- **/dev**: Provides access to devices and their drivers (like `/dev/sda` for storage devices).
- **sockfs**: A virtual file system used for managing sockets in the kernel. It allows socket-related operations to be done through file descriptors.
- **pipefs**: A file system for managing pipes, which are used for inter-process communication (IPC) in Unix-like systems.

2. **Shared Memory and VFS**:
    - **Shared memory** (such as via `shmget` and `shmat` on Linux) allows processes to communicate by accessing the same memory region.
    - Shared memory is treated as a **file** by the OS, which means it can be mapped to the VFS and exposed through the `/dev/shm` directory. This allows processes to share data by reading/writing to a memory-mapped "file."
    - Shared memory is faster than traditional IPC methods like pipes or message queues, as it avoids the overhead of copying data between user space and kernel space.
3. **File Descriptors and Open File Descriptions Table**:
    - **File descriptors** are integers returned when a process opens a file (or any file-like resource, such as a socket). These descriptors are used by system calls (like `read`, `write`, `close`) to perform file operations.
    - The **Open File Descriptions Table** is a data structure used by the kernel to track open files. It stores details about the file (like its offset, file status flags, and the file system in use). Multiple processes can share file descriptors pointing to the same open file description.
    - This separation allows for **shared file state** (like file position) between processes. If two processes open the same file, they might share the same open file description, meaning they both read/write from the same position.
4. **NFS (Network File System)**:
    - **NFS** is a distributed file system protocol that allows files to be accessed over a network as if they were part of the local file system.
    - VFS enables support for NFS by providing an abstraction layer that allows the OS to treat remote file systems (on an NFS server) like local files. The NFS client makes remote procedure calls (RPC) to interact with the remote file system.
    - VFS provides the standard interface for reading, writing, and managing files, while NFS handles network communication and file system operations on remote systems.
5. **LVM (Logical Volume Manager)**:
    - **LVM** is a system for managing disk storage that allows administrators to create flexible storage volumes that can be resized dynamically.

    - LVM works at a level above traditional partitions. VFS interacts with logical volumes as if they were regular block devices (like partitions). The abstraction allows easier management of storage space across multiple physical devices.
    - **LVM** enables volume groups (VGs) and logical volumes (LVs) to be combined into larger, more flexible storage pools.
6. **Software RAID**:
    - **RAID** (Redundant Array of Independent Disks) is a technology for combining multiple disk drives to improve performance, redundancy, or both.
    - **Software RAID** is managed by the operating system (rather than dedicated hardware) and is usually implemented using the **MD (Multiple Devices) driver** in Linux.
    - VFS interacts with RAID arrays in the same way it handles regular disk partitions. The RAID array, managed by the OS, presents a single logical block device to the VFS, which abstracts it as a normal file system.
7. **Capabilities**:
    - **Capabilities** are a way of dividing up root privileges into finer-grained controls for processes.
    - Instead of giving a process full root access, a process can be granted specific privileges (e.g., ability to bind to low ports or change system settings) without granting full root access. This improves security by reducing the risk of privilege escalation.
    - These capabilities are managed by the kernel and can be used to control what privileged operations a process can perform.
8. **Extended File Attributes**:
    - Extended file attributes (XATTRs) allow additional metadata to be associated with files beyond the standard attributes (e.g., permissions, owner, timestamps).
    - Examples of extended attributes include access control lists (ACLs), SELinux security contexts, and file-specific information used by applications.
    - The **VFS** provides support for reading and writing extended attributes, which are stored in the file system.
9. **ACLs (Access Control Lists)**:
    - **ACLs** provide a more fine-grained way of managing file permissions compared to traditional Unix file permissions.
    - They allow defining permissions for multiple users and groups, offering more flexibility than the standard owner/group/other model.
    - VFS interacts with the file system to enforce ACLs when a user tries to access a file.
10. **SELinux (Security-Enhanced Linux)**:

    - **SELinux** is a security module in Linux that implements mandatory access control (MAC). It enforces security policies that restrict how processes can interact with files, devices, and other system resources.
    - SELinux uses **labels** on files (stored in the **inode** of the file) and associates them with processes. The **VFS** checks these labels during file access to ensure compliance with SELinux policies.
    - SELinux helps improve security by preventing unauthorized access, even by privileged users or compromised applications

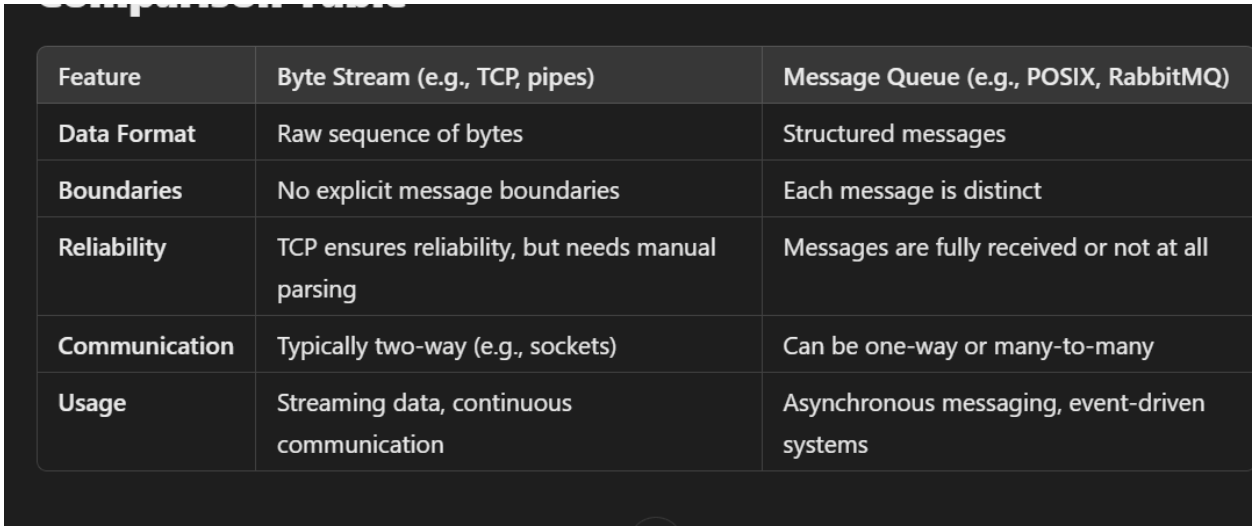
- interruptible and uninterruptible Sleep,
    - Uninterruptible - Can't be interrupted by signals. Waiting for i/o like disk read/write usually. Done when operation may cause corruption if interrupted (like disk write). Only woken when event done.
    - Interruptible - process is waiting for an event like user input or i/o. Can wake up safely with signal or when event done
- Runqueue/Scheduler latency,
    - This type of latency is how long wait in runqueue once it becomes ready. Context switch time + wait time
    - Check runqueue size
        - cat /proc/loadavg
    - Check scheduler (latency, decisions, size) - cat /proc/sched_debug | grep "cpu#" -A10
- How load is measured
    - Processes using cpu, processes in run queue, processes in uninterruptible sleep
- How sudo works
    - /etc/sudoers has users who can use it
    - Give root permission if correct password given
    - Controls root access so not everything is run with root, which can damage if you're not careful (sudo is almost like a warning). Sudo also logs
- Crash vs panic
    - Crash - sudden halt or failure due to unexpected event
    - Panic - crash where os needs to handle cirtical error before damage is done. Hardware errors, really bad software errors, corrupt system state are examples. Usually system halt/reboot happens, with err msg
- Systemd
    - Runs long running services
        - Create service file: has description, exec start=pathToBin, restart flag, User to run as, working dir, output dir

```
[Unit]
Description=My Custom Service
After=network.target

[Service]
ExecStart=/path/to/your/script_or_binary
Restart=always
User=myuser
WorkingDirectory=/path/to/working/directory
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

        - To run, recog service, enable it to start at boot, start
        - # Reload systemd to recognize the new service
        - sudo systemctl daemon-reload
        -
        - # Enable the service to start at boot
        - sudo systemctl enable myservice
        -
        - # Start the service immediately
        - sudo systemctl start myservice
    - Can start at boot
    - Ensures service restarts if it crashes
    - Ex: databases, api, nginx
- Cron
    - Schedule for repeating job
    - How to add job:
        - Edit crontable
            - Crontab -e
        - Look at table
            - Crontab -l
                - -r to remove all
        - Add entry in this format:
            - MIN HOUR DAY MONTH DAYOFWEEK COMMAND/binary-path
        - Check logs
            - journalctl -u cron --no-pager --since "1 hour ago”
            - Also is in var – grep CRON /var/log/syslog
- Use at for one time
    - echo "/home/user/myscript.sh" | at 2:00 AM tomorrow
        - Or – at now + 1 hour
- user-space tracing, kernel tracing
    - Pretty high overhead while counters don’t havea s much (just adding stuff in file)
    - Strace - see system calls made by program
    - Ltrace - see lybrary calls made by program
    - Ftrace - trace kernel actions
    - Perf - can also be used for kernel tracing

- Containerization
    - Containerism - running multiple servers on one machine without changing machine
    - Can do so with namespaces, cgroups
        - Can also use other third party stuff like docker
- Raid - redundancy for avilailbity, parallelism through striping for drives
    - 0 is fastest; striping (smaller chunks spread across devices) in 0/5/6/10 faster due to parallel access
    - 1 has mirroring, but mirroring results in temporary backups (good secondary avail) but slower
    - 5 has 1 failure drive and 6 have 2 failure drives, making good secondary availability (not true backups)
    - 0 has nor redundancy
    - 10 is expensive but has mirroring, striping, failure drives
- serspace threads/lightweight threads/coroutines,
- Control groups
    - Manage resource allocation and limits for specific containers to prevent hogging
    - Cpu time allocation
    - Ram and swap usage
    - i/o bandwidth (limit to specific rate)
    - Network bandwidth
    - Number of pids
    - Access to devices
    - Huge page total size
- Namespaces - illusion of own machine
    - pros
        - Gives illusion of being its own machine due to its own space separate from global (some pid-hard coded legacy programs depend on that)
            - Good for multiple webservers on a machine
        - No conflicts of resouce management
        - Ecapsulation - can't see other namespaces
    -
    - Can create specific spaces where a particular attribute's set space is unique to that space
        - Ex: pid namespace means that all pid's are considered unique within that space and do not match globals
            - Create pid namespace: one process is chosen as root of namespace. All its childrens are added and their pid's are made unique for that namespace, different from global
    - Ex namespaces
        - Pid, starts at pid 1 (so can have own set of pids)
        - Mount (so can have own file system)
        - Network interfaces, ip, routing tables

        - Uts (hostname/domain) - container can set its own hostname without affecting whole machine
        - Ipc - no leaks/peaking at other ipc namespaces
        - User (isolates user, groupids) - that way have hierarchy between users in global and in namespace, where namespace root might just map to a global normal user type
        - cgroup
    -
- Disks vs ram (in terms of physical build)
    - Ram is volatile but fast
    - Ssd is fast but expensive
    - Hdd is slow but long duration and cheap and made of disks
    - RAM: Uses transistors arranged in memory cells on a silicon chip. Data is stored as electrical charges, enabling fast access but requiring constant power to maintain.
    - SSD:
    - Employs flash memory chips, where data is stored by trapping electrons. These chips retain data without power and offer faster access than HDDs but have limited write cycles.
    - HDD: Consists of spinning magnetic platters and a read/write head. Data is stored magnetically on the platters, offering high capacity at low cost but with slower access due to mechanical movement.
- Cache vs buffer
    - just temporary storage (queue before process), or Buffer is a cache for output so can batch output
    - Cache is so don't have to read from disk or network all the time, reducing latency
- Elf
    - Executable and linkable format (ELF) - file used to create executable
    - Contents
    - Elf header - has metadata like little/big endian (readelf -h)
    - Program header - has all the address segments to load such as code/stack. Specifies to os which parts of file should be in memory (virtual memory) and which parts are readable, executable, writable
    - Section header - outlines logical divisions in text so linker knows what section is code, which are variables, etc.
    - Text - the instructions (read-execute)
    - rodata/data - variables (read-write) and rodata (read only data)
    - Bss - used during runtime to store global/static (zeroed out during compile)
    - https://www.baeldung.com/linux/executable-and-linkable-format-file
    -
- Byte stream - continuous info like tcp or pipes; two way // message queue - structure messages with boundaries (received/not) one to many / etc.



Feature	Byte Stream (e.g., TCP, pipes)	Message Queue (e.g., POSIX, RabbitMQ)
Data Format	Raw sequence of bytes	Structured messages
Boundaries	No explicit message boundaries	Each message is distinct
Reliability	TCP ensures reliability, but needs manual parsing	Messages are fully received or not at all
Communication	Typically two-way (e.g., sockets)	Can be one-way or many-to-many
Usage	Streaming data, continuous communication	Asynchronous messaging, event-driven systems

-
- Message queue implementation
    - Sender doesn't wait for receiver to be ready. Just puts in specific queue and then receiver picks up whenever ready
    - Many to one or one to one. Many to one needs synch methods to avoid race conditions
- sbrk() - moves program break, which defines heap segment
    - sbrk(n) expands it (-n shrinks it)
    - sbrk(0) resets it
    - Contiguous but may have holes frominternal fragmentation when shrink/grow by set amou nts (may have more than need)
    - Not as fine tuned as mmap
- Mmap() - heap alternative. Allocates space by mapping space directly to virtual address space. Page allocator instead of break. Used for large allocations (malloc uses it for large allocations)
    - Can be used for anon mapping (heap like) or file backed (reading/writign files)
    - More flexible
    - Non contiguous
    - Can be used for shared memory avoiding ipc
    - Resize via remap()
- calloc() - like malloc but sets allocated memory to 0 for heap
    - Argos: number of elements and size of each element.
- realloc() - resizes previously memory in heap
    - realloc(arr, 10 * sizeof(int)
- malloc() for c / new for c++ - sets data manually in heap to uninit. Malloc returns void pointer
    - New is calloc for standard types. Calls constructor for user defined classes
        - New returns pointer to object
        - New uses delete cmd
-

- SMEP stops the kernel from executing user-space code while running in kernel mode, which prevents certain types of code-injection attacks.
- SMAP prevents the kernel from accessing user-space memory unless explicitly allowed, limiting the chances of kernel-space vulnerabilities from compromising user-space data.
    - Raise error if any of these are broken

The **Virtual Filesystem (VFS)** is an abstraction layer in the Linux kernel that allows different file systems (e.g., NTFS, ext4, NFS) to be accessed uniformly by client applications. It enables interoperability by translating generic system calls like `read()` and `write()` into filesystem-specific operations.

Key Points of VFS:

- **VFS Objects**: The primary objects in VFS include:

    1. **File Object**: Represents an open file in a process.
    2. **Inode Object**: Contains metadata about a file, such as ownership and permissions.
    3. **Dentry Object**: Links inodes to file names, aids in path resolution and caching.
    4. **Superblock Object**: Stores filesystem metadata, such as type and size.
- **Supported Filesystems**: VFS supports disk-based filesystems (e.g., ext2, ext3, NTFS), network filesystems (e.g., NFS, CIFS), and special filesystems (e.g., /proc).

- **System Calls**: VFS manages file system operations through system calls like `mount()`, `stat()`, `open()`, `read()`, `write()`, and `rename()`.

- **Directory and File Management**: Directories act as containers for files, with file operations such as creation, deletion, and modification managed via inodes. File metadata is stored separately from the data itself.

- **Operations Objects**: Each object in VFS has associated operation tables that define methods for manipulating files, directories, inodes, and superblocks. Examples include `write_inode()`, `create()`, and `sync_fs()`.

- **File System Operations**: The superblock contains a table of operations for managing the filesystem, such as syncing or unmounting.

VFS provides a unified interface for interacting with different file systems, allowing various file operations to be handled efficiently in a common structure.

- Signal masking - like interrupt masking but for signals. Helpful to prevent spam or prevent critical events from being interrupted

- shoudl systems end signal to notify corresponding process whens wapping memory? if so what are the disadvantages?
    - Yes, some do. Linux doesn't
    - Bad because sending signal means overhead for switching context
    - Signal delivery introduces latency
    - More overhead for keeping track of which memory is associated with what process (no page table to walk up from (usually walk down)
    -
- Swapping is usually only for dirty!!!!