OS Mode B:

The provided sources cover various aspects of **Virtualization Systems**, encompassing **File Systems**, **Distributed Systems**, **Virtualization Principles**, **Hypervisors**, **Virtual CPU (vCPU)**, **Virtual Memory**, **Virtual I/O**, **Containers and Microservices**, **VM Migration**, and specific tools like **VirtualBox**, **Docker**, and **sed**.

1. File Systems

File systems are fundamental components of Operating Systems (OSs) that manage and organize persistent data.

- **Basic Functions of File Management in OSs**: Present a **logical (abstract) view** of files and directories, **hide hardware complexity**, facilitate **efficient use of storage devices** (optimize access), support **sharing** (files persist), and provide **protection (control access)**.
- **Layered File System Implementation**: File systems often use a layered approach to reduce complexity and redundancy, though this can add overhead. Layers include high-level data management (translates file name, directory management, protection), translation of logical to physical addresses (file allocation, free space management), and generic commands to the device driver (read/write blocks, memory buffers, caches).
- **File Concept**: A file is a **digital container** storing and organizing persistent data. It has a **contiguous logical address space** and types like data (numeric, character, binary) or program (text, source, executable). The concept is generic, with special cases like `/proc` (virtual file system) and `/dev` (device files) in Unix/Linux.
- **User View of Files**: Users interact with files via their **name and type**, **attributes**, and **operations**.
    - **File Name and Type**: A valid name has character limits and case sensitivity. The **extension** helps the OS interpret the file, associating it with programs. The file type is recorded in its header and **cannot be changed** even if the extension does. Executable files have specific requirements (e.g., "x" flag in Unix, "MZ" magic number in Windows EXE).
    - **File Attributes**: Include **Name** (human-readable), **Identifier** (unique tag), **Type**, **Location** (pointer on device), **Size**, **Ownership**, **Time** (creation, access, modification), and **Protection** (access control). This information is stored in the **directory structure** on disk.
    - **File Operations**: A file is an **Abstract Data Type (ADT)**. Basic system calls include **Create**, **Write**, **Read**, **Reposition (seek)**, **Delete**, and **Truncate**. **Open** and **Close** operations manage file entries in memory and on disk, respectively. The **Open File Table (OFT)** tracks currently open files, managing access rights, buffers, and file information.
- **File Locking**: Mediates access to a file, similar to reader-writer locks. It can be **shared** (multiple processes) or **exclusive** (single process), and **mandatory** (OS manages) or **advisory** (application manages).
- **File Structure**: Refers to how files are organized. Types include **sequence of bytes** (Unix), **simple record structure** (lines), or **complex structures** (formatted documents). The goal is **minimizing data transfer** from disk.
- **Access Methods**: Needed for efficient read/write operations on fixed-length logical records.
    - **Sequential Access**: `read_next()`, `write_next()`, `rewind()`.
    - **Direct Access**: `read(n)`, `write(n)`, `position(n)` using **relative block numbers**.
    - **Indexed File Access**: Combines sequential and direct access using an **index file** that maps logical keys to physical positions. Requires an extra index to track blocks.
- **Directory Structure**: The way an OS arranges files, residing on disk.

    - **Purpose**: **Efficiency** (locate files quickly), **Naming** (convenient, allows same name for different files, multiple names for same file), **Grouping** (logical organization).
    - **Operations**: Search, Create, Delete, List, Rename, Traverse.
    - **Types**:
        - **Single-Level Directory**: Simplest, all files in one directory, suffers from naming and grouping problems.
        - **Two-Level Directory**: Each user has a **User Files Directory (UFD)**, managed by a **Master File Directory (MFD)**. Allows same file names for different users and prevents cross-user access. Disadvantages include no file sharing or subdirectories, and poor scalability.
        - **Tree-Structured Directories**: Resembles an upside-down tree with a **root directory**. Users have **Home directories** (e.g., `/home/<username>` in Linux, `\Users\<username>` in Windows) and can create subdirectories. Files are accessed via **path names**. Can be complicated to search with many subdirectories and lacks direct file sharing between users.
        - **Acyclic-Graph Directories**: Allows a file to be accessed from **multiple directories** using **links**.
            - **Hard links** act as a copy, accessing original data; data persists even if the original is deleted (as long as one hard link survives). Cannot span file systems or link directories.
            - **Soft links (Symbolic links)** act as a pointer; fail if the original file is deleted. Are versatile and can link directories.
        - **General-Graph Directory Structure**: More complex, allows cycles, requires mechanisms like garbage collection or cycle detection algorithms.
- **File System Implementation**: General-purpose computers can have multiple storage devices, sliced into **partitions** that hold **volumes**. A volume is a single accessible storage area with a file system.
    - **File System Types**: Many exist (e.g., UFS, FFS, FAT, NTFS, ext4, XFS, ZFS).
    - **File System Data Structures**:
        - **On-Disc**: **Boot control block** (OS startup code), **Volume control block** (superblock, volume details), **Directory structure** (names, IDs, info about files), **File control blocks (FCB)** or **inodes**.
        - **In-Memory**: **Mount Table**, **Directory Structures cache**, **Global Open File Table**, **Per-process Open File Table**.
- **Directory Implementation**: Directories are collections of entries associating names to FCBs/inodes.
    - **Linear list**: Simple to program but time-consuming to execute (linear search time).
    - **Hash Table**: Decreases search time but can suffer from collisions and is only good for fixed-size entries.
- **Disk Blocks Allocation Method**: How disk blocks are allocated for files.
    - **Contiguous Allocation**: Each file occupies a set of **contiguous blocks**. Offers best performance but has problems like finding space, knowing file size, and **external fragmentation**. **Extent-based systems** are a modification.
    - **Linked Allocation**: Each block contains a pointer to the next. Simple, no external fragmentation, easy to grow files, but **inefficient random access** and **poor locality of reference**.

        - **File Allocation Table (FAT)**: Pointers to next blocks are stored separately in a FAT. Improves seeking performance as FAT can be in memory. Used in DOS/Windows. Large disks require large cluster sizes, leading to **internal fragmentation** with small files.
        - **Indexed Allocation**: Each file has its own **index block(s)** of pointers to data blocks. Provides easy **random access** and no external fragmentation, but requires index blocks and can have **poor locality of reference** or **internal fragmentation in the index table**. Can use linked lists of blocks, hierarchical index tables, or **combined schemes** (e.g., **UNIX UFS inode** with direct, single, double, and triple indirection) to handle large files.
- **Free-Space Management**: Tracks available blocks for file creation/growth.
    - **Bit Vector (Bit Map)**: Each block represented by a bit (1 for free, 0 for allocated). Simple to understand and find first free block, but inefficient for large disks.
    - **Linked List**: Each free block points to the next. No need to traverse entire list, but scanning is costly (disk I/O) and hard to get contiguous space. **Grouping** modification stores addresses of `n-1` free blocks in the first free block.
    - **FAT**: The FAT itself can indicate free blocks with a 0 entry.

2. Cloud Computing

Cloud computing addresses challenges like over/under-provisioning by offering scalable, on-demand resources.

- **Core Concepts**: Provides the illusion of **infinite computing resources on demand**, **eliminates up-front commitment** by users, and allows users to **pay for resources on a short-term, as-needed basis**.
- **Definition and Characteristics**: An umbrella term for **Internet-based development and services**. Characterized by being **remotely hosted**, **ubiquitous** (available anywhere), and **commodified** (utility computing model – pay for what you use). Platforms **hide infrastructure complexity** via simple interfaces or APIs, offer **on-demand services** (always on, anywhere, anytime), and are **flexible and elastic** (scale up/down CPU, storage, server capacity, etc.). Often built on clusters of commodity hardware and open-source software.
- **NIST Cloud Computing Reference Architecture**: Established in 2011, it defines "what" a cloud system can do, providing a common reference language.
- **Cloud Service Models**:
    - **Infrastructure as a Service (IaaS)**: Delivery of **technology infrastructure as an on-demand scalable service**, usually billed based on usage in a multi-tenant virtualized environment.
        - **Virtualization is its basis**, providing **virtual workspaces** (abstracted execution environments with resource quotas and software configurations) implemented on **Virtual Machines (VMs)** using a **Hypervisor**.
        - Offers **scalability** through **vertical scaling** (increasing existing hardware capacity) and **horizontal scaling** (connecting multiple entities to work as a single logical unit).
        - Sub-categories include **Computing as a Service (CaaS)** (raw computing power on virtual servers/VMs) and **Storage as a Service** (online storage).
        - Examples: Amazon EC2, WorkSpaces, S3 (Simple Storage Service), DynamoDB.
    - **Platform as a Service (PaaS)**: Provides facilities for the **complete life cycle of building and delivering web-based applications** entirely from the Internet. Applications are typically developed with a specific platform in mind, in multi-tenant, highly scalable environments.
        - Examples: Google App Engine (hosting and development platform for web apps, supports multiple languages, services, versions, and instance classes), Windows Azure Platform

(on-demand platform for distributed applications, including storage, hosted services, AppFabric, and SQL Azure).
    - **Software as a Service (SaaS)**: Model where an **application is hosted as a service** and provided to customers across the Internet. It **alleviates software maintenance burden** but users relinquish control over software versions. A SaaS is not only a web application; it can be installed in a particular environment (e.g., Facebook/Google platforms).
- **Benefits of Cloud Computing**:
    - **Lower computer costs** (less powerful client PCs needed).
    - **Improved performance** (fewer programs on client PC).
    - **Reduced software costs** (many free cloud applications).
    - **Instant software updates** (web-based apps update automatically).
    - **Improved document format compatibility** (shared formats in the cloud).
    - **Universal document access** (accessible from anywhere with Internet).
    - **Latest version availability** (cloud always hosts latest document version).
    - **Unlimited storage capacity** (virtually limitless).
    - **Increased data reliability** (data replicated, safe from local crashes).
    - **Easier group collaboration** (multiple users on documents).
    - **Device independence** (access data/apps from any device).
- **Disadvantages of Cloud Computing**:
    - **Requires constant Internet connection**.
    - **Does not work well with low-speed connections**.
    - **Features might be limited** compared to desktop applications.
    - **Can be slow** (network latency, server load).
    - **Stored data might not be secure** (risk of unauthorized access).
    - **Stored data can be lost** (reliance on cloud provider).
    - **Dependence on others** can limit flexibility/innovation.
    - **Security issues** (ownership of data, server downtime, account lockouts).

3. Distributed Systems

Distributed systems leverage interconnected nodes to perform tasks, providing benefits like resource sharing and fault tolerance.

- **Evolution of Operating Systems**: Major advances include processes, memory management, information protection/security, scheduling, resource management, and system structure.
- **Parallel Systems**: Systems with more than one CPU in close communication, often **tightly coupled** (share memory, clock, I/O).
    - **Advantages**: Increased throughput, economical, increased reliability.
    - **Symmetric Multiprocessing (SMP)**: Each processor runs an identical OS copy, many processes run concurrently, communication via shared memory. Most modern OSs support SMP.
    - **Asymmetric Multiprocessing (AMP)**: Each processor assigned a specific task, master processor schedules work for slave processors, no communication between processors. More common in large systems.
    - **SMP vs. AMP Comparison**: SMP uses a ready-queue, AMP uses master-slave. SMP has single OS accessed by all, AMP master accesses OS. AMP has more memory footprint. SMP design is easier (identical processors). SMP implementation of routines is easier. SMP has communication

overhead, AMP does not. In SMP, processor failure reduces capability; in AMP, a slave can become master.

- **Network-based Systems**: Computer networks create opportunities for **sharing resources** (files, printers, distributed databases, specialized hardware like GPUs) and **increased computing performance** (computation speedup, load balancing). They also offer **reliability** (detect/recover from failure). Nodes are interconnected by a network, with a **server** providing resources to a **client node**. Examples include Microsoft Windows Server, UNIX, Linux, Mac OS X.
- **Computer Networks vs. Distributed Systems**:
    - **Computer Networks**: Collection of **autonomous computers** interconnected by a single technology, primarily for **communication and resource sharing**. Less complex, transparency not primary, scalable but may require reconfiguration, vulnerable to single points of failure, limited resource sharing (bandwidth, storage, peripherals).
    - **Distributed Systems**: Collection of **independent computers** that **appear to users as a single coherent system**. A software system built on top of a network, providing high **cohesiveness and transparency**. Designed to perform **complex tasks by distributing workload**. Decentralized control, more complex management, provide transparency, inherently scalable, highly fault-tolerant, share processing power and software.
- **Network Operating Systems (NOS) vs. Distributed Operating Systems (DOS)**: The distinction lies in the **software (OS)**.
    - **NOS**: **User-driven management**. Users are **aware of the multiplicity of machines** and explicitly access remote resources (e.g., SSH, FTP).
    - **DOS**: **Task-driven management**. Users are **not aware of the multiplicity of machines** and access remote resources similarly to local ones. Achieved through:
        - **Data Migration**: Transferring entire files or portions, with translation if sites are incompatible.
        - **Computation Migration**: Transferring the job rather than data, via Remote Procedure Calls (RPCs) or messaging systems.
        - **Process Migration**: Executing entire processes or parts at different sites for **load balancing**, **computation speedup**, **hardware/software preference**, or **data access**.
- **Design Issues of Distributed Systems**:
    - **Robustness**: Ability to **withstand failures** (link, site, message loss). Involves **failure detection** (e.g., heartbeat protocol), **reconfiguration** (broadcasting failures/recoveries), and **recovery**.
    - **Transparency**: System should appear as a **conventional, centralized system** to the user. User interface should not distinguish local/remote resources (e.g., NFS). **User mobility** allows logging into any machine and seeing one's environment (e.g., LDAP with desktop virtualization).
    - **Scalability**: System should easily **accept new resources** to accommodate increased demand, reacting gracefully to load. Can be enhanced by data compression or deduplication.
- **Distributed File Systems (DFSs)**: Organize and share data across multiple machines. Challenges include **naming and transparency**, **remote file access**, and **caching/cache consistency**.
    - **Client-Server DFS Model**: Servers store files/metadata, clients request them. Server handles authentication and permissions. Examples include **NFS (Network File System)** and **OpenAFS**. Suffers from single point of failure and server bottleneck.
    - **Cluster-based DFS Model (CFS)**: File system shared by being simultaneously mounted on multiple servers. Partitions and distributes large datasets using **file striping**. Built to be more fault-tolerant and scalable. Examples: **Google File System (GFS)** and **Hadoop Distributed File System (HDFS)**.

- **Sun Network File System (NFS)**: A client-server DFS for remote file access across networks.
    - **Heterogeneous Environment**: Designed to operate across different machines, OSs, and network architectures using **Remote Procedure Call (RPC)** primitives (high-level communication protocol) and **eXternal Data Representation (XDR)** protocol (standard data serialization format).
    - **Mount Mechanism**: Remote directories are **mounted over local file system directories**, appearing as integral subtrees. The **Mount Protocol** establishes initial logical connection and returns a **File handle**.
    - **Architecture Layers**: **UNIX file-system interface**, **Virtual File System (VFS) layer** (distinguishes local/remote files, uses **vnodes** unique across the network), and **NFS service layer** (implements NFS protocol).
    - **NFS Protocol**: Provides RPCs for remote file operations (searching, reading/writing, manipulating links/directories, accessing attributes).
    - **Stateless Service**: A classical NFS server maintains **no in-memory hard state** (no client records, open files, file offsets, write-back caching). This **simplifies failure recovery** as there's no state to rebuild.
    - **Caching**: Employs **buffering and caching** (file-blocks cache, file-attribute cache) for performance. **Modified data must be committed to server's disk** before results return to client.
- **Google File System (GFS)**: A cluster-based DFS designed for large files with frequent appends, assuming hardware failures are common.
    - **Architecture**: Single master, multiple chunkservers, multiple clients.
    - **Chunkservers**: Store files divided into **fixed-size chunks (64MB)**, replicated for reliability (default 3 replicas). Large chunk size reduces client-master interaction and metadata size.
    - **Master**: Maintains file system metadata (namespace, access control, mapping, locations). Does not persistently record chunk locations but polls chunkservers periodically.
- **MapReduce Paradigm**: Programming model developed at Google for distributed computing, parallelizable across clusters. User-written Map and Reduce functions partition input and aggregate results.
- **Hadoop Distributed File System (HDFS)**: Open-source framework for managing big data and analytics, a key component of Hadoop systems.
    - **Components**: **NameNode** (heart of HDFS, manages metadata) and **DataNode** (stores actual data).
    - **Features**: **Failure tolerant** (data duplicated across DataNodes, default replication factor 3), **scalable** (read/write capacity scales with DataNodes), **space-efficient** (add DataNodes for more space), **industry standard**. Designed for **write-once-read-many** semantics, not low latency.
- **Other DFS Features**:
    - **Naming**: How entities (hosts, files, processes) are referred to.
        - **Flat naming**: Random bit string identifier, no location info, good for machines.
        - **Structured naming**: Human-readable, combines host and local name, unique system-wide but not location transparent/independent.
        - **Attribute-based naming**: Entity described by (attribute, value) pairs, allows effective searching (e.g., directory services, RDF).
    - **Remote File Access**: Data transfer from server to user. **Caching** reduces network traffic by retaining recently accessed blocks.
    - **Cache Location**: **Disk caches** (more reliable, persistent data) vs. **Main-memory caches** (faster access, allows diskless workstations, performance speedup).

- **Cache Update Policy**:
    - **Write-through**: Writes data to disk as soon as placed in cache. Reliable but poor performance.
    - **Delayed-write (write-back)**: Modifications written to cache, then later to server. Faster write accesses, but poor reliability if crash occurs before write-back. Variations include scan-and-flush or write-on-close.
- **Consistency**: Ensuring cached copy matches master copy.
    - **Client-initiated approach**: Client performs validity check.
    - **Server-initiated approach**: Server records cached files and reacts to potential inconsistency.
    - **DFS Specifics**: In cluster-based DFS, consistency is complicated by metadata servers and replicated chunks (e.g., HDFS append-only, GFS allows random writes with concurrent writers).

4. Virtualization Principles

Virtualization decouples hardware and software, allowing multiple OSs or instances on a single machine.

- **The Problem Virtualization Solves**: Traditionally, applications run directly on one OS per machine. Virtualization aims to **decouple hardware/software behavior from physical realization**, use data center hardware efficiently, improve OSs, and enhance software distribution.
- **Definition**: **Creating a virtual (rather than actual) version of something**. Examples include virtual memory and context switching.
- **Technology in Computing**: Enables a single machine to **simultaneously run multiple OSs** or multiple sessions of a single OS. A **Virtual Machine (VM)** is an instance of a full computer system, supporting multi-operating systems on a **host** (underlying hardware) and **guests** (VMs, usually OSs).
- **Java Virtual Machine (JVM)**: A specific example of a VM that creates a machine different from the underlying processor architecture, essentially always emulated.
- **Benefits**:
    - **Dynamic reconfigurability** for changing needs.
    - Support for **multiple OSs concurrently** on single hardware.
    - **Isolation**: Failures are isolated to the VM, and VMs are protected from each other, limiting virus spread.
    - **Sandbox environment** for isolation.
    - **Features**: Freeze/suspend/resume, snapshot (restore to a state), cloning (copying VM), templating (creating preconfigured VM molds), and **live migration** (moving a running VM without interruption). These features contribute to **cloud computing**.
- **Infrastructure Management**:
    - **Server consolidation**: Migrating services from multiple physical computers to fewer, or to multiple virtual computers on one host. Reduces power consumption, simplifies administration, lowers costs, and **increases hardware resource utilization**. The **consolidation ratio** measures VMs per server (e.g., 8:1).
    - **Load Balancing**: Distributing workloads across VMs.
- **Advantages of VMs**: **Security** (isolated), **Flexibility** (easy to spin up new VMs, scalable), **Ease of Use** (simple management), **Platform Independence** (multiple OSs on same hardware), **Quick Disaster Recovery** (easy backup and recovery).

- **Disadvantages of VMs**: **Performance Issues** (shared hardware resources), **High Initial Costs** (VM software, storage), **Complexity** (requires expert help).
- **Types of Virtualization**:
    - **Full Virtualization**: Provides an environment **identical to physical hardware**, allowing **unmodified guest OSs** to run. Can lead to **performance degradation** due to hypervisor translation of OS calls.
    - **Paravirtualization**: Provides a virtual hardware abstraction that is **similar but not identical**. Requires a **modified guest OS** that cooperates with the VMM via APIs. Results in **lower overhead and better performance**.
    - **Emulation**: Allows applications written for one hardware environment to run on a very different one. Reproduces functionality. Guest OS runs as a user-mode process with a translator in the VMM.
    - **Container-based Virtualization**: Provides **multiple isolated user-space instances** on a **single kernel** of the OS. **Not true virtualization** of hardware but segregates applications, making them secure and manageable. Examples include Oracle Solaris Zones, BSD Jails, Linux containers (LXC), and **Docker**.

5. Hypervisor

A Hypervisor, also known as a Virtual Machine Manager (VMM), is software or firmware that creates and runs virtual machines.

- **Definition**: Formally, virtualization involves constructing an **isomorphism that maps a virtual guest system to a real host system**. The hypervisor manages this mapping.
- **Characteristics (Popek and Goldberg)**: Provide an **environment identical to physical hardware**, ensure **minimal performance cost**, and **retain complete control of system resources**.
- **Types of Hypervisors**:
    - **Type 0 Hypervisors**: **Hardware-based solutions** that provide VM support via **firmware**. Resources are **dedicated** to each guest. Examples include **IBM LPARs** and **Oracle LDOMs** (Logical Domains) for SPARC architectures. LDOMs partition resources into logical domains, with the hypervisor maintaining separation. Domains can have roles: **Control Domain** (manages other domains), **I/O Domain** (direct physical I/O access), **Service Domain** (provides virtual device services), and **Guest Domain** (consumes virtual services). Type 0 can support virtualization-within-virtualization.
    - **Type 1 Hypervisors (Bare-Metal Hypervisors)**: **Run directly on server hardware without an OS beneath it**. They are **OS-like software** built to provide virtualization. They communicate directly with hardware, making them efficient. Examples include VMware ESX, Joyent SmartOS, Citrix XenServer, RedHat Enterprise Linux with KVM, and Windows with Hyper-V. They run in kernel mode and manage guest OSs, often implementing device drivers. They are common in data centers for **consolidation**, load balancing, snapshots, and cloning.
    - **Type 2 Hypervisors (Hosted Hypervisors)**: **Run as applications on standard operating systems**. Examples include VMware Workstation/Fusion, Parallels Desktop, and Oracle VirtualBox. They require extra steps for hardware interaction (VM -> hypervisor -> host OS -> hardware), adding overhead and potentially leading to poorer performance. However, they require no changes to the host OS. Often used in desktop development environments.
- **Type 1 vs. Type 2 Comparison**:
    - **Efficiency**: Type 1 is more efficient as it directly communicates with hardware.

    - **Security**: Type 1 is more secure; a guest cannot affect the hypervisor.
    - **Overhead**: Less processing overhead for Type 1, allowing more VMs per host.
    - **Hardware Support**: Type 2 supports a larger range of hardware (inherited from host OS).
    - **Installation**: Type 2 is easier to install/deploy.
    - **Reliability**: Type 2 is less reliable due to more points of failure (host OS issues affect hypervisor/guests).
- **Ways to Virtualize**: The key challenge is to **prevent guest OSs from accessing shared hardware** and corrupting other programs/OSs, ensuring **isolation** of guest behavior from other guests and host processes. This involves managing shared CPU registers, physical memory, and I/O devices.
- **Protection Rings**: Hierarchical protection domains (numbered 0 to highest) enforced by CPU architectures to protect data and functionality. Ring 0 has the most privileges, interacting with physical hardware.
    - **Hypervisor Placement**: **Type 1 hypervisors usually sit in Ring 0**. **Type 2 hypervisors often put the guest kernel code in Ring 1**.
    - **Intel Architecture with Negative Rings**: Introduced negative rings for higher privilege levels: **Ring -1** (VMX Root for Type 1 Hypervisor), **Ring -2** (System Management Mode/SMM for OEM software), and **Ring -3** (Management Engine/ME, highest privilege, always running, full access to system).
- **Trap-and-Emulate**: When a guest attempts a privileged instruction in user mode, it causes a **trap** to the hypervisor, which then analyzes the error, emulates the operation, and returns control to the guest. This causes **overhead** and slows down guest kernel mode operations.
- **Hardware-assisted Virtualization**: Uses special CPU instructions (e.g., Intel VT-x, AMD-V) to aid virtualization, typically by defining more CPU modes ("guest" and "host"). This makes hypervisor implementation less complex, more maintainable, and improves performance.
- **Binary Translation**: A software-based method to solve the problem of "special instructions" (privileged instructions) not causing a trap. The hypervisor reads instructions ahead of time; non-special instructions run natively, while special instructions are translated into new instructions that perform an equivalent task. Performance relies on optimizations like caching.

6. Virtual CPU (vCPU)

Virtual machines use virtual CPUs (vCPUs) to represent the CPU state as the guest machine perceives it, allowing multiple VMs to share physical CPU resources.

- **vCPUs in a VM**: Most Hypervisors implement **virtual CPUs (vCPUs)**, which represent the CPU state for the guest VM. When a guest is context-switched onto a physical CPU (pCPU), vCPU information is loaded/stored. The **hypervisor schedules vCPU cycles** on the host's available pCPUs.
- **Choosing vCPUs**: Multiple vCPUs benefit **multi-threaded applications** but can hinder performance if they cannot be scheduled simultaneously. It's best to **start with a single vCPU** and adjust upward as needed.
- **Hyper-Threading**: Intel technology presenting **two logical processors for each physical processor**, allowing more threads to be scheduled. Improves efficiency by about 30%.
- **vCPU-to-pCPU Ratio**: The number of vCPUs that can be supported by physical cores/threads. VMware vSphere 6.0 allows up to 32 vCPUs per physical core, with recommended ratios like 1:1 to 3:1 for good performance.

- **vCPU Scheduling**: Hypervisors schedule vCPUs onto pCPUs. Often, **CPU overcommitment** occurs (more vCPUs than pCPUs), leading to the hypervisor scheduling time slices. Overcommitment can cause poor response times and incorrect time-of-day clocks.
- **VMware vSphere**: A cloud computing virtualization platform with ESXi as its Type 1 hypervisor, offering centralized VM management and features like workload migration.
    - **vSphere CPU Scheduler**: Enforces fairness using a **proportional-share algorithm** based on **shares** (relative priority), **reservations** (guaranteed resources), and **limits** (maximum resource usage).
    - **Gang Scheduler (Co-Scheduling)**: For multi-vCPU VMs, all vCPUs are "ganged" together and scheduled simultaneously to maintain synchronization. If **skew** (difference in vCPU progress) exceeds a threshold, the VM is **co-stopped** and only resumed when all vCPUs can be scheduled together, which can cause **CPU fragmentation**.
    - **Relaxed Co-scheduling**: A modification where a "leading vCPU" can decide to co-stop itself if skew is too high, allowing individual vCPUs to make scheduling decisions. This decreases required pCPUs and increases CPU utilization, solving fragmentation but introducing skew issues.
- **Microsoft Hyper-V Scheduler**: Handles CPU calls independently, making oversubscription less of an issue.
- **Citrix Hypervisor (XenServer)**: An open-source Type 1 hypervisor. It uses a **Control Domain (dom0)**, a privileged Linux VM, to run the management toolstack (XAPI) which controls VM lifecycle, networking, and storage. **Guest domains** request resources from dom0.
    - **Xen CPU Schedulers**: Supports various schedulers (e.g., Credit Scheduler) optimized for different use cases, allowing user interaction via global parameters (Timeslice, Context-Switch Rate) and VM scheduling parameters (Weight, Cap).

7. Virtual Memory

Virtual memory techniques are employed by hypervisors to optimize the use of physical memory across multiple virtual machines.

- **Memory in VMs**: VMs are allocated specific amounts of memory, which can be reconfigured. The hypervisor also reserves a portion of memory for its own processes. Memory is managed in **pages**, and **paging** copies less-recently used pages to a **page file** on disk to free up memory.
- **Memory Overhead**: Additional memory is reserved per VM for operational functions like **memory mapping tables** (connecting VM memory addresses to physical memory addresses).
- **Nested Page Tables (NPTs)**: A common method for managing memory in virtualized environments. Guests maintain their own page tables (virtual to physical addresses), while the **hypervisor maintains per-guest NPTs** to represent the guest's page-table state. Changes to guest page tables are reflected in NPTs and the hypervisor's own page tables. This can cause **Translation Lookaside Buffer (TLB) misses** and lead to **slower performance**.
- **Memory Optimizations**:
    - **Memory Ballooning**: A feature (e.g., in VMware ESXi, Hyper-V, Xen, KVM) to **reclaim unused memory** from VMs. A "balloon driver" inflates inside the guest OS, forcing it to flush less-recently used pages to disk. Once flushed, the driver deflates, and the hypervisor reclaims the physical memory. This process enables **memory overcommitment** (allocating more virtual memory than physically exists). Issues include high balloon utilization impacting hypervisor performance and spikes in CPU/disk usage.

    - **Page Sharing**: Reduces data duplication by storing only one copy of identical memory pages across multiple VMs. If a VM needs to write to a shared page, the hypervisor creates a new copy for its exclusive use (known as **Copy-on-Write**).
    - **Compression**: Deferring swapping pages to disk (an expensive operation) by compressing them and moving them to a **compression cache** reserved by the hypervisor.

8. Virtual I/O

Virtual I/O manages how virtual machines interact with storage and network resources, abstracting the physical hardware.

- **Resource Contention**: If any resource (CPU, memory, I/O) suffers contention, the entire virtual server's performance can degrade.
- **Virtual Storage**: Drives are regions of disk space on a shared storage device. The **hypervisor manages their presentation to the VM**, abstracting the underlying physical connection (Fibre Channel, iSCSI, NFS).
    - **Shared Storage**: **SAN (Storage Area Network)** or **NAS (Network Attached Storage)** solutions allow multiple computers (physical or virtual) to access the same physical drives, easing transition to virtual environments.
    - **Data Path (VMware model)**: Application request -> guest OS -> virtual disk adapter -> **hypervisor** -> hypervisor's storage device driver -> physical host's storage controller -> physical storage.
    - **Data Path (Citrix Hypervisor model)**: Application request in user domain (DomU) -> front-end device driver -> **back-end driver in Dom0** (privileged guest with direct hardware access) -> Dom0 device driver -> hardware device. In this model, the **hypervisor is bypassed** as Dom0 handles the direct connection.
    - **Storage Virtualization**: An **abstraction of physical resources** presented to VMs as if they controlled physical devices. The virtual drive is a logical representation.
    - **Tuning Practices for VM Storage**:
        - **Thick Provisioning**: **Entire amount of virtual disk space is pre-allocated** on physical storage upon creation, even if not immediately used. Can waste space. Types include **Lazy Zeroed** (data not erased during creation, zeroed on demand) and **Eager Zeroed** (data zeroed at creation, slower to create).
        - **Thin Provisioning**: Storage space is **allocated only as needed**, allowing **overprovisioning** of physical storage. Minimal performance impact, but critical if all allocated space is consumed.
        - **Disk Configuration**: Many smaller disks often perform better than fewer larger disks due to parallel I/O capabilities of storage arrays.
        - **Disk Mirroring**: Uses a second disk to **perfectly mirror data blocks**, preventing data loss if one disk fails and providing two copies for read operations.
        - **Deduplication**: Locates **identical chunks of data**, flags the original, and replaces duplicates with pointers, significantly saving disk space (30-90% reclaim).
- **Virtual Networks**: Allow VM applications to connect to services outside the host. The **hypervisor manages network traffic** in/out of each VM and the host.
    - **Citrix Xenserver Network Model**: All network traffic goes through **Dom0 (parent partition)**, where the **virtual switch** resides, connecting to the physical NIC.

- **Virtual NICs and Virtual Switches**: Each VM can have virtual NICs connected to a **virtual network** composed of **virtual switches (vSwitches)**. A vSwitch is a software program enabling VM-to-VM communication and connections between virtual/physical networks. Ports on a vSwitch can be adjusted without replacing hardware.
- **Software-Defined Networks (SDN)**: Solutions that **move network configuration and management from specialized hardware to software**, enabling deployment of functions like firewalls, load balancing, and VPNs as virtual appliances.
- **Types of Virtual Switches**:
    - **External Virtual Switches**: Bound to a physical network card, provides external network access. Host OS can communicate across it.
    - **Internal Virtual Switches**: Not linked to physical adapter, entirely software-defined. Allows communication between connected VMs and the hypervisor host, but not to external networks. Useful for isolated environments.
    - **Distributed Virtual Switches**: Extend beyond a single host to meet the demands of clustered hosts, allowing cluster nodes to share the same switch across nodes.
    - **Private Virtual Switches**: Completely isolates VMs; only allows communication among VMs connected to that private switch.
- **Unique Address**: Every network device has a unique address (can be assigned or via DHCP).
- **VirtualBox Network Modes**:
    - **Bridged Network**: Allows VMs to have an **IP address recognized outside the host**, acting as a virtual switch routing traffic to the physical NIC.
    - **Internal Network**: Works on an **internal virtual switch**, allowing VM-to-VM communication but **without external connection**. Can use local DHCP.
    - **Network Address Translation (NAT)**: Blends host-only and bridged networks. VMs have **isolated IP addresses** internally but **share the physical host's IP address** for external access. Hypervisor translates addresses. Can create a private subnet and protect network topology.

9. Containers and Microservices

Containers offer a lightweight alternative to VMs for application deployment, especially in the context of microservices architecture.

- **Container vs. Virtual Machine**:
    - **VMs**: Each runs its **own Guest OS** on top of a Hypervisor. Boot-up time is in minutes. Not version-controlled. Fewer VMs per laptop.
    - **Containers**: Only virtualize the **User Space of the OS**, sharing the **same kernel of the hosting system**. Run on a Container Engine (like Docker) directly on the OS. Instantiate in seconds. Images are layered and version-controlled. Many containers can run on a laptop.
- **Why Use Containers?**:
    - **Isolated environment** with preferred OS, without dedicated hardware.
    - Cheaper than VMs due to kernel sharing.
    - **Application-centric**: Bundle application code, runtime, and dependencies into **container images** for reproducibility ("recipe" analogy).
    - **Portability**: Can create a Linux container on Windows.
    - **High utilization** of hardware resources due to lightweight nature.

- **Applications Evolution**: From **monolithic** (single stack, long-lived, single server) to **loosely coupled components** (constantly developed, deployed often to multitude of servers).
- **Monolithic Architecture**:
    - **Description**: All components (e.g., Java web app in a WAR file on Apache Tomcat) are packaged into a single unit.
    - **Benefits**: Simple to develop, test, deploy, and scale (the whole application).
    - **Disadvantages**: Difficult to maintain, **single component failure can crash whole system**, difficult to create patches, challenging to adapt to new technologies, long startup times.
- **Microservice Architecture**:
    - **Description**: Each microservice is designed to address a **particular aspect and function** of an application (e.g., logging, data search).
    - **Portability**: Combined with container technology (like Docker), microservices achieve **extremely high utilization** and portability.
    - **Scalability**:
        - **Non-uniform Scaling**: Replicate only bottlenecked parts of the application, deploying multiple instances of particular services to get higher utilization with fewer server resources.
        - **Elasticity**: Dynamically stop/start service replicas and servers based on request load, saving power and optimizing resource usage.
        - **High Availability**: Updates can be deployed to services without shutting down the entire application.
        - **Robustness**: If one server fails, other servers can still provide all services.
        - **Distribution**: Use several relatively weaker servers, run different microservices on each, leading to better I/O performance. Can combine with **edge computing** for local services.
    - **Usage Considerations**: Not suitable if heavily reliant on **in-memory state** or cannot accept latency from distributed data sharing; microservices should be designed as **stateless functions**.
    - **Implementation**: Programming language doesn't matter, as long as **solid API documentation** is provided for communication. **Message queue** (e.g., RabbitMQ, Azure Service Bus) can be used as eventbus middleware for asynchronous logic and better I/O performance.
- **Containers & Microservices Relationship**: Containers provide **better application packaging** for improved deployment and runtime management. Microservice components offer a **more granular application structure** for agility and diversity. They enable **same packaging and distribution for any application type**, **independent team development**, and **services that scale/fail over based on business demands**.
- **Microservice Design Concerns**: Identifying components, communication between components, **state management and data bounding/residency** (stateless services, service-managed state, platform-managed state), security, service discovery, resilience/transaction management, and no-downtime upgrades.
- **Platforms for Microservices**: Include generalized (Docker Swarm, Kubernetes, AWS Lambda) and specialized (Azure Functions, Google App Engine) container/cloud platforms, IoT platforms, analytics platforms, etc..
- **API Categories**: Microservices interact via various APIs: **Open web APIs**, **B2B APIs**, **Internal APIs**, and **Product APIs**. Business APIs serve as entry points, while private APIs and messaging connect internal components.

10. VM Migration

VM migration is a crucial technique for dynamically managing resources and ensuring high availability in data centers.

- **Enterprise Data Centers**: Composed of large server clusters and network-attached storage, with multiple applications per server, adhering to **Service Level Agreements (SLAs)**. Workloads are highly dynamic, leading to **hotspots** if demand exceeds provisioned capacity.
- **Provisioning Methods**:
    - **Static over-provisioning**: Allocates for peak load, but wastes resources and is unsuitable for dynamic workloads.
    - **Dynamic provisioning**: Adjusts resources based on workload, easier with virtualization.
- **Goals of Data Center Migration**:
    - **Consolidation**: Reduces physical servers, cuts costs, and increases hardware utilization.
    - **Colocation**: Relocates data/apps to specialized facilities for predictable costs and stability.
    - **Load Balancing**: Re-allocates resources to VMs to prevent server overload and performance degradation at peak times.
    - **Maintenance**: Transfers executing VMs to allow physical host updates without disruption.
- **Migration vs. Replication**:
    - **Migration**: **Moving data and applications** from one location/infrastructure to another, usually a **one-time or occasional process** for relocation or upgrade. Involves downtime planning, network connectivity, data consistency, and application compatibility.
    - **Replication**: **Creating and maintaining copies** of data across locations for high availability, redundancy, and disaster recovery. A **continuous or periodic process**. Involves network latency, consistency guarantees, and failover mechanisms.
- **VM vs. Process Migration**: VM migration is preferred as it avoids complex dependencies between processes and local services, and provides **separation of concerns** between users (control software within VMs) and operators (don't care what's inside the VM).
- **VM Migration Components**:
    - **CPU State**: Includes processor registers and cache.
    - **Memory Information**: Includes running processes memory and guest OS memory. Only used memory is migrated to minimize transfer.
    - **Storage Content**: For Network Attached Storage (NAS), storage contents don't need transfer. For local storage, hypervisor identifies unused space to minimize transfer time.
- **Handling Resources during Migration**: Keeping IP addresses while migrating (using ARP to map IP to new host MAC), assuming Network Attached Storage for simplicity.
- **Types of Migration**:
    - **Cold Migration**: **VM is shut down** on source, then restarted on destination after files are moved. Easy to implement but causes service interruption.
    - **Warm Migration**: Suspend VM on source, copy RAM/CPU registers, continue on destination (seconds of disconnection).
    - **Live Migration (Hot/Real-Time Migration)**: **Movement of a powered-on VM** from one physical host to another **without disconnecting clients or applications** (invisible, milliseconds of downtime). VM treated as a black box.
- **Live Migration Total Cost**: Influenced by **scale** (number of servers), **frequency** (load balancing, maintenance, fault tolerance, power management), and **cost of migration** (hardware, network bandwidth, workload performance, service availability).

- **Live VM Migration Phases**:
    - **Push**: Source VM sends its data (including **dirty pages** - modified pages) to destination while running.
    - **Stop and copy**: Source VM stops, transfers control to destination, and remaining data is transferred, causing downtime.
    - **Pull**: Destination VM retrieves missing data from source.
- **Live Migration Approaches**:
    - **Pre-copy**: Phases are **Push + Stop and copy**. State transferred iteratively while source VM runs. Source is suspended when enough state is transferred, then remaining state is copied. Focuses on **small downtime** but **increases total migration time** due to repeated dirty page transfers. Used by Xen, KVM, VMware.
    - **Post-copy**: Phases are **Stop and copy + Pull**. Execution switches to destination almost immediately after migration starts (only CPU state transferred initially). Remaining state is then pulled from source. Minimizes total migration time but VM may suffer **performance penalties due to network page faults**.
    - **Hybrid-copy**: Phases are **Push + Stop and copy + Pull**. Aims for **fast execution transfer and short total duration** for contending VMs under tight time requirements.
- **Management Platforms**:
    - **XO (Xen Orchestra)**: Solution to visualize, manage, backup, and delegate XenServer/XCP-ng infrastructure.
    - **Proxmox Virtual Environment**: Open-source server virtualization platform managing KVM (VMs) and LXC (containers). Provides a single web-based interface for HA, software-defined storage, networking, and disaster recovery. Includes `ha-manager` for automatic error detection and failover. Supports **Online Migration (Live Migration)** and **Offline Migration (Cold Migration)**.
        - **Requirements for Online Migration**: No locally attached devices (except migratable local disks), shared cluster membership, reliable network connectivity, compatible Proxmox versions, and similar CPU architectures.

11. VirtualBox

Oracle VM VirtualBox is a popular Type 2 hypervisor for creating and managing virtual machines on desktop computers.

- **Key Files**:
    - **ISO Images**: Archive files containing identical copies of data from optical discs, used for distributing OSs.
    - **VM Images**: Fully configured Virtual Machines.
    - **VDI file**: A `.vdi` file is a **virtual disk image** specific to VirtualBox, used to start VMs.
- **Open Virtualization Format (OVF)**: A standard for packaging VMs for transport between platforms. **OVF templates** create multiple files, while **OVA** encapsulates all information in a single file.
- **Start Modes**:
    - **Normal Start**: Opens a VM window. Options on close: **Save machine state** (hibernate), **Send shutdown signal**, or **Power Off** (like unplugging).
    - **Headless Start**: VM starts without a video output window, managed via remote desktop (e.g., VRDP, SSH).

    - **Detachable Start**: Combination of normal and headless. VM window can be closed while VM continues running in background.
- **Snapshots**: Capture a VM's state at a point in time, allowing easy reversion to a previous state. Preserves VM state, data, and hardware configuration. Useful for testing/development but **not a substitute for backups**.
- **Cloning a VM**: Copying VM files to quickly create a new VM.
    - **Full Clone**: A complete copy, requiring the same disk storage as the original.
    - **Linked Clone**: Uses the original as a reference and stores changes in a smaller amount of disk storage, but **requires the original VM to be available**.
- **Templates**: A preconfigured, preloaded VM that serves as a mold for creating new VMs. A template **cannot run** and must be converted to a VM for modifications.
- **Clone vs. Template Comparison**: Clones are exact duplicates, can be powered on/off, best for test/dev, not mass deployment. Templates act as a baseline image, cannot be powered on/off or edited directly, best for mass deployment in production.
- **Interaction with VirtualBox**: Primarily via **VBoxManage**, a command-line interface. Commands include `list`, `createvm`, `startvm`, `controlvm`, `showvminfo`. VMs can be specified by name or **UUID** (a 128-bit unique identifier).

12. Docker

Docker is a containerization platform that simplifies application deployment by packaging applications and their dependencies into lightweight, portable units.

- **What is Docker?**: A tool for creating, deploying, and running applications using **containers**.
- **Container Definition**: A software package containing **everything the software needs to run** (executable, tools, libraries, settings). Unlike VMs, containers **do not include a guest OS**; they run on a "container platform" like Docker, which is installed on an OS.
- **Docker Components**: A **daemon process (`dockerd`)**, a **REST API** for communication, and a **command-line interface (CLI) client (`docker` command)**.
- **Docker Image**: A **read-only template** containing instructions for creating a container.
- **Basic Commands**: `docker run` to create and run containers, `-it` for interactive shell, `-p` for port mapping (e.g., `docker run -p 80:80 nginx`).
- **Docker Volume (Persistent Data Management)**: Containers are stateless by default, so Docker provides mechanisms for persistent data.
    - **Bind Mounts**: Maps a custom folder on the host file system to a container file system.
    - **Volume**: Managed by Docker Daemon, stores data produced by a container on a reserved part of the host file system. (e.g., `docker run -v myvolume:/data/db -d mongo`).
- **Docker Network**: Provides default networking for container-to-container and container-to-host communication. Allows creating **user-defined networks** with internal DNS for easier container communication.
- **Dockerfile**: A text document containing **instructions to automatically build Docker images**. Commands define the base image, working directory, copy files, run commands (e.g., `pip install`), and the command to execute when the container starts (e.g., `CMD`). Images are built using `docker build -t image-tag .`.
- **Multi-Container Applications**: Docker supports deploying custom web applications with front-end and back-end services.

- **Docker Compose**: A tool to **define and share multi-container applications** using a **YAML file**. The YAML file defines services, networks, volumes, etc., allowing management with a single command (e.g., `docker-compose up`).

13. Unix/Linux File System

This section delves into specific structures and features of Unix/Linux file systems, particularly `ext4` and the concept of links.

- **File Access Methods**: As mentioned in the general File Systems section, Unix/Linux file systems also use Sequential, Direct, and Indexed access methods.
- **Basic Filesystem Structures**:
  - **Inode (`index node`)**: Every file and directory is represented by an **inode**, which contains **metadata** (owner, access rights) and the **location of the file's blocks on disk**. The `ls -i` command shows the inode number.
  - **Directories**: Special files containing a list of **(filename, inode number) pairs**. Specific inode numbers are reserved (e.g., 0 for NULL, 1 for bad blocks, 2 for root directory).
  - **Pathname Resolution**: Involves starting at the root directory and traversing down the chain of inodes to locate a file.
  - **Locating Inodes on Disk**: Inodes are stored in a top part of the filesystem. The inode number acts as an index to compute the block address of a given inode. This implies a fixed number of potential inodes, set at filesystem creation. The **superblock** stores crucial metadata about the filesystem layout.
- **`ext4` File System**: The fourth extended file system, proposed in 2006 and added to the Linux Kernel in 2008, becoming the default for many Linux distributions.
  - **Structure**: `ext4` filesystems are divided into **block groups**. Each block group has:
    - **Block group descriptor**: Stores details for the group.
    - **Block bitmap**: Tracks block usage status (1 for in-use, 0 for free).
    - **Inode bitmap**: Tracks inode usage.
    - **Inode table**: A table of all inodes in the group.
  - **Group Descriptor Table (GDT)**: Contains block group descriptors for all groups.
- **Soft and Hard Links**:
  - **Hard Links**: Act as a **mirrored copy** of a file, accessing the data available in the original. If the original file is deleted, the data persists as long as one hard link remains. They make **efficient use of disk space** and are fast. However, they **cannot span several file systems** and **directories cannot be hyperlinked**.
  - **Soft Links (Symbolic Links)**: Act as a **pointer or reference to the file name**. They do not access the data directly; if the original file is deleted, the soft link points to a non-existent file. They offer **versatility in linking files across different localities and file systems** and **can link directories**. They are slightly slower than hard links.
  - **Management**: Created using the `ln` command. Removing a hard link (`rm`) only deletes that link, not the file data if other links exist. Removing a soft link deletes the pointer. Orphaned soft links (pointing to non-existent files) can be found and cleaned using the `find` command.

14. sed - Stream Editor

`sed` (stream editor) is a non-interactive command-line tool for performing automatic edits on text files.

- **What is `sed`?**: A **non-interactive stream editor** that interprets instructions to perform edits on files, simplify edits across multiple files, or write conversion programs. It **does not change the input file** directly but sends modified output to standard output.
- **Flow of Control**: `sed` reads an input line, copies it to a **temporary buffer called pattern space**, applies editing commands, sends the modified line to output (unless `-n` option is used), and then removes the line from pattern space before reading the next line.
- **Command Syntax**: `sed -e 'script'` or `sed 'script'` directly. A script is a list of commands, where **each command consists of up to two addresses and an action**.
- **Addressing**: Determines which lines are processed. If no address, applied to all lines.
    - **Types**:
        - **Single-Line Address**: Specific line number or `$` for the last line.
        - **Set-of-Lines Address**: Uses **regular expressions (`/pattern/`)** to match lines, which may not be consecutive. Regular expressions use **Basic Regular Expressions (BREs)** like in `grep`.
        - **Range Address**: Defines a set of **consecutive lines** using `start-addr,end-addr` (inclusive), where addresses can be line numbers or patterns.
    - **Exclamation Point (`!`)**: When used with an address (e.g., `/pattern/!`), the instruction applies to all lines that **do not match** the address.
- **`sed` Commands**:
    - **Line Number (`=`)**: Writes the current line number before each matched/output line.
    - **Print (`p`)**: Forces the pattern space to be output. Useful with `-n` (suppresses automatic printing). If `-n` is not used, `p` will cause the line to be output twice.
    - **Delete (`d`)**: **Deletes the entire pattern space**. Commands following it are ignored for that line.
    - **Insert (`i`)**: **Adds one or more lines to the output before the address**. The inserted text does not appear in pattern space. Cannot be used with a range address.
    - **Append (`a`)**: **Adds one or more lines to the output after the address**. Similar to `i`, inserted text not in pattern space, cannot use range address.
    - **Change (`c`)**: **Replaces an entire matched line with new text**. Accepts all four address types (single-line, set-of-line, range, nested).
    - **Transform (`y`)**: **Translates one character to another** (e.g., `y/abcd/wxyz/`). Cannot use regular expressions or character ranges. Requires the same number of characters in both sets.
    - **Substitute (`s`)**: **Replaces text selected by a search string with a replacement string**. The search string can be a **regular expression**.
        - **Flags**:
            - `g` (global): Replaces **all occurrences** in the line.
            - `p` (print): Prints the line if a replacement was made.
            - `num`: Replaces only the `n`-th occurrence.
            - `w file`: Writes the modified line to a specified file.
- **Multiple Commands**: Can be chained using multiple `-e` options (e.g., `sed -e 'cmd1' -e 'cmd2'`) or grouped using **braces `{}`** applied to an address (e.g., `/pattern/{cmd1; cmd2;}`).