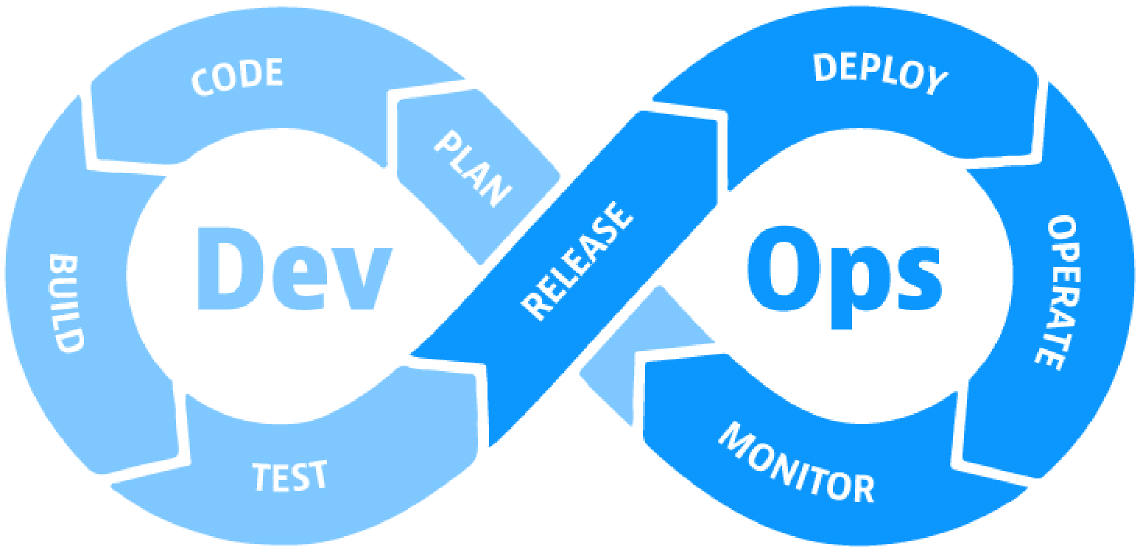
In the name of god

Mohsen Sarabi

First homework of Devops mentorship-lpic1

L1-JahanPahlevanan



1-

a-Motherboard(System Board)

The main printed-circuit board that interconnects all other parts; houses the CPU socket, chipset, memory slots, power connectors, and expansion buses.

b-CPU (Central Processing Unit)

The "brain" of the server that executes instructions and performs arithmetic/logic operations; core count, clock speed largely determine compute performance.

c-RAM (Memory)

Volatile storage that holds data and code currently in use. Larger capacity and bandwidth allow heavier workloads.

d-Storage Drives (HDD / SSD / NVMe)

Non-volatile media for OS and data. HDDs offer high capacity at lower cost but slower I/O; SATA/SAS SSDs give higher speed and durability; NVMe drives on PCIe deliver the highest throughput and lowest latency.

e-RAID Controller (Smart Array)

Hardware/firmware that groups multiple disks into RAID sets for redundancy or speed (RAID 0, 1, 5, 6, 10, etc.). Often equipped with write-back cache plus battery or super-cap to protect data in transit.

f-Power Supply Unit (PSU)

Converts incoming AC to stable DC voltages for the system. Server PSUs are usually hot-swappable and redundant, so another unit can take over instantly if one fails.

g-Network Interface Card (NIC)

Provides the server's connection to Ethernet or fiber networks (1 – 100 GbE+). Supports features such as PXE boot, VLAN tagging, and teaming/bonding for higher bandwidth or fail-over.

h-Cooling System (Fans & Heat Sinks)

Dissipates heat generated by CPUs, GPUs, and PSUs. Heat sinks spread thermal energy, while variable-speed fans expel it; precise thermal control is essential for stability and component lifespan.

i-Expansion Slots (PCIe)

High-speed lanes for add-in cards such as extra NICs, GPUs, storage HBAs, or RAID/NVMe adaptors. Bandwidth scales with lane count and PCIe generation (x4, x8, x16, Gen 3/4/5).

j-Chassis (Rack / Tower / Blade)

Physical enclosure. Rack-mount units (1U–4U) suit dense data centers; tower servers fit office environments; blade servers slot into a shared enclosure for maximum density and consolidated power/networking.

k-BIOS / UEFI Firmware

Low-level firmware that initializes hardware, runs POST, and hands control to the bootloader. UEFI adds a graphical setup, GPT disk support, and Secure Boot.

l-Backplane

A mid-plane PCB inside the chassis that drives plug directly into, distributing power and data without individual cables and enabling hot-swap of drives or blades.

2-

IPMI (Intelligent Platform Management Interface) is an open standard for out-of-band hardware management. A dedicated management controller on the motherboard lets administrators power a server on, off, or reset it even when no operating system is running. IPMI reports sensor data such as temperatures, voltages, fan speeds, and PSU status, records hardware events in the System Event Log, and—depending on the implementation—offers Serial-over-LAN or basic KVM-over-IP plus remote firmware and BIOS updates.

iLO (integrated Lights-Out) is Hewlett Packard Enterprise's proprietary enhancement of the same idea. An onboard iLO ASIC provides a full-featured web GUI, CLI, and REST APIs. Beyond standard IPMI functions, it delivers advanced KVM-over-IP with virtual media mounting, predictive power and thermal management, and tight integration with HPE orchestration tools like OneView. Security is strengthened with encrypted sessions, Secure Boot, and a Silicon Root of Trust that validates firmware at boot.

3-

IPMI and iLO do not replace the BIOS or UEFI firmware; rather, they form a management layer that operates alongside the system firmware and remains reachable even when the OS—or the firmware setup screen itself—is unavailable.

1. Out-of-band power and control:
   The IPMI/iLO management controller has its own power and dedicated network path, so it can power the server on, off, or reset it and force entry into the BIOS/UEFI setup regardless of the system's current state.

2. Remote console tunneling:
   Both technologies capture the video (or serial) output generated by BIOS/UEFI during POST and stream it over KVM-over-IP. Administrators can thus edit firmware settings, change boot order, or initiate firmware updates exactly as if they were in front of the machine.

3. Firmware flashing:
   Through the web UI or CLI, you can upload a BIOS/UEFI image to the management controller, which then handles flashing, verifies integrity, and can fall back to a prior image if the update fails—reducing the risk of bricking the motherboard.

4. Continuous hardware monitoring beyond POST:
   While BIOS/UEFI checks sensors only during boot, IPMI/iLO keep monitoring temperatures, voltages, and fan speeds in real time, logging anomalies in the System Event Log and pre-emptively ramping fans to avert overheating.

4-

A CPU socket on a server motherboard is the physical receptacle that holds a processor package and links it electrically to the rest of the system. Its purposes include:

1. Reliable electrical interface: Hundreds or thousands of pins or pads (in LGA sockets) carry power, data, and address signals between the CPU and chipset, memory buses, and I/O lanes, maintaining solid contact under vibration and thermal cycling.

2. Serviceability and upgrades: Because the processor is socketed rather than soldered, it can be swapped out—say, to add a higher-core-count model—as long as the board and firmware support the new chip.

3. Multi-socket scalability: Enterprise motherboards often feature two or four sockets, enabling multiple physical CPUs to share workload, expand total memory capacity, and multiply PCIe/IO resources for demanding server applications.

4. Thermal conduction platform: The socket provides a flat, secure surface to clamp heat sinks or liquid-cooling blocks onto, which is critical for dissipating the high thermal output of server CPUs.
5. Standardization: Each CPU generation defines a specific socket (e.g., Intel LGA-4189, AMD SP5) with set pin counts, voltages, and signal mappings, allowing interoperable designs for motherboards and coolers

----

5-

Linux introduced *pseudo-file systems*—such as /proc, /sys, and /dev—because of its design mantra that "everything is a file." Instead of special ioctl calls or binary control utilities, the kernel exposes internal data structures and device interfaces as ordinary files and directories that can be read or written.

Key motivations:

1. Uniform, simple interface
   Users and scripts interact with kernel information using the same commands they use for regular files (cat, echo, grep, etc.), eliminating the need for specialized tools or complex APIs.
2. Alignment with Unix philosophy
   Classic Unix design encourages small programs that read from stdin and write to stdout. Presenting kernel state as plain text files lets standard text-processing tools inspect, filter, and pipe that information effortlessly.
3. Ease of extension
   Adding a new kernel feature often means creating a new file or directory under /proc or /sys, without breaking existing user-space interfaces. This keeps kernel development and maintenance straightforward.
4. In-memory, low-overhead access
   These file systems are ram-backed; they reveal live kernel data without disk I/O, providing very fast access while keeping the on-disk file system uncluttered.
5. Portability of utilities
   Because all major distributions expose kernel details through the same pseudo-file paths, scripts and monitoring tools work consistently across kernels and vendors

----

6-

A regular file system—such as ext4 or XFS—stores its data on persistent media (HDD or SSD). Whatever you write there survives power-off. It's designed for long-term storage of user files and system resources, complete with on-disk metadata like block allocation tables, journals, and inodes. Every read or write ultimately touches physical hardware, so it inherits disk latency and bandwidth limits.

A pseudo file system—like /proc, /sys, or /dev—is not backed by disk at all. The Linux kernel creates these trees in RAM or synthesizes their contents on the fly. Their purpose is to expose live kernel and device state: you can cat CPU temperatures from /proc or echo new parameters into /sys to tweak the kernel. The data vanishes on reboot, but access is virtually instantaneous because no disk I/O occurs. Permissions are set dynamically by the kernel, and many pseudo-files are read-only, write-only, or undeletable.

7-

sys/ is a live, in-memory view of the Linux kernel's device model. Each top-level directory represents a different slice of hardware or kernel subsystems:

- block – All block devices (disks, SSDs, NVMe). Contains sector sizes, performance flags, hardware IDs, and runtime I/O stats.
- bus – Enumerations of buses (PCI, USB, I2C, etc.) with every device hanging off each bus, showing addresses, resources, and hierarchy.
- class – Logical groupings such as net (network interfaces) or leds, letting you list all devices of a given class regardless of physical bus.
- dev & devices – dev maps major/minor numbers; devices is the complete physical tree from root buses down to each device node.
- firmware – Hooks the kernel uses to locate and load external firmware blobs for hardware that needs them.
- fs – Runtime details of mounted file systems and kernel-internal mounts like cgroups.
- hypervisor – Present when running under virtualization, exposing information about the host hypervisor (KVM, Xen, etc.).
- kernel – Tunables and stats tied directly to the kernel itself: printk settings, scheduler, tracing switches, and more.
- module – Status and parameters of loaded kernel modules (drivers, fs). Shows exported symbols and reference counts.
- power – System-wide and per-device power-management knobs (autosuspend, low-power states, wake-up capabilities)

8-

Direct Memory Access (DMA) is a mechanism that lets hardware devices move data between main memory and the device itself without the CPU touching every byte. A DMA controller orchestrates the transfer; the CPU merely programs the source, destination, and length, then gets interrupted when the job completes, freeing processor cycles for other work.

Typical Linux use cases include:

- Disk and NVMe I/O – Storage drivers rely on DMA to copy data blocks straight between an SSD/HDD and kernel buffers, delivering high throughput while keeping CPU load low.
- High-speed network cards – Ethernet frames land directly in ring buffers in RAM (and vice versa), enabling millions of packets per second.
- Audio/Video devices – Continuous streams move via DMA into ring buffers, preventing glitches in real-time playback or capture.
- GPUs and accelerators – Textures or matrices are shuttled across PCIe using DMA (or RDMA) between host memory and the device's VRAM.
- Embedded systems – On ARM SoCs, DMA engines read sensor data or feed ADC/DAC peripherals autonomously, letting the CPU sleep.

9-

lsblk (List Block Devices) does not talk to disk drivers directly. Instead, it gathers information from kernel-exposed pseudo-files and then formats it:

1. Primary data lives under /sys/class/block/. Each block device has a directory; lsblk reads files like size, ro, queue/*, and device/*.
2. To map partitions, parent/child relationships, and labels, it also examines /sys/block/<dev>/ and looks at udev links in /dev/disk/by-*.
3. It performs no heavy ioctl calls; almost everything comes from sysfs plus the udev database in /run/udev/data/.
4. The userspace program builds a tree from that data and prints the nicely aligned output you see.

Do lsusb, lspci, and lshw work the same way?

- lsusb mainly parses /sys/bus/usb/devices/ and /proc/bus/usb/devices, but to fetch detailed descriptors it issues ioctl reads on /dev/bus/usb/*/*.
- lspci starts with /sys/bus/pci/devices/, yet for raw PCI config registers it still hits /proc/bus/pci/*/* or even /dev/mem via ioctl, which is why deep info requires root.
- lshw is the most exhaustive; it combines data from /proc, /sys, dmidecode (SMBIOS via /dev/mem), SCSI ioctls, and more, even probing NICs over the network. Hence its output is richer but slower to generate.

10-

To trigger a shutdown through the /sys interface:

# Tell the kernel to power off after hibernating

```
echo shutdown | sudo tee /sys/power/disk


# Request the 'disk' (hibernate) power state
echo disk | sudo tee /sys/power/state
```

echo shutdown > /sys/power/disk sets the post-hibernate action to power-off.

echo disk > /sys/power/state makes the kernel save RAM to swap, then cut the power—effectively a shutdown.

For an immediate power-off without hibernation, /sys has no direct node, but you can use the SysRq mechanism:

```
echo 1 | sudo tee /proc/sys/kernel/sysrq   # enable SysRq if needed

echo o | sudo tee /proc/sysrq-trigger      # 'o' = power off
```

11-

Monolithic kernels bundle nearly all OS services—process scheduler, memory manager, filesystems, drivers, networking—inside a single privileged address space.

Pros: very fast in-kernel calls; simple tight integration; rich driver ecosystem (e.g., Linux).
Cons: any driver bug can crash the whole OS; large codebase harder to maintain; most updates require reboot.

Microkernels keep only minimal mechanisms (scheduling, basic MMU, IPC) in kernel space; everything else runs as user-space servers communicating via message passing (e.g., Minix 3, seL4).

Pros: strong fault isolation; improved security; services can be restarted or upgraded on the fly.
Cons: message-passing overhead hurts performance; design & driver development more complex; smaller driver ecosystem.

Hybrid kernels (Windows NT, Apple XNU, QNX) adopt a microkernel posture but keep high-traffic components in kernel space for speed.

Pros: better performance than pure microkernels while retaining some separation; easier access to large driver libraries.
Cons: high architectural complexity; significant kernel-space code still risks system crashes; blurry boundary may bloat the kernel over time.

Bottom line:

- *Monolithic* = speed and simplicity, less isolation.

- *Microkernel* = robustness and security, potential performance hit.
- *Hybrid* = compromise: aims for both but inherits some drawbacks of each

----

12-

On IBM-PC–compatible systems with legacy BIOS, the firmware's boot routine is hard-wired to read exactly one 512-byte block—sector 0 of the drive—into memory and jump to it. That sector became the Master Boot Record (MBR) for several reasons:

1. Hardware simplicity: Early disk controllers and BIOS ROM space were tiny. Reading a single, fixed-location sector required minimal code in ROM.
2. Fixed addressing: BIOS issued a straightforward CHS command: "Read Cylinder 0, Head 0, Sector 1" (sector numbers started at 1, but firmware treated it as the first physical block). Having the boot code there meant BIOS needed no knowledge of partitions.
3. Dual purpose space: 512 bytes could accommodate a tiny bootstrap program (≈ 446 bytes), a 64-byte partition table (four entries), and the 2-byte boot signature 0x55AA. This let the MBR both start the boot chain and describe how the rest of the disk is divided.
4. Backward compatibility: Once operating systems and tools depended on this convention, keeping the boot record in the first sector became an industry standard that persisted until GUID Partition Table (GPT) and UEFI redefined the process

----

13-

Because the MBR's 446 bytes of code are too small to understand modern file systems, boot menus, or even GRUB 2 itself, BIOS/MBR booting is a staged process:

1. Read the partition table:
   The last 64 bytes of the MBR hold four partition entries. The MBR code scans them for the *bootable* flag and notes the starting LBA of that partition.
2. Jump to the Partition Boot Record (PBR/VBR):
   Using that LBA, the MBR loads one more sector—the first sector of the active partition. That sector contains tiny bootstrap code (Stage 1 of GRUB or another loader).
3. Load GRUB Stage 1.5 / Stage 2:
   - Stage 1 can't parse file systems, so at install time grub-install stores the next few blocks (Stage 1.5) in the post-MBR gap—the unused space between cylinder 0 sector 1 and the first partition—or in contiguous early blocks of the partition.
   - Stage 1.5 understands basic file-system drivers; it locates and loads Stage 2, the full GRUB binary residing under /boot/grub/.

    - Stage 2 reads grub.cfg, presents a menu, and finally loads the Linux kernel vmlinuz and initrd.

4. How does the MBR know those block addresses?
   During installation, GRUB records the exact disk blocks of Stage 1.5 inside unused bytes of the MBR/PBR (block-list). If the files move, re-running grub-install refreshes those addresses.

14-

efi files are executable binaries designed for the UEFI firmware environment. These files reside in the EFI System Partition (ESP)—typically a small FAT32-formatted partition—and they play a central role in booting modern systems.

Role of .efi files in the boot process:

1. UEFI replaces BIOS:
   UEFI firmware reads files from disk like a small OS. It can parse FAT32 and run .efi binaries directly—no need for MBR chainloading.
2. Boot starts by running .efi files:
   At power-on, the UEFI firmware searches the ESP for a default or registered boot entry (like BOOTX64.EFI), then loads and executes that .efi file. This could be a boot manager (GRUB, systemd-boot) or even the kernel itself.
3. Support for multiboot and Secure Boot:
   UEFI systems support multiple .efi files—each representing a different OS or boot config. In Secure Boot mode, the firmware checks the digital signature of .efi files to ensure authenticity before executing.

15-

The ESP (EFI System Partition) is a special disk partition used by UEFI-based systems to store the files required to boot the operating system. It contains bootloaders, kernel images, and UEFI applications in the form of .efi files.

Purpose and Usage of ESP:

1. Bootloader Storage: The ESP hosts bootloaders like grubx64.efi, bootmgfw.efi (Windows), systemd-bootx64.efi, and sometimes even the OS kernel itself (as .efi).
2. Standardized UEFI Boot Path: UEFI firmware searches for a bootable .efi file within the ESP either at a default path (like /EFI/BOOT/BOOTX64.EFI) or based on entries stored in NVRAM.
3. Multi-OS Friendly: Multiple OS bootloaders can coexist within the ESP, each in its own subfolder.
   Examples:

/EFI/ubuntu/grubx64.efi

/EFI/fedora/shimx64.efi

/EFI/BOOT/BOOTX64.EFI

4. Secure Boot Integration: Files in the ESP can be digitally signed. If Secure Boot is enabled, UEFI firmware checks signatures before executing .efi files to ensure trusted boot.

16-

In this section of the GRUB configuration file (/etc/grub/grub.conf), an entry is defined to boot the Ubuntu operating system. This entry contains information for identifying and loading the Linux kernel and other necessary files for booting the system. First, the menu item is set up with the name "Ubuntu" and related classes for proper categorization.

GRUB then uses the set root command to specify the root partition (in this case, the second partition on the first disk using GPT). It loads the necessary file system and partition modules like part_gpt and ext2. Afterward, GRUB searches for the partition containing the operating system using the search command, and the Linux kernel is loaded from the specified path. Additionally, the initrd is used to load essential drivers during the early stages of the boot process.

Finally, graphical settings for the boot process are applied. These configurations allow GRUB to properly boot Ubuntu and enable features like the installer if needed.