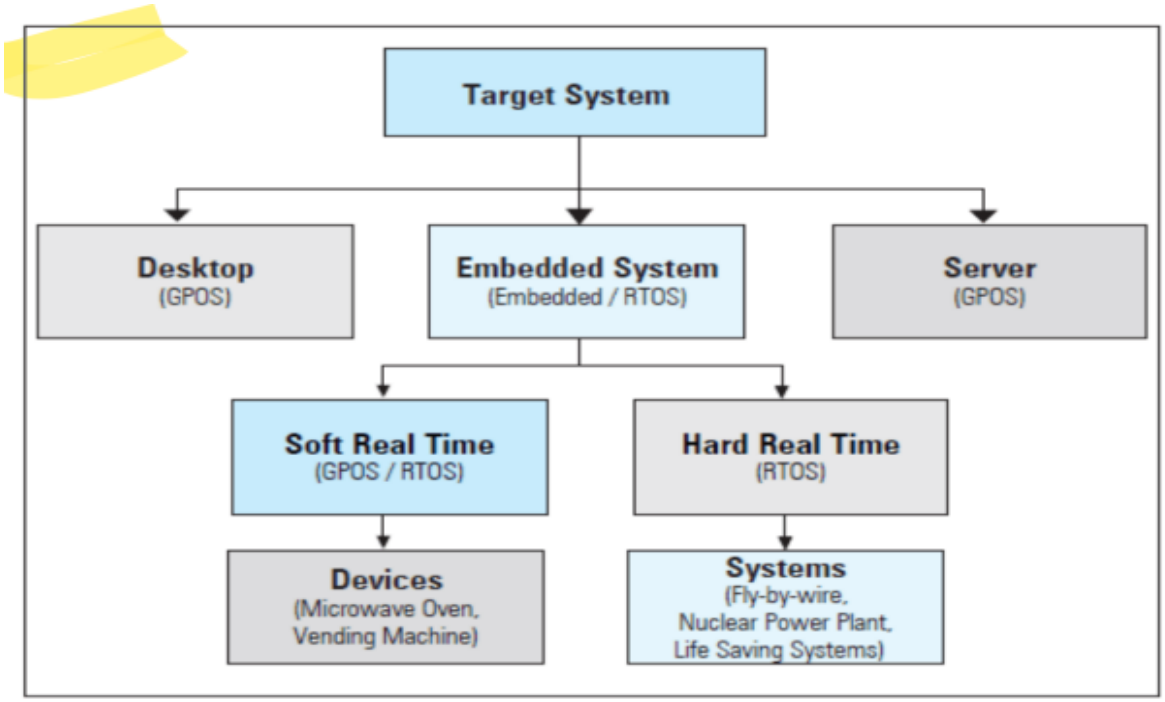
Structure of Computing System

Memory

Processor

Input & Output

Devices Types of Systems



1. Target System

The top-most box is Target System → it represents any computing system we are designing (desktop, server, embedded, etc.).

---

2. Main Types of Systems

From Target System, we branch into:

1. Desktop (GPOS)
    - Runs a General Purpose Operating System (GPOS) like Windows, Linux, macOS.
    - Designed for multitasking and throughput (maximizing performance).
    - Not strict about deadlines.
2. Embedded System (Embedded / RTOS)
    - A dedicated system performing specific tasks.
    - Examples: Printers, washing machines, routers.
    - Can run either:
        - GPOS (for simple tasks)
        - RTOS (for strict real-time deadlines)

3. Server (GPOS)
    - Uses GPOS like Linux, Windows Server.
    - Optimized for handling multiple users, high throughput, networking, database services.

---

3. Inside Embedded Systems

Embedded systems can further be divided into Soft Real-Time and Hard Real-Time:

a. Soft Real-Time Systems (GPOS / RTOS)

- Missing a deadline is not fatal, but performance suffers.
- Example: Microwave oven, Vending machine.
- Usually runs on Linux with real-time extensions.
- Prioritizes throughput but tolerates minor delays.

b. Hard Real-Time Systems (RTOS only)

- Deadlines are critical → missing them may cause failure or danger.
- Example: Nuclear power plant controls, Fly-by-wire (aircraft systems), Life-saving medical devices.
- Requires RTOS → ensures tasks always meet deadlines.
- The OS is stripped down to include only what's essential, so real-time tasks never get delayed by non-real-time tasks.

---

4. GPOS vs RTOS

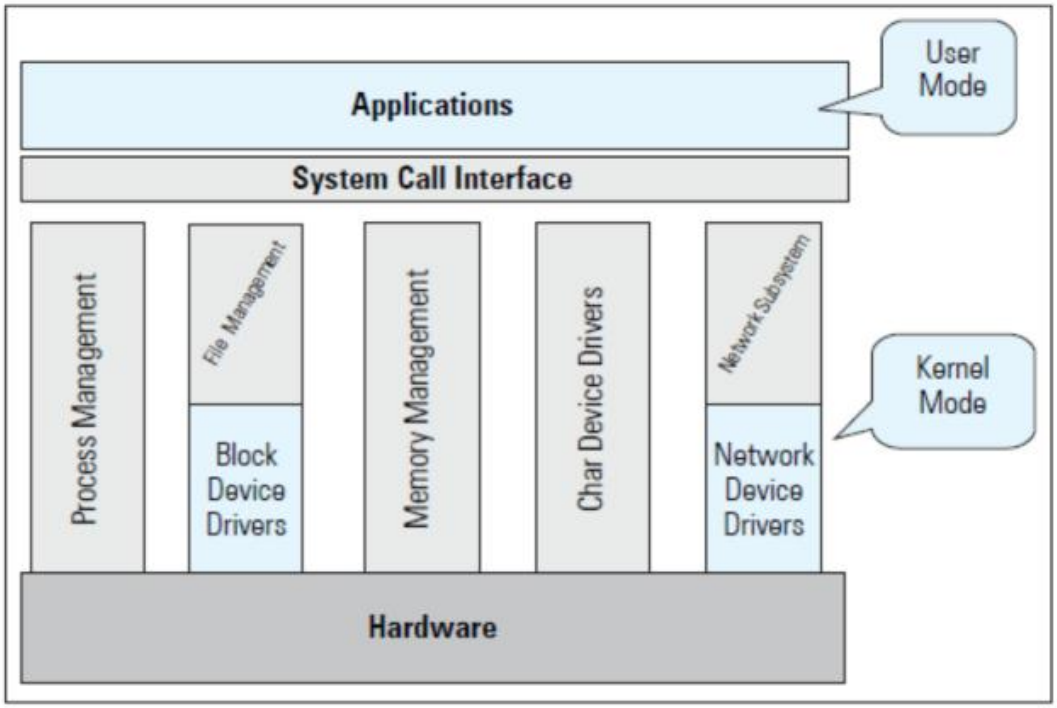
- GPOS (General Purpose OS):
    - Focus: High throughput, fairness among tasks.
    - Example: Windows, Linux, macOS.
    - Not deadline-sensitive.
- RTOS (Real-Time OS):
    - Focus: Predictability & deadlines.
    - Always schedules real-time tasks first.
    - Example: VxWorks, FreeRTOS, QNX, RT-patched Linux.

---

5. Key Concepts

- Throughput: How many tasks are done per second. Important in desktops/servers.
- Deadlines: Very important in RTOS (especially hard real-time).
- Priority in Linux: If you type top in terminal → processes with "rt" priority are real-time tasks.

GPOS



1. Two Modes of Operation

- User Mode → Where your apps (like Chrome, Word, games) run.
- Kernel Mode → Where the OS core (kernel) runs, managing hardware and resources.
  👉 Apps cannot directly talk to hardware, they must go through the kernel.

---

2. System Call Interface

- This is like a gate/bridge between apps and the kernel.
- When an app needs hardware access (e.g., saving a file, printing, sending data online), it makes a system call.
- System calls always run in kernel mode.

---

3. Kernel Components (things inside Kernel Mode)

- Process Management → Handles creation, execution, and termination of processes.
- File Management → Works with files stored in devices (using block device drivers).
- Memory Management → Allocates/deallocates memory to processes.
- Device Drivers → Special software that helps OS talk to hardware.
    - Block Device Drivers → For storage devices (HDD, SSD, CD-ROM). Data is accessed in blocks with caching.
    - Character Device Drivers → For stream-based devices (keyboard, serial ports, tape drives). Data flows as a stream, no random access.

- Network Device Drivers → Connect network hardware (NIC, Wi-Fi, modem) to the network subsystem (TCP/IP stack).

- Network Subsystem → Implements networking following 7 OSI layers (application → physical).

---

4. Hardware

- The lowest level: actual physical devices (CPU, RAM, disk, network card, etc.).

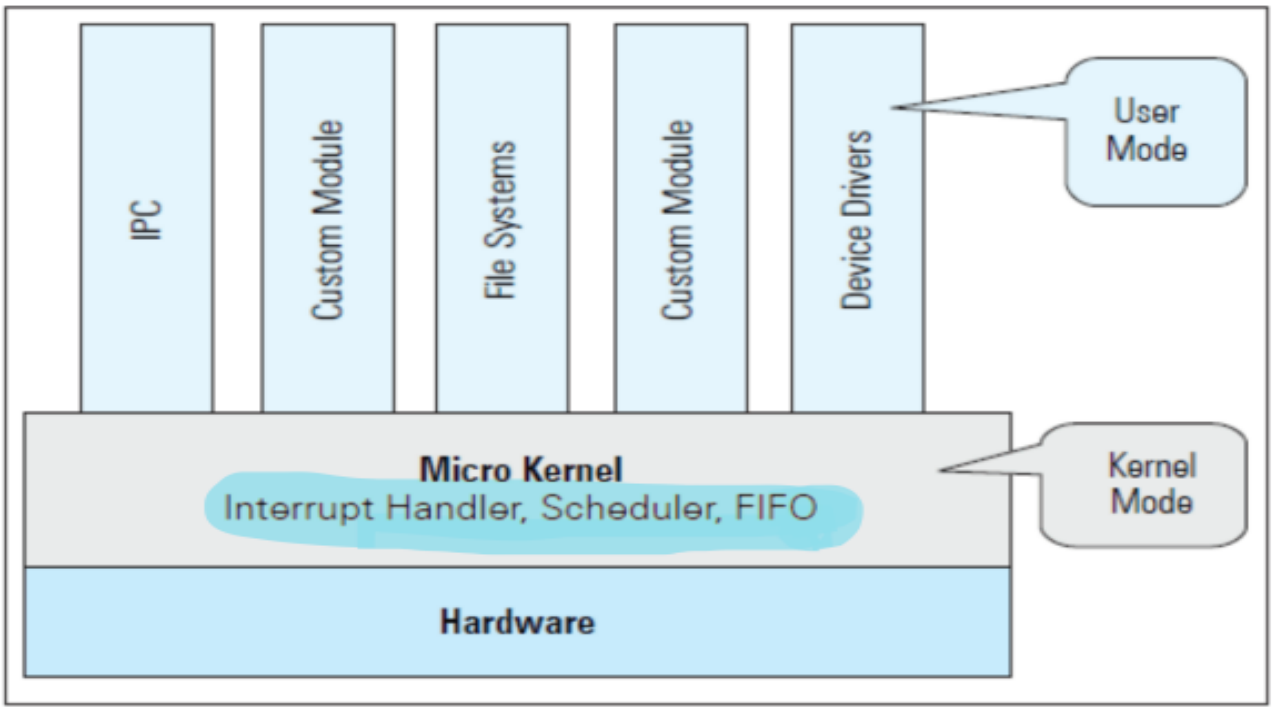
---

5. Flow Example

Suppose you want to download a file from the internet:

1. Your browser (app → user mode) makes a system call.
2. System call passes request to kernel.
3. Kernel uses network subsystem + network device drivers to receive packets.
4. File management + block device driver store the file onto disk.
5. Hardware (disk + network card) actually performs the physical actions.

Microkernel



1. What is a Microkernel?

A microkernel is the smallest possible kernel (core part of an operating system).
It only keeps the most basic and essential functions inside the kernel, and moves everything else outside into user mode.

---

2. What stays inside the Microkernel (Kernel Mode)?

Only very basic things:

- Interrupt Handler → manages signals from hardware (like keyboard, mouse, network).
- Scheduler → decides which process should run next.
- FIFO (communication pipe) → a simple way for processes to talk to each other.

---

3. What goes outside in User Mode?

Other services like:

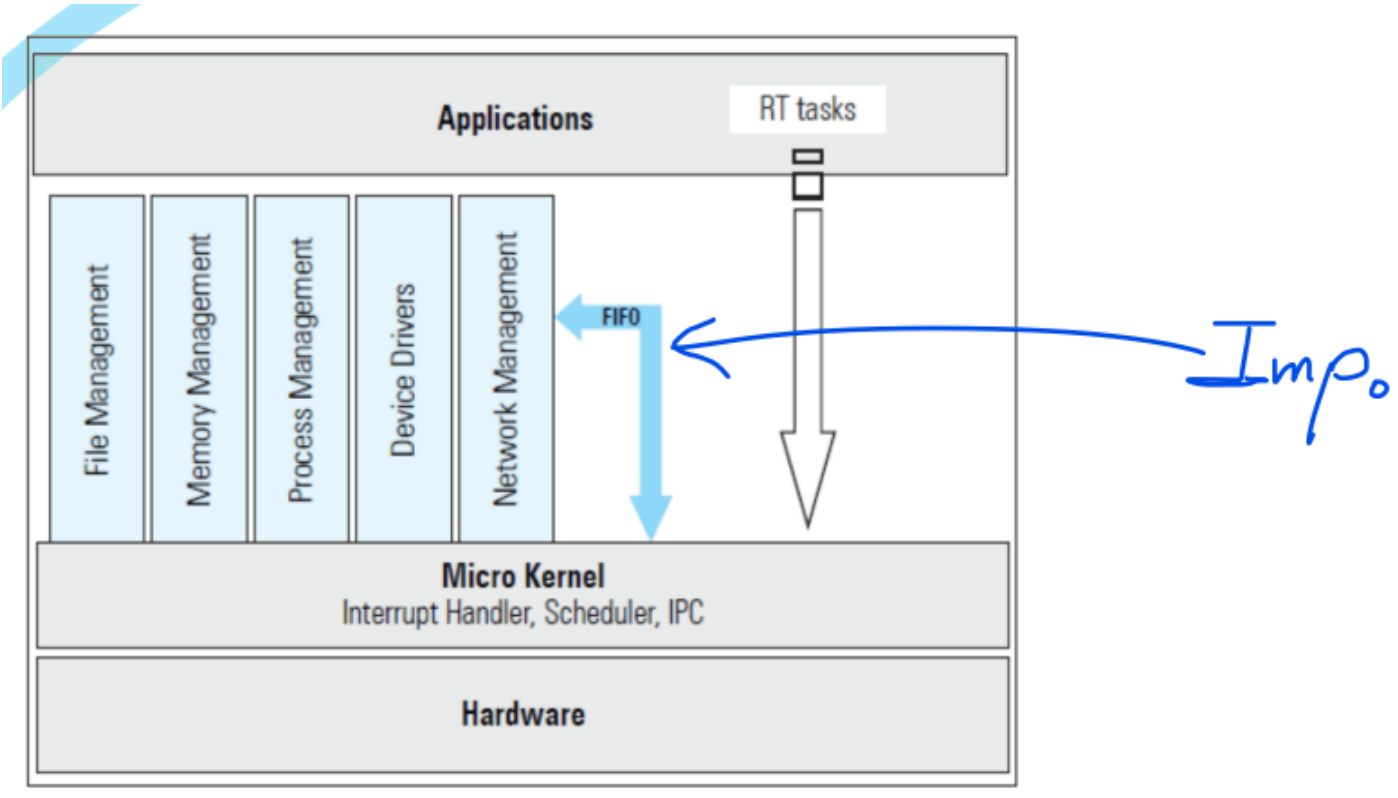
- File systems
- Device drivers
- Memory management
- Extra/custom modules

These are not part of the kernel, they run in user space.

4. Why is this useful?

- Customized for tasks → You only keep what you need.
- Safe & Stable → If a file system or driver crashes, the kernel won't crash (since they are outside kernel mode).
- Deterministic behavior → Predictable and reliable performance.

Hybrid Kernel



How a Hybrid Kernel Works

A traditional monolithic kernel, like the one in most Linux distributions, handles all core services—like file, memory, and process management—within a single, large block of code. A microkernel, on the other hand, keeps the kernel as small as possible, moving most services out into separate user-space processes.

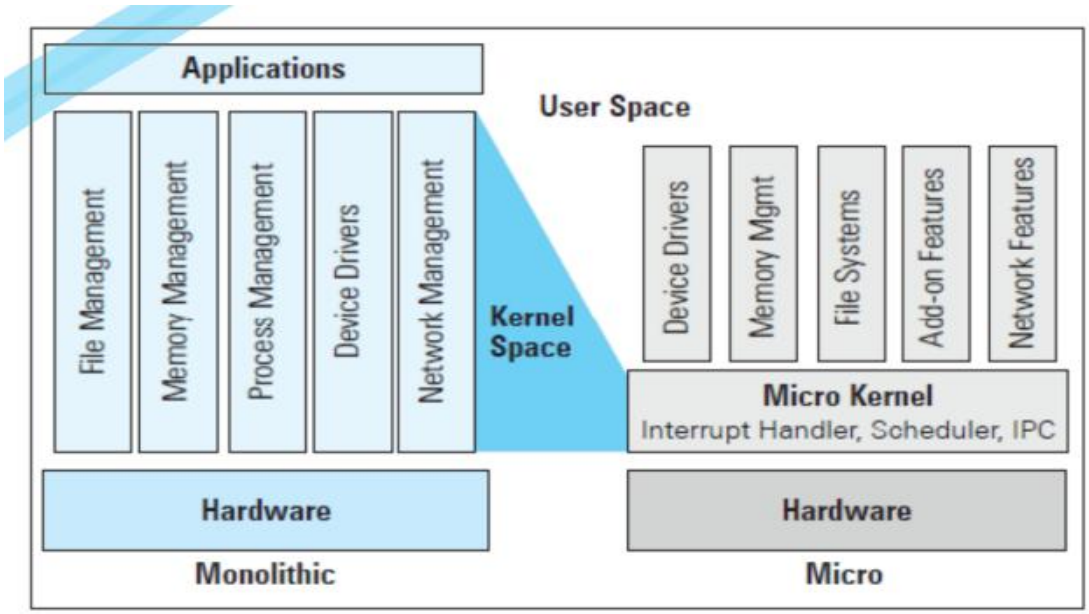
A hybrid kernel for real-time systems takes a microkernel and runs the entire Linux kernel as a single, prioritized process within it. This Linux kernel is often referred to as the "GPOS" (General Purpose Operating System) kernel. This setup allows for two main types of tasks:

- Real-Time (RT) Tasks: These tasks have strict timing requirements. When an RT task needs to run, the microkernel recognizes it and gives it direct access to the hardware and its own scheduling mechanisms (the Interrupt Handler, Scheduler, and IPC shown in the image). This bypasses the GPOS kernel entirely, ensuring the task is executed quickly and reliably.
- General Purpose (GPOS) Tasks: These are your regular applications (like web browsers, word processors, etc.). When there are no RT tasks to run, the microkernel takes on tasks from the GPOS kernel. These tasks use a FIFO (First-In, First-Out) communication method to pass requests to the microkernel.

Key Concepts Explained

- Real-Time (RT) System: A system where the correctness of a task depends not only on the result but also on the time it takes to produce that result. Think of a car's anti-lock braking system—it must react within milliseconds to be effective.
- GPOS Kernel: This is the standard Linux kernel that handles everyday tasks. It's not designed for strict timing and can have delays due to its complex functions.
- FIFO (First-In, First-Out): A simple communication method where the first request that comes in is the first one to be handled. In this hybrid kernel, the GPOS kernel uses FIFO to send requests for resources (like a device driver) to the microkernel.
- Device Driver vs. Device Controller:
    - A device controller is the hardware component—the electronic brain that manages a physical device (e.g., a hard drive).
    - A device driver is the software that acts as a translator, allowing the operating system to talk to the device controller.
- Operating System (OS) = Kernel + Utilities: The kernel is the core of the OS that manages hardware. The utilities are the tools and programs that make the OS useful to the user. Different Linux distributions (like Red Hat, Ubuntu, etc.) are built on the same core Linux kernel but include different sets of utilities, which is what gives them their unique flavor.

Monolithic vs Micro Kernel



1. Monolithic Kernel (left side of the image)

In a **monolithic kernel**, **everything runs inside the kernel space**.

What's in "kernel space"?

Kernel space means **a protected area of memory** where only the OS kernel runs.
It has full control over the system.

What runs inside it?

In the monolithic design:

- File management
- Memory management
- Process management
- Device drivers
- Network management

All these big services run **together inside the kernel**.

So the kernel is **large and powerful**, but also **complex** and **riskier** —
if one part crashes (say a driver), it can crash the whole system.

---

2. Micro Kernel (right side of the image)

In a **microkernel**, only the **most essential** parts stay in the kernel space:

- Interrupt handling
- Scheduling (deciding which process runs)
- Inter-process communication (IPC)

Everything else — file systems, device drivers, memory management, etc. —
runs in **user space** (outside the kernel).

---

3. Main difference

Feature	Monolithic Kernel	Micro Kernel
Kernel Size	Large (many services inside)	Small (only core functions)
Performance	Faster (no user–kernel switching)	Slightly slower (more communication)
Stability	Less stable (one crash can bring down system)	More stable (faults isolated in user space)
Example OS	Linux, Unix	Minix, QNX, early Mac OS X

---

4. The highlighted line in your image

"Monolithic places all of the services in the Kernel space."

That means:

- In a **monolithic kernel**, all important OS services (drivers, file systems, etc.) live **inside the kernel**.
- In a **microkernel**, they're moved **outside**, making the kernel much **smaller** and **more modular**.

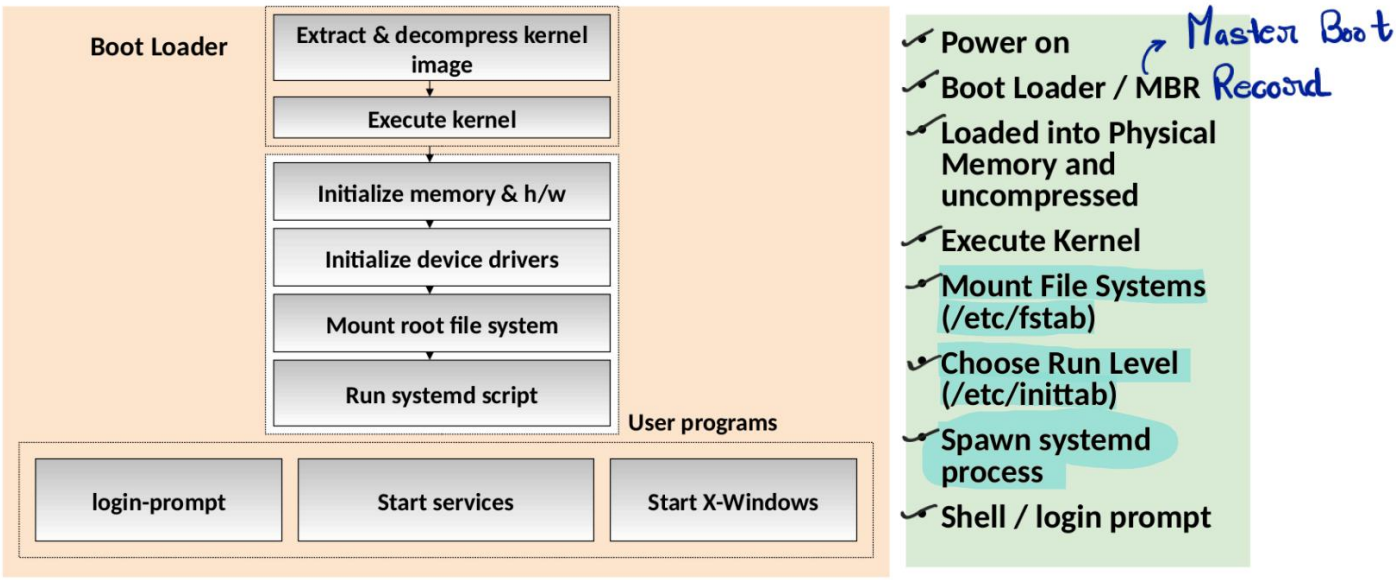
Easy Analogy:

Imagine your **kernel** is a **restaurant kitchen**

Monolithic Kernel → Everything (chefs, waiters, cashiers, cleaners) works inside the kitchen.
→ It's fast, but if something goes wrong (like a fire), the whole kitchen shuts down.

- **Microkernel** → Only the main chefs stay inside the kitchen.
  → Waiters, cleaners, and cashiers work outside.
  → It's safer and more organized, but communication takes slightly longer.

Booting Procedure



Step 1: Power On → Bootloader

- You press the power button.
- The bootloader (like GRUB) runs first.
- It shows you options (like Ubuntu, Windows, recovery mode).
- To let you type and see the menu, a temporary mini filesystem (ramfs) loads drivers for keyboard & screen.

Step 2: Kernel

- The Linux kernel (the brain of the OS) is stored in a compressed form (like a zip file).
- It gets unzipped (decompressed) and then starts running.
- Once the kernel takes over, the temporary **ramfs** is no longer needed and gets removed.

Step 3: Systemd (the manager)

- The first program that runs is called systemd (old systems used init).
- Systemd has PID = 1 (you can check with pstree).
- Its job: start all other programs → network, logins, background services, etc.
- It can do many things at once (multi-core), unlike the old init.

Step 4: Filesystem

- The kernel mounts the root filesystem /, which is the top of everything.
- Every file, folder, and device lives under /.

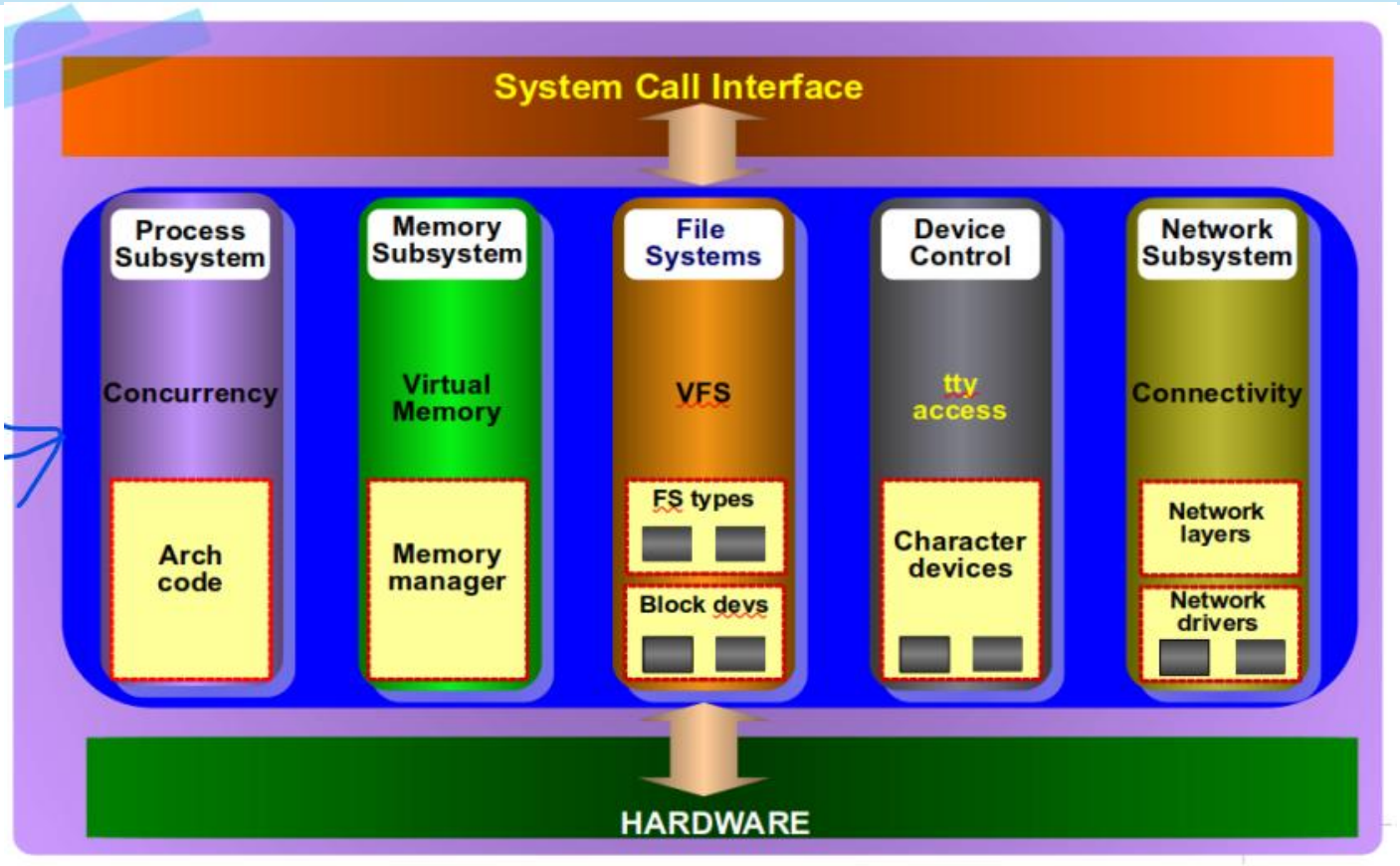
Step 5: User Interface

- If you use a desktop, the X Window system (or Wayland) provides the GUI (what you see on the screen).
- Without it, you'd just see a black terminal.

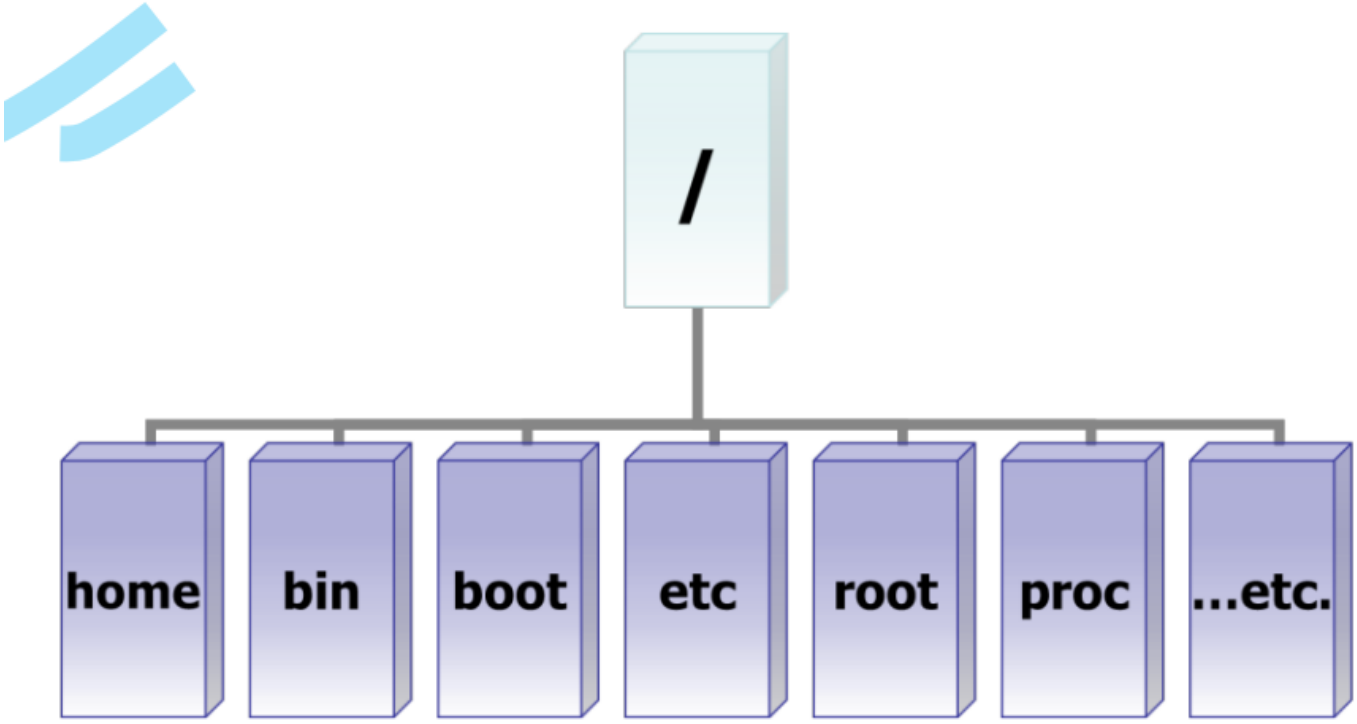
Step 6: Kernel Strength

- Linux has a monolithic kernel (a big block with many services inside).
- You might think: "If one thing breaks, will the whole OS crash?"
    - Answer: No.
    - Linux uses kernel hardening → if one kernel thread fails, it restarts just that thread, not the whole OS.

Linux Kernel Architecture



File Management

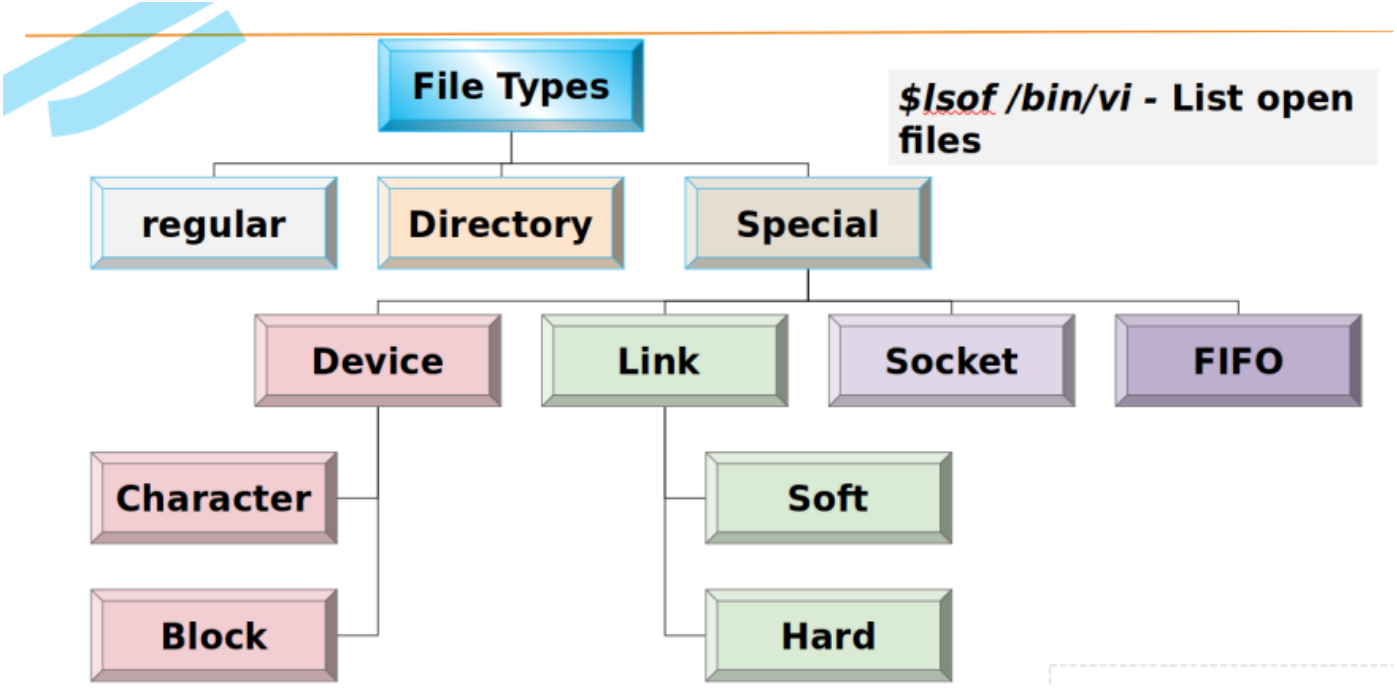


1. Linux File System Structure

Think of Linux files like a tree turned upside down.
At the top is the root /, and everything grows below it.

- /home → where **user files** are stored (your documents, downloads, etc.)
- /bin → **basic commands** (like ls, cp, cat)
- /boot → **boot loader files** (kernel, GRUB stuff)
- /etc → **system configuration files** (like network configs, passwords)
- /root → home directory of the **root (admin) user**
- /proc → **virtual files** that give info about processes and kernel (not real files, just views into memory/CPU status)
- ...etc. → many more, each with its own job.

File Type in Linux



Main file types:

1. **Regular file** → normal text, code, images, etc.
2. **Directory** → folders (just special files that list other files).
3. **Special files** → used for devices and communication. Examples:
    - Character device (c) → sends/receives data one character at a time (e.g., keyboard).
    - Block device (b) → transfers blocks of data (e.g., hard disk).
    - Socket (s) → file for network communication.
    - FIFO (p) → also called *named pipe*, used for one-way communication between processes.

---

3. Device Special Files

- They don't store data themselves.
- They just act as a bridge between the OS and the device.
- They have two numbers:
    - Major → tells which driver handles it.
    - Minor → tells which device it is (e.g., which disk if you have many).

---

4. Links (Shortcuts to Files)

Linux has two types of links:

Soft Link (Symbolic Link) → like a shortcut in Windows

- Just a reference to the file.

- If the original file is deleted, the soft link breaks (becomes useless → "dangling pointer").
- Different inode number than the original file.

Command:

ln -s source target

Hard Link → real extra name for the same file

- Points directly to the file in memory (inode).
- Both the file and hard link share the same inode number.
- Deleting one doesn't delete the data until all hard links are gone.

Command:

ln source target

---

5. Inodes

- Every file in Linux has a unique inode number (like an ID card).
- It stores metadata: file size, permissions, owner, etc. (but not the filename).
- A filename is just a pointer to an inode.

---

6. Quick Way to Identify File Types

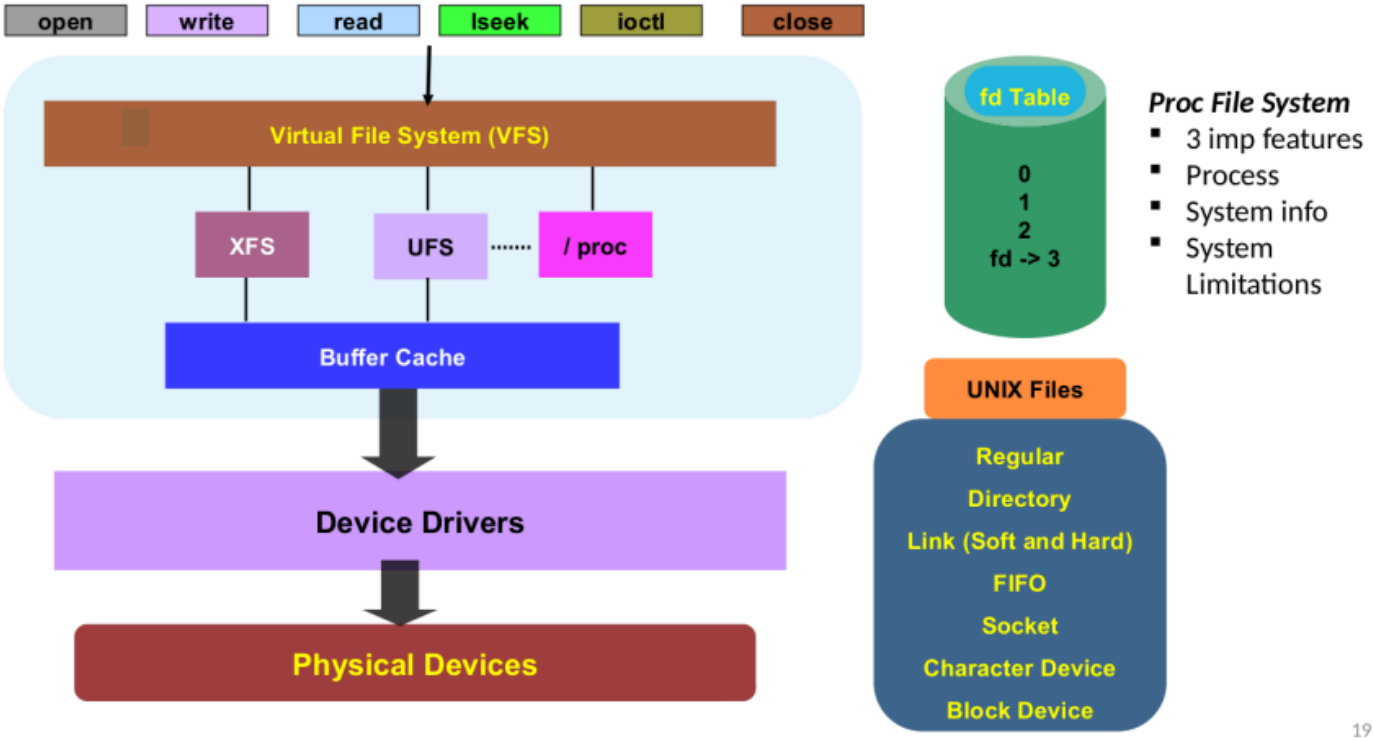
Run:

ls -l

Look at the first character of each line:

- - → Regular file
- d → Directory
- c → Character device
- b → Block device
- l → Soft link
- s → Socket
- p → FIFO (pipe)

File System Architecture



1. Virtual File System (VFS)

- Linux supports many file systems (ext4, XFS, FAT, NTFS, proc, etc.).
- Instead of writing separate code for each one, Linux uses a Virtual File System (VFS).

👉 Think of VFS as a translator between programs and the actual file system.

- Your program doesn't care if the file is on ext4, NTFS, or FAT.
- You just call open(), read(), write(), etc., and the VFS passes it to the right underlying file system driver.

---

2. Why VFS is Useful

- One interface for all file systems → same system calls (open, read, write, close) work everywhere.
- You can mount many partitions (ext4, FAT, NTFS) but still see them as one logical tree starting at /.

👉 Example:
Your / (root) might be ext4, /media/usb might be FAT32, but to you it looks like one system.

---

3. File System Architecture (Simplified Flow)

1. User Program → calls open(), read(), write().
2. VFS → catches the request and decides which real file system (ext4, FAT, proc, etc.) should handle it.
3. Buffer Cache → stores frequently accessed data in RAM for faster performance.
4. Device Drivers → talk to the actual hardware (disk, SSD, etc.).
5. Physical Device → the real storage.

---

4. File Descriptor Table (FD Table)

- When you open a file in Linux, it's given a number called a File Descriptor (FD).
- Each process has its own FD table.
- Standard FDs:
  - 0 → Standard input (keyboard)
  - 1 → Standard output (screen)
  - 2 → Standard error (error messages)

👉 **Example:**

- If you open a new file, it will get FD = 3.
- Next new file → FD = 4, and so on.

---

5. Proc File System

Special filesystem /proc → doesn't exist on disk, it's generated by the kernel.

- It provides system info (CPU, memory, running processes, etc.).
- Example: cat /proc/cpuinfo shows CPU details.

Ext File system:

1. File system basics

- **File data is stored in blocks.**
- Here, **block size = 4 KiB** (1 block = 4096 bytes).
- Even if a file is smaller (say 100 bytes), it will still take **1 full block = 4 KiB** on disk.

🔹 **2. Addressing in an inode (index node)**

Each file has an inode that stores pointers to blocks.

- **Direct pointers:** Point directly to data blocks. (Here, **12 direct pointers**)
- **Indirect pointers:** If file is bigger, inode uses *extra blocks* to store addresses.
  - **Single Indirect** → points to a block that contains addresses of data blocks.
  - **Double Indirect** → points to a block, which points to another block, which finally points to data blocks.
  - **Triple Indirect** → 3 levels of pointing.

🔹 **3. How many blocks can 1 indirect block address?**

- Block size = 4 KiB = 4096 bytes
- Each address = 4 bytes
- Number of addresses in 1 block = $4096 \div 4 = 1024$

👉 So **1 indirect block can point to 1024 data blocks**.
Since each data block = 4 KiB,
Single Indirect = 1024 × 4 KiB = 4 MiB of file data.

🔹 **4. Capacity at each level**

1. **Direct blocks (12 pointers):**

$$12 \times 4\,\text{KiB} = 48\,\text{KiB}$$

2. **Single Indirect:**

$$1024 \times 4\,\text{KiB} = 4\,\text{MiB}$$

3. **Double Indirect:**
   - One block can hold 1024 addresses.
   - Each of those 1024 addresses points to a single-indirect block (which can hold 1024 data blocks).

$$1024 \times 1024 \times 4\,\text{KiB} = 4\,\text{GiB}$$

4. **Triple Indirect:**

$$1024 \times 1024 \times 1024 \times 4\,\text{KiB} = 4\,\text{TiB}$$

🔹 **5. Example exam-type questions**

✅ **Q1: What's the maximum file size with only direct + single indirect?**

- Direct = 48 KiB
- Single Indirect = 4 MiB
- **Total ≈ 4 MiB + 48 KiB**

✅ **Q2: How many blocks can double indirect address?**

$$1024 \times 1024 = 1{,}048{,}576\ \text{blocks}$$

✅ **Q3: If one file = 1 block, how many files can be stored in a disk of size 1 GiB?**

- $1\,\text{GiB} = 1024 \times 1024\,\text{KiB} = 262{,}144\,\text{blocks}$
- So **262,144 files**

✅ **Q4: How many inodes are needed?**

- Same as number of files (since **1 inode = 1 file**).

👉 So the big picture:

- Direct = small files
- Indirect = medium files
- Double/Triple Indirect = huge files (up to TiB range)

Copy-On-Write Technique (COW)

What is COW?

- **Normal way:** When you copy a file (or resource), the system creates a full duplicate immediately. This takes **time + space**.
- **COW way:** When you "copy" a file, the system **does not actually copy it right away**. Instead:
  - Both processes/users just **share the same file (same memory/data block)**.
  - The copy is only created **if someone tries to modify it**.

👉 Until modification, both copies just point to the same data = **saving memory + improving speed**.

🔹 **Example**

1. Process A creates a file notes.txt.
2. Process B "copies" it (with COW).

- **No new file data written.** Both A and B just point to the same file blocks.

3. If **Process B only reads**, still no copy made.
4. If **Process B writes something**, then **a separate copy is made only for B**, so A's file is unaffected.

🔹 **Where is it used?**

- **Operating Systems:**
  - Forking a process (fork() in Linux) → Instead of copying all memory pages immediately, COW is used. Pages are copied only when the child or parent modifies them.
- **File Systems:**
  - Modern file systems (e.g., ZFS, Btrfs) use COW to manage snapshots and reduce disk writes.
- **Virtualization & Docker:**
  - Containers and VMs use COW to share base images, creating new copies only when changes happen.

🔹 **Key Advantages**

1. Saves **memory and disk space**.
2. **Faster** than eager copying.
3. Efficient for **read-heavy workloads**.

XFS

When you create a filesystem

Linux divides the disk into **blocks**. The main parts are:

1. **Boot Block** → info to boot the system (bootloader, kernel location).
2. **Super Block** → summary of the filesystem (like a control center).
3. **Inode Table** → keeps records (metadata) of files.
4. **Data Blocks** → actual file contents.

🔹 **Superblock**

The **superblock** stores important info such as:

- A **bitmap** (which blocks are free or used).
- The **size of each block** (e.g., 4 KiB).
- How many **inodes** exist.
- **Timestamps** (last checked, last modified).

👉 Every device keeps multiple copies of the superblock (backup).
If the main one is corrupted → backups can be used.

🔹 **Inode Table**

- An **inode** describes a file (not the name, but details about the file).
- Each inode contains:
  - File owner (UID, GID)
  - Permissions (read/write/execute)
  - Timestamps (created, modified, accessed)
  - File size
  - Number of hard links
  - Pointers to where file data is stored (direct/indirect blocks)

👉 **Each inode = one file.**
👉 The total number of inodes = max number of files you can store.

🔹 **Data Blocks**

- These are where the **actual file content** is stored (text, images, programs, etc.).
- Each file's inode points to its data blocks.

🔹 **File Creation Process**

- When you create a file:
  - A **free inode** is assigned (for metadata).
  - A **data block** is allocated (to store content).
  - The inode points to that data block.
- Filenames themselves are stored in directories, which just map a **name → inode number**.

🔹 **Summary (for exams)**

- **Boot block** → Boot info.
- **Superblock** → Filesystem control info (block size, free/used blocks, inodes).
- **Inode table** → Metadata about each file (owner, permissions, size, pointers).
- **Data blocks** → Actual content of files.

Device Special Files

What are Device Special Files?

- In Linux/UNIX, **devices are treated like files**.
- These special files live in /dev directory (e.g., /dev/sda, /dev/tty, /dev/null).
- They are not normal data files → they represent **hardware devices** or **communication channels**.

🔹 How are they accessed internally?

When you open any file (normal file or device special file), the OS goes through several tables:

1. **File Descriptor (FD):**
    - Small integer returned by open() (e.g., 0, 1, 2 for stdin, stdout, stderr).
    - Points to the **File Descriptor Table** (per process).
2. **File Descriptor Table:**
    - Contains pointers to the **File Table** (system-wide).
3. **File Table:**
    - Holds info like current file offset, open mode (read/write).
    - Points to the **inode table**.
4. **Inode Table:**
    - Stores metadata (permissions, owner, type).
    - For **device special files**, the inode contains a **major number** and **minor number**.
5. **Switch Table (Device Switch Table):**
    - **Only for device special files.**
    - Maps the **major number** to the correct **device driver** function.
    - Example: major=8 → hard disk driver; major=4 → terminal driver.

🔹 Major vs Minor Number

- **Major number:** Identifies **which driver** to use (e.g., disk driver, printer driver).
- **Minor number:** Identifies a **specific device/instance** managed by that driver.
    - Example:
        - /dev/sda1 (first partition of first hard disk → minor 1)
        - /dev/sda2 (second partition → minor 2)
    - Max minor number = **255** → so one driver can handle up to **256 devices**.

- **Types of Device Special Files**

1. **Character device:** Transfers data one character at a time (e.g., keyboard, mouse, serial port).
2. **Block device:** Transfers data in blocks (e.g., hard disks, USB drives).
3. **FIFO (Named Pipe):** Special file for inter-process communication.
4. **Socket:** Special file for network communication.

Mount Command

- **$mount command**

- **Files live inside file systems** → Every file you use is stored in a file system.
- **File systems live on devices** → Like hard disks, SSDs, partitions.
- **To use a file system, it must be "mounted"** → Mounting means connecting that file system to a directory (mount point) so you can access it.

👉 Example:
If you plug in a USB drive, Linux mounts it to /media/usb. Now you can open files.

Important points:

1. **/etc/fstab file**
    - This file tells Linux which partitions to mount automatically at startup.
    - Example: Root /, Home /home, Swap, etc.
2. **Mounting process**
    - At boot time, Linux reads /etc/fstab and mounts all the file systems.
    - You can also mount manually using the mount command.
3. **Mount point**
    - A directory where the file system gets attached.
    - Example: /, /home, /mnt/data, /media/usb.
4. **Unmounting**
    - You can detach a file system using unmount command.
5. **Check mounted systems**
    - Run mount → shows all file systems currently mounted.
    - Example from image:
    - /dev/sda0 on / type ext4 (rw)
    - none on /proc type proc (rw)

procfs

- **What is procfs?**
  - It is a **virtual file system** (not stored on disk).
  - It shows live information about the system and processes.
  - Mounted at /proc.
- **Why special?**
  - It doesn't exist on disk → created in memory when the system runs.
  - Shows details like CPU info, memory info, running processes, etc.
- **Process info**
  - Every running process gets a folder in /proc named by its PID (process ID).
  - Inside that folder, you can see details like status, memory used, etc.

👉 Examples:

- cat /proc/cpuinfo → Shows CPU details
- cat /proc/meminfo → Shows memory details
- cat /proc/1234/status → Shows info about process with PID 1234
- cat /proc/interrupts → Shows system interrupts

✅ **Simple analogy**

- **mount command**: Like plugging in a USB drive → you must connect (mount) it to use files.
- **procfs**: Like a live dashboard of your system → shows CPU, memory, and process info, but it's not stored permanently, it's created only while system is running.

Watch Command

🔹 **watch command**

- The watch command **repeats a command automatically** at regular intervals and shows the output updating live.
- Useful for **monitoring changes**.

👉 Example:

watch -n 0.1 cat /proc/interrupts

- Runs cat /proc/interrupts every **0.1 seconds**.
- You'll see how often **hardware interrupts** are happening (keyboard, mouse, network card, etc.).

Other uses:

- watch -n 1 ls -l → See files being created/deleted in a folder every second.

- watch -n 2 ps aux → See running processes update every 2 seconds.

🔹 **Running commands in the background**

- If you add **&** at the end of a command, Linux runs it in the **background**.
- This means you can still use the terminal for other tasks.

👉 Example:

./a.out &

- Runs your program in the background.
- The shell will show you a **PID** (process ID).
- You can check its status using:
- ps <pid>

✅ Useful if your program is long-running (e.g., server, simulation, infinite loop).

---

ipcs command

- ipcs = **Inter-Process Communication Status**.
- Shows which **IPC resources** are being used:
    1. **Shared memory segments** (for sharing data between processes).
    2. **Message queues** (for sending messages between processes).
    3. **Semaphores** (for process synchronization, like traffic signals).

👉 Example:

ipcs

Might show something like:

------ Shared Memory Segments --------

key shmid owner perms bytes nattch status

0x00000000 32768 user 600 524288 2 dest

------ Semaphore Arrays --------

key semid owner perms nsems

------ Message Queues --------

key msqid owner perms used-bytes messages

Buffer Cache

🔹 **What is Buffer Cache?**

- **Buffer cache = temporary storage in RAM** (not on disk).
- It keeps copies of data read from or written to a disk.
- Purpose: **speed up file access** because RAM is much faster than disk.

🔹 **How it works**

1. **When process needs data**
    - If data is already in buffer cache → ✅ take directly from RAM (very fast).
    - If data is not in buffer cache → ❌ system reads it from disk (slow), then stores it in buffer cache for next time.
2. **If buffer cache is full**
    - System picks some old buffer, writes it back to disk if modified, and reuses it.
3. **While reading/writing**
    - Process may wait until data is fetched into buffer cache.
    - Once ready, process continues.

🔹 **Why use Buffer Cache?**

Without buffer cache:

- Every read/write → direct communication with disk.
- Disks are slow → CPU would have to wait for I/O, wasting time.

With buffer cache:

- Data is fetched once from disk, then repeatedly reused from RAM.
- **Less disk traffic → faster performance.**
- CPU doesn't sit idle waiting for disk.

🔹 **Key points**

- Stored in **physical memory (RAM)**.
- **Reduces disk traffic** & speeds up access.
- No fixed size → depends on available free RAM.
- Processes can use sync() to make sure buffer cache contents are written back to disk.

✅ **Real-life Analogy**

Think of **buffer cache** like a **fridge**:

- Instead of going to the market (disk) every time you need milk, you store it in the fridge (buffer cache).
- If you already have milk in the fridge → instant access.
- If fridge is empty → you go to the market once, bring it, and keep it in fridge for later.

I/O Handling

- Input/Output (I/O) in Unix/Linux is handled as a **stream of bytes** (like water flowing through a pipe).
- There are **no special formats** → files, devices, and sockets are all seen as byte streams.
- To access I/O streams, processes use **file descriptors (FDs)**.

🔹 **File Descriptors (FDs)**

- A **file descriptor** is just a small non-negative integer assigned by the kernel to represent an open file (or device, socket, etc.).
- Each process has its **own FD table**.
- Standard descriptors:
  - **0 → stdin** (keyboard input)
  - **1 → stdout** (normal output to screen)
  - **2 → stderr** (error messages)

When a process opens new files, it gets **FD = 3, 4, 5…** and so on.

FD Table

- Each process has its own **separate FD table**.
- FD numbers are **local to the process**.
- Example:
  - Process 1 opens File1 then File2 → File1 = FD 3, File2 = FD 4.
  - Process 2 opens File1 then File3 → File1 = FD 3, File3 = FD 4.

So, the same file can have **different FD numbers in different processes**.

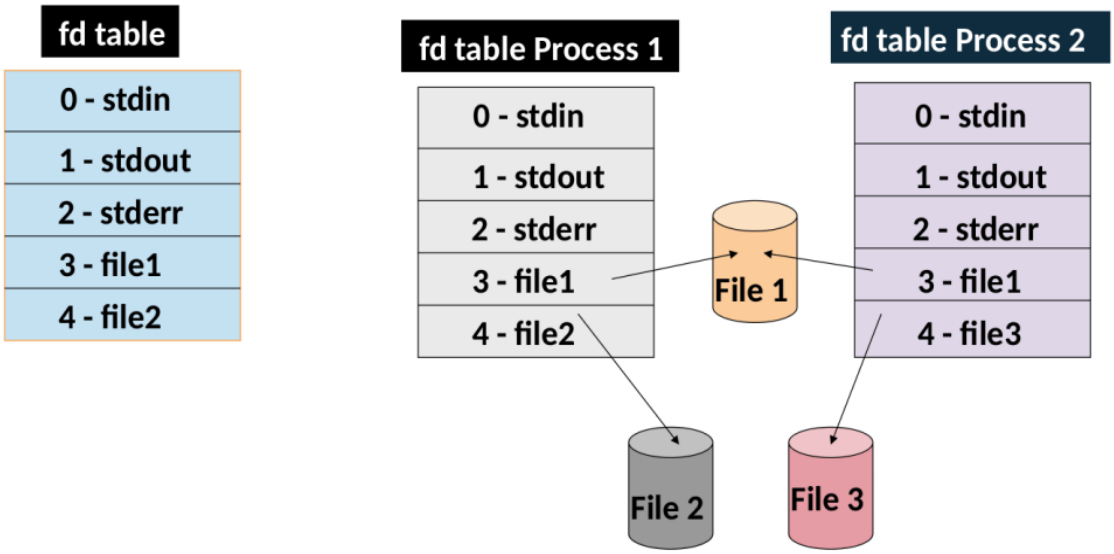
🔹 **Key Points**

1. Kernel always assigns the **lowest available FD number** when a new file is opened.
2. If opening a file fails, FD = **-1** is returned.

3. FD tables are created for each process when it starts.
4. Even if two processes open the same file, their FDs are independent.

✅ **In short**:

- FDs are small integers used by processes to access files/devices.
- Each process has its own FD table.
- 0, 1, 2 are reserved (stdin, stdout, stderr).
- After that, files/devices opened get 3, 4, 5… depending on order.

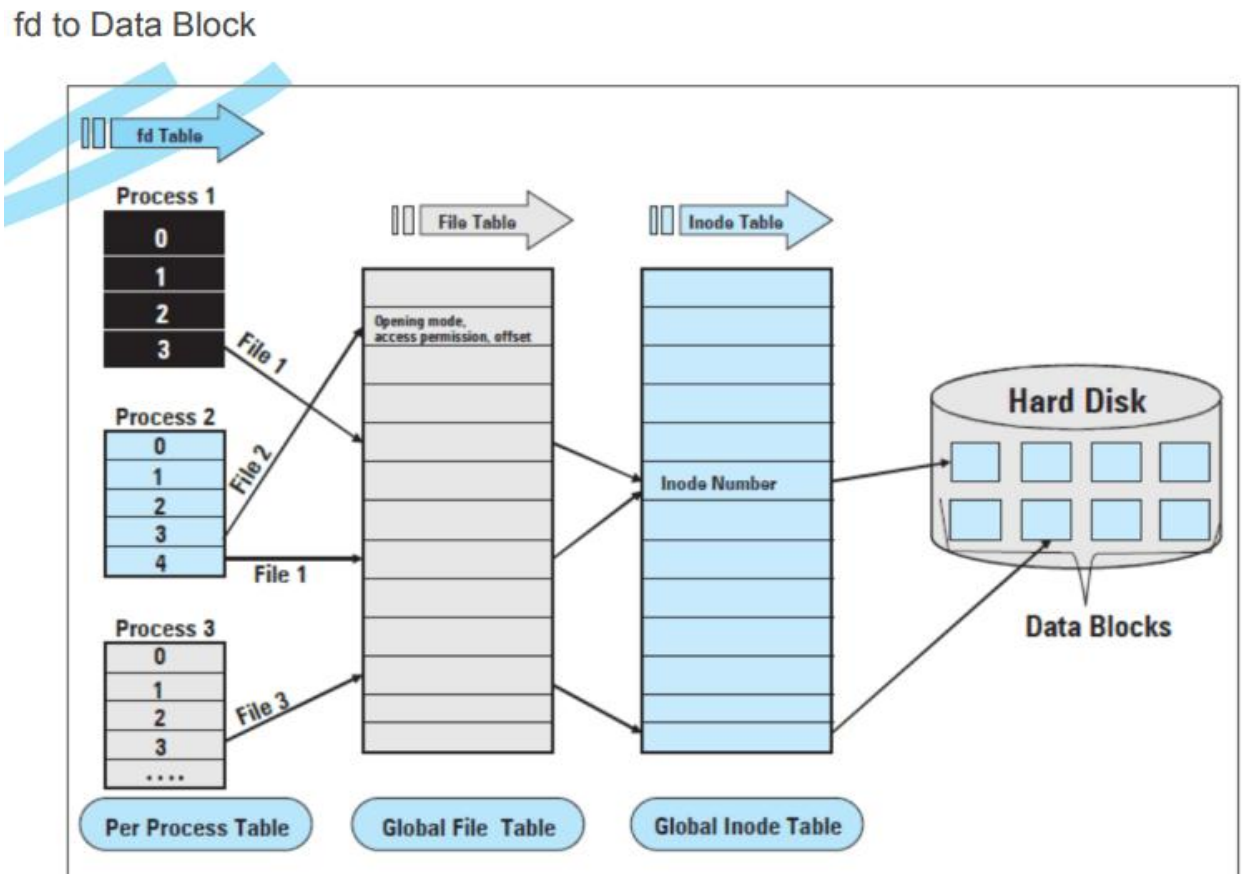


File Table / Open File Table

- This is **different from the fd (file descriptor) table**.
- **Purpose:** Keeps track of all files that have been opened.
- It stores:
    - Metadata about the file (like mode, permissions, offsets, etc.)
    - A **pointer to the inode table entry** (which uniquely identifies the file in the filesystem).

👉 **Key Point:**
Every open() system call creates a **new entry in the Open File Table**, even if it is the same file.

- If the same file is opened 10 times, there will be 10 entries in the Open File Table.
- However, the **inode table** will only have **one entry per file**, because inode represents the file itself (not each opening).



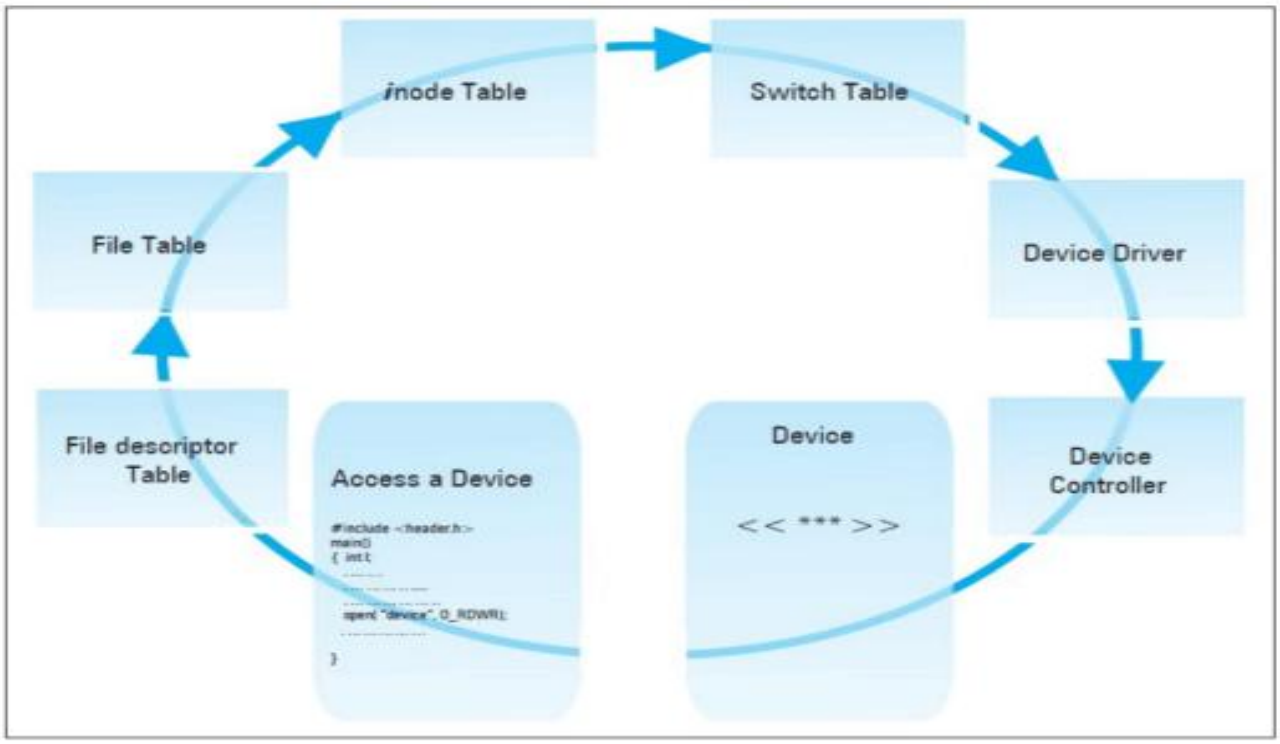
Switch Table

- The **switch table** is used for **device special files** (like /dev/sda, /dev/tty, etc.).
- Normally, inode table entries point to data blocks (for regular files).
- But for **device files**, the inode entry instead points to a **switch table entry**.
- The switch table maps the device to the **correct device driver**, using a **major number** to identify the driver.

👉 **Summary:**

- Regular files → inode → data block
- Device files → inode → switch table → device driver

User Process Accessing Device Special File



System Calls – Notes

- **What is a System Call?**
  - A way for a **program to talk to the OS kernel**.
  - Needed for tasks like opening files, reading, writing, creating processes.
  - Example system calls: open(), read(), write(), close().
  - **Library functions** (e.g. fopen()) are just **wrappers** around system calls.

- **File Handling Internals**

When a file is opened, Linux uses:

1. **File Descriptor Table** – Per process, stores file descriptors (0=stdin, 1=stdout, 2=stderr).
2. **File Table** – Stores mode, offset, pointer to inode.
3. **Inode Table** – Stores file metadata & points to data blocks.
4. **Data Blocks** – Actual content of the file.

- **File Permissions**
  - Users: **Owner, Group, Others**.

- Values:
  - Read = 4
  - Write = 2
  - Execute = 1
- Example:
  - rwx = 7
  - rw- = 6
  - r-- = 4
- **chmod 777 file.txt** → Everyone gets all permissions (⚠️ unsafe).
- **chmod 755 file.txt** → Owner: all permissions, others: read & execute (✅ safe for scripts)
- **chmod 644 file.txt** → Owner: read & write, others: read only (📄 standard for text files)

---

open() System Call

int open(const char *pathname, int flags, mode_t mode);

- **pathname** → file name.
- **flags** → how to open:
  - O_RDONLY = read only
  - O_WRONLY = write only
  - O_RDWR = read + write
  - O_CREAT = create file if not exists
  - O_EXCL = fail if file already exists (use with O_CREAT)
- **mode** → permissions (used only if O_CREAT). Example: 0744.

✅ Example:

int fd = open("file.txt", O_RDWR | O_CREAT, 0744);

---

umask (Default permissions)

- If no mode is given, **default permissions = (0777 – umask value)**.
- Example:
  - If umask = 022, then new file permissions = 755.

creat() System Call

🔹 **creat() Example**

int fd = creat("myfile.txt", 0644);

- Creates a file named **myfile.txt**.
- Permissions 0644 → Owner can read/write, Group & Others can only read.
- Returns a file descriptor (like 3, 4, etc.).
- If file already exists → contents are cleared (truncated).

🔹 **Adding a New User Example**

- Suppose you create a new user **alex**.
- Now alex can log in, but **cannot run admin tasks**.
- Example: If alex tries
- sudo apt-get update

→ It will **fail**, because alex is not a sudoer.

🔹 **Making User a Sudoer Example**

- If you give alex sudo rights, now alex can run admin commands.
- Example:
- sudo apt-get update

→ This time it **works**, because alex is added as a sudoer.

✅ So,

- **creat("file.txt", 0644)** → makes a new file with given permissions.
- **New user (like alex)** → created but no admin power.
- **After giving sudo** → alex can run root-level commands.

read() System Call

🔹 **What it does**

- read() → Reads data from a file into a buffer.
- Syntax:
- **int read(int fd, void *buf, size_t count);**
    - **fd** = file descriptor (from open()).

- **buf** = where the data will be stored.
- **count** = how many bytes to read.

- **Returns**:
  - -1 → if failed
  - n (number of bytes actually read) → if success

🔹 **Example**

char ch;

int fd = open("abc.txt", O_RDONLY);

read(fd, &ch, 1);

- Opens file abc.txt in read-only mode.
- Reads **1 byte** into variable ch.
- If file = "Hello", first call to read() stores 'H' in ch.

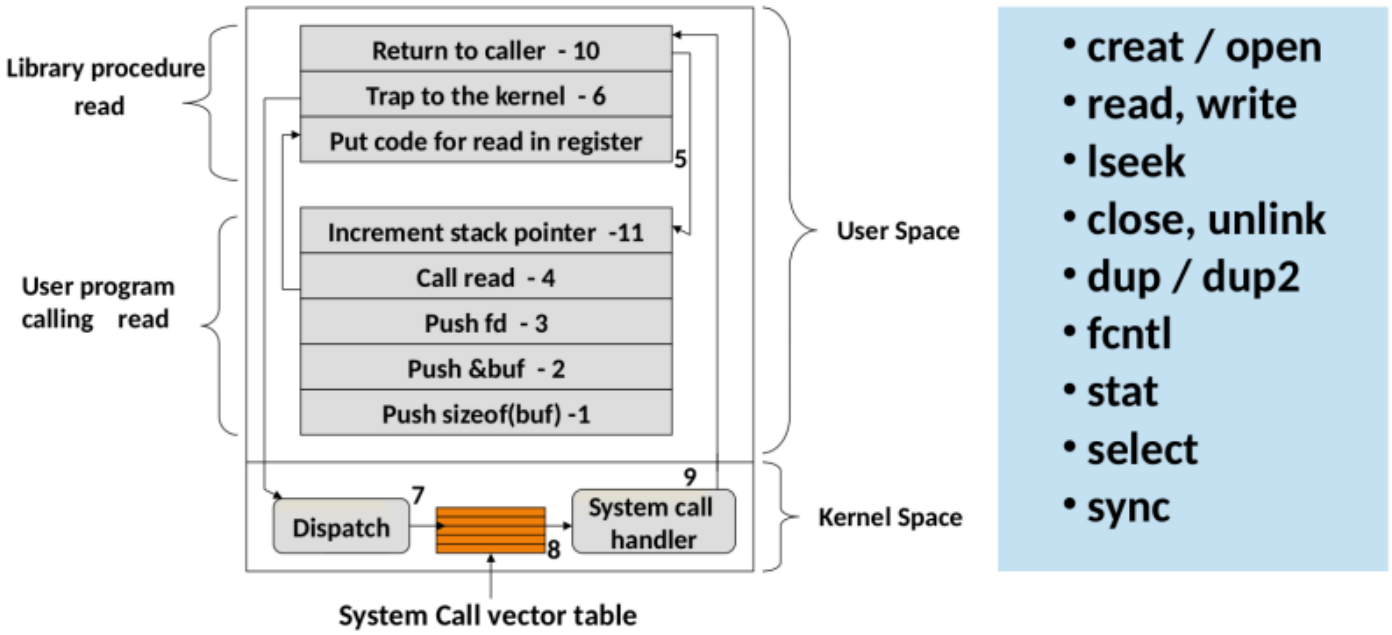
🔹 **Behind the Scenes (Steps)**

1. User program calls read(fd, buf, size).
2. Function pushes arguments (fd, buffer address, size) onto stack.
3. Puts system call number for read into a register.
4. Sends **trap instruction** → switches to **kernel mode**.
5. Kernel looks into **System Call Table** to find the read handler.
6. Kernel executes the read → copies data from file (disk) into buffer.
7. Returns number of bytes read.
8. Control goes back to user program (user space).

🔹 **Key Idea**

- User programs run in **user space** (can't directly touch hardware or disk).
- System calls run in **kernel space** (can access hardware, memory, devices).
- trap = special instruction to switch from user space → kernel space.

✅ In short:
read() is just a **doorway from user space → kernel space** to get data from files.



lseek() System Call

- **What it does**
  - Used for **random access** in a file (not just reading sequentially).
  - Moves the **file pointer** to a specific location.
  - Library version = fseek().

- **Syntax**

off_t lseek(int fd, off_t offset, int whence);

- **fd** → file descriptor
- **offset** → how many bytes to move
- **whence** → starting point for offset:
  - SEEK_SET → from beginning of file
  - SEEK_CUR → from current file position
  - SEEK_END → from end of file
- **Returns**: new file position (in bytes from start), or -1 if error.

- **Example**

```
int fd = open("data.txt", O_RDONLY);
lseek(fd, 5, SEEK_SET);
```

- Moves file pointer to the **6th byte** of file (0-based index).
- Next read() will start reading from that position.

✅ Example use cases:

- Skip some bytes in the file.
- Go back to the beginning.
- Find size of file (by lseek(fd, 0, SEEK_END)).

🔹 **Extra Trick (Input in C)**

Normally:

scanf("%s", str);

If input = Hello World → str = "Hello" (stops at space).

To read full line including spaces:

scanf("%[^\n]", str);

Now str = "Hello World".

✅ In short:

- lseek() = "move file pointer around".
- Example: lseek(fd, 5, SEEK_SET) → jump to 6th byte.
- Useful for skipping, jumping, or finding file size.

close()

int close(int fd);

Task:

- Releases a file descriptor so it can be reused.
- Frees resources if it was the last reference to the file.

✅ Example:

int fd = open("data.txt", O_RDONLY);

close(fd);

👉 Close the file so you don't waste system resources.

---

unlink()

int unlink(const char *pathname);

Task:

- Removes a file name (link) from filesystem.
- If no process has it open and it was the last link → file is deleted permanently.

✅ Example:

unlink("oldfile.txt");

👉 Delete file from disk.

---

dup()

int dup(int oldfd);

Task:

- Creates a **duplicate file descriptor** using the lowest available number.
- Both descriptors share the same file pointer.

✅ Example:

```
int fd1 = open("log.txt", O_WRONLY);
int fd2 = dup(fd1);
write(fd1, "A", 1);
write(fd2, "B", 1);
```

👉 Share the same file using two different fd values.

---

dup2()

int dup2(int oldfd, int newfd);

Task:

- Like dup() but you can choose the new fd number.
- If newfd is already used → it is closed first.

✅ Example:

```
int fd = open("output.txt", O_WRONLY);
dup2(fd, 1);       // redirect stdout (1) to output.txt
printf("Hello");   // goes into output.txt
```

👉 Redirect one fd to another (useful for input/output redirection).

fcntl()

- fcntl() = **File control** system call.
- It is like a **"Swiss Army knife"** for file descriptors (fd).
- You pass it a **file descriptor** (like the number you get when you do open()), and tell it **what operation** to do using cmd.

🔹 Syntax

int fcntl(int fd, int cmd, ... /* arg */);

- **fd** → the file descriptor (returned by open(), socket(), etc.).
- **cmd** → tells fcntl what action to do.
- **arg** → optional extra parameter (depends on the command).

🔹 Common cmd Options

1. **F_DUPFD**
   → Duplicates the file descriptor.
   Similar to dup().
   Example:
2. int newfd = fcntl(oldfd, F_DUPFD, 0);

Now newfd refers to the same file as oldfd.

---

2. **F_GETFD / F_SETFD**
   → Get or set *file descriptor flags*.
   - F_GETFD → check if FD_CLOEXEC is set (this closes fd when exec() is called).
   - F_SETFD → set/clear FD_CLOEXEC.

---

3. **F_GETFL / F_SETFL**
   → Get or set *file status flags*.
   - These include:
     - **Read/Write mode** (read-only, write-only, read-write)
     - **Append mode**
     - **Non-blocking mode**

Example (set non-blocking mode):

```
int flags = fcntl(fd, F_GETFL, 0);    // get current flags
fcntl(fd, F_SETFL, flags | O_NONBLOCK); // set non-blocking
```

🔹 **Why is fcntl() useful?**

- Lets you **duplicate file descriptors** (like piping output).
- Lets you **control blocking vs non-blocking behavior**.
- Useful in **sockets, pipes, and advanced I/O**.
- Acts as a **general-purpose control tool** for file descriptors.

stat()

- stat() = **Status of a file**.
- It gives you **metadata (information about a file, not the file's contents)**.
- Example: "When was this file last modified? How big is it? Who owns it?"

🔹 **Syntax**

int stat(const char *pathname, struct stat *statbuf);

- pathname → the file name (like "myfile.txt").
- statbuf → pointer to a struct stat, where file info will be stored.

👉 Returns 0 if successful, -1 if there's an error.

🔹 **Important members of struct stat**

The struct stat (in **<sys/stat.h>**) has many fields. The most commonly used:

- **st_ino** → inode number (unique ID of file on disk).
- **st_mode** → file type + permissions (regular file, directory, etc.).
- **st_nlink** → number of hard links.
- **st_uid** → user ID (owner).
- **st_gid** → group ID.
- **st_size** → size of file in bytes.
- **st_atime** → last access time.
- **st_mtime** → last modification time.
- **st_ctime** → last status change time (permissions, owner, etc.).

🔹 **Variants**

- **stat(path, buf)** → by file path.
- **fstat(fd, buf)** → by file descriptor (useful if file is already opened).

- **lstat(path, buf)** → like stat(), but if the file is a **symbolic link**, it gives info about the link itself, not the file it points to.

👉 So in short:

- **stat()** → "Tell me info about this file path."
- **fstat()** → "Tell me info about this file I already opened."
- **lstat()** → "Tell me info about the symlink, not the target file."

perror()

- **What it does:** Prints a **human-readable error message** based on the global variable **errno**.
- errno is automatically set when a system call or library function fails.

Example

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

int main() {
   int fd = open("file.txt", O_RDWR);
   if (fd == -1) {
      perror("Error opening file");
   }
   return 0;
}
```

👉 If file.txt doesn't exist, output will be:

Error opening file: No such file or directory

So, perror() = "Your custom message: explanation from errno".

---

select()

- **What it does:** Lets you **watch multiple file descriptors (fd)** at once.
- It waits until at least one fd is **ready for reading, writing, or has an exception**.
- Useful in **network programming** (handling many sockets in one process without threads).

🔹 **Syntax**

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

- **nfds** → highest fd number + 1 (basically range to check).
- **readfds** → fds you want to watch for **reading**.
- **writefds** → fds you want to watch for **writing**.
- **exceptfds** → fds you want to watch for **exceptions**.
- **timeout** → how long to wait (in seconds + microseconds).
    - If NULL → wait forever until something happens.
    - If {0,0} → return immediately (polling).

🔹 **Return Value**

- > 0 → number of fds that are ready.
- 0 → timeout happened, no fd ready.
- -1 → error.

🔹 **Example (monitor stdin for input)**

```
#include <stdio.h>
#include <sys/select.h>
#include <unistd.h>

int main() {
    fd_set readfds;
    struct timeval timeout;

    FD_ZERO(&readfds);        // clear the set
    FD_SET(STDIN_FILENO, &readfds); // watch stdin (fd = 0)

    timeout.tv_sec = 5;   // wait max 5 sec
    timeout.tv_usec = 0;

    printf("Waiting for input (5 sec)...\n");
```

```
    int ret = select(STDIN_FILENO + 1, &readfds, NULL, NULL, &timeout);

    if (ret == -1) {
        perror("select");
    } else if (ret == 0) {
        printf("Timeout! No input.\n");
    } else {
        if (FD_ISSET(STDIN_FILENO, &readfds)) {
            printf("Data is available on stdin!\n");
        }
    }
    return 0;
}
```

👉 Run this program, if you type something within 5 seconds, it prints "Data is available".
If you don't, it prints "Timeout! No input.".

✅ So in short:

- **perror()** = quick way to print error messages when syscalls fail.
- **select()** = lets you **monitor multiple fds at once** (like stdin, sockets, pipes), and acts only when they're ready.

Non-Blocking Operations

- **Blocking**: The process waits until the operation finishes.
  Example: read() waits until data is available.
- **Non-blocking**: The process **doesn't wait**.
  - read() → returns immediately with whatever data is available (could be 0).
  - write() → returns immediately (often by buffering data in memory).

👉 Useful when you don't want your program to "freeze" while waiting for I/O.

🔹 **select() and fd sets**

select() checks multiple file descriptors (fds) at once to see if they're ready for I/O.

- **readfds** → watch fds for reading.
- **writefds** → watch fds for writing.
  - A writable fd means "a small write won't block".

- ⚠️ But a very large write might still block.

- **exceptfds** → watch fds for exceptional conditions (e.g., socket errors).

🔹 **nfds**

- Must be **highest fd number in all sets + 1**.
- Example: if you're checking fd = 4, 7, 9, then nfds = 10.

🔹 **Helper Macros for fd sets**

Used with fd_set structures.

- FD_ZERO(&set) → Clear the set (all 0).
- FD_SET(fd, &set) → Add an fd to the set (mark as 1).
- FD_CLR(fd, &set) → Remove an fd from the set (mark as 0).
- FD_ISSET(fd, &set) → Check if fd is still set (ready after select()).

🔹 **Mini Example (check stdin readiness)**

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(STDIN_FILENO, &readfds);

struct timeval tv;
tv.tv_sec = 3;   // wait max 3 sec
tv.tv_usec = 0;

int ret = select(STDIN_FILENO+1, &readfds, NULL, NULL, &tv);
if (ret == -1) {
    perror("select");
} else if (ret == 0) {
    printf("No input within 3 seconds\n");
} else {
    if (FD_ISSET(STDIN_FILENO, &readfds))
        printf("You typed something!\n");
}
```

✅ Quick Recap

- Non-blocking I/O = no waiting.
- select() = watch multiple fds → tells which are ready.
- nfds = max fd + 1.
- Use FD_ZERO, FD_SET, FD_CLR, FD_ISSET to manage fd sets.

File Locking

Why do we need File Locking?

- When multiple processes (programs) access the **same file at the same time**, problems can happen.
- Example:
    - Process A is writing "100" into a file.
    - At the same time, Process B writes "200".
    - Final file content may be **corrupted** (e.g., "1200" or half-written values).
- This is called a **Race Condition** (two processes racing to access the file).
- File locking ensures:

1. **No corruption** (only one process modifies at a time).
2. **Synchronization** (everyone sees the correct, updated data).
3. **Avoid deadlocks** (two processes waiting forever).

📝 Types of File Locking

There are **two types**:

1. Mandatory Locking

- Enforced by the **kernel (operating system)**.
- If one process locks a file, no other process can read/write until the lock is released.
- **Strict but slower** because the kernel blocks access.

2. Advisory (Record) Locking

- A **gentleman's agreement**: processes "agree" to use locks but the kernel doesn't enforce it.
- If a process ignores the rule, it can still access the file.
- Advantage: Faster + allows multiple processes to work on **different parts** (records) of the file.

🆚 Mandatory vs Record Locking Example

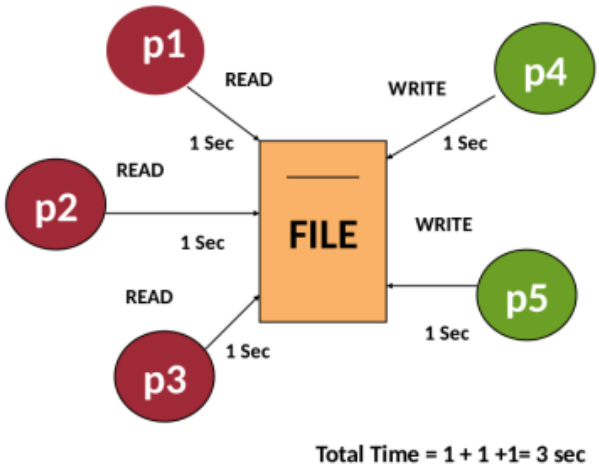
Imagine 3 people (processes) working with a shared notebook (file):

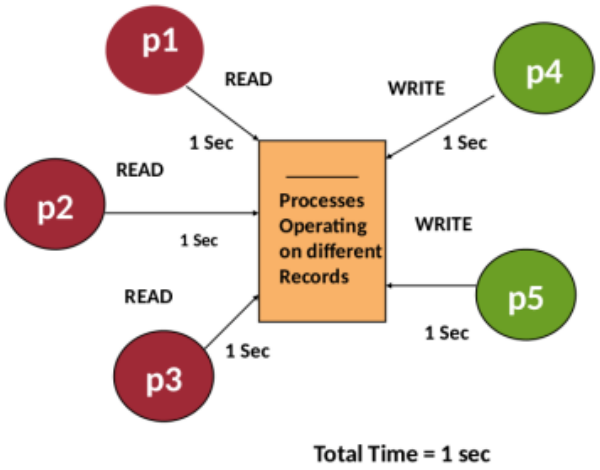
- **Mandatory Locking**:
  - If one person is using the notebook, others must wait.
  - Total waiting time = longer.
- **Record Locking (Advisory)**:
  - Notebook is divided into pages (records).
  - One person can work on page 1, another on page 2 at the same time.
  - Saves time, avoids conflicts.

✅ In short:

File locking is like reserving a seat so no two people try to sit on it at the same time.

- **Mandatory lock** = the guard (kernel) blocks others until you leave the seat.
- **Advisory lock** = people follow the rule to not sit on the same seat, but the guard won't stop someone who ignores it.





File lock structure

The flock structure

```
struct flock {
   short l_type;    // Lock type: read, write, or unlock
   short l_whence;  // Reference point (like lseek): start, current, or end of file
   off_t l_start;   // Offset (from l_whence) where lock begins
   off_t l_len;     // Number of bytes to lock (0 = till end of file)
   pid_t l_pid;     // PID of the process holding the lock
};
```

👉 **Think of it like reserving seats in a theater:**

- l_type → Are you reserving for **reading only** (many readers allowed) or **writing** (exclusive)?
- l_start and l_len → Which row & how many seats you're reserving.
- l_pid → Which person (process) is holding the reservation.

🔑 **Types of Locks**

- **Read lock** → Many processes can read at the same time (shared).
- **Write lock** → Only one process can write, no one else can read or write at that moment (exclusive).
- **Unlock** → Releases the lock.

⚙️ **How locking/unlocking is done → fcntl() system call**

int fcntl(int fd, int cmd, struct flock *lock);

- **F_SETLK** → Try to set/release a lock. If it can't (because someone else already locked), return -1 immediately.
- **F_SETLKW** → Same as above, but **waits** until the lock can be set/released. (W = Wait).
- **F_GETLK** → Check if the file is already locked by someone else.

👉 Typically:

- Use F_SETLKW to **acquire** a lock (so it waits until free).
- Use F_SETLK to **release** (unlock).

🔄 **Automatic Unlocking**

- Each lock also stores the **PID of the process**.
- If a process **crashes or is killed**, the kernel automatically frees all its locks → prevents "file locked forever" problem. ✅

🆚 **flock() vs fcntl()**

- flock() = simpler library function (works on whole files).
- **int flock(int fd, int operation);**
  - LOCK_SH = Shared (read)
  - LOCK_EX = Exclusive (write)
  - LOCK_UN = Unlock
- fcntl() = more powerful, **supports record locking** (specific byte ranges). That's why it's preferred in system programming.

✅ **In short:**

- Use **fcntl()** when you need fine-grained control (lock specific parts of file).
- Use **flock()** when you just need a quick whole-file lock.

Process Management

- **Process = program in execution**
  When a program runs, it becomes a process. It has its own memory, registers, and execution context.
- **Process contains important components**:
  - **Program Counter (PC):** shows the next instruction to execute.
  - **CPU registers:** hold current working values.
  - **Process stack:** stores temporary data like function calls, parameters, local variables.
- **Linux is multiprogramming + multiprocessing**
  Multiple processes can reside in memory (multiprogramming), and multiple CPUs/cores allow true parallel execution (multiprocessing).
- **Kernel responsibilities**:
  - Manage resources (CPU, memory, files, I/O devices).
  - Keep track of all running processes.
  - Schedule processes efficiently to maximize CPU utilization.
- **Reentrant Kernel**:
  Means the kernel code can be safely used by multiple processes at the same time.
  - Example: On a multicore CPU, two processes may make system calls at the same time.
  - The kernel handles them safely because its code is reentrant (doesn't keep private state that could get corrupted).
- **Goal:**
  High **CPU utilization** (CPU should be busy most of the time) and low **CPU waiting time** (processes should not wait too long in the queue).

Context Switch

- **Definition:**
  Changing the CPU's current running process to another one.
- **Why it happens:**
  - A process finishes.
  - A higher-priority process needs CPU.
  - A process waits for I/O.

- **Steps in context switching**:

1. Save the current process's state (all registers, PC, stack pointer, etc.) into its **Process Control Block (PCB)**.
2. Load the state of the next process from its PCB.
3. Resume execution of the new process.

- **PCB (Process Control Block):**
  A big structure (in Linux: task_struct, ~800 lines long) that stores everything about a process:
  - Process ID, priority, state (ready, running, waiting)
  - CPU registers, program counter
  - Memory info (page tables, address space)
  - Open file descriptors, accounting info
- **Virtual Address Space:**
  Each process thinks it has its own private memory, but actually the kernel maps this to real physical memory using **page tables**.
- **Overhead:**
  Context switching takes time (saving/restoring state), but it is necessary for multitasking.

✅ **In short:**

- The kernel manages processes and resources, ensuring fairness and efficiency.
- Context switching is the mechanism that lets multiple processes share the CPU, by quickly saving one process's state and loading another's.

Execution Context

Execution context = the environment in which kernel functions or code execute.
There are **two main types**:

1. Process Context

- Related to a particular **process in execution**.
- When user programs run → they run in **user mode + process context**.
- When a process makes a **system call** or receives a **signal**:
  - Execution switches to **kernel mode** (since kernel functions must handle it).
  - But still remains in **process context**, because the action belongs to that specific process.
- **Access:** Can access both **process space** and **system space**.

---

2. System Context

- Related to the **system as a whole**, not tied to any one process.
- Example: **interrupt handling**, **scheduler adjusting process priorities**, system-wide timers.

- Always runs in **kernel mode**, but not on behalf of any single process.
- **Access:** Only **system space**, cannot access a specific process's address space.

🔹 **Key Points in Slide**

- Every process has a **task_struct** (PCB in Linux).
- New process → kernel creates a new task_struct.
- Terminated process → kernel deletes its task_struct.
- task_struct is a **doubly linked list**, so the kernel can maintain all processes in a list.

🔹 **Examples**

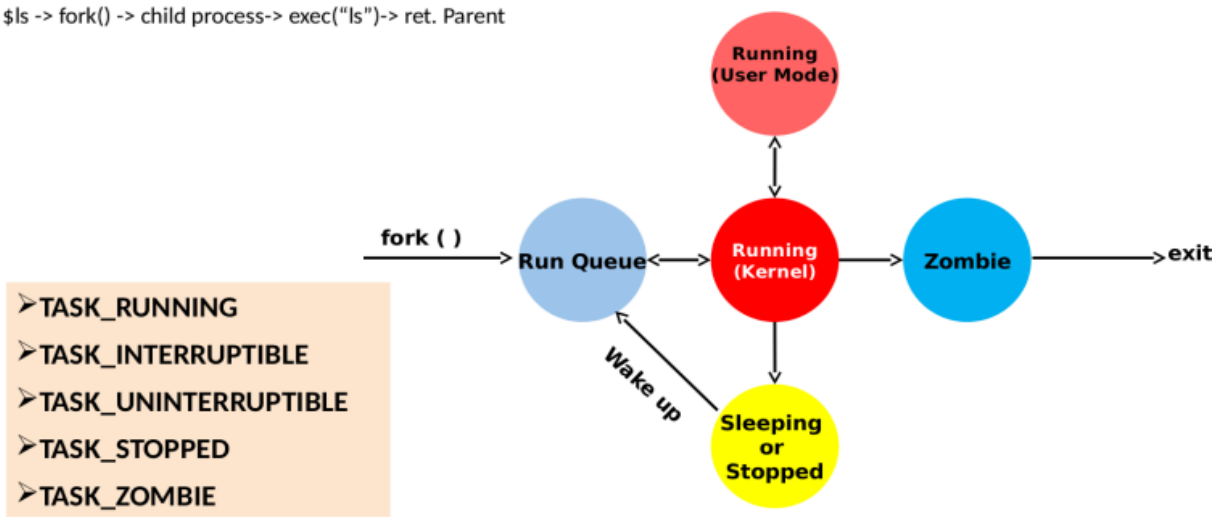
- **Process Context:**
  - A user program calls read() system call → kernel executes it in process context.
  - A signal sent to a process → handled in process context.
- **System Context:**
  - Interrupt occurs (like keyboard press, disk I/O completion).
  - Scheduler adjusts CPU time slices.
  - Load balancing between CPUs.

✅ **In short:**

- **Process Context = Execution tied to a particular process.**
- **System Context = Execution tied to the whole system, not any one process.**

Process States in Linux

A process in Linux can be in one of several states depending on what it is doing and how it interacts with the kernel.



1. Running (R)

- Process is **actively executing** on CPU or is **ready to run** (in the run queue).
- Two sub-modes:
    - **User Mode** → process executes user program instructions.
    - **Kernel Mode** → process executes kernel functions (system calls, handling signals, etc.).

---

2. Sleeping

When process is waiting for an event/resource. Two types:

- **Interruptible Sleep (S)**
    - Process is waiting, but can be interrupted by a **signal** (like kill -9 pid).
    - Example: A program waiting for keyboard input.
- **Uninterruptible Sleep (D)**
    - Process is waiting **without being interruptible** (cannot be killed by signals).
    - Usually in **kernel mode**, waiting for I/O operations (disk read, network).
    - Example: Waiting for disk I/O to complete.
    - ⚠️ You cannot create this state at user level.

---

3. Stopped

Process execution is **paused (suspended)** but not terminated.
It can resume later. Two cases:

- **T (Stopped by Job Control Signal)**
    - Example: Pressing Ctrl + Z sends SIGSTOP.
- **t (Stopped by Debugger)**
    - Example: Process being traced/debugged (with gdb).

⚠️ Note: Stopped ≠ Terminated. A stopped process still exists in the process table.

---

4. Zombie (Z)

- Process has **finished execution** but its **parent has not yet collected its exit status** (via wait() system call).
- Leaves an entry in the process table → shown as **Zombie**.
- Example: Orphan child process before the parent calls wait().

🔹 **Symbol Mapping (as shown by ps or /proc/[pid]/status):**

- **R** → Running
- **S** → Interruptible Sleep
- **D** → Uninterruptible Sleep
- **T/t** → Stopped (by signal / by debugger)
- **Z** → Zombie

🔹 **Example Demo**

1. Run a C program with an infinite loop → process state = R.
2. cat /proc/<pid>/status | head
3. Press Ctrl + Z → process state = T (Stopped).
4. Use sleep(10) in C program → process state = S (Sleeping).
5. You cannot put process into D at user level (kernel only).

🔹 **$ ls Example (from diagram)**

When you type ls:

1. **Shell** calls fork() → creates child process.
2. **Child process** calls exec("/bin/ls").
3. ls runs in child process → executes & prints.
4. **Parent (shell)** waits, then resumes control.

✅ **In short:**

- Linux processes keep changing states: Running → Sleeping/Stopped → Zombie/Exit.
- Symbols (R, S, D, T, Z) help you track their current state via ps, top, or /proc.

fork()

- When you call fork(), the OS:
    1. Creates a **new process** (child) → almost identical to parent.
    2. Copies **program counter, registers, memory space, file descriptors**.
    3. Makes both processes continue **from the next instruction after fork()**.
- **Return values:**
    - **Parent process**: gets **child's PID** (positive number).
    - **Child process**: gets **0**.

- **Failure**: returns -1.

So effectively:

```
pid_t pid = fork();
if (pid == 0) {
  // child
} else if (pid > 0) {
  // parent
} else {
  // error
}
```

🔹 2. Example 1 — Printing fork return value

```
int main() {
   printf("Fork returns: %d\n", fork());
   wait(0);
}
```

Timeline:

Parent (P) calls fork()

|

|--> OS creates Child (C)

|

+--> In Parent: fork() returns child_pid = 9245

+--> In Child: fork() returns 0

Execution:

- Parent executes: printf("Fork returns: 9245")
- Child executes: printf("Fork returns: 0")

So output:

Fork returns: 9245

Fork returns: 0

(The order can swap sometimes, depending on scheduling, but both always appear.)

- **3. How to detect parent/child**

```
if (!fork()) {
   // child → fork() returned 0
} else {
   // parent → fork() returned >0
}
```

Orphan process

Definition:
An orphan process is a **still running process** whose **parent has already terminated**.

```
int main() {
   if (!fork()) {
      printf("In child - Before orphan PPID = %d\n", getppid());
      sleep(1);
      printf("In child - After orphan PPID = %d\n", getppid());
   } else {
      printf("This is parent process %d\n", getpid());
      exit(0);
   }
}
```

Step by step:

1. Parent runs fork() → creates a child.
2. **Child process before sleep:**
    - Runs first printf → shows its current parent's PID (the real parent).
3. **Parent process:**
    - Prints its PID.
    - Calls exit(0) → terminates.
    - At this moment, the **child becomes orphan**.
4. OS immediately reassigns the child to **systemd (PID = 1)**.
5. **Child wakes up after 1 second:**
    - Calls getppid() again → returns 1.

Timeline diagram:

Time →

Parent (P) ---------- exit(0)

Child (C) --- print before orphan (ppid=P)

| sleep(1)

|--- parent already gone → becomes orphan

|--- adopted by systemd (pid=1)

|--- print after orphan (ppid=1)

Sample output:

This is parent process 15324

In child - Before orphan PPID = 15324

In child - After orphan PPID = 1

- **5. Shell prompt "weird" behavior**
  - Normally, shell waits for **parent process** to finish.
  - In our case, **parent exits early** (exit(0)), so the shell thinks program is done and prints prompt.
  - But child is still alive → after waking up from sleep(1), it prints In child - After orphan....
  - That output appears **after shell prompt**, making it look mixed.
  - After child exits, shell is already in foreground → so no new prompt appears until you press **Enter**.

- **6. Key Concepts Recap**
  - **fork()** → creates child, both run same code.
  - **wait()** → makes parent wait for child → prevents zombies.
  - **orphan** → child whose parent died → gets adopted by systemd.
  - **zombie** → child exited but parent didn't wait() → entry still in process table.
  - **Shell behavior** → tied to parent's lifecycle, not child's.

zombie process creation

Definition:
A zombie process is a process that has **finished execution (terminated)** but still has an entry in the process table because its **parent has not yet read its exit status** using wait().

- **The Code**

```
int main() {
   if (!fork()) {
      printf("In child - PID = %d\n", getpid());
      // child ends immediately
   }
   else {
      printf("This is parent process %d\n", getpid());
      sleep(30);     // parent is "ignoring" child for now
      wait(0);       // after sleep, parent reaps child
   }
}
```

- **Step-by-Step Execution**

1. **Parent calls fork()**
    - OS creates a **child process**.
    - In child: fork() returns 0 → enters if block → prints its PID → then exits.
    - In parent: fork() returns child PID → enters else block → prints its own PID → then calls sleep(30).
2. **Child exits immediately**
    - At this moment, parent is **still sleeping**.
    - OS marks child as **"terminated but not reaped"**.
    - Child's entry remains in the process table with state = **Z (Zombie)**.
3. **Zombie phase**
    - For ~30 seconds, child stays in **Zombie state**.
    - If you run:
    - ps -l | grep <child_pid>

you'll see status Z (zombie) Or:

- cat /proc/<child_pid>/status | head

→ will show: State: Z (zombie)

4. **Parent finally calls wait(0)**
    - After 30s sleep, parent executes wait(0).
    - This **reaps** the child: removes its entry from process table.
    - Zombie disappears.

🔹 **Why can't you kill a Zombie?**

- A zombie **is already dead** — it has exited.
- What remains is only a **process table entry**, waiting for parent to acknowledge via wait().
- Sending signals like kill -9 won't work, because there's no actual running process to kill. The only way to clear zombies is:
- Parent calls wait() → normal case.
- If parent never calls wait(), and parent exits → child (still zombie) gets re-parented to systemd, which will reap it.

Quick Questions => How Many Times Hello World Prints in Each Code?

1️⃣ **Example 1:**

```
int main() {
   fork();
   fork();
   fork();
   printf("Hello World\n");
   wait(0);
}
```

- 3 fork() calls → $2^3 = 8$ processes in total.
- Each process executes printf("Hello World\n"); once.
- So 8 "Hello World"s are printed. ✅

=>Why it works?

- Because \n flushes the buffer immediately, so each process prints its own line.

2️⃣ Example 2:

```
int main() {
   printf("Hello World\n");
   fork();
   fork();
   fork();
}
```

- Here the printf happens before any fork.
- At that time, only 1 process (the parent) exists → prints once.
- After that, 3 fork() calls → 7 more children created, but none of them re-run the earlier printf.
- So output = 1 “Hello World” ✅

3️⃣ Example 3 (Interview Question):

```
int main() {
   printf("Hello World");
   fork();
   fork();
   fork();
   wait(0);
}
```

- Looks similar to Example 2, but notice → NO \n in printf.
- Without \n, the text stays in the output buffer (not yet shown on screen).
- Then 3 fork() calls happen → making 7 children.
- Each child is an exact copy of the parent’s memory, including the buffer (which already has "Hello World" inside).
- When each process exits, its buffer is flushed to the screen.
- Total processes = 8 → each flushes once.
  So output = 8 “Hello World”s ✅

🔑 Key Idea:

- fork() duplicates processes.
- printf with \n → prints immediately.

- printf without \n → text stays in buffer → copied to children by fork().
- At exit, each process flushes its own buffer → multiple prints.

👉 Quick Formula:

- With n forks → total processes = 2^n.
- With \n → each process prints individually.
- Without \n → buffer gets copied, so multiple processes may print the same text again.

Process Scheduling

---

1. Process Identifier (PID)

- Every process in Linux has a **unique ID number** (PID).
- This number is **not an index**, it's just a label.
- Each process also has **User ID (UID)** and **Group ID (GID)** → these control **permissions** (what files/devices it can access).

---

2. CPU Time & Clock

- The kernel (OS core) keeps track of:
  - **When a process started**.
  - **How much CPU time it used**.
- This timing uses a special **system clock** (software + hardware).
- It does **not depend on CPU speed**.

---

3. Jiffy (Clock Tick Unit)

- A **clock tick** = smallest unit of time the kernel uses.
- Called a **jiffy**.
- It's different from CPU frequency.
- Example:
  - If **jiffy = 10ms** → 100 ticks per second.
  - If **jiffy = 1ms** → 1000 ticks per second.

👉 System responsiveness depends on this tick value.

- Larger jiffy (10ms) → less accurate timing.
- Smaller jiffy (1ms) → more accurate but more CPU overhead.

---

4. Example of Jiffy in Sleep

- Suppose jiffy = 10ms.
- If you call sleep(11ms), the process will actually sleep **20ms** (rounded up to the nearest tick).
- If jiffy = 1ms, then sleep(11ms) really sleeps 11ms.
  👉 So, **smaller jiffy = better accuracy**.

---

5. What happens at each clock tick?

- The kernel updates:
  - How long the process ran in **system mode** (inside kernel).
  - How long in **user mode** (normal code).
- Linux also allows **interval timers** → processes can request signals after some time.
  - Can be **one-time (single-shot)** or **repeating (periodic)** timers.

---

6. Role of the Scheduler

The **scheduler** decides **which process runs next**.
Its main jobs:

1. Pick the **most deserving process** from the ready queue.
2. Ensure **fairness** (no process should starve).
3. Apply the chosen **scheduling policy**.
4. Update process states every **clock tick (jiffy)**.

---

7. Throughput & Real-Time Systems

- The **goal of scheduling** = maximize **throughput** (number of tasks finished per unit time).
- In **real-time systems**, meeting deadlines is more important than throughput.
  - So scheduling there must be **deterministic** (predictable, strict timing).

---

8. Linux Scheduling

- Linux uses **priority-based scheduling**.
- **Important rule**:
  - **Lower priority number = higher priority**.
  - Example: Priority 0 > Priority 10.

✅ **Quick Recap**:

- **PID** = unique process number.
- **Jiffy** = basic time unit for scheduling.
- **Scheduler** = picks the right process fairly.
- **Linux scheduling** = priority-based (lower value = higher priority).
- **Real-time systems** need strict timing, not just throughput.

Priority Ranges & Scheduling in Linux

---

1. Priority Ranges

- **0 – 99 → Real-Time Processes** (static priority, fixed by user/root).
- **100 – 139 → Normal (Non-Real-Time) Processes** (dynamic priority, kernel adjusts).

👉 These values are mapped to **nice values**:

- **-20 → +19** (where -20 = highest priority, +19 = lowest).

⚠️ Note: Priority +19 for real-time ≠ Priority +19 for general processes.
Real-time processes **always have higher priority** than normal processes.

---

2. Real-Time Processes

- Need **root/sudo** to create.
- They have **fixed priority (static)**.
- Two scheduling policies:
    1. **SCHED_FIFO (First-In-First-Out)**
        - For **short tasks** → avoids unnecessary context switches.
    2. **SCHED_RR (Round Robin)**
        - For **longer tasks** → ensures fairness among real-time processes.
- How to decide which one?
    - Use **heuristics (estimates)** of execution time.
    - This estimation must not depend on current system load.

---

3. General (Normal) Processes

- Scheduling algorithm = **SCHED_OTHER**.
- Under the hood: **Priority-based Round Robin**.
- Kernel automatically adjusts priorities → **dynamic priority**.

4. Dynamic Priority Adjustment (to avoid starvation)

- If a process **keeps running** → its priority is **decreased** (lowered).
- If a process **waits too long** → its priority is **increased**.
  👉 This ensures fairness (no process starves forever).

---

5. Time Slice (Quantum)

- Higher priority = **more CPU time per turn**.
- Lower priority = **less CPU time per turn**.

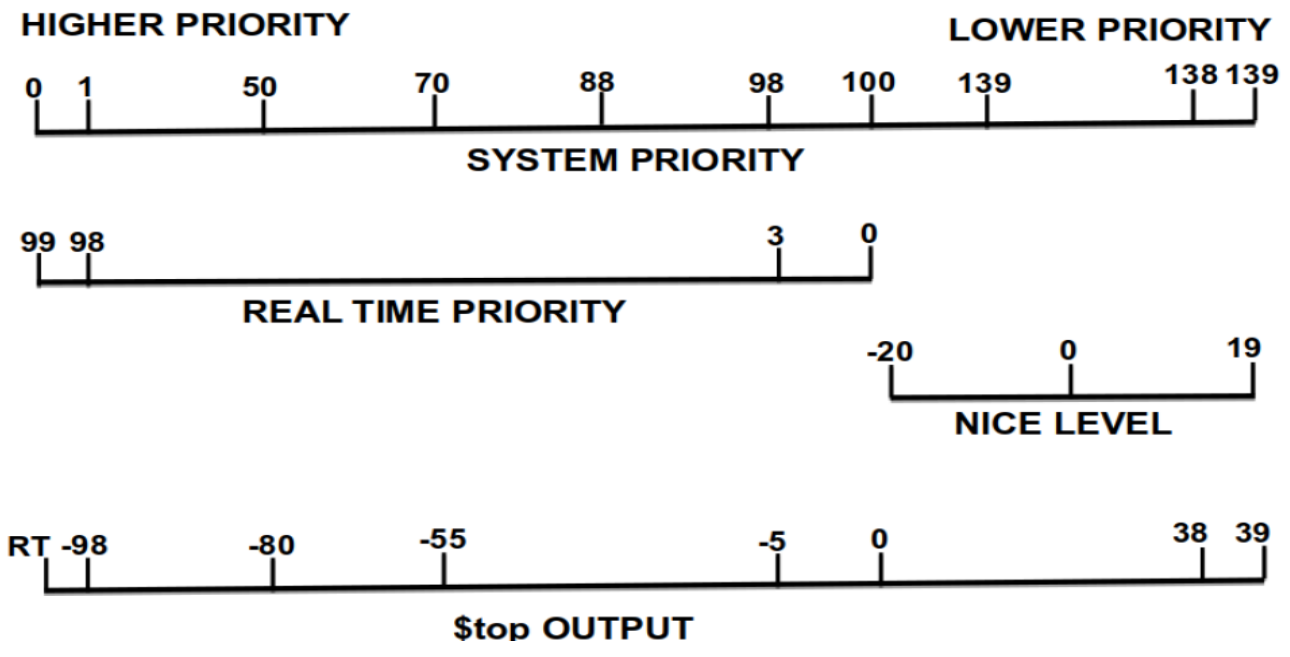
📌 Example values:

- **Priority = -20** → max time slice = **300ms**.
- **Priority = 0** → default time slice = **150ms**.
- **Priority = +19** → min time slice = **10ms**.

✅ **Quick Recap**

- Real-Time: Priority 0–99, root only, FIFO/RR.
- General: Priority 100–139 → mapped to -20..+19 (dynamic).
- Linux adjusts normal priorities to prevent starvation.
- More priority → bigger time slice.
- All real-time processes > any general process.

Understanding Process Priority in Linux



1. Why it's confusing

- Originally, Linux only had **nice values**:
  - Range: **-20 (highest priority) → +19 (lowest priority)**
  - Used for **normal (non-real-time) processes**.
- Later, **real-time process support** was added (with priorities **0–99**).
- To avoid breaking old programs, Linux had to **mix both systems together**.
  👉 That's why the top command shows **weird-looking priority values**.

---

2. What you see in top

When you run top, you'll notice:

- **RT (real-time processes)**
  - Displayed as rt (or sometimes as a large negative number).
  - Example:
    - rt → priority 99
    - -98 → real-time priority 98
    - … and so on.
- **Normal processes**
  - Displayed as positive values.
  - These correspond to **nice values (-20 to +19)**.
  - Next column in top shows the actual **nice value**.

---

3. Priority translation

- **General processes**:
  - Use **nice values** (-20 → 19).
  - Kernel maps them into scheduling priorities internally (100–139).
- **Real-time processes**:
  - Use **static priority (0–99)**.
  - Always higher than normal processes.
  - Can only be created by **root/sudo**.

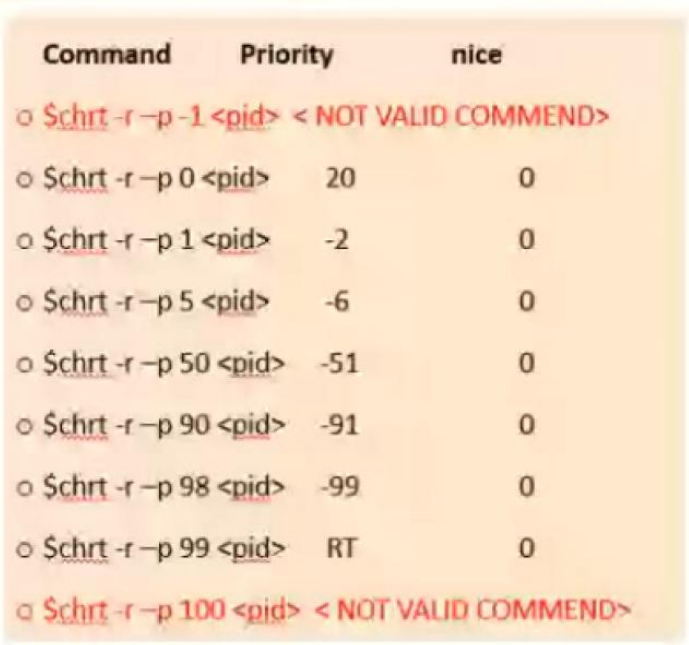
---

4. Key Rule

- **Real-time processes ALWAYS win over normal processes** (no matter the nice value).
- Among real-time:

- Higher priority number (99) = more important.
- Among normal:
  - Lower nice value (-20) = more important.

nice and chrt - Manipulating Scheduling Priorities



- **Normal (non-real-time) processes** use a **nice value** from **-20 … +19**
  - Lower nice (e.g., **-10**) ⇒ **higher** priority
  - Higher nice (e.g., **+10**) ⇒ **lower** priority
- **Real-time processes** use **static priorities 1 … 99** and a **policy** (SCHED_FIFO or SCHED_RR).
  - Larger number ⇒ higher priority (e.g., **99** is highest).
- **Real-time always outranks normal** no matter the nice value.
- **Permissions:** Without sudo, you can only make yourself **nicer** (increase nice value → lower your priority). Setting negative nice or any real-time scheduling generally needs sudo/CAP_SYS_NICE.

---

1) nice — start a command with a given *nice* value

What it does: Launches a new process with modified nice (default start is 0).

Syntax

nice -n <nice_value> command [args...]

-n can be omitted on many systems: nice <nice_value> command

Examples

nice -n 10 ./a.out # start a.out with lower priority (be nicer to others)

sudo nice -n -5 make # start with higher priority (requires sudo)

How to verify

ps -o pid,pri,nice,cmd -p <pid>

top # check PR and NI columns

Tip: nice affects **CPU scheduling** only. For disk I/O priority, use ionice.

---

2) renice — change the nice value of an *already running* process

What it does: Adjusts the nice value for PIDs, process groups, or users.

Linux Syntax (util-linux)

renice <new_nice> -p <pid> ...

renice <new_nice> -g <pgrp> ...

renice <new_nice> -u <user> ...

(The -n flag exists on some systems but on Linux you usually give the absolute value.)

Examples

```
renice 5 -p 1234          # set PID 1234 to nice=+5 (lower priority)
sudo renice -10 -p 1234   # set PID 1234 to nice=-10 (higher priority)
renice 15 -u www-data     # make all www-data processes nicer (lower prio)
```

Notes / Gotchas

- Non-root users can only **increase** nice (e.g., 0 → 5).
- Root can set any value (−20 … +19).
- Changing nice doesn't change real-time policy; it only applies to **SCHED_OTHER/BATCH/IDLE** tasks.

---

3) chrt — view/set real-time policy and priority

What it does: Gets/sets a process's **scheduling policy** and **real-time priority**.

Policies

- **SCHED_FIFO** (First-In-First-Out): Runs until it blocks, yields, or is preempted by a *higher* RT priority. Best for **short, latency-sensitive** tasks.
- **SCHED_RR** (Round Robin): Like FIFO but with a **time slice** among equal-priority RT tasks. Better for **longer** tasks that must share CPU.

Priority range: 1 … 99, where **99** is highest.

Common commands

```
# Inspect an existing process
chrt -p <pid>
```

```
# Run a new command as real-time Round Robin prio 50
sudo chrt -r 50 ./my_rt_task

# Run a new command as real-time FIFO prio 80
sudo chrt -f 80 ./my_short_deadline_task

# Change an existing process (PID 4242) to RR prio 60
sudo chrt -r -p 60 4242

# Change an existing process to FIFO prio 90
sudo chrt -f -p 90 4242

# See min/max supported priorities on this kernel
chrt -m
```

How scheduling behaves

- **FIFO:** No time slice; the task runs until it blocks or a higher-priority RT task appears.
- **RR:** Each equal-priority RT task gets a **fixed time slice** and they rotate.

Safety warnings

- Misusing RT (e.g., FIFO 99 loop) can **starve the system** and make it feel frozen.
- Always test with conservative priorities, ensure the task can block/yield, and keep a root shell handy to undo changes.

🧭 **Reading top/ps columns (quick decode)**

- PR (priority) and NI (nice) show different things:
  - **Normal tasks:** NI = -20…+19; PR is the kernel's computed priority (roughly mapped from NI).
  - **Real-time tasks:** PR often shows rt (or a special value); NI is usually -.
- Remember: **any RT task outranks normal tasks**.

💡 **Which tool when?**

- I'm about to start something and want it to be gentle → **nice**
- Something is already running and I want to adjust its CPU priority → **renice**

- I need strict, deadline-style behavior (audio, robotics, HFT, etc.) → **chrt** with SCHED_FIFO/SCHED_RR (requires sudo)

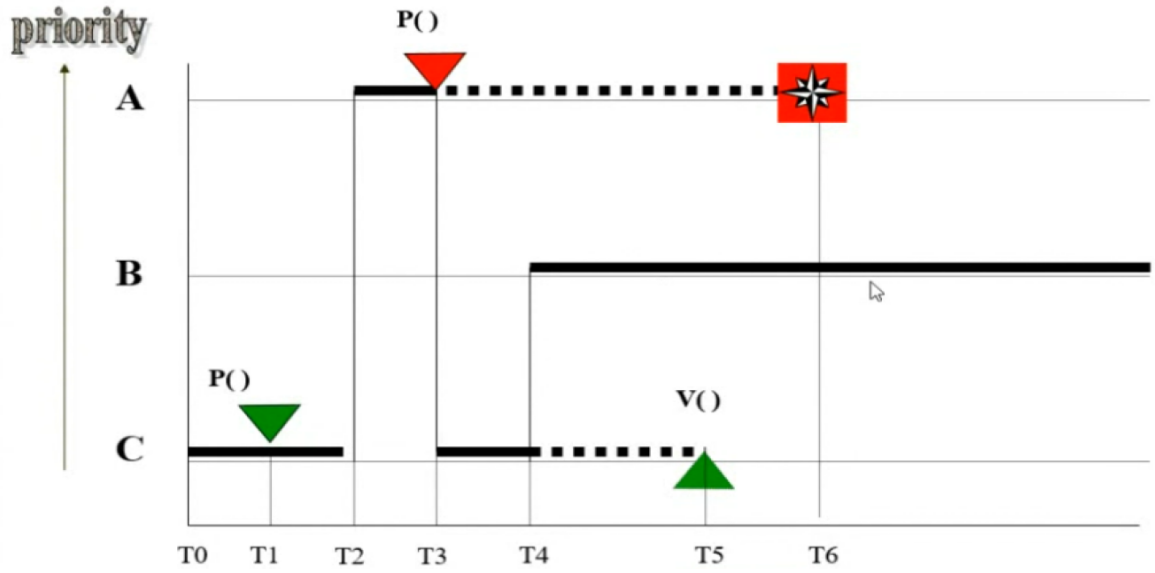
📚 Quick Cheat Sheet

```
# Start low-priority job
nice -n 10 long_job

# Bump a running job higher (root)
sudo renice -5 -p <pid>

# Make a running process real-time RR prio 60 (root)
sudo chrt -r -p 60 <pid>

# Launch a new FIFO real-time task at prio 80 (root)
sudo chrt -f 80 ./cmd
# Inspect
ps -o pid,cls,rtprio,pri,nice,cmd -p <pid>
chrt -p <pid>
```

Unbounded Priority Inversion

This is a problem that happens in preemptive scheduling algorithms and is very important forReal-time scheduling.

Consider the following example,



Process C (low priority) is running in the critical section (CS).

Process A (high priority) requests the CS. Since C is already inside, A is blocked and put to sleep.

Now, Process B (medium priority) arrives. B doesn't need the CS, but since it has higher priority than C, it preempts C and starts running.

Result:

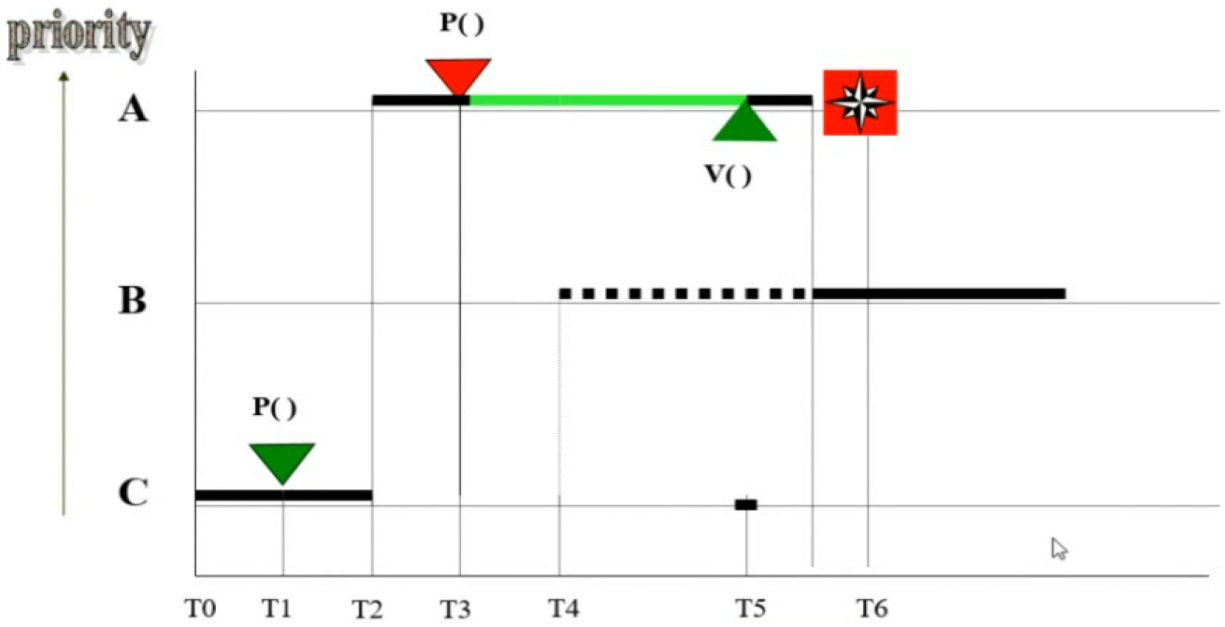
C is paused, so it cannot finish and release the CS.

A is blocked, waiting for the CS.

B keeps running, preventing C from completing.

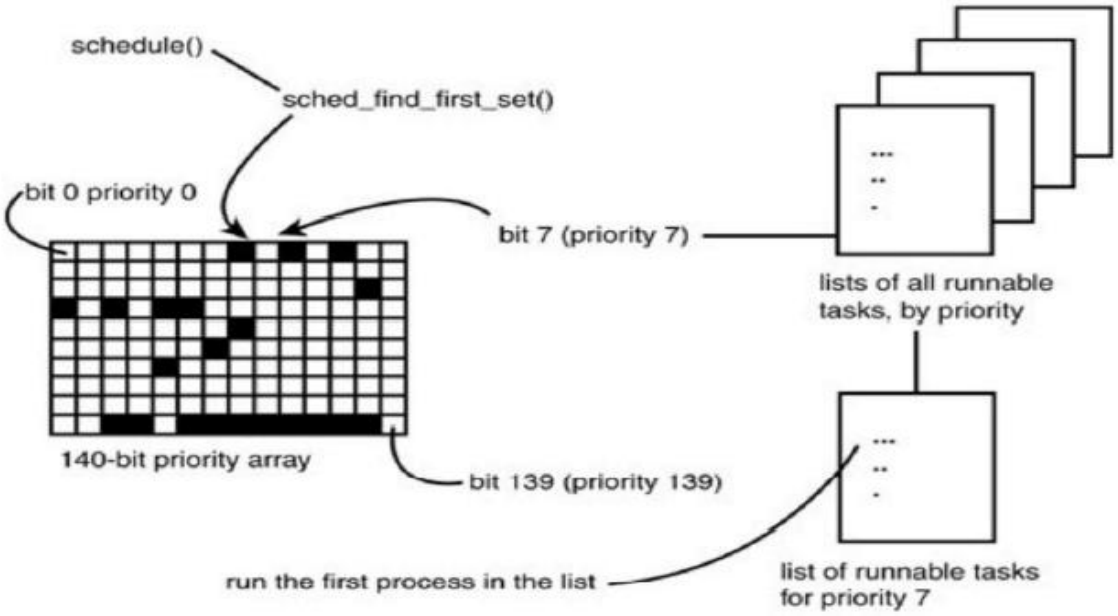
If A had a deadline, it will miss it because it's stuck waiting while a medium process (B) runs unnecessarily in between.

Priority Inheritance -> Solution to Priority Inversion



1. **Process C (low priority)** is executing inside the **critical section (CS)**.
2. **Process A (high priority)** requests the CS but cannot proceed since C is holding it.
   - At this moment, **C's priority is temporarily boosted** to the same as A's priority (so that it can finish quickly).
   - A goes to sleep, waiting for the CS.
3. Now **Process B (medium priority)** arrives.
   - Normally, B would preempt C (since B > C).
   - But because C's priority was boosted to A's level, B **cannot preempt C**.
   - So B goes to sleep.
4. **C finishes its CS work** and releases the lock.
   - Its priority is then reduced back to its original low level.
5. **A immediately acquires the CS** and executes in time to meet its deadline.
6. Once A completes, **B resumes execution**, and finally, C continues its normal execution.

O(1) Scheduler



- **How O(1) Scheduler Works**

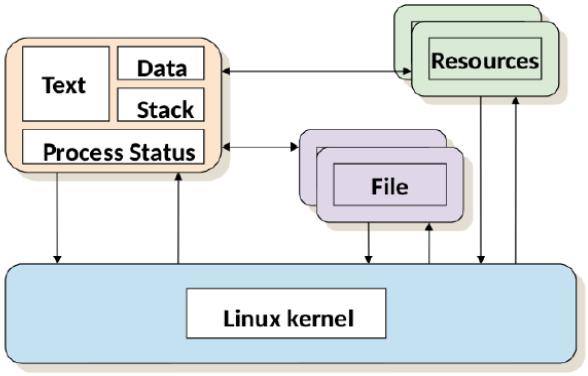
1. **Priority Levels**
   - Linux defines **140 priority levels**:
     - **0–99** → *Real-Time priorities (RT)*
     - **100–139** → *Normal (non-RT) priorities*
   - So, each process has a static priority value in this range.
2. **Priority Bitmap**
   - A **140-bit array** (bitmap) is maintained.
   - Each bit corresponds to a priority level.
   - If bit = 1 → at least one process is waiting at that priority level.
   - If bit = 0 → no process is waiting there.
3. **Wait Queues**
   - Each priority level has its **own queue** of processes waiting for CPU.
   - New processes are added at the tail of their priority's queue.
   - The kernel always takes from the head (first process in line).
4. **Scheduling Decision** (O(1) step)
   - The kernel checks the **highest set bit** in the bitmap (fast operation).
   - That tells it the **highest-priority non-empty queue**.
   - It then **dequeues the first process** from that queue → gives it the CPU.

🔹 **Why O(1)?**

- Finding the highest set bit in a bitmap is a **constant-time operation** (hardware instructions or optimized bit tricks).
- Dequeuing a process from a linked list (per priority queue) is also **O(1)**.
- Therefore, the scheduler makes its choice in **constant time**, regardless of:
    - number of processes in the system, or
    - number of processes in the wait queue.

Process Creation

When you run a command in Linux (say ls or gcc), a lot happens in the background:



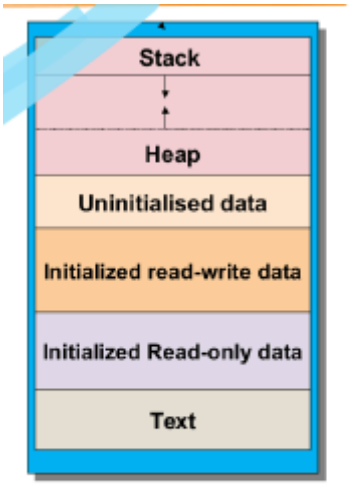
1. **Command execution in shell**
    - You normally type commands inside a shell (like bash, zsh).
    - Example: typing ls.
2. **Finding the command's executable image**
    - The shell looks for the binary file (executable image) of the command.
    - It uses the PATH environment variable (a list of directories).
    - Example: ls is usually found at /bin/ls.
3. **Fork and exec mechanism**
    - The shell does not run the program directly. Instead, it:
        - **forks** → creates a new child process (a copy of the shell process).
        - **executes (exec)** → replaces the child process's memory with the new program's executable image.
    - Now the child process becomes ls, not a copy of the shell anymore.
4. **Completion & Exit status**
    - Once the child process (say ls) finishes, it returns an exit code (success, failure, etc.).
    - This status is passed back to the **parent process** (the shell).
    - Shell then waits for the next command.

👉 So, the key idea:

- **Shell forks → child created → exec replaces child → child runs program → shell gets exit status.**

Process Image

Now, what actually is a **process image**?
It's the layout of a process in memory when it runs. It has different sections:




```
$ size a.out  (man size )
   text    data     bss     dec     hex filename
    920     268      24    1212     4bc
```

Components of Process Image

1. **Text (Code Segment)**
   - Contains machine instructions (compiled code).
   - Read-only → you can't modify program instructions while running.
   - Loaded into memory from the executable file.
2. **Data Segment**
   - Holds program variables.
   - Divided into 3 parts:
     - **Initialized Read-Only Data** → constants, e.g., const int x = 5;
     - **Initialized Read-Write Data** → global/static variables initialized by the program, e.g., int a = 10;
     - **Uninitialized Data (BSS)** → variables declared but not initialized, e.g., int b; (set to 0 at runtime).
3. **Heap**
   - Dynamically allocated memory during execution (malloc, new in C/C++).
   - Grows upwards (towards higher memory).
4. **Stack**
   - Stores function calls, local variables, return addresses.
   - Grows downwards (towards lower memory).
   - Each function call creates a *stack frame*.
5. **Process Status / Context**
   - Maintained by the **kernel**.
   - Includes registers, program counter, and scheduling information.
   - Kernel uses it to pause/resume processes.

---

3. Key Points

- When a process is created using exec(), the **old process image is completely replaced** with the new one.
- That's why the old code does not continue after exec().

- The **heap** and **stack** allow flexible memory use while running.
- Kernel keeps track of the process and manages files/resources linked to it.

✅ **Example in Action**
Say you run:

```
gcc program.c -o program
./program
```

- Shell finds gcc → forks → exec → gcc runs.
- gcc finishes, shell resumes.
- When you run ./program:
    - Shell forks → exec replaces child → memory layout created: **Text + Data + Heap + Stack**.
    - Program executes inside this layout.
    - After completion, exit status is returned to shell.

Copy-on-Write (COW) with fork()

Normally, when you use fork() in Linux, the **child process** is a copy of the **parent process**.
But copying the entire memory immediately is wasteful. So Linux uses an optimization called **Copy-on-Write (COW):**

- **How it works:**
    1. After fork(), parent and child share the same memory pages.
    2. These pages are marked as **read-only**.
    3. If neither modifies them, both keep sharing (saves memory).
    4. If **either parent or child tries to modify**, then the kernel **creates a private copy** of that page for the process making the change.
       → This way, only the modified pages are copied, not the entire memory.
- **Advantage:**
    - Saves memory and improves performance.
    - Kernel only copies what is *actually modified*.

👉 Example:

- You fork() a process running a program with 100 MB memory.
- Without COW → full 100 MB copied.
- With COW → nothing copied initially, only pages modified later are duplicated.

Process Tree Structure

- In Linux, **no process is completely independent** (except the very first one, usually init or systemd).
- Each process is created by another process → forming a **process tree**.

Key Points:

- **Parent process** creates **child processes**.
- Children can also create new processes, forming a **tree structure**.
- You can see this tree using the command:
- pstree

Characteristics:

- **Resource sharing**:
  - Parent & child may share all, some, or none of the resources.
- **Execution**:
  - Parent and child may run **concurrently**.
  - Or parent may **wait** until child finishes.
- **Address space**:
  - By default, child has a **duplicate** of the parent's address space.
  - But after exec(), child's memory is replaced with a new program.

Resource Sharing

When a child process is created via fork(), resource sharing depends on how the program is written:

1. **Parent and Child share all resources**
   - Example: printf() inside both will write to the same terminal.
2. **Child shares only a subset of resources**
   - Example: Using if(fork()) → parent runs one part, child another.
   - Both may share files, but their execution is separate.
3. **Parent and Child share no resources**
   - Example: If the child calls exec() → it loads a completely new program, discarding parent's resources.
   - Now parent and child run totally independent code.

Time Stamp Counter (TSC)

- A **TSC** is a special CPU register that counts the number of cycles (ticks) since the processor was powered on (boot).
- It increases continuously as the CPU runs.
- Since it counts CPU cycles, it can be used for **very high-resolution time measurements** (much more precise than time() function).

👉 Example: If your CPU runs at **1.5 GHz**, then in **1 second**, the counter increases by **1.5 billion ticks**.

---

1. Why use TSC?

- To measure **tiny time intervals** (like nanoseconds).
- More accurate than system calls like time() or gettimeofday() because it's handled directly by the CPU using assembly instructions.

---

2. The rdtsc Instruction

- rdtsc = **Read Time Stamp Counter**.
- It's a CPU instruction that returns the current value of the TSC register.

Since it's not provided by a standard C library, we need to write **inline assembly** to use it:

```
unsigned long long rdtsc() {
   unsigned long long dst;
   __asm__ __volatile__("rdtsc" : "=A" (dst));
   return dst;
}
```

Explanation:

- __asm__ → lets you write assembly inside C code.
- __volatile__ → tells compiler "don't optimize this away" (the value can change anytime).
- "rdtsc" : "=A" (dst) → runs the rdtsc instruction and stores the result into dst.
- Returns the tick count as a 64-bit integer.

---

3. Measuring Time with TSC

Here's the example program:

```
main() {
   long long int start, end;
   start = rdtsc();
   /* Code or job to measure */
   end = rdtsc();
   printf("Difference is : %llu\n", end - start);
}
```

- start = rdtsc(); → reads the counter before job starts.
- end = rdtsc(); → reads the counter after job ends.
- end - start → gives the **number of CPU ticks** spent on the job.

---

4. Converting Ticks into Time

- Since ticks depend on CPU speed, we must know CPU frequency.
- Use command:
- lscpu

→ Shows CPU MHz (say 1552 MHz = 1.552 GHz).

$$\text{Each tick duration} = \frac{1}{\text{CPU Frequency}}$$

$$\text{CPU} = 1.552\,\text{GHz} = 1.552 \times 10^9 \text{ ticks per second}$$

$$1\,\text{tick} = \frac{1}{1.552 \times 10^9} \approx 0.66\,\text{nanoseconds}$$

So to convert ticks → seconds:

$$\text{Time (sec)} = \frac{end - start}{\text{CPU Frequency}}$$

---

5. Why is this useful?

- Benchmarking code performance (how many cycles a function takes).
- Measuring **micro-optimizations** (like difference between two sorting algorithms).
- More accurate than normal timers because it directly uses CPU hardware.

Daemon Process

A **daemon process** is a background process that runs independently of user interaction.
Key points:

1. **Starts during system startup**
    - Usually started by initialization scripts (e.g., /etc/rc).
2. **Executed as background process**
    - Not tied to any terminal.
3. **Orphan process**
    - Its parent exits, so it is adopted by systemd (or init in older systems).
4. **No controlling terminal**
    - So closing the shell does not affect the daemon.
5. **Process leaders**
    - Acts as **session leader** and **process group leader**.
6. **Runs with superuser privileges**
    - For system-level services (e.g., sshd, cron).
7. **Examples**
    - cron (periodic jobs)
    - Print server (cupsd)
    - Web server (httpd, nginx)

Daemon Process Creation

Typical steps in creating a daemon:

```
int init_daemon(void) {
   if (!fork()) {          // Step 1: Fork so parent can exit
      setsid();           // Step 2: Start new session, become session leader
      chdir("/");         // Step 3: Change working dir to root
      umask(0);           // Step 4: Reset file permissions mask
      /* Do your job here */
      return 0;
   } else {
      exit(0);            // Step 5: Parent exits -> child becomes orphan
   }
}
```

Explanation of functions:

- **fork()**
  Creates a child process. The parent exits so the child becomes an orphan.
- **setsid()**
  Creates a new session and makes this process the **session leader** and **process group leader**. This detaches it from any terminal.
- **chdir("/")**
  Changes the working directory to root (/) so the daemon does not keep any directory in use.
- **umask(0)**
  Removes restrictions on file permissions for newly created files.
- **exit(0)**
  Parent process exits, leaving the child running in the background.

🔹 **Process Types**

- **Parent** – spawns child.
- **Child** – created via fork().
- **Orphan** – child whose parent exited (gets re-parented to systemd).
- **Daemon** – special orphan background process with no terminal.

What is a Pipe (|)?

- A **pipe** is a way for two related processes (like parent & child processes) to talk to each other.
- On the command line, you write it as |.
  Example:
- ls -l | wc
  - ls -l lists files.
  - wc (word count) counts lines/words.
  - The output of ls -l is directly sent as input to wc using a pipe.

🔹 **Features of a Pipe**

1. **Two ends (descriptors)** → One end for writing, one end for reading.
2. **Half duplex** → Only **one-way communication** at a time.
   (If process A writes, process B reads; not both at the same time).
3. **Order preserved** → Data comes in the same order as written.
4. **Circular buffer with zero capacity** →
   - Imagine a pipe as a water pipe: as soon as data is written, it must be read.
   - There is no storage inside the pipe (no buffering like in files).

5. **Blocking calls →**
    - If **writer** writes but reader isn't ready, the writer waits.
    - If **reader** tries to read but no data is written, the reader waits.

🔹 **Communication Example**

- Suppose we run:
- ls -l | wc
- Here:
    - The **shell** is the parent process.
    - It creates two child processes:
        - One runs ls -l (writer).
        - One runs wc (reader).
    - The pipe connects them: output of ls -l flows into wc.

Inter Process Communication (IPC)

Processes often need to **share data or resources**. IPC is the general term for these techniques.

1. Primitive IPC

- **Unnamed pipe** (just |, used between related processes).
- **Named pipe (FIFO)**: A pipe that has a name (file-like entry) so **unrelated processes** can communicate.

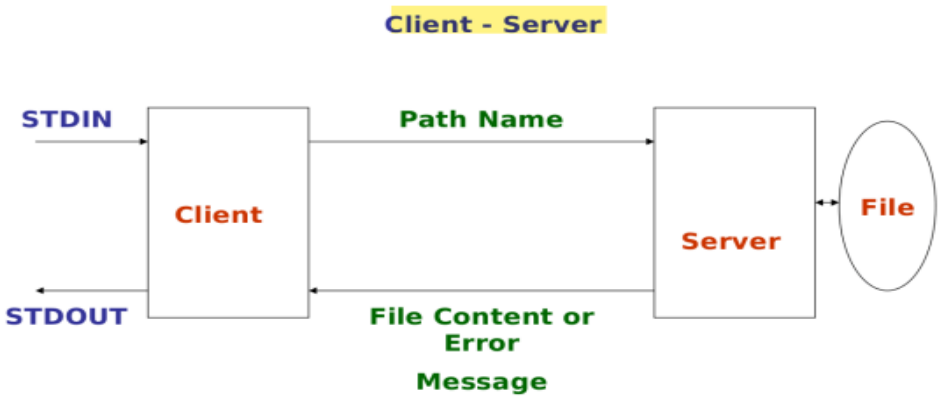
2. System V IPC

- **Message queues** → Send structured messages between processes.
- **Shared memory** → Fastest method, processes share a memory region.
- **Semaphores** → Used for synchronization (to avoid conflicts).

3. Socket Programming

- Works like a **pipe** but more powerful:
    - **Full duplex** → Both processes can read and write at the same time.
    - Used for **client-server communication** (even over networks).

Pipes



🔹 **Analogy to Understand**

- **Pipe (|):** Like a **straw**—water flows one way at a time.
- **Named Pipe (FIFO):** Like a **tube with a label**—anyone can put water in or take it out if they know the label.
- **Message Queue:** Like a **postbox**—you drop letters (messages), others pick them up.
- **Shared Memory:** Like a **whiteboard**—everyone can write and read, but need rules to avoid overwriting.
- **Semaphores:** Like a **traffic signal**—controls access to shared resources.
- **Socket:** Like a **telephone line**—both sides can talk and listen at the same time.

🔹 **Pipes in Client-Server Example**

In the second slide's diagram:

- **Client** sends a request (via STDIN) → passes through a path (pipe).
- **Server** receives it, processes it (maybe reads a file).
- **Server** sends back response (via STDOUT) → client gets file content or error message.

✅ **In short:**

- Pipe (|) connects commands for one-way communication.
- IPC methods (pipes, message queues, shared memory, semaphores, sockets) allow processes to share data/resources.
- Sockets are more advanced (support full duplex, network communication).

pipe() system call

int pipe(int pipefd[2]);

- Creates a **unidirectional channel**.
- Returns **two file descriptors**:
  - pipefd[0] → **read end**
  - pipefd[1] → **write end**
- Returns 0 if success, -1 if error.

🟢 How it works

1. Whatever you **write** to pipefd[1] can be **read** from pipefd[0].
2. Data is stored in the **kernel buffer** until it's read.
3. **Blocking behavior**:
   - If no data → read() waits.
   - If buffer full → write() waits.
4. Pipes are **half duplex** → data flows only one way.

🟢 Simple Example

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int fd[2];
   char buf[80];
   pipe(fd);
   write(fd[1], "Hello\n", 6);      // write end
   read(fd[0], buf, 6);             // read end
   printf("From pipe: %s", buf);
   return 0;
}
```

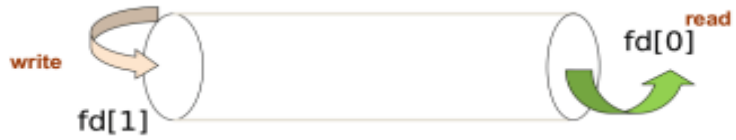
Output:

From pipe: Hello

With fork()

- When you call fork(), both parent and child inherit the pipe.
- Common usage:
  - **Parent writes** to fd[1].
  - **Child reads** from fd[0].
- Used to connect two processes (like ls | wc).

Data Transfer Using Pipe



When we call fork(), both parent and child get **both ends** of the pipe.
But we want:

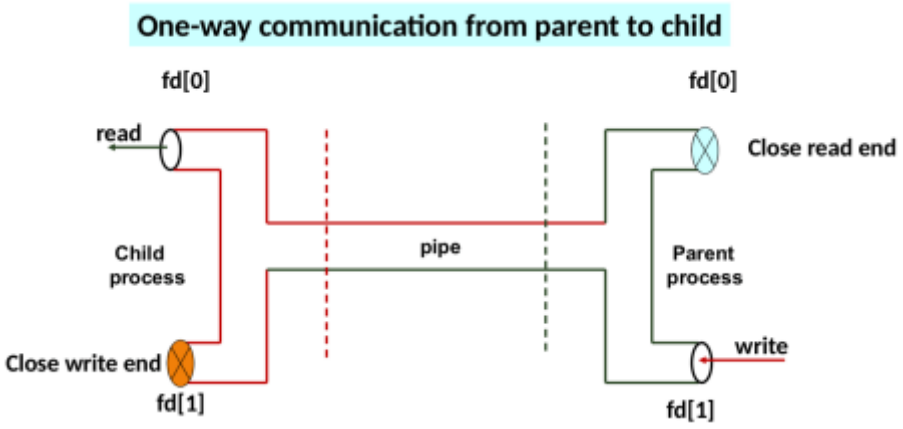
- Parent = **writer** → only needs write-end (fd[1]).
- Child = **reader** → only needs read-end (fd[0]).

So:

- Parent closes fd[0] (read-end).
- Child closes fd[1] (write-end).

If we don't close unused ends:

- The kernel gets confused (who should read?).
- read() may never return EOF because a write-end is still open somewhere.

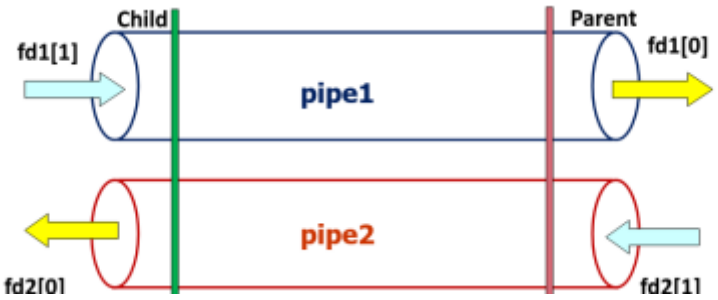


- **Program for One-Way Data Transfer**

```
int main() {
   char buff[80];
   int fd[2];
   pipe(fd);

   if (fork()) {
      // Parent
      close(fd[0]); // close read-end
      printf("Enter message to the child: ");
      scanf(" %[^\n]", buff); // input from user
      write(fd[1], buff, sizeof(buff)); // send to child
   }
   else {
      // Child
      close(fd[1]); // close write-end
      read(fd[0], buff, sizeof(buff));  // wait for parent's data
      printf("Message from parent: %s\n", buff);
   }
   wait(0); // wait for child to finish
}
```

- **Why this works?**
  - **Parent writes** to the pipe, **child reads** from it.
  - read() and write() are **blocking calls**:
    - Child will wait until parent writes.
    - Parent will wait if no reader exists.
  - This makes sure the communication is synchronized.
  - Once parent closes write-end, child's read() gets EOF and stops.

Two Way Data Transfer



- A **pipe** is a communication channel between two processes.
- But a single pipe is **half duplex**, meaning data can only flow in **one direction at a time**.
- If we want the parent and child to **both send and receive messages**, one pipe is not enough.
- That's why we create **two pipes**:
    - **Pipe 1 (fd1):** used by the child to send data to the parent.
    - **Pipe 2 (fd2):** used by the parent to send data back to the child.
- With this setup, communication becomes **two-way (full duplex)**.

🔹 File Descriptors

When we create a pipe, the system gives us **two file descriptors**:

- fd[0] → the **read end** of the pipe.
- fd[1] → the **write end** of the pipe.

Since we are using **two pipes (fd1 and fd2)**, we get four descriptors:

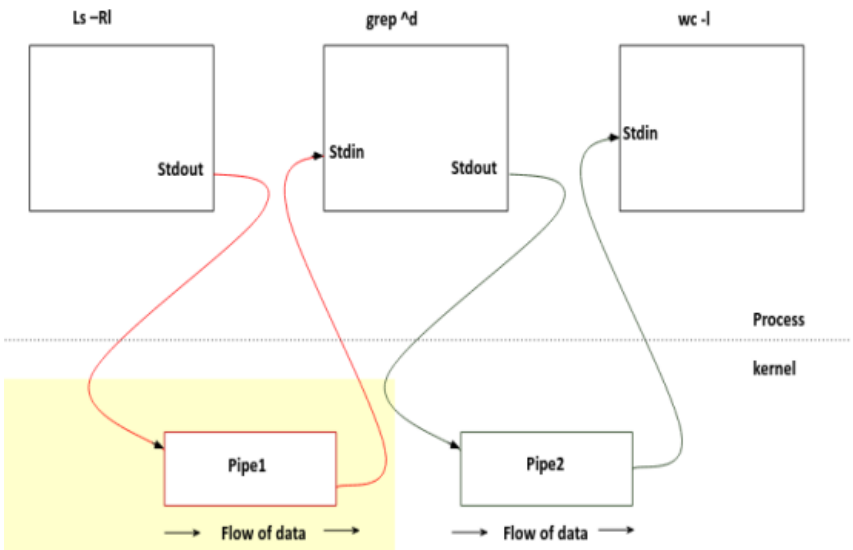
- fd1[0] → read end of pipe1
- fd1[1] → write end of pipe1
- fd2[0] → read end of pipe2
- fd2[1] → write end of pipe2

Each process (parent and child) inherits **all four descriptors** after fork().
But both don't need all of them, so we **close the unused ends** to avoid mistakes.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int fd1[2], fd2[2];
   char buff1[80], buff2[80];
   pipe(fd1);
   pipe(fd2);
```

```
    if (!fork()) {
        // Child
        close(fd1[0]); // close read end of fd1
        close(fd2[1]); // close write end of fd2
        printf("Enter message to parent: ");
        scanf(" %[^\n]", buff1);
        write(fd1[1], buff1, sizeof(buff1));   // child → parent
        read(fd2[0], buff2, sizeof(buff2));    // wait for parent
        printf("Message from parent: %s\n", buff2);
    }
    else {
        // Parent
        close(fd1[1]); // close write end of fd1
        close(fd2[0]); // close read end of fd2
        read(fd1[0], buff1, sizeof(buff1));    // wait for child
        printf("Message from child: %s\n", buff1);
        printf("Enter message to child: ");
        scanf(" %[^\n]", buff2);
        write(fd2[1], buff2, sizeof(buff2));   // parent → child
    }
}
```

Execution of ls -Rl | grep ^d | wc



We need to create **three processes** to execute the three commands.

The first command, **ls -Rl**, normally prints to STDOUT, but we want its output redirected to the **write-end of pipe1**.
To achieve this, we first close the STDOUT file descriptor for that process, then call dup(fd[1]). This duplicates the write-end of pipe1 (fd[1]) onto the lowest available file descriptor, which is 1 (STDOUT).
As a result, the output of ls -Rl will go directly into the write-end of pipe1.

A similar approach is used for the second process: we redirect its STDIN to the read-end of pipe1 by closing file descriptor 0 and duplicating fd[0].

Program to execute: ls -l | wc

```
int main() {
   int fd[2];
   pipe(fd);


   if (!fork()) {
      // Child: run "ls -l"
      close(1);       // Close STDOUT
      close(fd[0]);   // Close unused read end
      dup(fd[1]);     // Redirect STDOUT to pipe write end
      execlp("ls", "ls", "-l", (char*) NULL);
      // Output of "ls -l" now goes to pipe write end
   } else {
      // Parent: run "wc"
      close(0);       // Close STDIN
      close(fd[1]);   // Close unused write end
      dup(fd[0]);     // Redirect STDIN to pipe read end
      execlp("wc", "wc", (char*) NULL);
      // "wc" reads input from the pipe
   }
}
```

Using dup2() instead of dup()

You can simplify the redirection with dup2():

- dup2(fd[0], 0) → replace STDIN with the read-end of the pipe

- dup2(fd[1], 1) → replace STDOUT with the write-end of the pipe

dup2() automatically closes the target descriptor if it's already in use, so there's no need to close STDIN or STDOUT manually.

Using fcntl() with F_DUPFD

Another option is fcntl(fd[1], F_DUPFD, 0).
In this case, you must close the existing file descriptor beforehand.

Program to execute: ls -Rl | grep ^d | wc

```
int main() {
   int fd1[2], fd2[2];
   pipe(fd1);
   pipe(fd2);

   // Child 1: run "ls -Rl"
   if (!fork()) {
      dup2(fd1[1], 1);   // Redirect STDOUT to pipe1 write end
      close(fd1[0]);
      close(fd2[0]);
      close(fd2[1]);
      execlp("ls", "ls", "-Rl", (char*) NULL);
   } else {
      // Child 2: run "grep ^d"
      if (!fork()) {
         dup2(fd1[0], 0);   // STDIN from pipe1 read end
         dup2(fd2[1], 1);   // STDOUT to pipe2 write end
         close(fd1[1]);
         close(fd2[0]);
         execlp("grep", "grep", "^d", (char*) NULL);
      } else {
         // Parent: run "wc"
         dup2(fd2[0], 0);   // STDIN from pipe2 read end
         close(fd2[1]);
```

```
        close(fd1[0]);
        close(fd1[1]);
        execlp("wc", "wc", (char*) NULL);
    }
  }
}
```

Notes

- For quick testing, you can use system("ls -Rl | grep ^d | wc"); or popen().
- However, system() is discouraged and popen() is not ideal for systems programming. They are fine only for testing.

FIFO

- FIFO (named pipe) is like a **normal pipe**, but it has a **name in the filesystem**.
- You create it using:
    - mkfifo myfifo (simpler way)
    - or mknod myfifo p
- Once created, you can treat it like a file: processes can open(), read(), and write() on it.

How FIFO Works

- **Half duplex** → data flows in only one direction (one writes, the other reads).
- **FIFO order** → data is read in the same order it was written (First-In-First-Out).
- **Communicate between unrelated processes** → since it's a named file, any process can open it, not just parent-child.
- **Reusable** → as long as the FIFO file exists, it can be used again and again by new processes.
- **Persists** → the FIFO file stays until deleted with rm.

Difference from Pipe

- **Pipe (|)**: works only between **related processes** (e.g. parent-child created with fork()).
- **FIFO**: works between **any processes**, even if they are unrelated, because they just open the same named file.

How Processes Use FIFO

1. Process 1 opens FIFO in **write-only mode** and writes some data.
2. Process 2 opens FIFO in **read-only mode** and reads that data.
3. System calls like open(), read(), write(), close() are used.

Disadvantages of FIFO

1. **Data cannot be broadcast to multiple receivers** → only one reader consumes it.
2. **No way to target a specific reader** if multiple receivers exist.
3. **Cannot store data** → FIFO only transfers; once read, data is gone.
4. **Not for networks** → works only on the same machine.
5. **Less secure** → any process with file permissions can read/write.
6. **No message boundaries** → it's just a stream of bytes. One read may cut a message, or combine two writes.

FIFO Limitations

1. **Maximum number of files open in a process → OPEN_MAX**
    - Every process can only open a limited number of files at once.
    - This limit is stored in the macro **OPEN_MAX**.
    - You can find it at runtime with sysconf(_SC_OPEN_MAX).
2. **Maximum atomic write size → PIPE_BUF**
    - When multiple processes write to a FIFO/pipe at the same time, their data could get mixed.
    - But writes of size **≤ PIPE_BUF** are guaranteed atomic (safe, no mixing).
    - If you write more than PIPE_BUF bytes at once, the kernel may break it into pieces, and it could mix with other writers.
    - PIPE_BUF value depends on the system (often 4096 bytes).

```
#include <unistd.h>
#include <stdio.h>
int main() {
   long PIPE_BUF, OPEN_MAX;
   PIPE_BUF = pathconf(".", _PC_PIPE_BUF);       // max atomic write size
   OPEN_MAX = sysconf(_SC_OPEN_MAX);             // max open files per process
   printf("Pipe_buf = %ld\t OPEN_MAX = %ld\n", PIPE_BUF, OPEN_MAX);
   return 0;}
```

👉 This prints the system's FIFO/pipe limits.
Example output might be:

Pipe_buf = 4096 OPEN_MAX = 1024

mkfifo() vs mknod()

- mknod() is the **system call** that creates device special files (including FIFOs).
- To create a FIFO with mknod:
- mknod("filename", S_IFIFO | file_permissions, 0);

(Device number = 0 because FIFO is a pseudo device).

- mkfifo() is a **library function** that internally uses mknod() with S_IFIFO.
- mkfifo("filename", 0744);

👉 Which is better?

- mknod() is lower-level and faster (direct system call).
- mkfifo() is simpler and more readable (just wraps mknod).
- If you run strace mkfifo myfifo, you will actually see that it ends up calling mknod() inside.

So, **both create the same FIFO file**, but programmers usually prefer mkfifo() for clarity.

One-way FIFO Communication

Writer Program

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
   int fd;
   char buff[100];

   fd = open("myfifo", O_WRONLY);         // open FIFO for writing
   printf("Enter the text: ");
   scanf(" %[^\n]", buff);                // take input
   write(fd, buff, sizeof(buff));         // write into FIFO
   close(fd);
}
```

Reader Program

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
   int fd;
   char buff[100];

   fd = open("myfifo", O_RDONLY);          // open FIFO for reading
   read(fd, buff, sizeof(buff));           // read from FIFO
   printf("The text from FIFO file: %s\n", buff);
   close(fd);
}
```

👉 Steps to test:

1. First run the **reader** in one terminal. It will block, waiting for data.
2. Then run the **writer** in another terminal, type some text.
3. The reader displays that text.

This shows **blocking behavior**: the reader waits until a writer writes.

Two-way FIFO Communication

- Since FIFO is one-way, we need **two FIFOs** (myfifo1, myfifo2) for bidirectional chat.

Program 1

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
   int fd1, fd2;
   char buff1[100], buff2[100];

   fd1 = open("myfifo1", O_WRONLY);
```

```
    fd2 = open("myfifo2", O_RDONLY);

    printf("Enter the text: ");
    scanf(" %[^\n]", buff1);
    write(fd1, buff1, sizeof(buff1));   // write to fifo1

    read(fd2, buff2, sizeof(buff2));    // read from fifo2
    printf("The text from FIFO file: %s\n", buff2);

    close(fd1);
    close(fd2);
}
```

Program 2

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    int fd1, fd2;
    char buff1[100], buff2[100];
    fd1 = open("myfifo1", O_RDONLY);
    fd2 = open("myfifo2", O_WRONLY);

    read(fd1, buff1, sizeof(buff1));    // read from fifo1
    printf("Received: %s\n", buff1);

    printf("Reply: ");
    scanf(" %[^\n]", buff2);
    write(fd2, buff2, sizeof(buff2));   // write to fifo2

    close(fd1);
    close(fd2);
}
```

👉 This way, Program1 writes → Program2 reads → Program2 replies → Program1 reads.

Using select() with FIFO

Problem: If one end of FIFO is not opened, the other end **blocks forever** at open() or read().

Solution: use select() to wait only for some time.

Example

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/select.h>
int main() {
   int fd;
   char buff[100];
   fd_set rfds;
   struct timeval tv;

   fd = open("myfifo", O_RDONLY);

   FD_ZERO(&rfds);
   FD_SET(fd, &rfds);

   tv.tv_sec = 5;       // wait 5 seconds
   tv.tv_usec = 0;
   if (!select(fd + 1, &rfds, NULL, NULL, &tv)) {
      printf("No data is available for reading yet\n");
   } else {
      printf("Data is available now\n");
      read(fd, buff, sizeof(buff));
      printf("Data from FIFO: %s\n", buff);
   }
   close(fd);
}
```

👉 If no writer sends data within 5 seconds → it prints *No data available*.
👉 If data arrives within 5 seconds → it reads and displays it.

This avoids being stuck forever.

System V IPC (Inter-Process Communication)

Why System V IPC?

- Normal **Pipe** and **FIFO** are not enough for many applications.
- So, **System V IPC** was introduced as a **single package** that provides:
    - **Message Queues (MQ)**
    - **Shared Memory (SHM)**
    - **Semaphores (SEM)**

✅ These mechanisms remain in the **kernel** (kernel persistent) until:

- You **delete them explicitly**, or
- The **system is rebooted**.

👉 You can check them using:

```
man 7 svipc
```

👉 You can see active IPCs with:

```
ipcs
```

System V IPC Attributes

Each IPC object (MQ, SHM, SEM) has some attributes:

- **Key** → unique identifier.
- **ID** → internal ID assigned by kernel.
- **Owner** → who created it.
- **Permission** → access rights.
- **Size** → depends on type (msg size, shared mem size, number of semaphores, etc).

📌 These are stored in a structure called **ipc_perm**.

IPC Key (Very Important)

To create **Message Queue, Semaphore, or Shared Memory**, you need a **unique key**.

- This is called the **System V IPC Key**.
- Works like a **file descriptor** (used to identify IPC resources).

How to generate this key?

Use the **ftok()** function:

```
key_t key = ftok("somefile", proj_id);
```

- somefile → path to an existing file.
- proj_id → an integer (only lower 8 bits are used).

👉 The system uses **file path + proj_id** to generate a unique key.

Example:

- You can pass anything as proj_id (like roll number).
- Requirement = must be **unique**.
- That's why **ftok()** is useful.

ID

1. Get Function (xxxget)

This is used to **create or access** an IPC object (like message queue, shared memory, or semaphore).

👉 Syntax:

```
int xxxget(key_t key, int xxxflg);
```

- xxx → can be msg, shm, or sem (depending on what you want).
- key → a unique identifier to access the IPC object.
- xxxflg → flags (options) like **create new object** or **exclusive access**.

✅ **If successful** → returns an identifier (ID).
❌ **If failed** → returns -1.

🔑 Keys can be generated by:

- ftok() function (standard way)
- Using a positive integer
- Using IPC_PRIVATE (always gives a unique new key)

⚡ Common flags:

- IPC_CREAT → Create the object if it doesn't exist
- IPC_EXCL → Fail if the object already exists (use with IPC_CREAT)

---

2. Control Function (xxxctl)

This is used to **modify, check, or remove** an existing IPC object.

👉 Syntax:

```
int xxxctl(int xxxid, int cmd, struct xxxid_ds *buffer);
```

- xxxid → ID of the IPC object (obtained using xxxget)
- cmd → what action to perform
- buffer → structure to send/receive data about the object

✅ **If successful** → returns 0
❌ **If failed** → returns -1

📌 Commands:

- IPC_STAT → Get info about the object
- IPC_SET → Change settings of the object
- IPC_RMID → Remove (delete) the object

Message Queue

- It is a way for processes to **communicate by sending and receiving messages**.
- Unlike simple FIFO pipes, message queues:
    - Can **store multiple messages**.
    - Support **message boundaries** (each message is separate, not just a stream of bytes).
    - Allow **different types of messages**, not just plain text.

Think of it like a **mailbox**:

- Many processes can put letters (messages) into it.
- Other processes can take letters out.
- Each letter has a "label" (message type).

🔹 **Operations you can do**

1. **Create a message queue**
2. **Send messages** into the queue
3. **Receive messages** from the queue
4. **Remove (delete)** the queue when done

✅ Any process with permission can use the queue.

🔹 **Internal Structure**

- A message queue is like a **doubly linked list** of messages.
- New messages are **added at the tail**.
- Old messages are **read from the head** (FIFO order by default).

You can check existing queues using the command:

```
ipcs -q
```

🔹 Message Structure

When sending/receiving a message, it must fit a specific structure.

Standard structure:

```
struct msgbuf {
   long mtype;   // message type (like a label, must be > 0)
   char mtext[1]; // message text
};
```

Custom structure:

You can define your own with bigger message space or more fields:

```
struct My_msgQ {
   long mtype;       // message type
   char mtext[1024];  // message text (bigger buffer)
   void *xyz;        // extra field if needed
};
```

👉 This allows you to send **structured messages**, not just plain strings.

🔹 Example

Imagine multiple apps sharing updates:

- Process A sends: mtype=1, mtext="Hello"
- Process B sends: mtype=2, mtext="Urgent Alert"
- Receiver can choose to read:
    - Any message
    - Or only messages of type 2 (urgent ones)

msgget() – Create or Get a Message Queue

```
int msgget(key_t key, int msgflg);
```

- **Purpose** → Creates a new message queue or gets the ID of an existing one.
- **Returns** → A **message queue identifier (ID)** if successful, or **-1** if there's an error.

🔹 Parameters

1. **key** → A unique number that identifies the queue.

- Can be generated using ftok().
- Or can use a fixed number.
- Or IPC_PRIVATE → always creates a brand new queue.

2. **msgflg** → Flags/permissions (similar to open() for files).
   - IPC_CREAT → create queue if it doesn't already exist.
   - IPC_EXCL → with IPC_CREAT, ensures error if queue already exists.
   - File permissions → just like files (e.g., 0666 for read/write by all).

msgsnd() – Send a Message

```
int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
```

- **Purpose**: Put a message into the message queue.
- **Parameters**:
  - msqid → Queue ID (from msgget()).
  - msgp → Pointer to your message (must follow struct msgbuf).
  - msgsz → Size of the message text (mtext), not counting mtype.
  - msgflg → Flags (e.g., blocking/non-blocking).

✅ Returns 0 if success, ❌ -1 if error.

---

msgrcv() – Receive a Message

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

- **Purpose**: Read a message from the message queue.
- **Parameters**:
  - msqid → Queue ID (from msgget()).
  - msgp → Pointer to message buffer to store the received message.
  - msgsz → Max size of message text you can receive.
  - msgtyp → Which type of message you want:
    - 0 → first message in queue (FIFO).
    - >0 → first message of that type.
    - <0 → smallest type ≤ absolute value.
  - msgflg → Flags (e.g., IPC_NOWAIT).

✅ Returns **number of bytes received**. ❌ Returns -1 if error.

🔹 Message Structure

All messages must follow a structure like this:

```
struct msgbuf {
   long mtype;       // message type (>0), acts like a label
   char mtext[100];   // actual message content
};
```

👉 mtype helps in categorizing messages (urgent, normal, etc.).
👉 mtext can be plain text or even a custom struct.

🔹 Example

Send:

```
struct msgbuf message;
message.mtype = 1;                  // type 1
strcpy(message.mtext, "Hello");     // message text
msgsnd(msqid, &message, strlen(message.mtext)+1, 0);
```

Receive:

```
struct msgbuf message;
msgrcv(msqid, &message, sizeof(message.mtext), 1, 0);
printf("Received: %s\n", message.mtext);
```

Message Type (mtype)

- Every message has a **type (id)** stored in msgbuf.mtype.
- This type doesn't have to be unique → multiple messages can share the same type.
- If multiple messages have the same type, they are delivered in **FIFO order**.

👉 Example:

- You send 3 messages with mtype=5.
- When receiving msgtyp=5, you'll get them in the order they were sent.

🔹 Receiving Rules (msgtyp parameter in msgrcv)

When you call msgrcv(), the **msgtyp** argument decides which message you'll get:

1. **msgtyp = 0**
   → Get the **first message** in the queue (FIFO).
2. **msgtyp > 0** (positive integer)
   → Get the **first message with exactly this type**.

3. **msgtyp < 0** (negative integer)
   → Get the **first message with type ≤ |msgtyp|**.
   (So, -5 means get the first message whose type ≤ 5.)

- **msgsz (size of message text)**
  - This tells the system **how big your message buffer is** (msgp->mtext).
  - If the actual message is larger than msgsz, behavior depends on flags (truncation vs error).

- **msgflg (flags)**

Flags control whether the process **waits** or **returns immediately** if things aren't available.

For msgsnd():

- msgflg = 0 → Wait until enough space in kernel buffer.
- msgflg = IPC_NOWAIT → Don't wait, just return -1 (error) if no space.

For msgrcv():

- msgflg = 0 → Wait until a message of that type arrives.
- msgflg = IPC_NOWAIT → Don't wait, return -1 (error) if no such message is available.

- **Important Note: Kernel Persistence**
  - Messages are stored in the **kernel's buffer**, not just in your process.
  - That means messages **stay in the queue** until someone receives them or the queue is deleted.
  - This allows processes to communicate **asynchronously**.

Create Message Queue:

```
int main() {
   key_t key;
   int msgid;
   key = ftok(".", 'a');
   msgid = msgget(key, IPC_CREAT | IPC_EXCL | 0744);
   printf("key=0x%0x\t msgid=%d\n", key, msgid);
   return 0;
}\
```

Sender Message Queue:

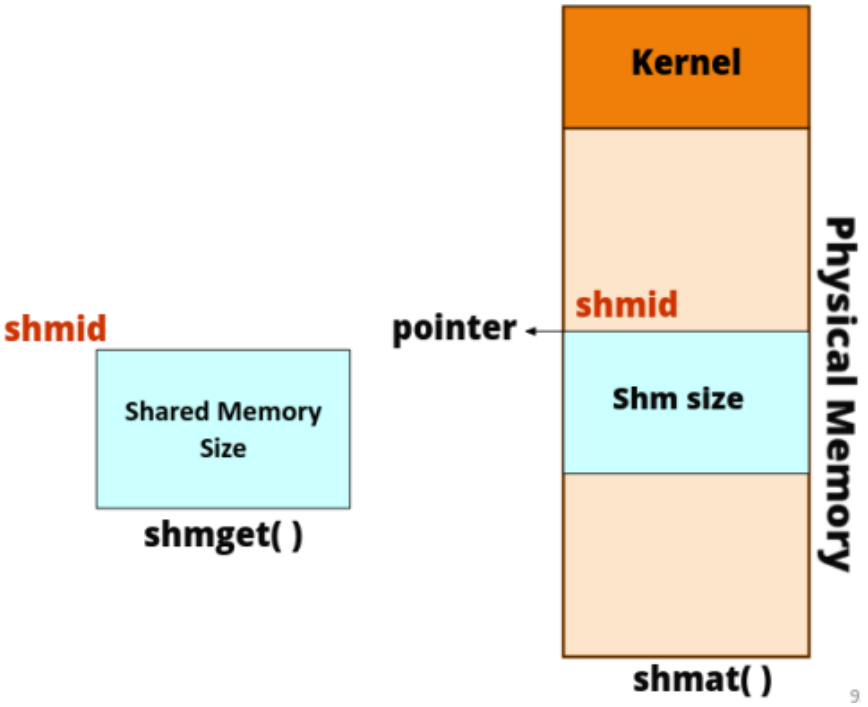
```
int main() {
    struct msg {
        long int m_type;
        char message[80];
    } myq;

    key_t key;
    int mqid;
    size_t size;
    key = ftok(".", 'a');
    mqid = msgget(key, 0);
    printf("Enter message type: ");
    scanf("%ld", &myq.m_type);
    getchar(); // clear newline
    printf("Enter message text: ");
    scanf("%[^\n]", myq.message);
    size = strlen(myq.message);
    // size + 1 to include '\0'
    msgsnd(mqid, &myq, size + 1, 0);
    return 0;
}
```

Receiver Message Queue:

```
int main() {
    struct msg {
        long int m_type;
        char message[80];
    } myq;

    key_t key;
    int mqid;
    int ret;
```

```
    key = ftok(".", 'a');
    mqid = msgget(key, 0);
    printf("Enter message type: ");
    scanf("%ld", &myq.m_type);

    ret = msgrcv(mqid, &myq, sizeof(myq.message), myq.m_type, 0);
    //ret = msgrcv(mqid, &myq, sizeof(myq.message), myq.m_type, IPC_NOWAIT);
    if (ret == -1)
        exit(-1);
    printf("Message type: %ld\nMessage: %s\n", myq.m_type, myq.message);
    return 0;
}
```

Shared Memory



Why Shared Memory?

- In **message queues**, every time data goes from one process to another, it has to copy twice: **process → kernel → process.**
  This is slow and wasteful.
- Shared memory solves this by letting **both processes directly use the same memory area.**

- **Benefits**
  - **Very fast IPC (Inter-Process Communication)** → because no extra copying.
  - **Flexible & easy to use.**
  - Acts like a **normal memory area** once set up.
  - Useful for many things:
    - Global variables
    - Shared libraries
    - Word processors
    - Multiplayer games
    - Web servers (HTTP daemons)
    - Programs in C, Perl, etc.
- **How it works**
  1. **Create shared memory** using shmget().
     - Returns an **ID (shmid)** for the memory block.
  2. **Attach it to your process** using shmat().
     - This links it into your **user space** so you can read/write directly.
     - That's why it's the **fastest IPC method**.
  3. **Use it like normal memory** → read/write directly.
  4. **Detach & delete** when done.
- **Important Notes**
  - You can make shared memory **read-only or read/write**.
  - The **OS manages the actual physical memory** behind the scenes (virtual memory mapping).
  - In embedded systems (without virtual memory), you may need to know the **physical address**.

1. shmget() → Create shared memory

```
int shmget(key_t key, size_t size, int shmflg);
```

- key: Unique number (like an identifier).
- size: How much memory you want.
- shmflg: Flags (e.g., create new, permissions).
- Returns: **shmid** (shared memory ID).

👉 If memory doesn't exist and you use IPC_CREAT, it creates a new one.

2. shmat() → Attach to process

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

- shmid: ID from shmget().
- shmaddr: Starting address (usually NULL, OS chooses).
- shmflg: Read/Write permissions.
- Returns a **pointer** to the shared memory → now you can use it like a normal array.

Reading & Writing Shared Memory

- Once you attach shared memory using shmat(), you get a **pointer**.
- Reading and writing is just like normal memory access with that pointer.

Example

```
// Reading
printf("SHM contents: %s\n", data);


// Writing
printf("Enter a string: ");
scanf("%[^\n]", data);
```

👉 Here data is the pointer returned by shmat().

Detaching & Removing Shared Memory

1. Detach (disconnect from process)

- Use shmdt(pointer);
- This removes the shared memory segment from your process's address space.
- But the segment still exists in the system.

👉 Syntax:

```
int shmdt(void *shmaddr);
```

2. Remove (delete from system)

- Even after detaching, the memory block still exists until it's **removed**.
- Use shmctl() with IPC_RMID.

👉 Syntax:

```
int shmctl(shmid, IPC_RMID, NULL);
```

- This permanently deletes the shared memory segment.
- Returns 0 on success, -1 on error.

Disadvantages of Shared Memory

1. No Appending

- Shared memory is just a **fixed-size memory block**.
- You **cannot find the "end"** like in a file where you keep appending.
- So, you can only:
    - **Read existing data**
    - **Write new data (overwrite old content)**
- If you want append-like behavior, you must implement your own **data structure** (e.g., linked list, queue, or offset tracker).

---

2. Race Condition (Simultaneous Access)

- Multiple processes can read and write **at the same time**.
- If two processes write together → **data corruption**.
- Example:
    - Process A writes "Hello"
    - Process B writes "World" at the same time → shared memory might end up with "Horld" or "Wello".

👉 **Solution:**

- Allow multiple **readers** at once.
- But allow only **one writer** at a time.
- Use **synchronization tools** like:
    - **Semaphores**
    - **Mutex (locks)**
    - Or implement a **custom locking protocol**.

Programs to demonstrate Shared Memory

1. Create and Write (writer.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_SIZE 1024
```

```
int main() {
    key_t key = ftok("shmfile", 65);  // generate key
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);

    char *data = (char *)shmat(shmid, NULL, 0);

    printf("Enter some text: ");
    fgets(data, SHM_SIZE, stdin);

    printf("Data written to shared memory: %s\n", data);

    shmdt(data); // detach
    return 0;
}
```

---

2. Read-Only Attach (reader.c)

```
int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, SHM_SIZE, 0666);

    // Attach in read-only mode
    char *data = (char *)shmat(shmid, NULL, SHM_RDONLY);

    printf("Data read from shared memory: %s\n", data);

    // Try to overwrite (this will fail or crash)
    printf("Trying to overwrite in read-only mode...\n");
    strcpy(data, "This should fail!");

    shmdt(data); // detach
    return 0;
}
```

3. Detach Only (detacher.c)

```
int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, SHM_SIZE, 0666);

    char *data = (char *)shmat(shmid, NULL, 0);

    if (shmdt(data) == 0) {
        printf("Shared memory detached successfully.\n");
    } else {
        perror("shmdt");
    }

    return 0;
}
```

---

4. Remove Shared Memory (remover.c)

```
int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, SHM_SIZE, 0666);

    if (shmctl(shmid, IPC_RMID, NULL) == 0) {
        printf("Shared memory removed successfully.\n");
    } else {
        perror("shmctl");
    }

    return 0;
}
```

Semaphores

- **Synchronization Tool**: Used to coordinate multiple processes or threads so they can share resources without conflict.
- **Not for message passing**: Unlike message queues or pipes, semaphores don't transfer data; they only control access.
- **Integer value**: Each semaphore holds an integer that represents availability of a resource.
    - Initialized at the start (e.g., max users, slots, or resources available).
- **Types of Semaphore**:
    1. **Binary Semaphore (Mutex)** → Max value = 1
       Used for locking/unlocking (like a switch: either free or occupied).
    2. **Counting Semaphore** → Max value > 1
       Used when multiple instances of a resource exist (e.g., 100 users allowed).

Example (Counting Semaphore)

- A website allows only 100 users at a time.
- Semaphore initialized to 100.
- Each login → decrement.
- Each logout → increment.
- If semaphore = 0 → new logins must wait (blocked).

P() and V() Operators

- **P()** (Proberen, "test") → Decrement operation (request resource).
- **V()** (Verhogen, "increment") → Increment operation (release resource).
- Both are **atomic** → cannot be interrupted.

Busy Waiting

- Normally: a process may keep using CPU while waiting (wasteful).
- With semaphores: waiting process goes to **sleep state** → avoids CPU wastage.

Applications

- Semaphores are used for synchronization in:
    - **Shared Memory segments**
    - **Message Queues**
    - **Files (locking mechanisms)**

Semaphore Operations

Semaphores mainly do two operations:

Increment (V operation / Unlock)

```
int v(int i) {
   i = i + 1;   // unlock
   return i;
}
```

- Used when a process **releases** a resource.
- Increases the semaphore value.

Decrement (P operation / Lock)

```
int p(int i) {
   if (i > 0)
      i--;    // lock
   else
      wait till i > 0;   // process must wait
   return i;
}
```

- Used when a process **wants to use** a resource.
- If the value is **> 0**, it decrements (lock successful).
- If the value is **0**, process must wait (blocked).

👉 This is how semaphores control access to the **critical section**.

System V Semaphores

Unlike a single semaphore, **System V** allows a **set of semaphores** (like an array).

- One semaphore = controls **one critical section**.
- If a program has **10 critical sections**, you'd normally need 10 separate semaphores.
- But with **System V**, you can group them together into a **semaphore set**.

Features:

- You decide how many semaphores to keep in the set.
- Each semaphore in the set can be **binary** (0/1) or **counting** (>1).
- Each semaphore controls access to one resource.
- The whole set is controlled by a **single semaphore ID** (simplifies management).

🔹 **Example (Semaphore Set)**

Suppose a program has 3 shared resources:

- Database connection
- Printer
- File system

Instead of creating 3 separate semaphore IDs, you create **one semaphore set with 3 semaphores**.

- sem[0] → controls database
- sem[1] → controls printer
- sem[2] → controls file system

All managed by **one semaphore ID**.

Semaphore Creation

- To create or get a semaphore set, we use:
- semid = semget(key, 1, IPC_CREAT | 0644);
    - key → unique identifier (generated by ftok() usually).
    - 1 → number of semaphores in the set (here just one).
    - IPC_CREAT | 0644 → create if doesn't exist, and give read/write permission (octal 644).
- Then we initialize it with:
- union semun {
- int val; // for SETVAL
- struct semid_ds *buf; // for IPC_STAT, IPC_SET
- unsigned short int *array; // for GETALL, SETALL
- };
- union semun arg;
- arg.val = 1; // Initial semaphore value
- semctl(semid, 0, SETVAL, arg);

---

1. Semaphore Definition

Each semaphore has some internal fields (inside the kernel):

- semval → current semaphore value.
- semzcnt → number of processes waiting for it to become **0**.
- semncnt → number of processes waiting for it to **increase**.
- sem_pid → PID of the process that last operated on it.

These help the kernel track semaphore usage.

---

2. Union semun

- Used when calling semctl() to pass parameters.
- **Why union instead of struct?**
  - A **union** shares memory for all its members; only one member is active at a time.
  - Size of a **union** = size of its **largest** member.
  - Size of a **struct** = sum of sizes of **all** its members.

Since semctl() uses only **one** field at a time (val, buf, or array), union saves memory.

---

3. semget() Method

- Used to create/get semaphore set:
- int semget(key_t key, int nsems, int semflg);
  - key → unique identifier (from ftok()).
  - nsems → number of semaphores in the set.
  - semflg → permissions + flags (IPC_CREAT etc.).
- Returns semaphore ID (semid), which is then used by other functions (semctl, semop).

Prototype

int semop(int semid, struct sembuf *sops, size_t nsops);

- **semid** → ID of the semaphore set (from semget()).
- **sops** → pointer to an array of operations (struct sembuf).
- **nsops** → number of operations in that array (you can operate on multiple semaphores atomically).

---

sembuf Structure

```
struct sembuf {
   unsigned short sem_num; // semaphore index in the set
   short sem_op;          // operation to perform
   short sem_flg;         // operation flags
};
```

Meaning of sem_op:

- **sem_op > 0 → increment** semaphore value (unlock / signal).
  Example: sem_op = 1 means release one resource.

- **sem_op = 0** → wait until semaphore value becomes zero.
  Example: useful for synchronization barriers.
- **sem_op < 0** → **decrement** semaphore value (lock / wait).
  - If semval ≥ |sem_op| → subtract and proceed.
  - If semval < |sem_op| → process sleeps until enough resources are available.
    Example: sem_op = -1 means acquire one resource.

Meaning of sem_flg:

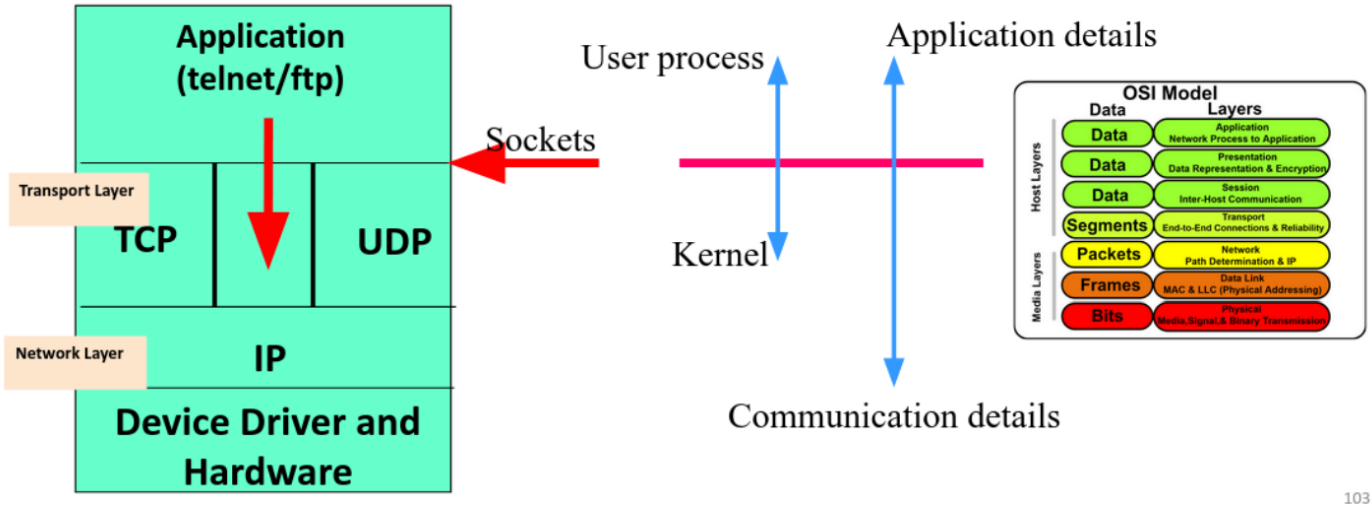
- **0** → default blocking behavior (wait if not possible).
- **IPC_NOWAIT** → don't wait, return -1 immediately if not possible.
- **SEM_UNDO** → automatically undo operation if process dies (safely release locks).

FIFO vs MQ vs SHM vs SEM

Feature	FIFO (Named Pipe)	Message Queue	Shared Memory	Semaphore
Purpose	Data transfer between processes	Data transfer via structured messages	Share large data directly	Synchronize process access
Type of IPC	Byte stream	Message-based	Memory-based	Synchronization
Communication Direction	Unidirectional (one-way)	Bidirectional (can send/receive)	Bidirectional (read/write shared data)	No data transfer, only signaling
Speed	Moderate	Moderate	Very fast	Not for data transfer
Data Stored In	Kernel buffer (temporary)	Kernel-managed queue	Common memory segment	Kernel counter
Persistence	Exists until explicitly removed	Persists until removed	Persists until detached/removed	Persists until removed
Used Between	Related or unrelated processes	Related or unrelated processes	Related or unrelated processes	Related or unrelated processes
Synchronization Required?	No (blocking read/write handled by kernel)	No (kernel queues messages)	Yes (must use semaphores/mutexes)	N/A (used for synchronization)
Complexity	Simple	Moderate	High (requires semaphore)	Simple to moderate

Feature	FIFO (Named Pipe)	Message Queue	Shared Memory	Semaphore
Example Use	mkfifo, pipes in Linux	POSIX message queues	Shared buffers, databases	Critical section locks
Data Format	Stream of bytes	Structured message with type	Raw memory	Integer counter

Socket Programming



103

Think of **sockets** as a way for two computers (with different IP addresses) to talk to each other over a network.

1. What is a Socket?

- A **socket** is like a "door" through which data goes in and out of a computer.
- It lets two programs (even on different machines) send messages back and forth.
- It supports **full-duplex communication** → both sides can talk and listen at the same time (like a phone call).

A **socket address = IP address + Port number**.

2. Where is Socket Used in the OSI Layers?

- Socket sits **between the Application layer and Transport layer**.
- Above socket → Application (your program, e.g., browser, WhatsApp, telnet).
- Below socket → Transport (TCP/UDP), Network (IP), and Hardware.

3. TCP vs UDP Sockets

- **TCP socket (SOCK_STREAM)**
  Reliable, continuous, like a **phone call**. Data arrives in order.

- **UDP socket (SOCK_DGRAM)**
  Faster but unreliable, like sending **letters**. No guarantee they arrive in order.

4. How does data travel?

- Your application writes data into the socket.
- Socket hands it to TCP/UDP (Transport layer).
- Then it goes to IP (Network layer).
- Finally, it reaches the device driver + network hardware (like your Wi-Fi card).
- The other computer's socket receives it, and passes it up to the application.

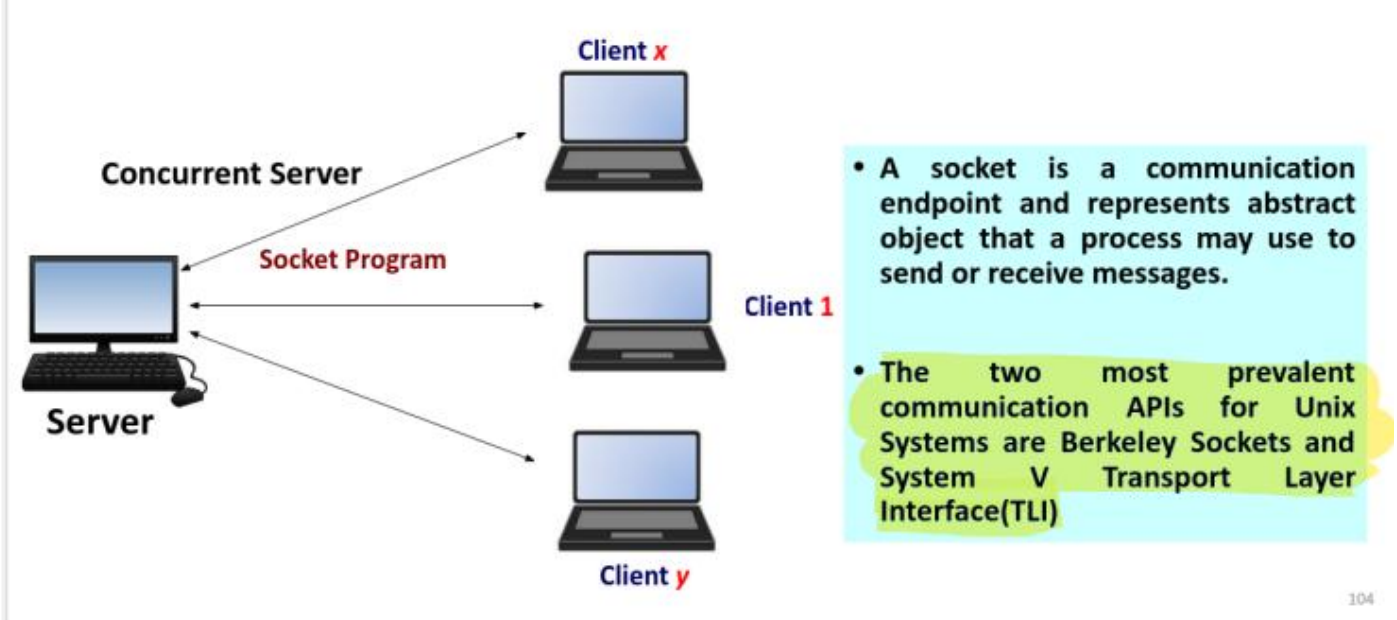
5. Why use Network Interfaces (not device files)?

- In Linux, everything is usually a "file". But network interfaces (like eth0, wlan0) are not disk files.
- They don't store data like files; instead, they just send/receive packets.
- You can check available interfaces with ifconfig.

6. User Space vs Kernel Space

- **User space**: Your program + socket.
- **Kernel space**: TCP, UDP, IP, drivers, and hardware.
- So only the **Application layer + Socket** are in **User Space**; everything else happens in **Kernel Space**.

Client-Server Model



Types of Servers

1. Iterative Server

- Works with **one client at a time**.

- If another client tries to connect → it must **wait** until the first client disconnects.
- Think of it like a **shop with only one counter**:
    - Customer 1 is being served.
    - Customer 2 has to wait in line until Customer 1 is done.

---

2. Concurrent Server

- Can handle **many clients at the same time**.
- When a new client connects, the server creates a **separate handler** for it (so it doesn't block others).
- Two ways to create this handler:
    1. **fork()** → creates a new process for each client.
        - Like opening a **new counter** for every customer.
        - Each process is independent.
    2. **Multithreading** → creates a new thread for each client.
        - Like one counter, but many employees (threads) working inside it.
        - More lightweight and efficient compared to fork.

Socket() System call

What is a socket()?

- socket() is a system call used to create a communication endpoint between two programs (like client and server).
- Think of it like opening a "phone line" between two computers/programs.

Important Points

1. **Client & Server are not the same**
    - A **client** asks for a service (like a browser asking for a webpage).
    - A **server** provides the service (like a web server sending the page).
    - You can't swap them because they behave differently.
2. **Connection Types**
    - **Connection-oriented** → like making a phone call (both sides stay connected). Example: **TCP**.
    - **Connectionless** → like sending a letter/postcard (just send it, no continuous connection). Example: **UDP**.
3. **UNIX I/O vs Network**
    - In Unix, normal input/output (like reading files) is **stream-based** (like reading a continuous flow of data).

- Network communication needs **more parameters** (domain, type, protocol).

4. **Network Interface**
    - Should support many different protocols, not just one.

---

socket() Function

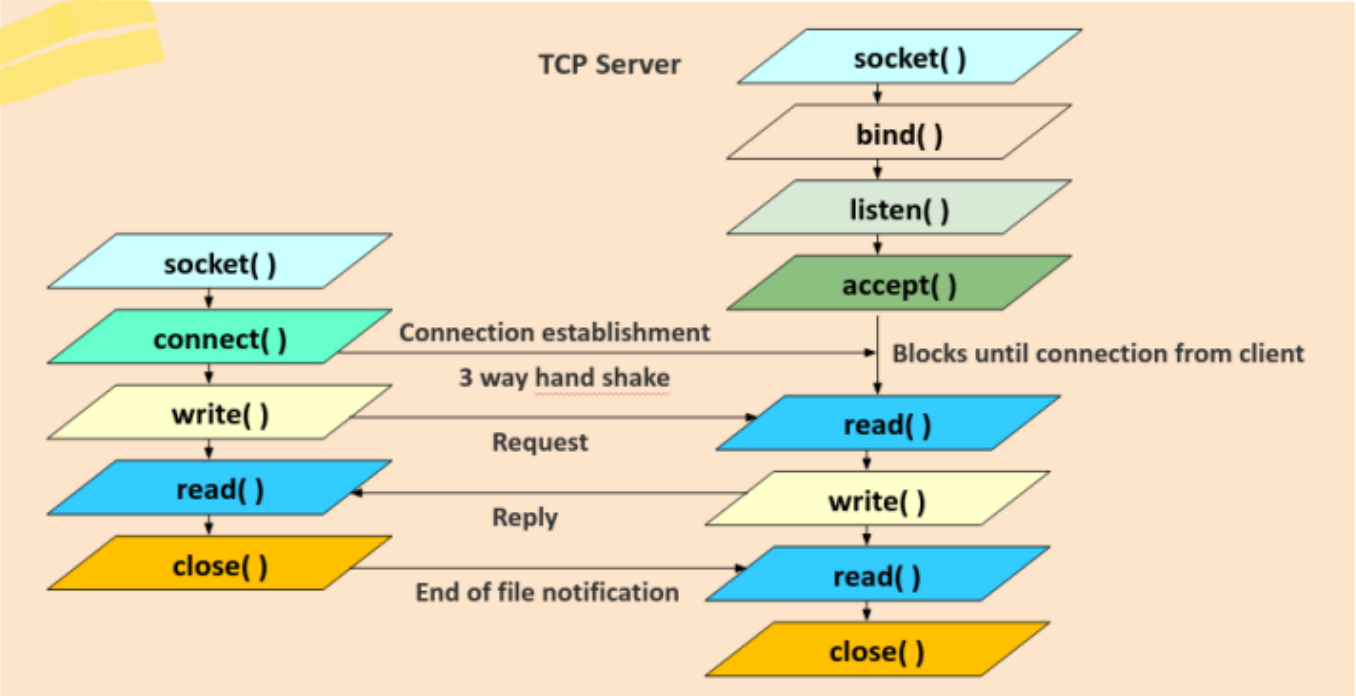
int socket(int domain, int type, int protocol);

- **Domain (Address Family)**
    - AF_UNIX → Client and Server are on the **same machine**.
    - AF_INET → Client and Server are on **different machines** (over Internet/Network).
- **Type**
    - SOCK_STREAM → TCP (connection-oriented, reliable).
    - SOCK_DGRAM → UDP (connectionless, faster but no guarantee).
- **Protocol**
    - Identifies the exact application protocol (like TCP, UDP).
    - Usually can just pass 0 (system will pick the default).
    - Protocol numbers are listed in /etc/protocols.

Key Notes

- Socket system call returns a **socket descriptor** (like a file handle) on success, or -1 if it fails.
- If you choose a protocol manually, make sure the port isn't already used by another service.
- To check, use cat /etc/services.

Socket Functions



[Remember this diagram for exam]

1. Making sockets

- Server makes a socket using socket().
- Client also makes a socket using socket().
  👉 Think of a socket as a *telephone line* that both sides set up.

---

2. Server gets ready

- The server ties its socket to an IP + Port using bind().
- Then it says "I'm ready for calls" using listen().
  👉 Like a shopkeeper putting a board: *"Shop open at address X, waiting for customers."*

---

3. Client connects

- The client calls connect() to reach the server.
- The server accepts this with accept().
  👉 Like the customer dialing the shop's phone number, and the shopkeeper picking up the call.

---

4. Handshake

- Before real talking, there's a 3-step handshake:
  1. Client says "SYN" (I want to talk).
  2. Server replies "SYN+ACK" (Okay, I'm ready).

3. Client says “ACK” (Let’s go!).
   👉 Now the connection is officially open.

---

5. Talking (Data transfer)

- Client sends data using write().
- Server receives it with read().
- Server replies using write().
- Client receives it with read().
  👉 Like chatting on the phone: *“Hello?”* → *“Hi there!”*

---

6. Closing

- When done, the client hangs up with close().
- Server notices this and also closes with close().
  👉 Conversation ends, both phones go down.

✅ That’s the full life cycle of client-server communication using sockets:
Create → Bind → Listen → Connect → Accept → Handshake → Talk → Close

Server Code:

```
int main() {
   struct sockaddr_in serv, cli;
   int sd, nsd;
   socklen_t sz;
   char buf[80];

   // Create socket
   sd = socket(AF_INET, SOCK_STREAM, 0);

   // Setup server address
   serv.sin_family = AF_INET;
   serv.sin_addr.s_addr = INADDR_ANY;   // Bind to all interfaces
   serv.sin_port = htons(3558);       // Port number

   // Bind socket
```

```
    bind(sd, (struct sockaddr*)&serv, sizeof(serv));

    // Listen for clients
    listen(sd, 5);

    sz = sizeof(cli);

    // Accept connection
    nsd = accept(sd, (struct sockaddr*)&cli, &sz);

    // Read message from client
    read(nsd, buf, sizeof(buf));
    printf("Message from client: %s\n", buf);

    // Send reply
    write(nsd, "ACK from Server\n", 17);

    close(nsd);
    close(sd);
    return 0;
}
```

Client Code:

```
int main() {
    struct sockaddr_in serv;
    int sd;
    char buf[80];

    // Create socket
    sd = socket(AF_INET, SOCK_STREAM, 0);

    // Setup server address
    serv.sin_family = AF_INET;
```

```
    serv.sin_addr.s_addr = inet_addr("127.0.0.1");  // Server IP
    serv.sin_port = htons(3558);

    // Connect to server
    connect(sd, (struct sockaddr*)&serv, sizeof(serv));

    // Send message
    write(sd, "Hello Server\n", 13);

    // Receive reply
    read(sd, buf, sizeof(buf));
    printf("Message from server: %s\n", buf);

    close(sd);
    return 0;
}
```

Socket Structure And htos():

1) sockaddr_in — what it is and its fields

This is the C structure you use to describe an IPv4 socket address (IP + port).

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t    sin_family; // usually AF_INET (IPv4)
    in_port_t      sin_port;   // 16-bit port number — **in network byte order**
    struct in_addr sin_addr;   // IPv4 address (32-bit) — **in network byte order**
    // plus padding bytes on some systems
};
```

- sin_family — tells the socket API this is IPv4 (AF_INET).
- sin_port — the TCP/UDP port (e.g. 80, 8080). **Important:** the integer must be stored in *network byte order* (see next sections) — so set it with htons(...).
- sin_addr.s_addr — a 32-bit IPv4 address. Functions like inet_addr() or inet_pton() give you this in *network byte order*, so you can assign directly.

Common code:

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);                // host -> network short (16-bit)
inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr); // preferred over inet_addr
bind(listen_sock, (struct sockaddr*)&addr, sizeof(addr));
```

2) Endianness — big vs little endian (easy picture)

Endianness = which byte of a multi-byte number goes first in memory.

Take the 32-bit integer 0x000164C1 (decimal 91329).

To get its bytes:

- low (least significant) byte = 0xC1 = 193
- next byte = 0x64 = 100
- next = 0x01 = 1
- highest = 0x00 = 0

Memory layout if stored starting at address A:

- **Big-endian (network byte order)** — MSB first:
- memory[A] = 00
- memory[A+1] = 01
- memory[A+2] = 64
- memory[A+3] = C1

(When you look at bytes left→right you see 00 01 64 C1.)

- **Little-endian (Intel PCs)** — LSB first:
- memory[A] = C1
- memory[A+1] = 64
- memory[A+2] = 01
- memory[A+3] = 00

(Bytes left→right: C1 64 01 00.)

Short definition:

- **Big-endian:** most-significant byte at the lowest address.
- **Little-endian:** least-significant byte at the lowest address.

(We computed 91329 = 0x000164C1 by dividing: 91329 % 256 = 193 (0xC1), (91329 >> 8) % 256 = 100 (0x64), (91329 >> 16) % 256 = 1 (0x01), rest = 0x00.)

3) Network byte order vs host byte order

- **Network byte order** is standardized as **big-endian**. All internet protocols specify big-endian.
- **Host byte order** is whatever your machine uses — e.g., Intel is little-endian.

That means: when you put multi-byte integers into protocol structures (port numbers, 32-bit fields) you must convert to network order so the other side interprets the bytes correctly.

4) The conversion functions (the ones you'll use)

Defined in <arpa/inet.h> / <netinet/in.h>:

- htons(uint16_t x) — **Host TO Network Short** (16-bit). Use for ports.
- htonl(uint32_t x) — **Host TO Network Long** (32-bit).
- ntohs(uint16_t x) — **Network TO Host Short** (on receive/print).
- ntohl(uint32_t x) — **Network TO Host Long**.

Meaning: htons(8080) turns 8080 into bytes arranged as big-endian so it can be stored into sin_port.

Example: port 8080

- 8080 decimal = 0x1F90.
- Host little-endian memory would be 90 1F.
- htons(8080) makes memory 1F 90 (network big-endian).
  So to set sin_port: addr.sin_port = htons(8080);

5) IP address helpers

- inet_pton(AF_INET, "1.2.3.4", &addr.sin_addr) — modern, recommended. sin_addr gets network byte order.
- inet_ntop(AF_INET, &addr.sin_addr, buf, buflen) — convert sin_addr back to string.
- inet_addr("1.2.3.4") — older; returns in_addr_t in network byte order (works but deprecated).

When you receive a struct sockaddr_in from accept() or getpeername(), to print the client's port and IP:

```
struct sockaddr_in cli;
socklen_t len = sizeof(cli);
accept(..., (struct sockaddr*)&cli, &len);

// print port:
printf("client port = %u\n", ntohs(cli.sin_port));

// print ip:
char ipstr[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &cli.sin_addr, ipstr, sizeof(ipstr));
```

```
printf("client ip = %s\n", ipstr);
```

6) Why this matters (a short story)

If a little-endian machine writes port 80 into memory as 0x50 0x00 and sends those bytes without conversion, a big-endian reader will see 0x5000 (20480) — wrong port. htons/ntohs guarantee both sides agree on byte order.

7) Quick cheat sheet / checklist

- When filling sin_port: always addr.sin_port = htons(port_number);
- When filling sin_addr.s_addr: use inet_pton() or inet_addr() (they return network order).
- When reading sin_port from a sockaddr_in: use ntohs() to get readable host order.
- For 32-bit numbers you send in a protocol (not IP payload bytes): use htonl / ntohl.
- h = host, n = network, s = short (16-bit), l = long (32-bit).

8) Minimal working example (server bind)

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void) {
   int s = socket(AF_INET, SOCK_STREAM, 0);

   struct sockaddr_in srv;
   memset(&srv, 0, sizeof(srv));
   srv.sin_family = AF_INET;
   srv.sin_port = htons(12345);              // 12345 -> network order
   inet_pton(AF_INET, "0.0.0.0", &srv.sin_addr); // bind to all interfaces

   bind(s, (struct sockaddr*)&srv, sizeof(srv));
   listen(s, 5);
   // accept(), recv(), send(), ...
   close(s);
}
```

1) connect() (for client)

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

- **sockfd** → like a phone you bought (created by socket()).
- **serv_addr** → the server's phone number (IP + port).
- **addrlen** → size of the address.

👉 Client uses connect() to "dial" the server.
If successful, now the client and server can talk.

---

2) listen() (for server)

int listen(int sockfd, int backlog);

- **sockfd** → server's socket (like a restaurant front door).
- **backlog** → how many customers can wait in line if the restaurant is busy.

👉 listen() tells the OS: "I'm open for business. Start accepting customers."
Without listen(), the server won't accept incoming requests.

---

3) accept() (for server)

int accept(int sockfd, void *addr, int *addrlen);

- **sockfd** → the server's listening door.
- **addr** → info about the client (like their name/phone/IP).
- **addrlen** → size of that info.

👉 accept() is like the waiter opening the door and seating the next customer.
It gives the server a **new socket (new table)** for that specific client.
Now, the server can talk privately with that client without disturbing others.

---

4) close()

close(sockfd);

- Just like hanging up the phone.
- Once closed, you cannot send or receive messages using that socket.
- If you try, you'll get an error.

---

5) Important Notes (from slide, simplified)

- **Sockets are like pipes** → data flows both ways (client ↔ server).

- **One socket stays at the server** (listening socket).
- **Each client connection gets its own new socket** (returned by accept()).
- After connect() succeeds, client ↔ server can use read() and write() directly.

---

6) Small Example — Restaurant Analogy 🍽️

1. **Server side**:
    - Creates a socket (buys a phone).
    - Calls bind() → publishes its phone number (IP + port).
    - Calls listen() → opens the restaurant and allows a waiting line.
2. **Client side**:
    - Creates its socket (buys its own phone).
    - Calls connect() → dials the server's phone number.
3. **Server accepts**:
    - Server's accept() picks up the phone (answers client).
    - Now there's a **new line** (new socket) just for that client.
4. **Communication**:
    - Both sides use read() / write() (or recv() / send()) like talking on the phone.
5. **Closing**:
    - Either side can close() the socket = hang up.

Shutdown()

int shutdown(int sockfd,int how);

Imagine you're on a phone call. The shutdown() function is like ending the call, but you have options for how you hang up. The function takes two inputs:

- **sockfd**: This is the phone line you want to hang up. In programming, it's called a **socket file descriptor**.
- **how**: This is how you want to hang up. You have three choices:
    - **how = 0**: You can still talk and send messages, but you can't hear or receive anything from the other person.
    - **how = 1**: You can still hear and receive messages, but you can't talk or send anything to the other person.
    - **how = 2**: You can't talk or hear anything. Both sides of the conversation are completely shut down.

So, while the close() function is like slamming the phone down and ending the call immediately, **shutdown() gives you more control**.

Iterative vs Concurrent

Iterative server → Handles **one client at a time**. Others must wait.

- **Concurrent server** → Can handle **many clients at once** by creating a **child process/thread** for each client.

The Question

You have a concurrent server and you want to restrict the number of client connections to 100. How will you implement this?

This means → The server should **not allow more than 100 clients to connect at the same time**.

Why Counting Semaphore?

- A **semaphore** is like a "counter + lock".
- It is initialized with a maximum value (say **100**).
- Each time a client connects:
  - The semaphore value is **decremented** (one slot taken).
- When the client finishes:
  - The semaphore value is **incremented** (slot freed).
- If 100 clients are already connected, the semaphore value is 0 → new clients must **wait** until a slot frees.

General Implementation (Step by Step)

1. **Initialize a counting semaphore** with value = 100.
2. **sem_init(&sem, 0, 100); // 100 client limit**
3. **Before forking/creating a thread for a new client**, do:
4. **sem_wait(&sem); // decrement (take one slot)**
   - If semaphore > 0 → client allowed.
   - If semaphore = 0 → block/wait until a slot is free.
5. **In the child process/thread**, after finishing communication:
6. **sem_post(&sem); // increment (free one slot)**

Easy Analogy

Think of the server as a **restaurant with 100 tables**:

- Semaphore = **tables available**.

- When a client enters → take a table (sem_wait).
- When client leaves → free the table (sem_post).
- If no table available → client must **wait outside** until someone leaves.

👉 So the implementation is:

- Use a **counting semaphore initialized to 100**.
- sem_wait when a new client connects.
- sem_post when the client disconnects.

Persistence Level of IPC (Inter-Process Communication) Mechanisms

1. Unnamed Pipe

- Scope: **Process level**
- Exists only as long as the process (and its children) are running.
- Once the process terminates, the unnamed pipe **disappears**.
  👉 **Think of it like a temporary water pipe inside a house that disappears when the house is destroyed.**

---

2. Named Pipe (FIFO)

- Scope: **File system level**
- Created as a special file in the filesystem.
- Exists until the file is **explicitly deleted** (using rm) or the filesystem is unmounted.
- Even if no process is using it, the FIFO file stays in the directory.
  👉 **Like a public telephone booth installed on the street. Even if nobody is talking, the booth is still there until someone removes it.**

---

3. Message Queue, Shared Memory, Semaphores

- Scope: **Kernel level**
- These are created and maintained by the **kernel**.
- They survive even after the process that created them dies.
- But they are cleared when:
  - Explicitly deleted (e.g., ipcrm)
  - Or when the system is rebooted (since kernel memory is reset).
    👉 **Like noticeboards, lockers, or keys managed by a building's manager (kernel). They remain until the manager clears them or the whole building is reset (reboot).**

Quick Summary Table

IPC Mechanism	Persistence Level	Terminates When
Unnamed Pipe	Process Level	Process ends
Named Pipe (FIFO)	File System Level	FIFO file deleted / filesystem unmounted
Message Queue	Kernel Level	Explicitly deleted or system reboot
Shared Memory	Kernel Level	Explicitly deleted or system reboot
Semaphores	Kernel Level	Explicitly deleted or system reboot

Signals

What are Signals?

- **Signals = messages between processes.**
- They are mainly used for **inter-process communication (IPC)**.
- Example: When you press Ctrl+C, your terminal sends a **signal** (SIGINT) to stop the running program.

When are signals generated?

A signal can be generated when:

1. **Event occurs** (like timer/alarm goes off).
2. **User quota exceeded** (too many processes, too much memory).
3. **I/O device ready** (keyboard/mouse).
4. **Illegal instruction** (divide by zero, segmentation fault).
5. **Keyboard interrupts**
    - Ctrl+C → SIGINT
    - Ctrl+Z → SIGSTOP
6. **Another process sends it** (e.g., kill -9 <pid>).

How do signals look?

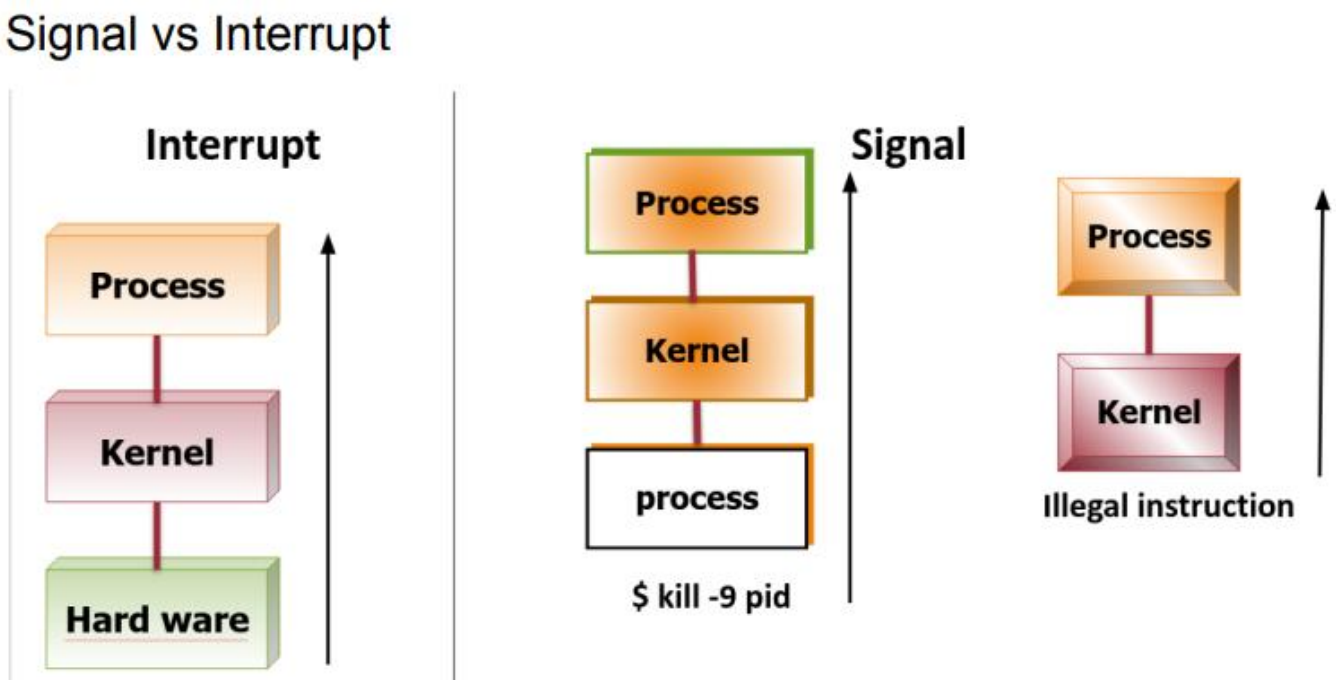
- All signals start with **SIGxxx** (like SIGINT, SIGKILL).
- They also have integer numbers (kill -l shows the list).
    - **1–32** → Standard signals (predefined behavior).
    - **33 onwards** → Real-time signals (SIGRTxxx, no default behavior).

Signal Handling

- Each signal has a **signal handler** → a small function that runs when the signal is received.
- Example: Default handler of SIGINT (Ctrl+C) is "terminate the program".
- You can **write your own handler** to override default.

⚡ Important points:

- If another signal comes while handling one → it waits until the first handler finishes.
- If process is in "uninterruptible sleep" (like waiting for I/O), the signal **waits** until process is ready.



Signals vs Interrupts

- **Interrupts** → from hardware (like keyboard press, disk ready).
  Kernel runs an **ISR (Interrupt Service Routine)**.
- **Signals** → from software/processes.
  Kernel runs the **signal handler**.

👉 So: **Interrupt = hardware → kernel → process**,
Signal = process → kernel → process.

Signal() System call

When does a process receive a signal?

1. **Process in Kernel Mode**
    - If the process is running in **kernel mode**, the signal **waits** until it switches back to **user mode**.
    - Reason: kernel data structures are **global** and critical. Interrupting them could cause **race conditions**.
2. **Process in Uninterruptible Sleep (D state)**
    - If the process is in an **uninterruptible sleep** (e.g., waiting on I/O), the signal **waits** until the process leaves that state.
    - Example: reading from disk, waiting for network response.
3. **All Other Cases**
    - The signal is delivered **immediately**.

Signal Handling

- Catching a signal is similar to **catching an exception** in programming:
    - By default, each signal has a **default handler**.
    - We can replace it with a **custom handler function**.
- Special handlers:
    - SIG_IGN → Ignore the signal.
    - SIG_DFL → Restore the default behavior for that signal.

Important Signals

- **SIGSTOP** → Suspends (pauses) the process.
    - Can be resumed with **SIGCONT** (typo in your note: should be SIGCONT, not SIGCNT).
- **SIGKILL** → Immediately kills the process (cannot be caught or ignored).
- **SIGINT** → Interrupts the process (usually from Ctrl+C).
    - Can be **caught or ignored**.
    - That's why SIGKILL is called the **sure-kill** signal.

sigaction() & kill() System Calls

sigaction()

Used to **examine** or **change** the action taken by a process on receipt of a specific signal.

Prototype

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Parameters

- signum → The signal number (e.g., SIGINT, SIGKILL, etc.).
- act → New action to be associated with signum.
- oldact → If not NULL, stores the old action.

sigaction structure

```
struct sigaction {
   void     (*sa_handler)(int);     // Pointer to signal handler (or SIG_IGN, SIG_DFL)
   void     (*sa_sigaction)(int, siginfo_t *, void *); // More advanced handler
   sigset_t sa_mask;               // Signals to block while handler runs
   int      sa_flags;              // Options (e.g., SA_SIGINFO, SA_RESTART)
};
```

Usage

- To install a **custom signal handler**.
- More powerful & portable than signal().
- Can specify extra behavior using flags.

👉 When you press Ctrl+C, instead of killing, it prints Caught signal 2.

---

kill()

Used to **send a signal** to a process or group of processes.

Prototype

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Parameters

- pid:
  - >0 → Signal sent to process with PID = pid.
  - =0 → Signal sent to all processes in same process group.
  - <-1 → Signal sent to all processes in the process group = |pid|.
  - -1 → Signal sent to **all processes** the user has permission to signal.
- sig:
  - The signal number (e.g., SIGKILL, SIGSTOP, SIGCONT).
  - 0 → Special case → no signal sent, just checks if process exists and if permission is allowed.

Usage

- To terminate, stop, resume, or send custom signals to a process.
- Commonly used by shell commands (kill -9 pid).

Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Custom signal handler
void my_handler(int sig) {
   printf("I got SIGINT number %d\n", sig);
   exit(0);
}

int main(void) {
   // Catching a signal with custom handler
   signal(SIGINT, my_handler);
   printf("Catching SIGINT\n");
   sleep(3);
   printf("No SIGINT within 3 seconds\n");

   // Ignoring a signal
```

```
    signal(SIGINT, SIG_IGN);  // <-- fixed ':' to ';'
    printf("Ignoring SIGINT\n");
    sleep(3);
    printf("No SIGINT within 3 seconds\n");

    // Calling default action for a signal
    signal(SIGINT, SIG_DFL);
    printf("Default action for SIGINT\n");
    sleep(3);
    printf("No SIGINT within 3 seconds\n");

    return 0;
}
```

Alarm and Timers

- **unsigned int alarm (unsigned int seconds);**
- **It is used to set an alarm for delivering SIGALARM signal.**
- **On success it returns zero.**

- **Three interval timers.**
  - **ITIMER_REAL**
    - This timer counts down in real (i.e., wall clock) time. At each expiration, a ***SIGALRM*** signal is generated.
  - **ITIMER_VIRTUAL**
    - This timer counts down against the user-mode CPU time consumed by the process. (The measurement includes CPU time consumed by all threads in the process.) At each expiration, a ***SIGVTALRM*** signal is generated.
  - **ITIMER_PROF**
    - This timer counts down against the total (i.e., both user and system) CPU time consumed by the process. At each expiration, a ***SIGPROF*** signal is generated.

The interval timers are used to determine how the time given to an alarm is to be interpreted, i.e. what time should be counted against the alarm.

ITIMER_REAL measures real-world time.
ITIMER_VIRTUAL only measures time taken to execute user-mode functions (no system calls) but doesn't include time taken for waiting.
ITIMER_PROF measures time taken to execute user-mode functions as well as kernel-mode functions (system calls) but doesn't include time taken for waiting for I/O.

In summary,
ITIMER_REAL = User-Mode (CPU + I/O) + Kernel-Mode time => SIGALRM
ITIMER_VIRTUAL = User-Mode (CPU) time => SIGVTALRM
ITIMER_PROF = User-Mode (CPU) + Kernel-Mode (CPU) time => SIGPROF

get and set timer

- get value of an interval timer
- int *getitimer* (int which, struct itimerval *val);
- On success it returns zero and the timer value is stored in the itimverval structure.
- Example: ret = getitimer (ITIMER_REAL, val);

- Set value for a interval timer
- int *setitimer* (int interval_timers, const struct itimerval *val, struct itimvferval *old_value);
- On success it returns zero.
- Example: ret = setitimer(ITIMER_REAL, &value, 0);

115

getitimer is used to return how much time is left for the timer to go off.

Resource Limits

- The OS imposes limits for certain system resources it can use.
- Applicable to a specific process.
- The “ulimit” shell built-in can be used to set/query the status.
- “ulimit –a” returns the user limit values

```
[root@localhost ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7892
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7892
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
[root@localhost ~]#
```

117

It is possible that depending on which user you are logged in as, you will get different outputs for “ulimit -a”. Like root vs regular user.

What is the core file mentioned in the output of “ulimit -a”?
It contains the core dump file which contains error logs. Only created for segmentation faults.
If core file size = 0 then that means core dump will not happen.
So that means we have to set it to something larger in order to have core dumps to happen.

08/10/21

Hard and Soft Limits

-c Maximum size of "core" files created.

-f Maximum size of the files created.

-l Maximum amount of memory that can be locked using mlock() system call.

-n Maximum number of open file descriptors.

-s Maximum stack size allowed per process.

-u Maximum number of processes available to a single user.

- Each resource has two limits –Hard and Soft
- Hard Limits
  - Absolute limit for a particular resource. It can be a fixed value or "unlimited"
  - Only superuser can set hard limit.
- "ulimit" command has –H or –S option to set hard/soft limits. Default is soft limit.
- Hard limit cannot be increased once it is set.

- Soft Limits
  - User-definable parameter for a particular resource.
  - Can have a value of 0 till <hard limit> value.
  - Any user can set soft limit.
- Limits are inherited (the new values are applicable to the descendent processes).

getrlimit() and setrlimit() System Calls

- getrlimit()/setrlimit() are system-call interfaces for getting and setting resource limits.
- Syntax
  - getrlimit(<resource>, &r)
  - setrlimit (<resource>, &r)
  - where r is of type "struct rlimit"

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
int prlimit(pid_t pid, int resource, const struct rlimit *new_limit, struct rlimit *old_limit);

struct rlimit {
        rlim_t rlim_cur;  /* Soft limit */
        rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
      };
```

Resource Usage: rusage struct

```
struct rusage {
        struct timeval ru_utime; /* user CPU time used */
        struct timeval ru_stime; /* system CPU time used */
        long   ru_maxrss;      /* maximum resident set size */
        long   ru_ixrss;       /* integral shared memory size */
        long   ru_idrss;       /* integral unshared data size */
        long   ru_isrss;       /* integral unshared stack size */
        long   ru_minflt;      /* page reclaims (soft page faults) */
        long   ru_majflt;      /* page faults (hard page faults) */
        long   ru_nswap;       /* swaps */
        long   ru_inblock;     /* block input operations */
        long   ru_oublock;     /* block output operations */
        long   ru_msgsnd;      /* IPC messages sent */
        long   ru_msgrcv;      /* IPC messages received */
        long   ru_nsignals;    /* signals received */
        long   ru_nvcsw;       /* voluntary context switches */
        long   ru_nivcsw;      /* involuntary context switches */
    };
```

```
int getrusage(int who, struct rusage *usage);
getrusage()  returns  resource usage
measures
for who, which can be one of the following:
RUSAGE_SELF
RUSAGE_CHILDREN
RUSAGE_THREAD
```

```
sysconf - get configuration information at
run time:

long sysconf(int name);

Example: ret= sysconf(_SC_CLK_TCK);

On success it returns the value of the given
system limits.
```

To know what you can give as input to sysconf() have a look at its man page.

Signals Continued: SIGSEGV

Consider the following program,

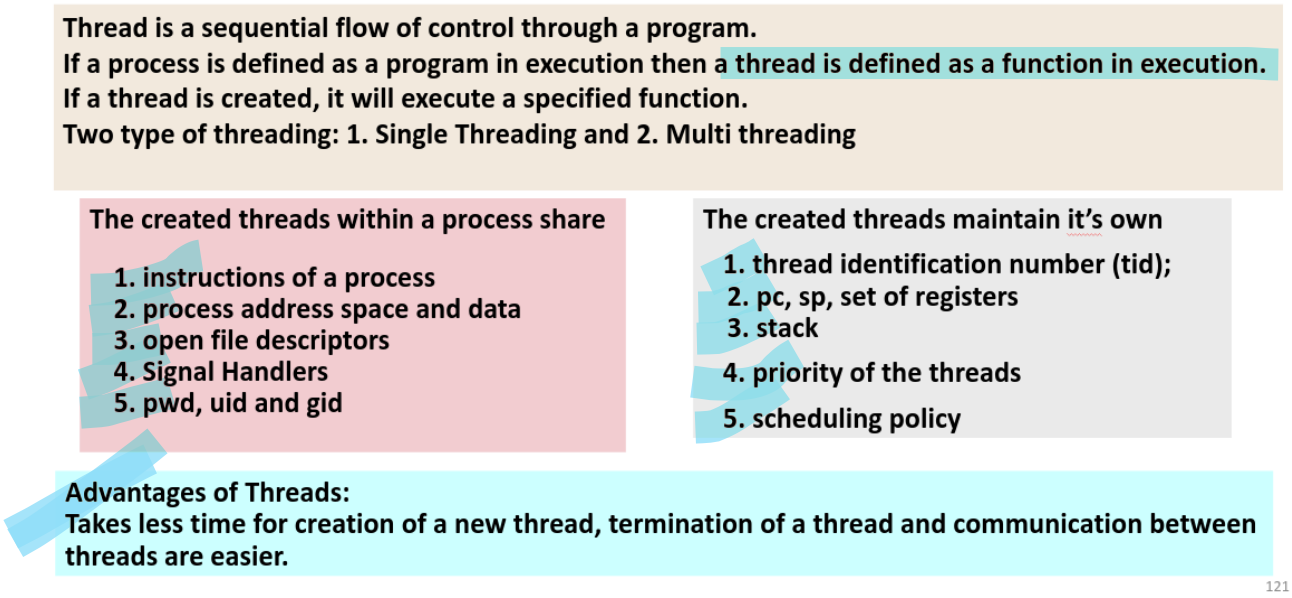
```c
int main() {
      int a;
      scanf("%d", a);
}
```

Does this program give an error?
Yes, Segmentation Fault (core dumped).
To get the core dump file (executable file) you have to set core file size to something greater than zero atleast using ulimit command.
We can debug the original program by using the generated core file with GDB.

When a segmentation fault occurs, a SIGSEGV (Segmentation Violation) signal is raised.

Note that we can also catch this signal using the signalling mechanism studied earlier.

Multithreading



[Single Thread => Just don't create any threads]

Communication between threads is easier than that of a process because threads of the same process share the same memory. So no inter-process communication involved.

The disadvantage of threads is that managing concurrency is difficult. For this we have to synchronize the threads for which we'll require POSIX Semaphores (not System-V Semaphores) which allows us to create unnamed semaphores which can be used by related processes like unnamed pipe.

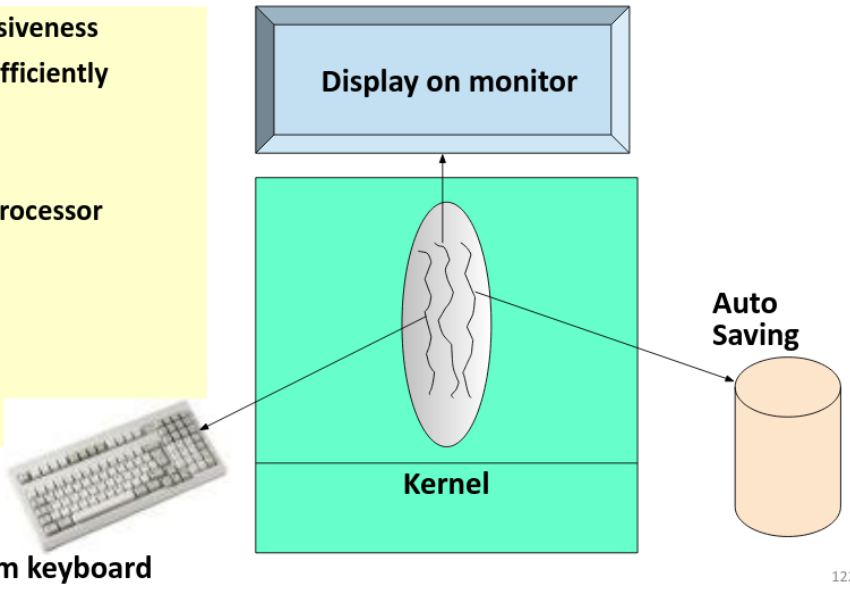
When we create threads, we can also specify the initial values for the parameters mentioned in the slide that are specific to each thread.
If we don't specify them, then the threads will inherit such values from the process that created them.

Advantages of Threads

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- use fewer system resources
- Specific applications in uniprocessor machines

Applications

- A file server on a LAN
- GUI
- web applications



It uses multiprocessors more efficiently because if we had multicore systems with no threading then it is possible that only one core will be in use at a time.
Improves program structure because it forces the code to be more modularized.
Specific applications in uniprocessor machines => Concurrent server

Note that the kernel treats each thread as a separate process; the only thing is that they will share the same address space and all the other properties mentioned before.

Thread Creation

```
#include <pthread.h>
void thread_func(void) {
    printf(" Thread id is %d", pthread_self());
}
main ( ) {
    pthread_t mythread;
    pthread_create ( &mythread, NULL, (void *) thread_func, NULL);
 }
```

This needs to be compiled as follows...$gcc pthread.c –lpthread

pthread_t is type-defined as unsigned long int. It takes the thread address as the first argument, the second argument is used to set the attributes for the thread-like stack size, scheduling policy, priority; if NULL is specified, then it takes default values for the attributes.

The third argument is the function that the thread should execute when created. The fourth argument is the argument for the thread function. If that function has a single argument to be passed, we can specify it here. If it has more than one argument, then we have to use a structure and declare all the arguments and pass the address of the structure.

pthread => p stands for POSIX
All of these are POSIX library functions and not system calls.

When you use the math library in a C program and try to compile it normally then most likely a warning message will be thrown. You resolve this by including a flag “-lm” while compiling to indicate that the math library is used. Similarly, when a pthread library is used, the “-lpthread” flag must be specified while compiling.
[lm = Link to math library, lpthread = Link to pthread library]

Sample Code:

```c
void *myThreadFunction(void *argvp) {
      printf("Printing HELLO WORLD from Thread\n");
      return NULL;
}
int main() {
      pthread_t tid;
      pthread_create(&tid, NULL, myThreadFunction, NULL);
      pthread_join(tid, NULL);
      exit(0);
}
```

The third argument of pthread_create() takes in the function name as the argument because the function name by itself is a pointer (like how an array is a pointer to the first element in the

array). Also note that the only argument given to the function has to be void*, even if you are not passing in any argument.
The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.
This function is useful in case there are some dependencies between the threads being created. So you can ensure that the threads on which the new thread is dependent on finish execution before it starts its own execution.
In this program, pthread_join() ensures that the thread terminates before main() itself terminates.

Summary:
pthread_create for threads => fork() for processes
pthread_join for threads => waitpid() for processes

If you use strace on the pthread program you can see that it uses the clone() system call in the backend which is similar to fork().

Memory Management

[Not much in terms of programming like system calls or library functions but there's a lot in terms of kernel programming => Not relevant for us right now atleast.
So just note the theory from this section]

Swap space is **a partition on the hard disk that is a substitute for physical memory**. It is used as **virtual memory** which contains process memory images.
We determine the size of the swap space and optimally it should have been twice the size of the RAM. Nowadays because of how large RAMs have become, swap space should be half the size of the RAM.

According to the kernel, **RAM + Swap Space = Actual RAM space**

Places to check memory size (including swap),
cat /proc/meminfo
free -m
vmstat 1 1 [1 to return just 1 entry, otherwise it will go on infinitely]
sudo slabtop [like top command but for memory]

Virtual Memory

- **Memory management: one of the most important kernel subsystems**
- **Virtual Memory: Programmer need not to worry about the size of RAM (Large address space)**
- **Static allocation: internal Fragmentation**
- **Dynamic allocation: External Fragmentation**
- **Avoid Fragmentation: Thrashing -overhead**

- **Large address space: virtual memory is many times larger than the physical memory in a system.**
- **For a 32 bit OS, the virtual memory size will be 2 to the power of 32 i.e 4GB. But the RAM size may be much smaller.**
- **Each process has a separate virtual address space.**
- **Each process space is protected from other processes.**
- **It supports shared virtual memory, i.e more than one process can share a shared page.**
- **Uses paging technique.**

130

Programmers need not worry about the size of RAM

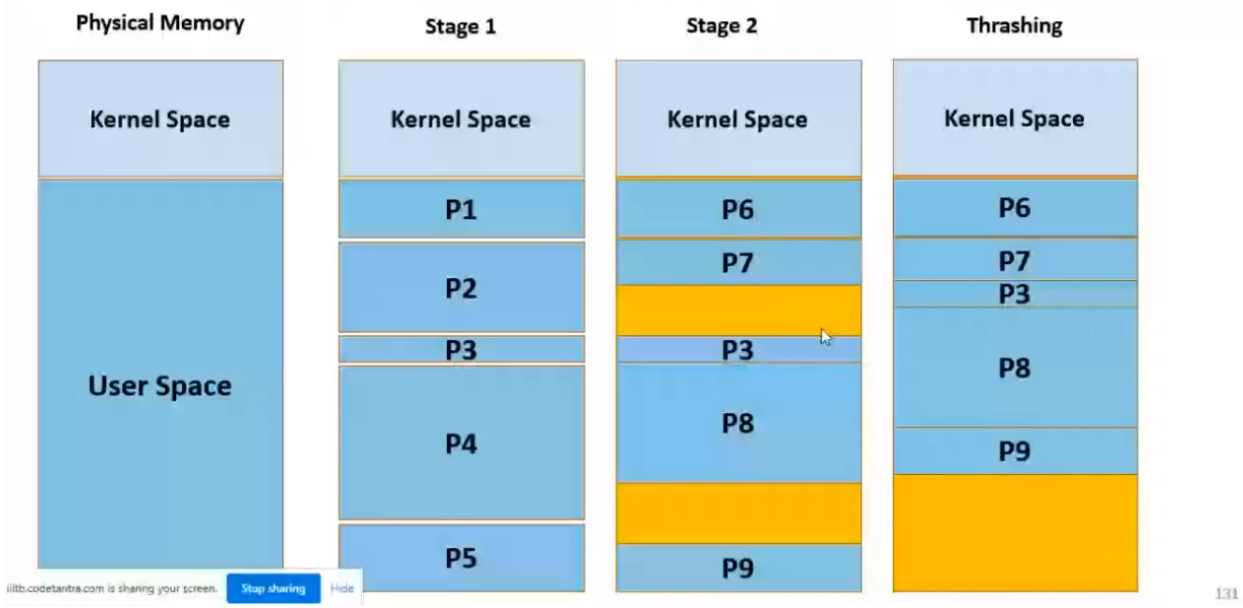
- Because the concept of demand paging is used. The process is split into pages of disk block size (4KiB typically) and pages will be brought into main memory only when they are required (demanded), i.e. a page fault happens for that page.

Internal Fragmentation happens with static allocation

- Suppose page size = 4KiB and your program size is 6KiB, then your program shall occupy 2 pages, one fully filled and one in which 2KiB is free.
- This 2KiB is wasted space and this occurrence is called internal fragmentation as wastage happens within a page/block.

However with small page sizes like 4KiB even internal fragmentation is minimal and this is a satisfactory enough solution.

Dynamic Allocation



In this example, no page sizes exist. However once P1 occupies that space, a page of that size has been created and once P1 leaves, new pages that are stored in that page frame must have the same size or less than it.

Little by little this will create small sized empty slots that will remain unused. And that's why this is called external fragmentation.

The solution to this is that in periodic intervals of time, a function must execute that pushes all of the externally fragmented blocks at the end of the user space. After this the entire fragmented space can be used as a single block again which makes it much less susceptible to being externally fragmented.

Page Table

- Hardware support (MMU, TLB) is required.
- Fair share allocation
- static allocation
- Minimize internal fragmentation
- identified by a PFN (Page Frame Number)
- virtual address is split into two parts namely an offset and a virtual page frame number.

- Translate a process virtual address into physical address since processor use only virtual address space
- The size of a page table is normally size of a page
- if it is 4kb, each page address size is 4byte, so 1024 page entries in a page table.
- holds info about
  - Whether valid page table or not?
  - PFN
  - access control information
- Stored in TLB (Translation Look-aside Buffer)

Fair-share allocation => Static allocation and it minimizes internal fragmentation because only the last page of a process will cause small amounts of internal fragmentation, all of the other pages will fully occupy their respective page frames.

You can link this page allocation and address translation concept to the single-indirect, double-indirect, etc. addressing discussed on the ext file system.

Memory Mapping

- Executable image split into equal sized small parts (normally the page size)
- Virtual memory assigns virtual address to each part
- Linking of an executable image into a process virtual memory

Swapping

- Swap space is in hard disk partition
- If a page is waiting for certain event to occur, swap it.
- Use physical memory space efficiently
- If there is no space in Physical memory, swap LRU pages into swap space

- Demand Paging - don't load all the pages of a process into memory
- Load only necessary pages initially
- if a required page is not found, generate page fault then the page fault handler brings the corresponding page into memory.

Kernel Data Structure

[Not that important right now]

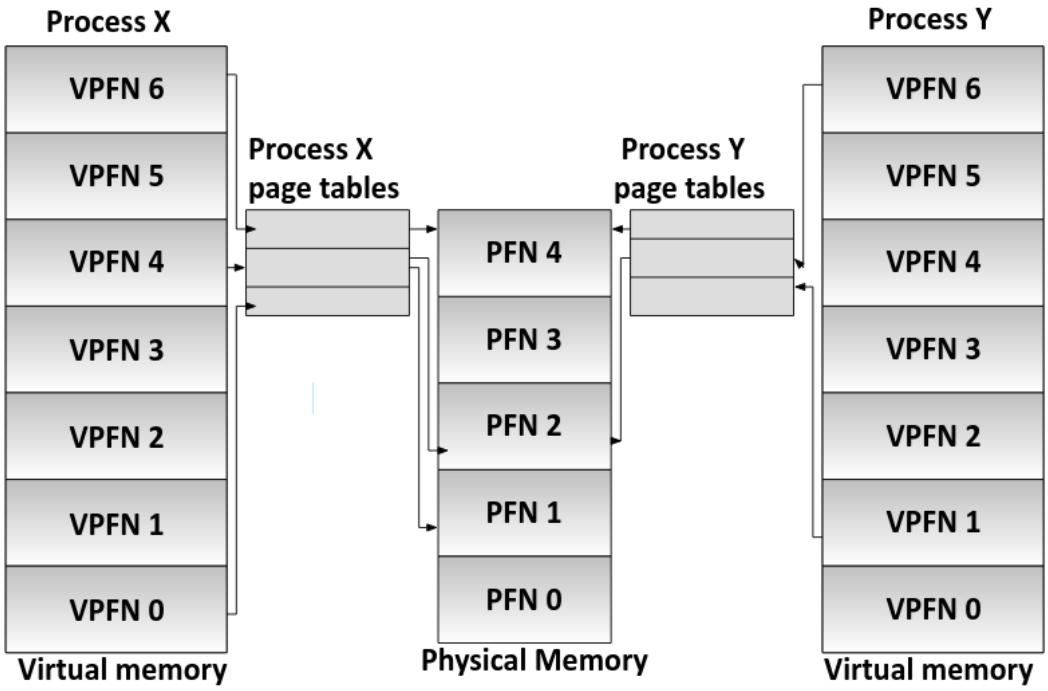
- **Source Code: /usr/src/linux-4.12/mm**
  /usr/src/linux-4.12/include/linux
- **virtual memory is represented by an mm_struct data structure**
- **it has pointers to vm_area_struct data structure**
  - **created when an executable image is mapped with the process virtual address**
  - **has starting and end points of virtual memory**
  - **represents a process's image like text, data and stack portion**
  - **has control access info**

mm_types.h

```
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    ……
```

```
struct vm_area_struct {
    /* The first cache line has the info for VMA tree walking. */
    unsigned long vm_start;
    unsigned long vm_end;
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;
    ……
```

Virtual to Physical Memory Translation

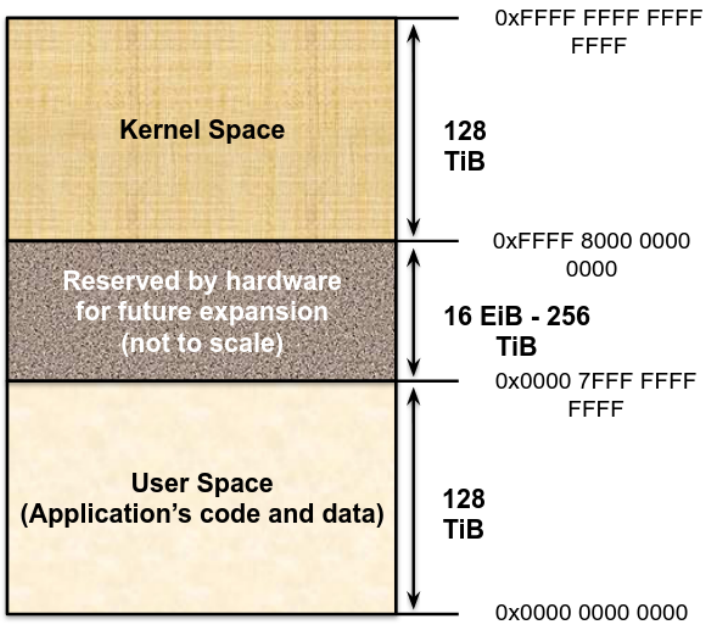


12/10/21

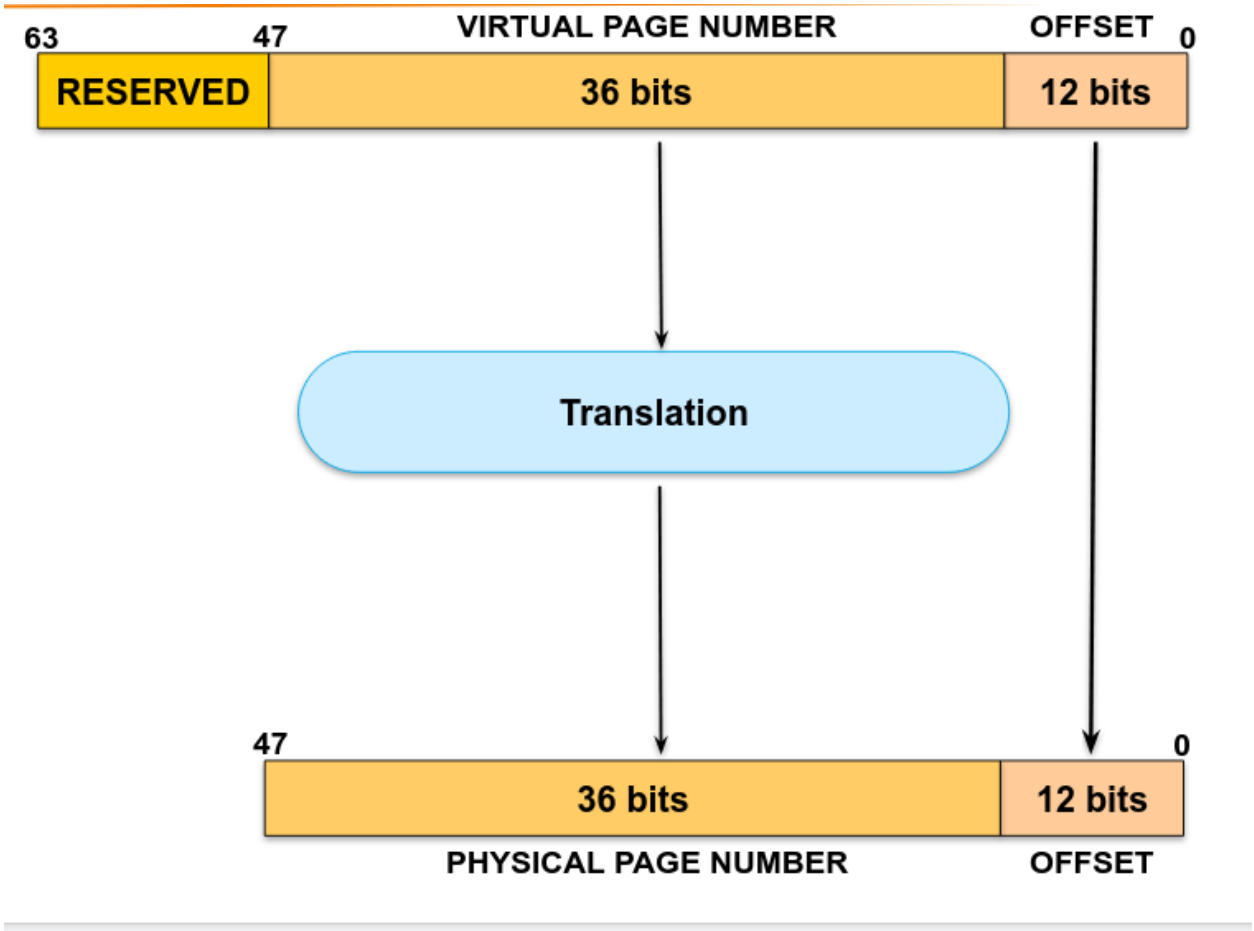
x86-64 Virtual Memory Layout

$pmap –x <pid>

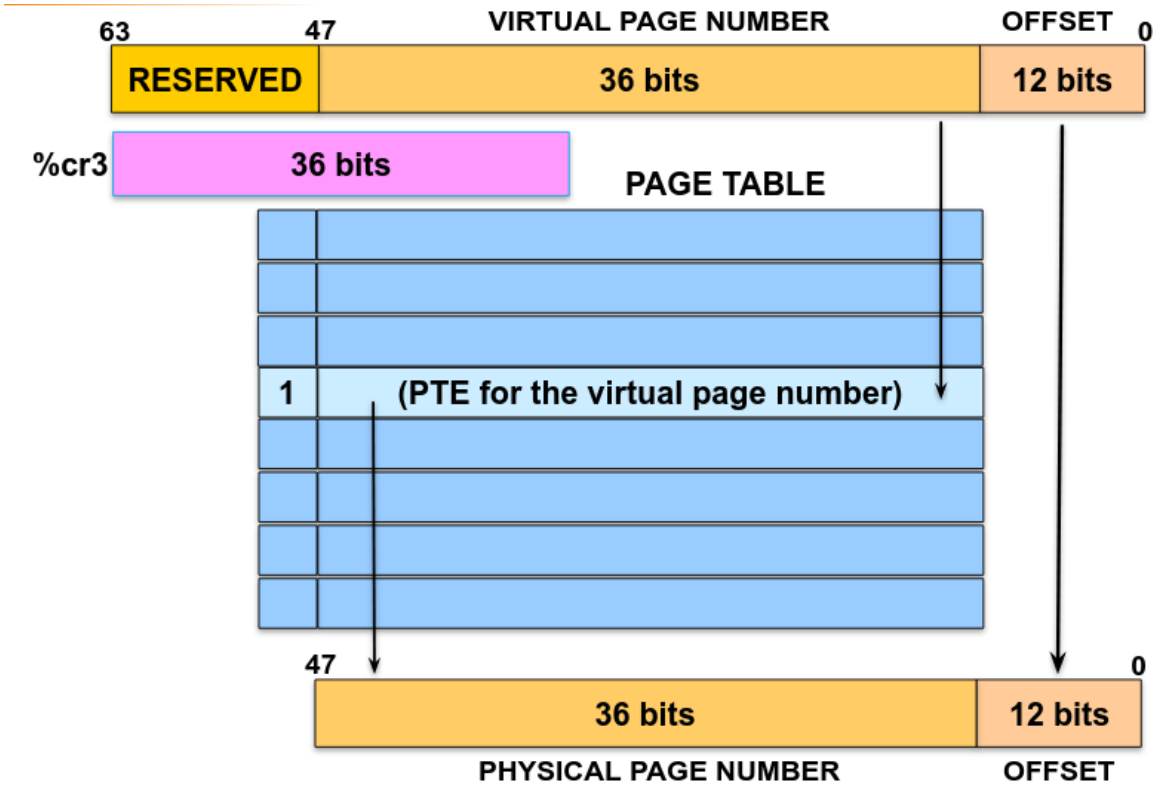
Prefix	Name			
Kibi	binary kilo	1 kibibyte (KiB)	2^{10} bytes	1024 B
Mebi	binary mega	1 Mebibyte (MiB)	2^{20} bytes	1024 KiB
Gibi	binary giga	1 Gibibyte (GiB)	2^{30} bytes	1024 MiB
Tebi	binary tera	1 Tebibyte (TiB)	2^{40} bytes	1024 GiB
Pebi	binary peta	1 Pebibyte (PiB)	2^{50} bytes	1024 TiB
Exbi	binary exa	1 Exbibyte (EiB)	2^{60} bytes	1024 PiB



Virtual to Physical Address Translation

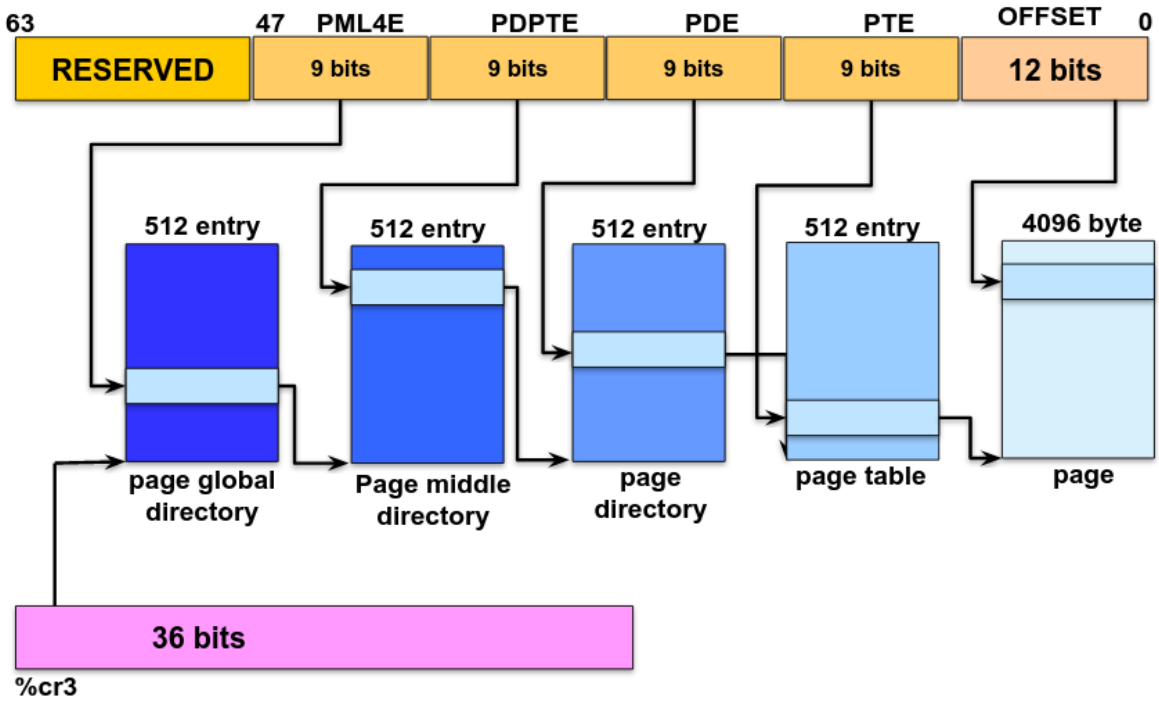


More details on what is happening here is given in the following slide,



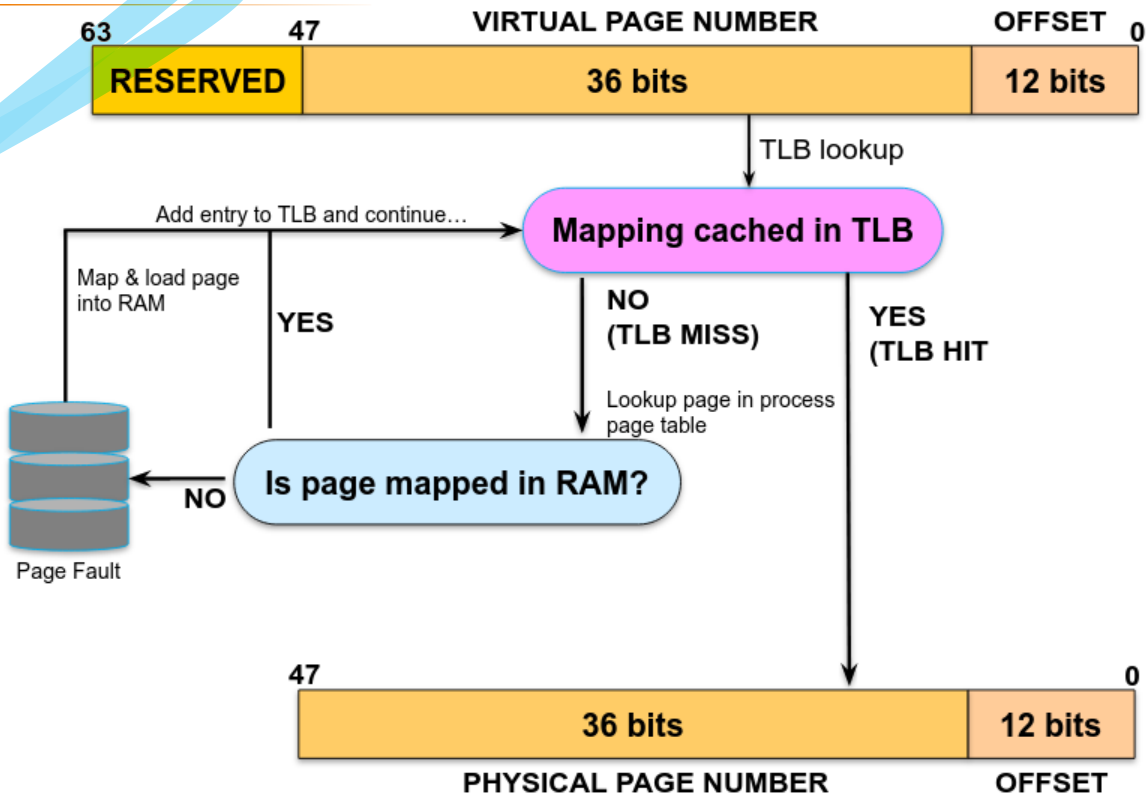
%cr3 = Translation Look-aside Buffer (TLB)
So Virtual Page Number points to the Page Table entry (PTE) which contains the Physical Page Number.
The Page Table is stored in the TLB and is responsible for translation from virtual to physical page number.

Page Table Hierarchy on x86-64



This shows you how multi-level indexing works in Page Table address translation.
Again, this is quite similar to the ext4 file system with its double and triple indirect addressing format.

Paging & TLB



Mapping cached in TLB = Page Table
“Is Page Mapped to RAM” is asking whether the page is present in swap space or not.
This happens when a page remains inactive for some time. It is moved from main memory to swap space.
If the page is not even present in swap space then page fault occurs and by principle of demand paging it is brought into main memory and an entry for it is made in the page table which is present in TLB.

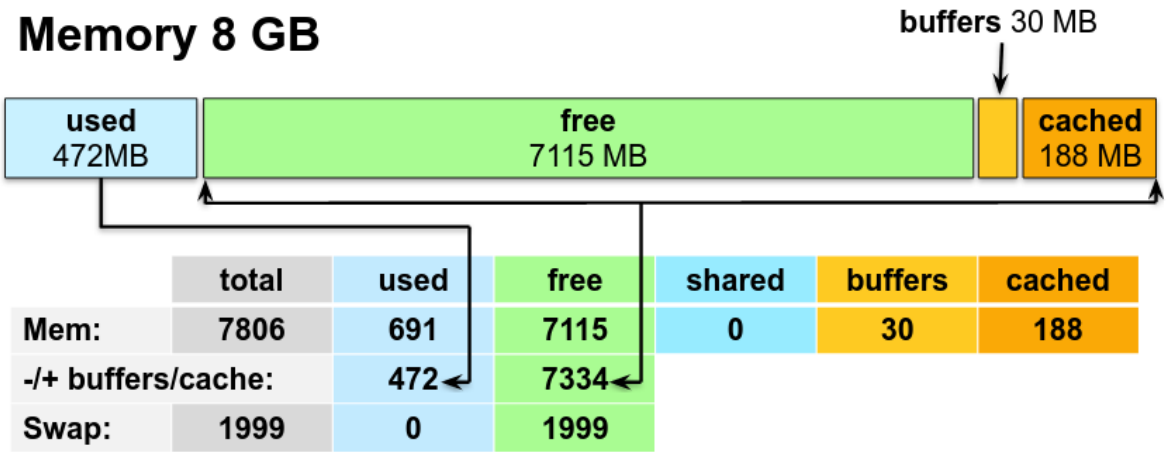
Major & Minor Page Faults

The page fault where page entry is not present in TLB but is present in swap space is called minor page fault.
While the page fault where a page is not even present in swap space but instead has to be fetched from the hard disk is called the major page fault.

Sidenote: SAR - System Activity Report

Prints system logs about all the activity that has happened since boot.
This is really in depth and gives a huge output file.
This is useful because this can be given to system monitoring programs as input and then we can monitor our system with a higher degree of control.

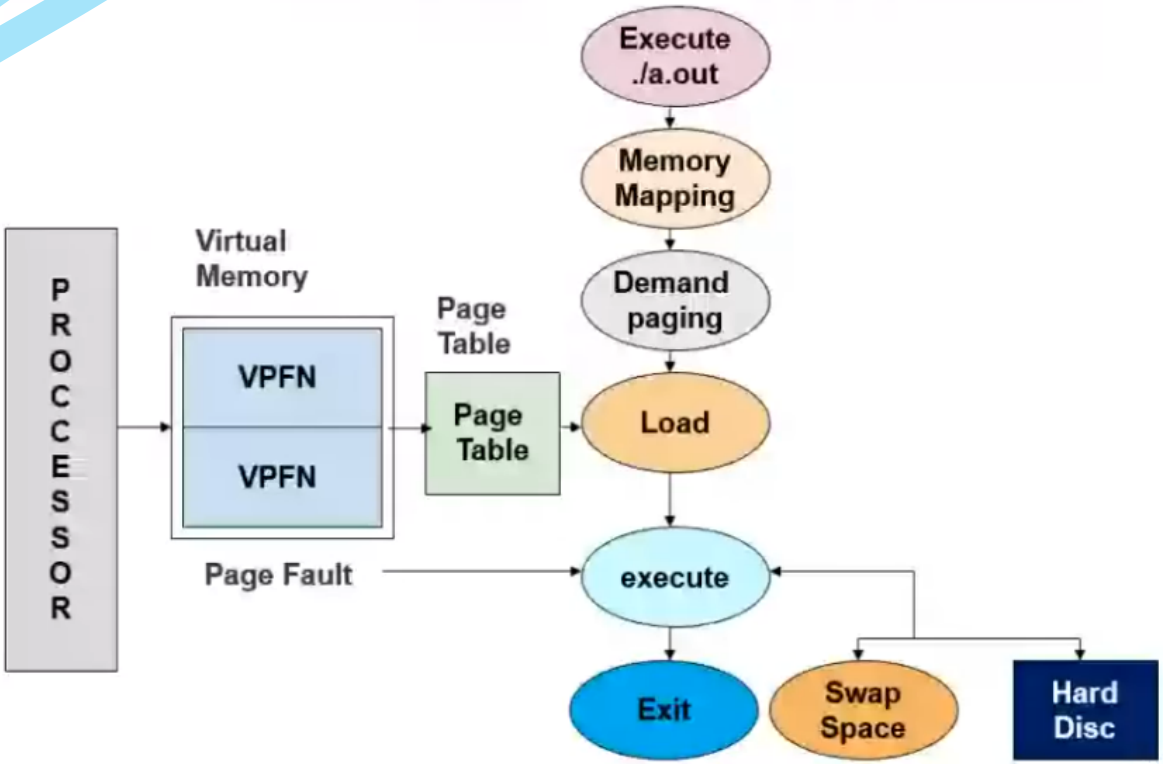
free -m



	total	used	free	shared	buffers	cached
Mem:	7806	691	7115	0	30	188
-/+ buffers/cache:		472	7334			
Swap:	1999	0	1999			

Mem: used = 472 + 30 + 188 = 690

Walk Through Program Execution



When you execute a program “a.out”, it is divided into chunks of size, PAGE_SIZE (4KiB). One virtual address is assigned for each chunk/page and this is the memory mapping part. Then the demand paging concept is used to load only the first page into main memory.

Loading a page involves translating the virtual address of the page (Virtual Page Frame Number, VPFN) into a physical address which requires the Page Table (TLB).
After which the page starts to execute.
Once it finishes execution, the second page has to be loaded into main memory for which it will generate a page fault and the page fault handler will load this into main memory.
The page that has to be loaded after a page fault can either be in swap space (minor page fault) or in the hard disk (major page fault).
The pages will keep on loading and executing till the program exits.