🧠 1. Basic OS Concepts

✅ What is an Operating System?

🔹 Definition

An **Operating System (OS)** is a system software that acts as an **interface between the user and the computer hardware**. It manages hardware resources and provides a set of services for computer programs, enabling efficient execution of software and effective hardware utilization.

It hides the complexities of the underlying hardware and provides a user-friendly environment for execution.

🔹 Key Points

- Controls and coordinates hardware usage among various application programs.
- Provides a stable and consistent way for applications to deal with hardware.
- Acts as an intermediary between users and hardware.

🔹 Core Components

- **Kernel**: Core of the OS responsible for low-level tasks like scheduling, memory management, etc.
- **Shell**: Interface through which users interact with the OS (e.g., command-line shell).
- **File System**: Organizes and stores data.
- **System Utilities**: Tools for managing the system.

✅ OS Roles and Responsibilities

🔹 1. Process Management

- Creating, scheduling, and terminating processes.
- Ensures that CPU time is shared fairly and efficiently among active processes.
- Handles context switching and inter-process communication (IPC).

🔹 2. Memory Management

- Allocates and deallocates memory space as needed.
- Keeps track of each byte of memory in the system.
- Provides virtual memory abstraction.

🔹 3. File System Management

- Organizes data in directories and files.
- Controls permissions and access rights.

- Manages storage devices and provides file I/O APIs.

🔹 4. Device Management

- Manages I/O devices via device drivers.
- Handles buffering, caching, and spooling.
- Provides a uniform interface for hardware interaction.

🔹 5. Security and Protection

- Enforces access control policies to protect data and resources.
- Prevents unauthorized access and malware threats.
- Manages user authentication and file permissions.

🔹 6. User Interface Management

- Provides Command Line Interface (CLI) or Graphical User Interface (GUI).
- Ensures usability and responsiveness for the end user.

🔹 7. Resource Allocation

- Manages hardware resources (CPU, memory, disk, etc.).
- Allocates resources to users and programs as needed.

🔹 8. Error Detection and Handling

- Detects hardware and software failures.
- Logs errors and attempts to recover from them gracefully.

🔹 9. Networking

- Supports communication over local and global networks.
- Implements networking protocols and stack layers.

✅ Real-World Analogy

Think of the OS as a **hotel manager**:

- Rooms = Memory
- Guests = Processes
- Keycards = Access permissions
- Staff = Kernel subsystems
- Front desk = User interface
- Manager = OS coordinating everything

✅ Code Example (Simple System Call)

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

```
int main() {
    write(1, "Hello from OS\\n", 15);  // File descriptor 1 = stdout
    return 0;
}
```

🔸 Output:

```
Hello from OS
```

✅ Diagram: OS as a Layered System

```
 +-----------------------------+
|      User Applications       |
+------------------------------+
|         System Calls         |
+------------------------------+
|          Operating System    |
|  - Process Management        |
|  - Memory Management         |
|  - File System               |
|  - Device Drivers            |
|  - Security                  |
+------------------------------+
|          Hardware            |
+------------------------------+
```

✅ Real-World Q&A

Q: Why can’t we run applications directly on hardware without an OS?
A: Because apps need services like memory allocation, CPU scheduling, file access, and I/O control — which the OS provides. Direct access to hardware would be inefficient, uncoordinated, and prone to conflict or failure.

Q: What happens when the computer is powered on?
A: BIOS/UEFI → Bootloader → Kernel loaded → OS initializes system → Shell/GUI started → User interface is presented.

✅ Types of Operating Systems

🔸 1. Batch Operating System

Definition:
A Batch OS executes batches of jobs with **no user interaction**. Users submit jobs to an operator who batches them together and runs them in sequence.

Characteristics:

- Jobs are grouped and processed in the order of arrival.
- No direct user input during execution.
- Common in early computing systems.

Real-World Example:
IBM OS/360

Advantages:

- Good for large computations and repetitive tasks.
- Maximizes throughput by reducing idle CPU time.

Disadvantages:

- No real-time interaction.
- Difficult error handling due to lack of user intervention.

🔸 2. Time-Sharing Operating System

Definition:
Time-sharing (or multitasking) OS allows **multiple users** to access the system **simultaneously** by giving each user a time slice of the CPU.

Characteristics:

- Rapid context switching.
- User gets impression of exclusive control.
- Uses scheduling algorithms like Round Robin.

Real-World Example:
Unix, Multics

Advantages:

- Interactive and responsive.
- Efficient resource sharing.

Disadvantages:

- Higher complexity and overhead.
- Security concerns due to multi-user environment.

Diagram:

```
 +------------+     +------------+     +------------+
| User 1     |     | User 2     |     | User 3     |
+------------+     +------------+     +------------+
       \                |                /
        \               |               /
         ---------> Time-Sharing OS <---------
```

🔸 3. Distributed Operating System

Definition:
A Distributed OS manages a **group of independent computers** and makes them appear as a **single system** to users.

Characteristics:

- Tasks are distributed among multiple machines.

- Transparency in access and location of resources.
- Requires network communication and synchronization.

Real-World Example:
Amoeba, Plan 9, Microsoft Azure, Google Borg

Advantages:

- Fault tolerance.
- Scalability.
- Resource sharing across systems.

Disadvantages:

- Complex design and synchronization.
- Dependency on reliable networking.

🔸 4. Real-Time Operating System (RTOS)

Definition:
A RTOS responds to inputs **within a guaranteed time**. It's used where timing is critical.

Types:

- **Hard RTOS** – Strict deadlines (e.g., flight control systems).
- **Soft RTOS** – Deadline is important but not fatal (e.g., video streaming).

Characteristics:

- Deterministic behavior.
- Prioritized task execution.
- Minimal interrupt latency.

Real-World Example:
VxWorks, FreeRTOS, QNX

Advantages:

- Predictable response.
- Efficient use of resources.

Disadvantages:

- Limited multitasking.
- Difficult to develop and test.

🔸 5. Network Operating System

Definition:
A Network OS enables **resource sharing** (files, printers, etc.) between computers connected via a network.

Characteristics:

- Requires user login to access shared resources.
- Runs on a central server.

- Common in client-server architectures.

Real-World Example:
Windows Server, Novell NetWare, UNIX

Advantages:

- Centralized security and administration.
- Easy file and printer sharing.

Disadvantages:

- Server dependency.
- Expensive setup and maintenance.

🔸 6. Multiprogramming Operating System

Definition:
Allows **multiple programs to reside in memory** and execute concurrently by utilizing idle CPU cycles.

Characteristics:

- Increases CPU utilization.
- Context switching between programs.

Real-World Example:
Early UNIX, IBM systems

Advantages:

- Efficient resource usage.
- Improves system throughput.

Disadvantages:

- No user interaction.
- Poor responsiveness to external input.

Diagram:

```
 Memory:
+-----------+-----------+-----------+
| Program A | Program B | Program C |
+-----------+-----------+-----------+
         |
         v
      CPU executes one at a time using scheduling
```

🔸 7. Multiprocessing Operating System

Definition:
Supports **multiple CPUs** working in parallel to execute different tasks.

Types:

- **Symmetric Multiprocessing (SMP)** – All CPUs share the same memory and I/O.
- **Asymmetric Multiprocessing (AMP)** – One CPU is master; others follow.

Real-World Example:
Linux, Windows with multi-core processors

Advantages:

- Increases performance and throughput.
- Reliability: if one CPU fails, others continue.

Disadvantages:

- Increased complexity.
- Expensive hardware requirements.

🔸 8. Multithreading Operating System

Definition:
Allows a single process to have **multiple execution threads** that run independently but share the same memory.

Characteristics:

- Lightweight context switching.
- Shared address space.
- Thread-level parallelism.

Real-World Example:
Java-based servers, modern Linux, macOS

Advantages:

- Efficient CPU usage.
- Faster context switching.

Disadvantages:

- Risk of race conditions.
- Needs synchronization.

Code Snippet (POSIX Thread Example):

```
 #include <pthread.h>
#include <stdio.h>

void* thread_func(void* arg) {
    printf(\"Hello from thread!\\n\");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread_func, NULL);
    pthread_join(tid, NULL);
    return 0;
}
```

🔸 9. Mobile Operating System

Definition:
An OS optimized for **smartphones and tablets** with touch interfaces and wireless communication.

Characteristics:

- Energy-efficient.
- App-centric.
- Security sandboxing.

Real-World Example:
Android, iOS, KaiOS

Advantages:

- Lightweight and responsive.
- Built-in support for GPS, sensors, etc.

Disadvantages:

- Limited multitasking.
- Fragmentation (esp. in Android).

✅ Summary Table

OS Type	Key Use Case	Example
Batch	Offline large jobs	IBM OS/360
Time-Sharing	Multi-user interactivity	UNIX, Multics
Distributed	Clustered environments	Plan 9, Amoeba
RTOS	Embedded, safety-critical apps	QNX, FreeRTOS
Networked	Centralized resource sharing	Windows Server
Multiprogramming	Efficient CPU use	Early UNIX
Multiprocessing	Multi-core execution	Linux, Windows
Multithreading	Parallelism within process	Java apps, Linux
Mobile	Smartphones	Android, iOS

✅ Kernel Types and System Calls

🔹 Kernel Types

The **kernel** is the core component of an operating system. It manages CPU, memory, I/O devices, and system calls. Different architectures define how the kernel interacts with system services and hardware.

🔸 1. Monolithic Kernel

Definition:
All OS components run in **kernel space** as part of a single large process.

Characteristics:

- Fast due to direct communication between modules.
- Poor fault isolation (a bug can crash the whole system).

Examples:

- Linux, UNIX, MS-DOS

Advantages:

- High performance
- Easy access to services

Disadvantages:

- Less modularity
- Harder to maintain and debug

Diagram:

```
 +---------------------------+
|        User Space         |
+---------------------------+
|        System Calls       |
+---------------------------+
|  File Sys | Scheduler | MM |
|  Drivers  | IPC       |    | <-- All in Kernel
+---------------------------+
|        Hardware           |
+---------------------------+
```

🔸 2. Microkernel

Definition:
Only the essential components (e.g., scheduling, IPC) are run in **kernel space**; other services (drivers, FS, etc.) run in **user space**.

Characteristics:

- Uses message passing for communication.
- Better fault isolation and modularity.

Examples:

- Minix, QNX, L4

Advantages:

- Stability and security
- Easier to extend or modify

Disadvantages:

- Performance overhead due to IPC

Diagram:

```
 +---------------------------+
|      User Applications     |
+---------------------------+
| Filesys | Drivers | Others |
+---------------------------+
|      Microkernel (IPC, MM, Scheduler)    |
+---------------------------+
|          Hardware          |
+---------------------------+
```

🔸 3. **Hybrid Kernel**

Definition:
Combines features of both **monolithic** and **microkernels** — runs some services in kernel mode and some in user mode.

Characteristics:

- Optimized performance with better modularity than monolithic kernels.

Examples:

- Windows NT, macOS (XNU)

Advantages:

- Balanced design
- Supports modular drivers

Disadvantages:

- Still complex
- Not as cleanly separated as pure microkernels

🔸 4. **Exokernel**

Definition:
An extremely minimal kernel that **exposes hardware resources directly to applications** with minimal abstraction.

Characteristics:

- Application-level libraries manage resources.
- Focuses on **efficiency and customizability**.

Examples:

- MIT's Exokernel

Advantages:

- Maximum performance
- Applications control hardware directly

Disadvantages:

- Complex app design
- Security risks due to low abstraction

Diagram:

```
 +---------------------------+
|        User Libraries      |
+---------------------------+
|        Exokernel           |
+---------------------------+
|        Hardware            |
+---------------------------+
```

🔷 System Calls and APIs

🔶 What is a System Call?

Definition:
A system call is a **programmatic way** in which a user-space program requests a **service from the kernel**.

System calls provide a controlled interface to interact with **hardware, files, processes, memory**, etc.

🔶 Categories of System Calls

- Process Control (fork, exec, exit, wait)
- File Management (open, read, write, close)
- Device Management (ioctl, read, write)
- Information Maintenance (getpid, time)
- Communication (pipe, shmget, send, recv)

🔶 Code Example: `write()` syscall

```
#include <unistd.h>

int main() {
    write(1, "Hello, Kernel!\\n", 15);  // 1 = stdout
    return 0;
}
```

Output:

```
Hello, Kernel!
```

🔸 User Mode vs Kernel Mode

Mode	Access Level	Purpose
User Mode	Restricted	Running user apps (low privilege)
Kernel Mode	Full Hardware Access	Runs OS core services (high privilege)

Context Switch:
Occurs when control is transferred from user mode to kernel mode (e.g., during a system call or interrupt).

🔸 Trap Instructions

Definition:
A **trap** is a software-generated interrupt that **switches execution from user mode to kernel mode**.

Usage:

- Executing a system call
- Handling exceptions (e.g., divide by zero)

Flow:

1. User app invokes syscall → trap instruction issued
2. CPU switches to kernel mode
3. Jumps to syscall handler in kernel
4. Executes service
5. Returns to user mode

Diagram:

```
 User Process
     |
     v
[System Call]
     |
     v
[Trap Instruction]
     |
     v
Switch to Kernel Mode
     |
     v
[Syscall Handler]
     |
     v
Return to User Mode
```

✅ Real-World Q&A

Q: Why use system calls instead of direct hardware access?
A: To ensure **security, stability, and abstraction**. Direct hardware access can lead to conflicts,

corruption, or crashes.

Q: What if a user app misbehaves in kernel mode?
A: Kernel mode has full control; any bugs can lead to system crashes. That's why system calls are carefully designed with **input validation and isolation**.

Q: Why aren't all services in user mode (like in microkernels)?
A: Because crossing the user-kernel boundary repeatedly (via IPC) introduces **performance overhead**.

🧠 2. Processes and Threads

✅ Process Concepts

🔸 Process vs Program

Program:

- A **static** set of instructions stored on disk.
- Passive entity.
- Example: A `.exe`, `.out`, or `.sh` file.

Process:

- A **dynamic** instance of a program in execution.
- Includes program counter, registers, stack, heap, etc.
- Actively utilizes CPU and memory.

Analogy:

- A **recipe** is a program.
- **Cooking** using that recipe is a process.

Feature	Program	Process
State	Static (on disk)	Dynamic (in memory)
Execution	Not executing	Executing
Lifespan	Permanent file	Temporary (until complete)
Example	`ls` binary	`ls` running in terminal

🔸 Process Lifecycle

A process goes through multiple states from creation to termination.

States:

1. **New** – Process is being created.
2. **Ready** – Waiting for CPU.
3. **Running** – Instructions are being executed.

4. **Waiting** – Waiting for I/O or event.
5. **Terminated** – Finished execution.
6. **Suspended** (optional) – Paused by OS or admin.

Diagram:

```
 +---------+
 |  New    |
 +---------+
      |
    admit
      v
 +---------+
 | Ready   |<------------------+
 +---------+                   |
      |                        ^
  dispatch                I/O complete
      v                        |
 +---------+              +---------+
 |Running | ------------>|Waiting |
 +---------+   I/O req    +---------+
      |                        ^
    exit                       |
      v                        |
 +------------+               |
 | Terminated | <------------+
 +------------+
```

🔸 Process Control Block (PCB)

Definition: A **PCB** is a data structure maintained by the operating system for every process. It contains all the **information required to manage and track** a process.

Key Fields in PCB:

- **Process ID (PID)** – Unique identifier.
- **Process State** – Ready, running, etc.
- **Program Counter** – Next instruction to execute.
- **CPU Registers** – Register snapshot.
- **Memory Limits** – Memory allocated to process.
- **Open File Descriptors** – Files the process has opened.
- **Priority** – Scheduling priority.
- **Parent Process** – Reference to parent PID.

Diagram:

```
+------------------------------+
|       Process Control Block |
+------------------------------+
| PID = 1042                  |
| State = Running             |
| PC = 0x0021AF               |
| Registers = {...}           |
| Memory Limit = 0x7fffffff   |
| Open Files = [fd0, fd1]     |
| Priority = 5                |
| Parent PID = 1041           |
+------------------------------+
```

Why is PCB Important?

- During context switching, PCB stores the current state.
- Enables process resumption without loss of state.
- Maintains isolation between processes.

✅ Real-World Q&A

Q: Why can't two processes share the same PCB?
A: Because the PCB uniquely identifies and maintains the state of **each process separately**. Sharing would cause race conditions and inconsistencies.

Q: How is PCB used during context switch?
A: OS saves the CPU state of the current process to its PCB and loads the state of the next process from its PCB into the CPU registers.

Q: Is PCB stored in user space?
A: No. PCB is maintained by the OS in **kernel space**, inaccessible to user processes.

Would you like to continue with:

- Threads
- Context Switching
- Fork/exec/wait
- Process vs Thread comparison?

✅ Thread Concepts

🔸 Thread vs Process

Process:
A self-contained unit with its own address space, code, data, stack, and system resources. Heavyweight to create and manage.

Thread:
A **lightweight subprocess** — smallest unit of CPU execution within a process. Multiple threads share the **same address space** and resources.

✅ Comparison Table

Feature	Process	Thread
Address Space	Own memory	Shared with other threads
Control Block	PCB	TCB (Thread Control Block)
Creation Overhead	High	Low
Context Switch Cost	High (different memory space)	Low (same memory space)
Communication	Inter-Process Communication	Shared memory (direct)
Crash Effect	One process crash is isolated	One thread crash can affect all

Feature	Process	Thread
Examples	Chrome tabs	Java threads in JVM

✅ Real-World Analogy

- **Process**: A house with its own walls (memory), rooms (code/data), and residents.
- **Thread**: People (threads) living in the same house (process) and sharing the kitchen, electricity, etc.

🔸 Thread Benefits

- Faster context switching
- Efficient CPU utilization on multi-core systems
- Easier inter-thread communication
- Useful for parallel tasks (e.g., web server handling multiple clients)

🔷 Types of Threads

🔸 1. User-Level Threads (ULT)

Definition:
Threads that are **managed entirely in user space**, without kernel support. The OS is unaware of the presence of multiple threads.

Characteristics:

- Lightweight and fast to create.
- Managed by a user-level thread library (e.g., POSIX threads, Java threads).
- Blocking one thread blocks all threads in the process.

Advantages:

- No kernel mode switch required.
- Custom scheduling strategies possible.

Disadvantages:

- If one thread makes a blocking system call, all threads are blocked.
- No true parallelism on multi-core systems.

Diagram:

```
 +-------------------------+
|     User Application    |
+-------------------------+
|  User-Level Thread Lib  |
+-------------------------+
|     Single OS Thread    |
|      (Kernel View)      |
+-------------------------+
```

🔸 2. Kernel-Level Threads (KLT)

Definition:
Threads that are **fully managed by the OS kernel**. Each thread is known and scheduled by the kernel.

Characteristics:

- True parallel execution on multi-core processors.
- OS handles context switching and scheduling.

Advantages:

- If one thread blocks, others can continue.
- Utilizes multiprocessor systems efficiently.

Disadvantages:

- Higher overhead (syscalls, kernel involvement).
- Slower than ULT in simple tasks.

Examples:
Linux, Windows, macOS use kernel-level threads.

Diagram:

```
+-------------------------+
|   User Application      |
+-------------------------+
|  Multiple Kernel Threads |
+-------------------------+
|     OS Kernel Schedules  |
+-------------------------+
|       Hardware          |
+-------------------------+
```

🔷 Comparison Table: ULT vs KLT

Feature	User-Level Threads (ULT)	Kernel-Level Threads (KLT)
Managed By	User-space library	Operating System Kernel
System Call Needed	No	Yes
Performance	High (low overhead)	Lower (kernel switches)
Blocking Impact	Blocks entire process	Other threads run independently
Parallelism	Not possible (1 core)	True parallelism (multi-core)

✅ Real-World Q&A

Q: Why use threads instead of processes?
A: Threads share memory and resources, allowing faster communication and better

performance for concurrent tasks.

Q: Why are ULTs still used if they can't run in parallel?
A: For simplicity and speed in single-core systems or cooperative multitasking environments (e.g., green threads in some runtimes).

Q: Can we mix ULT and KLT?
A: Yes — Many-to-One, One-to-One, and Many-to-Many models combine them. Some platforms (e.g., Java Virtual Machine) abstract the threading model based on OS capabilities.

Let me know when you're ready to proceed to:

- Multithreading Models (1:1, M:1, M:N)
- Thread lifecycle
- `pthread` example with output and dry-run

✅ Multithreading Models and Process Execution

🔹 Multithreading Models

Operating systems implement threading using one of three major models:

🔸 1. Many-to-One Model

Definition:
Maps **many user threads** to a **single kernel thread**.

Characteristics:

- Thread library manages all threads in user space.
- Only one thread can access the kernel at a time.

Diagram:

```
+----------------------+
|  User Thread 1       |
|  User Thread 2       |  -->  One Kernel Thread
|  User Thread N       |
+----------------------+
```

Pros:

- Fast context switching.
- Portable.

Cons:

- No parallelism on multicore systems.
- Blocking one thread blocks all.

Example:
Older Java Green Threads.

🔸 2. One-to-One Model

Definition:
Each **user thread** maps to its **own kernel thread**.

Characteristics:

- True parallelism possible on multicore systems.
- Kernel handles thread creation and management.

Diagram:

```
 +----------------------+
| User Thread 1 --> K1 |
| User Thread 2 --> K2 |
| User Thread 3 --> K3 |
+----------------------+
```

Pros:

- Concurrent execution on multiple cores.
- Non-blocking I/O per thread.

Cons:

- High overhead for creating threads.
- Limited number of threads per process.

Examples:
Linux `pthread`, Windows threads.

🔸 3. Many-to-Many Model

Definition:
Maps **many user threads** to **many kernel threads**.

Characteristics:

- OS can schedule any user thread onto any available kernel thread.
- Better scalability and flexibility.

Diagram:

```
 +-------------------------+
| U1 U2 U3 --> K1         |
| U4 U5    --> K2         |
+-------------------------+
```

Pros:

- Combines best of both worlds.
- High scalability.

Cons:

- More complex implementation.

Examples:
Windows fibers (to some extent), Solaris threads.

🔷 Context Switching

🔸 Definition:

Context switching is the act of **saving and restoring the state** of a process/thread when switching between tasks.

🔸 What Is Saved?

- Program Counter (PC)
- Stack Pointer (SP)
- CPU Registers
- Memory Mapping (Page Tables)
- Scheduling Information

🔸 Steps:

1. Save current process's PCB (state) to memory.
2. Select next process from ready queue.
3. Load its PCB values into CPU registers.
4. Resume execution from its program counter.

Diagram:

```
 [Process A Running]
     |
     | Save CPU State to PCB_A
     v
[Process B Running]
     ^
     | Load CPU State from PCB_B
```

🔸 Overhead:

- Can be expensive in time.
- Increases with number of threads/processes.
- Hardware support (e.g., TLB tagging) can reduce this.

✅ Process Creation & Execution in Linux

🔹 fork()

Creates a new child process by duplicating the parent.

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

int main() {
    pid_t pid = fork();
    if (pid == 0)
        printf("Child process\\n");
    else
        printf("Parent process\\n");
    return 0;
}
```

Output (non-deterministic order):

```
Parent process
Child process
```

◆ exec()

Replaces the current process image with a new program.

```
#include <unistd.h>

int main() {
    char *args[] = {"/bin/ls", "-l", NULL};
    execvp(args[0], args);
    return 0;
}
```

Output:
Displays `ls -l` listing.

◆ wait()

Makes the parent wait for the child to finish.

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

int main() {
    pid_t pid = fork();
    if (pid == 0)
         exit(0);
    else {
        wait(NULL);
        printf("Child finished\\n");
    }
    return 0;
}
```

◆ exit() and _exit()

- `exit()` flushes stdio buffers.
- `_exit()` terminates immediately.

✅ Real-World Q&A

Q: Why does `fork()` return twice?
A: It returns once in the parent (returns child's PID), and once in the child (returns 0), enabling both to run independently.

Q: Why use `exec()` after `fork()`?
A: To launch a new program in the child process without affecting the parent. `fork()` + `exec()` = foundation for `bash` pipelines.

Q: How is `wait()` useful?
A: Prevents zombie processes by allowing the parent to collect the child's exit status.

Let me know if you'd like the next set:

- Zombie vs Orphan processes
- Process hierarchy tree
- Pthread creation with dry run and output

✅ Process Hierarchy and Thread Pooling

🔷 Process Hierarchy: Parent, Child, Orphan, Zombie

🔸 Parent and Child Process

Parent Process:

- The process that creates a new process using `fork()`.
- Can manage, monitor, or wait for the child.

Child Process:

- Created by the parent; has a separate memory space.
- Receives a copy of the parent's data and code segment.

```
pid_t pid = fork(); // Creates a child process
```

🔸 Orphan Process

Definition:
A child process **whose parent terminates before the child does**.

What happens?

- The OS (Linux) reassigns the orphaned process to `init` (PID 1) or `systemd`.
- This ensures it still has a parent for cleanup.

Diagram:

```
Parent
  |
  |-- Child (running)
Parent exits
  |
Child → Orphan → Adopted by init/systemd
```

Code Example:

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

int main() {
    pid_t pid = fork();
    if (pid > 0) {
        printf("Parent exiting...\\n");
         exit(0); // Parent exits
    } else {
        sleep(5); // Child becomes orphan
        printf("Child now orphan, adopted by PID %d\\n", getppid());
    }
    return 0;
}
```

🔸 Zombie Process

Definition:
A process that **has completed execution but still has an entry in the process table** because its parent hasn't called `wait()`.

Why does it exist?

- The kernel keeps exit status information so the parent can retrieve it.
- If `wait()` is never called, the process becomes a zombie.

Lifecycle:

1. Child exits → OS marks it "terminated"
2. Parent doesn't `wait()` → remains in process table
3. Wastes system resources

Code Example:

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
         exit(0); // Child exits immediately
    } else {
        sleep(10); // Parent sleeps, doesn't wait
        printf("Check with: ps -ef | grep defunct\\n");
    }
    return 0;
}
```

Zombie state shown in `ps`:

```
defunct    Z
```

🔷 Daemon Processes

Definition:
A background process that **runs independently of terminal or user session**, typically to perform periodic or system-level tasks.

Examples:

- `cron`, `sshd`, `httpd`, `systemd`, `udevd`

Characteristics:

- Detaches from terminal using `setsid()`
- Redirects I/O to `/dev/null`
- Often started during boot time
- Has no controlling terminal

How to create a daemon:

1. `fork()` and exit parent
2. `setsid()` to become session leader
3. Change working directory and file mode mask
4. Close standard file descriptors

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

int main() {
    pid_t pid = fork();
    if (pid > 0) exit(0); // Parent exits
    setsid();             // Become session leader
    chdir("/");           // Change working directory
    close(0); close(1); close(2); // Close stdin, stdout, stderr
    while (1) {
        // Daemon task (e.g., logging)
        sleep(10);
    }
    return 0;
}
```

🔷 Thread Pools

Definition:
A thread pool is a collection of **pre-initialized threads** that are reused to execute tasks, rather than creating new threads each time.

Why thread pools?

- Avoids overhead of thread creation/destruction.
- Efficient resource usage under high load.
- Useful in server applications (e.g., handling requests).

🔸 How it Works

1. Fixed number of threads created at startup.
2. Tasks are placed in a **queue**.
3. Idle threads pick tasks from the queue and execute.
4. Threads remain alive and reusable.

🔸 Benefits

- Improved performance and responsiveness.
- Bounded resource usage (no thread explosion).
- Better CPU scheduling under load.

🔸 C++ Pseudocode Example (using std::thread + queue)

```
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>

std::queue<std::function<void()>> taskQueue;
std::mutex queueMutex;
std::condition_variable cv;

void worker() {
    while (true) {
        std::function<void()> task;
        {
            std::unique_lock<std::mutex> lock(queueMutex);
            cv.wait(lock, [] { return !taskQueue.empty(); });
            task = taskQueue.front(); taskQueue.pop();
        }
        task(); // Execute task
    }
}

// Initialization
for (int i = 0; i < 4; ++i)
    std::thread(worker).detach();
```

🔸 Real-World Use Cases

- Web servers (e.g., Apache)
- Job scheduling systems
- Background task managers

✅ Real-World Q&A

Q: Why do zombies occur?
A: Because the parent didn't `wait()` to collect the child's exit status.

Q: What if a daemon crashes?
A: It may get restarted by `systemd` or `init.d`, depending on service configuration.

Q: Can threads in a thread pool block each other?
A: Yes, if improperly synchronized or if too many tasks block (e.g., I/O).

🧠 3. CPU Scheduling

✅ Introduction

CPU Scheduling is the process of selecting a process from the **ready queue** and allocating the CPU to it. Since only one process can use the CPU at a time in a uniprocessor system, an efficient scheduling algorithm is essential for maximizing performance.

🔹 Preemptive vs Non-Preemptive Scheduling

🔸 Non-Preemptive Scheduling

Definition:
Once a process starts executing on the CPU, it **runs to completion or voluntarily yields** (e.g., for I/O). The CPU is not taken away.

Characteristics:

- Simple to implement.
- Suitable for batch systems.
- Less overhead.

Examples:

- First-Come-First-Serve (FCFS)
- Shortest Job First (non-preemptive)
- Priority Scheduling (non-preemptive)

Disadvantages:

- Can cause long wait times for short processes.

🔸 Preemptive Scheduling

Definition:
The operating system can **suspend a running process** and allocate the CPU to another process (usually with higher priority or shorter remaining time).

Characteristics:

- Enables better responsiveness and fairness.
- Common in time-sharing and real-time systems.
- Requires context switching.

Examples:

- Round Robin (RR)

- Shortest Remaining Time First (SRTF)
- Preemptive Priority Scheduling
- Multilevel Feedback Queue

Disadvantages:

- Context switching overhead.
- Potential for race conditions if not managed properly.

🔹 Comparison Table

Feature	Preemptive	Non-Preemptive
Control	OS can interrupt process	Process keeps CPU until done
Responsiveness	High	Low
Overhead	High (context switching)	Low
Complexity	More complex	Simpler to implement
Examples	RR, SRTF, MLFQ	FCFS, SJF, Non-preemptive Priority

🔹 Scheduling Criteria

Scheduling algorithms are evaluated based on these **performance metrics**:

🔸 1. CPU Utilization

Definition:
Percentage of time the CPU is actively working on processes (not idle).

Goal: Maximize
Typical range: 40–90%

🔸 2. Throughput

Definition:
Number of processes completed per unit time.

Goal: Maximize
Higher throughput means more work done.

Example:
If 5 processes finish in 10 seconds → Throughput = 0.5 processes/second

---

🔸 3. Turnaround Time

Definition:
Total time taken for a process from submission to completion.

Formula:

```
Turnaround Time = Completion Time - Arrival Time
```

Goal: Minimize
Includes waiting time + execution + I/O

🔸 4. Waiting Time

Definition:
Total time a process spends **in the ready queue** waiting for CPU.

Formula:

```
Waiting Time = Turnaround Time - Burst Time
```

Goal: Minimize
Affects overall user satisfaction and fairness.

🔸 5. Response Time

Definition:
Time from process submission until the **first response (CPU allocation)**.

Important for: Interactive systems

Goal: Minimize
Note: Not total execution time — only time to get first CPU slice.

🔹 Real-World Q&A

Q: Why not always use preemptive scheduling?
A: Because it incurs overhead (context switching), may lead to starvation, and adds system complexity.

Q: Why is response time important in GUI systems?
A: Users expect immediate feedback — even a 1-second delay can degrade UX.

Q: Can scheduling criteria conflict?
A: Yes. Maximizing throughput may increase turnaround time; minimizing waiting time may reduce CPU utilization.

🔹 Example for Clarification

Processes:

Process	Arrival	Burst
P1	0	4

Process	Arrival	Burst
P2	1	3
P3	2	1

Using FCFS (Non-preemptive):
Gantt Chart:

```
 | P1 | P2 | P3 |
0    4    7    8
```

Turnaround Time:

- P1 = 4 - 0 = 4
- P2 = 7 - 1 = 6
- P3 = 8 - 2 = 6

Waiting Time:

- P1 = 0
- P2 = 4 - 1 = 3
- P3 = 7 - 2 = 5

Let me know when you're ready to continue with:

- Scheduling Algorithms (FCFS, SJF, RR, Priority)
- Gantt chart problems and code
- Starvation, Aging, and Real-Time Scheduling

✅ CPU Scheduling Algorithms

🔸 1. First-Come, First-Served (FCFS)

Definition:
Processes are scheduled in the order they arrive (like a queue). **Non-preemptive.**

Characteristics:

- Simple to implement.
- Can cause high waiting time for short jobs.

Gantt Example:

Process	Arrival	Burst
P1	0	5
P2	1	3
P3	2	1

```
| P1 | P2 | P3 |
0    5    8    9
```

Waiting Time:

- P1 = 0
- P2 = 5 - 1 = 4
- P3 = 8 - 2 = 6

Pros:

- Simple and fair

Cons:

- Convoy effect (long process delays all others)

🔸 2. Shortest Job First (SJF)

Definition:
Selects process with **smallest burst time** first.

- **Non-preemptive**: Once a process starts, it runs till completion.
- **Preemptive (SRTF)**: If a new process arrives with a shorter burst, it preempts the current one.

Non-Preemptive Example:

Process	Arrival	Burst
P1	0	7
P2	1	2
P3	2	1

```
| P1 | P3 | P2 |
0    7    8    10
```

Preemptive SJF (SRTF): Continuously checks if a new process with a shorter remaining time has arrived.

Pros:

- Optimal for average waiting time

Cons:

- Needs future burst prediction
- Starvation possible for long jobs

🔸 3. Round Robin (RR)

Definition:
Each process gets a **fixed time slice (quantum)**. If it doesn't finish, it goes back to the queue.

Characteristics:

- **Preemptive**
- Fair, used in time-sharing systems

Example (Quantum = 2):

Process	Arrival	Burst
P1	0	4
P2	1	5
P3	2	2

```
 | P1 | P2 | P3 | P1 | P2 | P2 |
0    2    4    6    8    9    11
```

Pros:

- Fairness and responsiveness

Cons:

- Too small quantum → high overhead
- Too large quantum → becomes FCFS

🔸 4. Priority Scheduling

Definition:
Each process has a **priority**, and the CPU is assigned to the process with the highest priority.

- **Preemptive**: Higher priority can interrupt.
- **Non-preemptive**: Waits for the running process to finish.

Example:

Process	Priority	Burst
P1	3	5
P2	1	3
P3	2	2

Scheduling Order: P2 → P3 → P1

Pros:

- Flexible control over resource allocation

Cons:

- Starvation possible (low-priority jobs may never run)

Solution:

- **Aging**: Gradually increase priority of waiting processes.

🔸 5. Multilevel Queue Scheduling

Definition:
Processes are grouped into **queues based on type** (foreground, background), each with its own scheduling policy.

Example Queues:

- System processes (FCFS)
- Interactive jobs (RR)
- Batch jobs (SJF)

Scheduling:

- Fixed priority among queues.
- No movement between queues.

Diagram:

```
 Queue 1 (High priority): RR
Queue 2: FCFS
Queue 3: SJF
```

Pros:

- Separates job classes.

Cons:

- Rigid structure, starvation of low-priority queues.

🔸 6. Multilevel Feedback Queue (MLFQ)

Definition:
Improved version of multilevel queue — processes can **move between queues** based on behavior and age.

Rules:

- Start in high-priority queue with small quantum.
- If not completed → move to lower queue.
- If waiting too long → move up (aging).

Characteristics:

- Preemptive
- Adaptive to process behavior

Pros:

- Reduces starvation
- Good for mixed workloads

Cons:

- Complex implementation

🔸 7. **Earliest Deadline First (EDF)** – Real-Time Scheduling

Definition:
Schedules the task with the **nearest deadline** first.

Used in:
Hard and soft **real-time systems**

Pros:

- Proven to be optimal under certain utilization limits.

Cons:

- Requires knowledge of deadlines
- Susceptible to deadline misses under overload

🔸 8. **Rate Monotonic Scheduling (RMS)** – Real-Time Scheduling

Definition:
Assigns priorities based on **frequency of execution** — **shorter periods = higher priority**.

Characteristics:

- Static priorities
- Works well for periodic real-time tasks

Pros:

- Easy to analyze
- Optimal for static task sets

Cons:

- Not optimal for dynamic tasks or non-periodic workloads

✅ Comparison Table

Algorithm	Preemptive	Starvation	Suitable For
FCFS	No	Yes	Simple batch systems
SJF	Both	Yes	Performance-focused
RR	Yes	No	Time-sharing systems

Algorithm	Preemptive	Starvation	Suitable For
Priority	Both	Yes	Controlled environments
MLQ	Mixed	Yes	OS-level classification
MLFQ	Yes	No	Adaptive workloads
EDF	Yes	No	Real-time dynamic tasks
RMS	No	No	Periodic real-time tasks

Let me know if you'd like Gantt chart examples for each algorithm, or real-world scenarios (like scheduling in Linux or Android).

✅ Gantt Charts and Scheduling Metrics Calculation

🔸 What is a Gantt Chart?

A **Gantt chart** visually represents the order and duration of processes being scheduled on the CPU. It is essential for calculating performance metrics like **Turnaround Time**, **Waiting Time**, and **Response Time**.

🔸 Common Terms

- **Arrival Time (AT)**: When the process enters the ready queue.
- **Burst Time (BT)**: Time the process needs on CPU.
- **Completion Time (CT)**: When the process finishes execution.
- **Turnaround Time (TAT)** = CT - AT
- **Waiting Time (WT)** = TAT - BT
- **Response Time (RT)** = First CPU start - AT

🔸 Example Problem

Process	Arrival	Burst
P1	0	5
P2	1	3
P3	2	8
P4	3	6

Algorithm: FCFS (First Come First Served)

🔸 Step-by-step Gantt Chart:

```
  |  P1  |  P2  |  P3  |  P4  |
0      5      8      16     22
```

🔸 Completion Time (CT)

Process	CT
P1	5
P2	8
P3	16
P4	22

🔸 Turnaround Time (TAT = CT - AT)

Process	CT	AT	TAT = CT - AT
P1	5	0	5
P2	8	1	7
P3	16	2	14
P4	22	3	19

🔸 Waiting Time (WT = TAT - BT)

Process	TAT	BT	WT = TAT - BT
P1	5	5	0
P2	7	3	4
P3	14	8	6
P4	19	6	13

🔸 Response Time (RT)

For **non-preemptive FCFS**, Response Time = Waiting Time
(Since the process starts only once)

🔸 Average Metrics

- **Avg Turnaround Time** = (5 + 7 + 14 + 19) / 4 = 11.25
- **Avg Waiting Time** = (0 + 4 + 6 + 13) / 4 = 5.75

- **Avg Response Time** = 5.75 (same as WT for FCFS)

✅ Starvation and Aging

🔸 Starvation

Definition:
A condition where a **low-priority process waits indefinitely** because higher-priority processes keep executing.

Occurs in:

- Priority Scheduling
- Multilevel Queue Scheduling

Example:
If a system always schedules high-priority tasks, a low-priority background job may **never** get CPU time.

Real-World Analogy:
At a restaurant, if VIP customers keep coming, the regular customers never get served.

🔸 Effects of Starvation

- Wastes resources (process waits without progress).
- Degrades fairness.
- Affects system responsiveness and reliability.

🔸 Aging

Definition:
A **technique to prevent starvation** by **gradually increasing the priority** of a waiting process over time.

How it works:

- Every `x` units of time, increase the priority level of waiting processes.
- Eventually, even low-priority processes will reach execution.

Implementation:

```
If wait time >= threshold:
    priority += 1
```

Example:
After 10 minutes of waiting, increase priority of job by 1.

🔸 Diagram: Starvation Avoidance via Aging

```
Initial Priority Queue:
[ P1(high) > P2(med) > P3(low) ]

After 5 min:
P3 priority ↑ → Now: [ P1 > P2 > P3(med) ]

After 10 min:
P3 priority ↑↑ → Now: [ P3(high) > P1 > P2 ]
```

✅ Real-World Q&A

Q: Can Round Robin cause starvation?
A: No. Every process gets a fair time slice.

Q: Why is Aging necessary in MLFQ?
A: Because without aging, low-priority queues may never get CPU if higher-priority queues are busy (starvation risk).

Q: Is starvation a bug?
A: No, it's a **design flaw** if the algorithm doesn't handle long-waiting jobs fairly.

Let me know if you want to proceed to:

- Real-time scheduling guarantees (EDF, RMS)
- Practical Linux schedulers (CFS)
- Implementing scheduling in C++

🧠 4. Synchronization & Concurrency

✅ What is Synchronization?

Synchronization ensures **correct execution of concurrent processes or threads**, particularly when they **access shared data** or **critical resources**.

🔸 The Critical Section Problem

Definition:
A **critical section** is a part of the code where **shared resources** are accessed (e.g., global variables, files, buffers). If multiple threads/processes enter their critical sections at the same time, it may lead to **data races or corruption**.

🔹 Conditions to Solve the Critical Section Problem

1. **Mutual Exclusion** – Only one process/thread can enter the critical section at a time.
2. **Progress** – If no one is in the critical section, a process outside cannot prevent others from entering.
3. **Bounded Waiting** – A process must not wait forever to enter its critical section.

◆ Software Solutions

◆ 1. Peterson's Algorithm

Type: Software-based mutual exclusion for **2 processes**.

Idea:
Processes use two shared variables:

- `flag[i]`: Indicates if process `i` wants to enter.
- `turn`: Indicates whose turn it is.

```
// Process 0
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1);
// critical section
flag[0] = false;
```

```
// Process 1
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0);
// critical section
flag[1] = false;
```

Satisfies:
All 3 conditions: mutual exclusion, progress, bounded waiting.

Limitation:
Works only for 2 processes and relies on memory ordering (not safe on all CPUs).

◆ 2. Bakery Algorithm (Lamport's)

Type: Generalized Peterson's algorithm for `n` **processes**.

Idea:
Each process takes a "number" like in a bakery queue. The process with the **smallest number** gets to enter the critical section.

```
choosing[i] = true;
number[i] = 1 + max(number[0..n-1]);
choosing[i] = false;

for (j = 0; j < n; j++) {
    while (choosing[j]);
    while ((number[j] != 0) &&
           ((number[j] < number[i]) ||
           (number[j] == number[i] && j < i)));
}
```

Exit:

```
number[i] = 0;
```

Satisfies:
Mutual exclusion, bounded waiting, progress.

Limitations:

- Complex and inefficient.
- Not practical on modern CPUs due to relaxed memory models.

🔷 Hardware Solutions

Modern processors provide **atomic hardware instructions** for synchronization.

🔶 3. TestAndSet

Definition:
An atomic instruction that tests a memory location and sets it **in a single operation**.

```
bool TestAndSet(bool *target) {
    bool old = *target;
    *target = true;
    return old;
}
```

Usage:

```
while (TestAndSet(&lock)); // busy wait
// critical section
lock = false;
```

Pros:

- Ensures mutual exclusion via atomicity

Cons:

- Causes **busy waiting (spinlock)**
- Starvation possible

🔶 4. Swap Instruction

Definition:
Swaps values of two variables atomically.

```
void Swap(bool *a, bool *b) {
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

Usage:

```
bool key = true;
while (true) {
    Swap(&lock, &key);
    if (!key) break; // enter critical section
}
// critical section
lock = false;
```

Pros:

- Low-level atomic solution
- No need for flags or IDs

Cons:

- Still spinlocks
- Complex to manage fairness

✅ Comparison Table

Algorithm	Mutual Exclusion	Bounded Waiting	Progress	Type
Peterson's	✅	✅	✅	Software
Bakery	✅	✅	✅	Software
TestAndSet	✅	❌	❌	Hardware
Swap	✅	❌	❌	Hardware

✅ Real-World Q&A

Q: Why not use TestAndSet everywhere?
A: It causes **busy-waiting**, wasting CPU cycles. Better to use higher-level constructs like mutexes or semaphores.

Q: Can Peterson's algorithm work on modern CPUs?
A: No — it relies on strict memory ordering which is **not guaranteed** in modern multi-core CPUs without memory fences.

Q: Why is bounded waiting important?
A: It ensures **fairness** — no process/thread waits forever.

Let me know when you're ready to continue with:

- Mutex, Semaphore, Monitor
- Classic problems (Producer-Consumer, Readers-Writers, Dining Philosophers)

✅ Advanced Synchronization Concepts

◆ Hardware Support for Synchronization

Modern CPUs provide **atomic instructions** to implement efficient locking and avoid race conditions.

◆ Common Hardware Instructions:

Instruction	Description
`TestAndSet`	Tests a variable and sets it in one atomic step
`CompareAndSwap`	Compares memory content and swaps if equal
`FetchAndAdd`	Atomically adds and returns the previous value
`Load-Link/Store-Conditional`	Used in RISC processors

◆ Benefits:

- Faster than software-only solutions.
- Essential for implementing spinlocks, mutexes.
- Basis for synchronization primitives in OS.

◆ Mutex vs Semaphore vs Spinlock

◆ 1. Mutex (Mutual Exclusion Lock)

Definition:
A binary lock that allows only one thread to enter the **critical section**.

Characteristics:

- Only **owner** can unlock.
- Usually **blocking** (puts thread to sleep if locked).

Example (Pseudocode):

```
pthread_mutex_lock(&lock);
// critical section
pthread_mutex_unlock(&lock);
```

◆ 2. Semaphore

Definition:
A generalized counter used to control access to resources.

- **Binary Semaphore:** Works like a mutex (0 or 1).
- **Counting Semaphore:** Allows up to N concurrent accesses.

Operations:

- `wait()` or `P()` – decrement and block if zero
- `signal()` or `V()` – increment and wake up a thread

Example:

```
sem_wait(&s);   // wait
// critical section
sem_post(&s);   // signal
```

🔸 3. Spinlock

Definition:
A lock where threads **busy-wait** (continuously check) until it is free.

Characteristics:

- No context switching → very fast on short critical sections
- CPU is occupied while waiting
- Inefficient if held for long durations

Example:

```
while (__sync_lock_test_and_set(&lock, 1)) {}  // spin
// critical section
__sync_lock_release(&lock);
```

🔹 Comparison Table

Feature	Mutex	Semaphore	Spinlock
Owner Required	Yes	No	No
Blocking	Yes	Yes	No (busy)
Count	1	0 to N	1
Usage	Mutual Exclusion	Resource Count	Low-latency lock

🔹 Counting vs Binary Semaphore

🔸 Binary Semaphore

- Only two states: `0` (locked), `1` (unlocked)
- Used for **mutual exclusion**
- Similar to a mutex but **any thread** can `signal()`

🔸 Counting Semaphore

- Value can be >1
- Tracks number of available **resources**
- Useful for managing a pool (e.g., database connections)

Example:

```
sem_init(&sem, 0, 3);  // 3 resources
sem_wait(&sem);        // acquire
sem_post(&sem);        // release
```

🔹 Monitor

Definition:
A **high-level synchronization construct** that allows **only one thread** to execute a method (or block) at a time.

Encapsulates:

- Shared variables
- Synchronization code
- Condition variables

Languages with Monitor support:

- Java (synchronized)
- Python (threading.Lock)
- C++20 (via condition variables + scoped_lock)

Java Example:

```
synchronized void increment() {
    count++;
}
```

Monitor vs Semaphore

Feature	Monitor	Semaphore
Level	High-level	Low-level
Ownership	Enforced (by language)	Not enforced
Blocking	Yes	Yes
Usage	Encapsulated objects	Global/shared

🔹 Busy Waiting vs Blocking

🔸 Busy Waiting

- Continuously checks a condition in a loop
- Wastes CPU cycles
- Used in **spinlocks**, short waits

Example:

```
while (lock == 1); // spin
```

🔸 Blocking

- Puts the thread to **sleep** until condition is met
- Frees up CPU for other tasks
- Used in semaphores, condition variables

Example:

```
pthread_cond_wait(&cond, &mutex);
```

🔹 Thread Safety and Atomicity

🔸 Thread-Safe Code

Definition:
Code that works correctly when accessed by **multiple threads** concurrently.

Achieved by:

- Mutexes
- Semaphores
- Atomic operations
- Immutability

🔸 Atomic Operation

Definition:
An operation that appears **instantaneous and indivisible**.

Example:

```
__atomic_fetch_add(&x, 1, __ATOMIC_SEQ_CST);
```

🔹 Reentrant Functions

🔸 Definition:

A **reentrant function** is one that **can be safely interrupted and re-entered**, even by itself (recursively or from another thread).

🔸 Requirements:

- Does **not use static or global variables**
- Does **not modify shared data**
- Uses local variables or thread-local storage

Example:

```
int add(int a, int b) {
    return a + b;  // reentrant
}
```

Non-Reentrant:

```
int counter = 0;
int increment() {
    return ++counter;  // not thread-safe
}
```

🔷 Race Conditions

🔸 Definition:

A **race condition** occurs when the **correctness of a program depends on the sequence/timing** of uncontrollable events (e.g., thread execution order).

🔸 Example:

```
int counter = 0;

void* thread func(void*) {
    for (int i = 0; i < 1000; i++)
        counter++;
}
```

With 2 threads: final value may not be 2000!

Why? `counter++` is not atomic (read-modify-write).

🔸 Solution:

```
pthread mutex_lock(&lock);
counter++;
pthread_mutex_unlock(&lock);
```

✅ Real-World Q&A

Q: Can a function be thread-safe but not reentrant?
A: Yes. It may use locks (thread-safe) but still rely on global state (not reentrant).

Q: When is busy waiting acceptable?
A: For very short waits in low-level code (e.g., spinlocks inside OS kernels).

Q: Why use condition variables over polling?
A: Polling wastes CPU time; condition variables allow sleeping until notified.

Let me know if you want to continue with:

- Classical problems (Producer-Consumer, Readers-Writers)
- Semaphore-based synchronization code
- Condition variables & barriers

✅ Deadlock, Livelock & Starvation

🔹 What is a Deadlock?

A **deadlock** is a situation in a multi-process system where two or more processes are **permanently blocked**, each waiting for a resource held by the other.

Example:
Process A holds resource R1 and waits for R2
Process B holds R2 and waits for R1 → both are blocked

🔹 Necessary Conditions (Coffman's Conditions)

A deadlock can occur **only if all four conditions** hold simultaneously:

1. **Mutual Exclusion**
   At least one resource is held in a **non-shareable** mode.

2. **Hold and Wait**
   A process is **holding at least one resource** and waiting to acquire more.

3. **No Preemption**
   Resources cannot be forcibly taken; they must be **released voluntarily**.

4. **Circular Wait**
   A set of processes `{P1, P2, ..., Pn}` exist such that
   `P1` waits for a resource held by `P2`,
   `P2` for `P3`, ..., `Pn` for `P1`.

🔹 Resource Allocation Graph (RAG)

Used to visualize potential deadlocks.

- **Nodes**: Processes (circles) and Resources (squares)
- **Edges**:
  - Request edge: `P → R`
  - Assignment edge: `R → P`

Deadlock: Cycle exists (for single instance of each resource)

Example:

```
[P1] → [R1] → [P2] → [R2] → [P1]  ⇒ DEADLOCK
```

🔹 Deadlock Prevention

Idea: Eliminate one or more of the four Coffman conditions.

🔸 Strategy Summary:

Condition	Prevention Technique
Mutual Exclusion	Not always preventable (printers)
Hold and Wait	Require all resources at once
No Preemption	Preempt resources from waiting procs
Circular Wait	Impose strict ordering on resources

🔹 Deadlock Avoidance

Uses **future knowledge** to avoid unsafe states.

🔸 Banker's Algorithm (Dijkstra)

Used for:
Multiple instances of each resource
Based on **safe state detection**

✅ Key Terms:

- **Available**: Number of resources currently available
- **Max**: Maximum demand of each process
- **Allocation**: Currently allocated resources
- **Need** = Max - Allocation

✅ Algorithm Steps:

1. Let `Work = Available`
2. Find process `P` such that:
   - `Need[P] <= Work`

3. If found:
    - `Work += Allocation[P]`
    - Mark P as finished
4. Repeat until:
    - All processes can finish → Safe State
    - No such process → Unsafe (possible deadlock)

✅ Example:

Process	Max	Allocation	Need
P1	7	3	4
P2	5	2	3
P3	3	2	1

Available = 3

Safe sequence = P3 → P2 → P1

🔷 Deadlock Detection

Used when system doesn’t prevent or avoid deadlocks.
It periodically checks for cycles in the **Resource Allocation Graph**.

- For **single instance of each resource**: Cycle = Deadlock
- For **multiple instances**: Use **Wait-for Graph (WFG)**

When to run:

- Periodically
- On resource request failure

🔷 Deadlock Recovery

If a deadlock is detected:

🔸 Recovery Methods:

1. **Process Termination**

    - Kill all deadlocked processes
    - Kill one-by-one until deadlock breaks

2. **Resource Preemption**

    - Take resources from other processes
    - Rollback and restart

🔹 Livelock

Definition:
Processes **continuously change state** in response to each other **but never make progress**.

Example: Two people step side to side trying to pass in a hallway and continuously block each other.

In Code:
Processes repeatedly yield or retry without ever entering the critical section.

Difference from Deadlock:

- Deadlock → no progress and blocked
- Livelock → no progress but actively trying

🔹 Starvation

Definition:
A process **waits indefinitely** to gain access to a resource because others are constantly prioritized.

Occurs in:

- Priority scheduling
- Multilevel queues
- Readers-writers problems

Solution:

- **Aging** – Increase priority of waiting processes gradually

---

✅ Comparison Table

Issue	Progress?	Cause	Solution
Deadlock	❌	Circular hold of resources	Prevention/Avoidance
Livelock	❌	Repeated retry/yield	Backoff strategies
Starvation	❌ (for victim)	Unfair scheduling	Aging/Fairness

✅ Real-World Q&A

Q: Can deadlocks occur with threads?
A: Yes. If threads use mutexes/resources and follow circular waiting patterns.

Q: Can the Banker's algorithm work with dynamic resource requests?
A: Only if the **maximum claim is known in advance** — otherwise it fails.

Q: Is livelock harder to detect than deadlock?
A: Yes, because the system appears active even though no task is progressing.

Let me know if you want:

- Code examples for deadlock, semaphore usage
- Classic concurrency problems (Dining Philosophers, Readers-Writers)

✅ Classical Synchronization Problems

🔷 Bounded Buffer Problem (Producer-Consumer)

🔸 Problem Statement

A classic example of a multi-process synchronization problem involving **two processes**:

- **Producer** generates data and inserts it into a buffer.
- **Consumer** removes data from the buffer.

If the buffer is **full**, the producer must wait.
If the buffer is **empty**, the consumer must wait.

🔸 Constraints

- Only one thread can **access the buffer at a time** (mutual exclusion).
- **Bounded buffer** of fixed size `N`.

🔸 Solution using Semaphores

```
#define N 5
int buffer[N], in = 0, out = 0;

sem_t mutex = 1;       // for mutual exclusion
sem_t empty = N;       // initially all slots are empty
sem_t full = 0;        // initially no slots are full

// Producer
while (true) {
    int item = produce();
    sem_wait(&empty);          // wait for empty slot
    sem_wait(&mutex);          // enter critical section
    buffer[in] = item;
    in = (in + 1) % N;
    sem_post(&mutex);          // exit critical section
    sem_post(&full);           // signal item available
}

// Consumer
while (true) {
    sem_wait(&full);           // wait for item
    sem_wait(&mutex);          // enter critical section
    int item = buffer[out];
    out = (out + 1) % N;
    sem_post(&mutex);          // exit critical section
```

```
    sem post(&empty);          // signal empty slot
    consume(item);
}
```

◆ Reader-Writer Problem

◆ Problem Statement

- **Multiple readers** can read the shared data simultaneously.
- **Writers** require **exclusive** access.
- Goal: Avoid **reader starvation** and maintain **data consistency**.

◆ Variants

- **First Readers-Writers Problem**: No reader shall be kept waiting unless a writer has already obtained access.
- **Second Readers-Writers Problem**: Once a writer is ready, it performs write ASAP (readers can starve).

◆ Solution using Semaphores (1st Problem)

```
 int readcount = 0;
sem t mutex = 1;   // protects readcount
sem_t wrt = 1;     // for writer access

// Reader
sem wait(&mutex);
readcount++;
if (readcount == 1)
    sem wait(&wrt);  // first reader locks writer
sem_post(&mutex);

// critical section: read

sem wait(&mutex);
readcount--;
if (readcount == 0)
    sem post(&wrt);  // last reader unlocks writer
sem_post(&mutex);

// Writer
sem_wait(&wrt);

// critical section: write

sem_post(&wrt);
```

◆ Dining Philosophers Problem

◆ Problem Statement

- **5 philosophers** sit around a table with **5 forks**.
- Each needs **two forks** (left and right) to eat.

- Problem: **Avoid deadlock** and **ensure fairness**.

🔸 Naive (Wrong) Solution

```
// Each philosopher:
sem_wait(&fork[i]);
sem_wait(&fork[(i+1)%5]);
// eat
sem_post(&fork[i]);
sem_post(&fork[(i+1)%5]);
```

Problem: Deadlock if all philosophers pick left fork at the same time.

🔸 Solution (Asymmetric or Resource Hierarchy)

```
if (i % 2 == 0) {
    sem_wait(&fork[i]);
    sem_wait(&fork[(i+1)%5]);
} else {
    sem_wait(&fork[(i+1)%5]);
    sem_wait(&fork[i]);
}
// eat
sem_post(&fork[i]);
sem_post(&fork[(i+1)%5]);
```

Fix: Prevent circular wait by changing acquisition order.

🔸 Solution using Arbitrator (Waiter)

```
sem_t mutex = 1;

sem_wait(&mutex);
sem_wait(&fork[i]);
sem_wait(&fork[(i+1)%5]);
// eat
sem_post(&fork[i]);
sem_post(&fork[(i+1)%5]);
sem_post(&mutex);
```

Effect: Limits total philosophers trying to eat at once.

🔹 Barrier Synchronization

🔸 Problem Statement

Ensure that **multiple threads** wait for each other at a **synchronization point**, and only **proceed when all have arrived**.

Example: Used in parallel computation (e.g., matrix operations) where steps must be synchronized.

🔸 Pseudocode for Barrier (n threads)

```
int count = 0;
sem t mutex = 1;
sem_t barrier = 0;

void thread() {
    sem wait(&mutex);
    count++;
    if (count == N)
        sem post(&barrier); // last thread unblocks all
    sem_post(&mutex);

    sem wait(&barrier); // all threads wait here
    sem_post(&barrier); // allow others to pass
}
```

🔸 Using pthread*barriert* (POSIX)

```
pthread_barrier_t barrier;

pthread_barrier_init(&barrier, NULL, N);

void* thread(void* arg) {
    // Do some work
    pthread barrier wait(&barrier); // wait for all threads
    // Continue after all have reached
}
```

✅ Summary Table

Problem	Goal	Solution Type
Bounded Buffer	Synchronize producer & consumer	Semaphores
Readers-Writers	Allow multiple readers, 1 writer	Semaphore + counters
Dining Philosophers	Avoid deadlock, starvation	Fork ordering / waiter
Barrier Synchronization	Wait for all threads before proceed	Barrier primitive

🧠 5. Inter-Process Communication (IPC)

✅ Introduction

Inter-Process Communication (IPC) enables **data exchange between processes**, which are otherwise isolated by their memory spaces. IPC is vital for synchronization, coordination, and resource sharing among processes.

🔹 Shared Memory vs Message Passing

🔸 Shared Memory

Definition:
A region of memory is mapped into the address space of two or more processes, allowing them to communicate by reading/writing to it.

Characteristics:

- Fast (no kernel involvement after setup)
- Needs explicit synchronization (mutex/semaphore)

Example API (Linux): `shmget`, `shmat`, `shmdt`

```
 int shmid = shmget(IPC PRIVATE, 1024, IPC CREAT | 0666);
char* data = (char*) shmat(shmid, NULL, 0);
strcpy(data, "Hello");
```

Pros:

- High speed
- Suitable for large data

Cons:

- Complex synchronization
- More setup required

🔸 Message Passing

Definition:
Processes send and receive messages via the kernel (OS-mediated).

Characteristics:

- Safe (no shared memory)
- Simpler synchronization

Examples:

- Pipes
- Message Queues
- Sockets

Pros:

- Encapsulation of communication
- Simpler for unrelated processes

Cons:

- Slower than shared memory
- Limited data size

🔹 Pipes (Anonymous and Named)

🔸 Anonymous Pipes

Definition:
A unidirectional byte stream used for communication between **related processes** (e.g., parent and child).

API: `pipe()`

Example:

```
int fd[2];
pipe(fd); // fd[0]: read, fd[1]: write

if (fork() == 0) {
    close(fd[0]);
    write(fd[1], "msg", 3);
} else {
    close(fd[1]);
    char buf[4]; read(fd[0], buf, 3);
}
```

Limitations:

- Unidirectional
- Related processes only

🔸 Named Pipes (FIFO)

Definition:
A pipe with a name in the filesystem, allowing **unrelated processes** to communicate.

API: `mkfifo()` , `open()` , `read()` , `write()`

Example:

```
mkfifo mypipe
```

```
int fd = open("mypipe", O_WRONLY);
write(fd, "hello", 5);
```

Pros:

- Persistent and visible in filesystem
- Inter-process use

Cons:

- Still unidirectional unless two FIFOs used

🔹 Signals and Signal Handling

🔸 Signals

Definition:
Asynchronous notifications sent to a process to **interrupt or notify it of an event** (e.g., Ctrl+C, segmentation fault).

Common Signals:

Signal	Meaning
`SIGINT`	Interrupt (Ctrl+C)
`SIGKILL`	Kill immediately
`SIGTERM`	Terminate gracefully
`SIGCHLD`	Child terminated
`SIGSEGV`	Segmentation fault

🔸 Signal Handling

API: `signal()`, `sigaction()`

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

void handler(int signum) {
    printf("Caught signal %d\\n", signum);
}

int main() {
    signal(SIGINT, handler);
    while (1);
}
```

Output (on Ctrl+C):

```
Caught signal 2
```

Note: `SIGKILL` and `SIGSTOP` **cannot be caught** or ignored.

🔹 Message Queues (System V IPC)

🔸 Definition

A kernel-managed **queue of messages**, allowing processes to exchange data asynchronously using `msgsnd()` and `msgrcv()`.

Structure:

```
struct msgbuf {
    long mtype;
    char mtext[100];
};
```

Steps:

1. `msgget()` – create/get message queue
2. `msgsnd()` – send message
3. `msgrcv()` – receive message
4. `msgctl()` – control/delete queue

🔸 Example

```
key_t key = ftok("file", 65);
int msgid = msgget(key, 0666 | IPC_CREAT);

struct msgbuf msg;
msg.mtype = 1;
strcpy(msg.mtext, "Hello");

msgsnd(msgid, &msg, sizeof(msg), 0);
```

🔸 Advantages

- Works between unrelated processes
- Message-based (no shared memory needed)
- Can implement priority via `mtype`

🔸 Limitations

- Size limits per message and queue
- Slower than shared memory
- Not as flexible as sockets for network use

✅ Comparison Summary

IPC Method	Direction	Related Procs	Sync Needed	Size Limit	Speed
Shared Memory	Bi-dir	Any	Yes	No	Fastest
Anonymous Pipe	Uni	Related only	No (stream)	Yes	Fast
Named Pipe (FIFO)	Uni	Any	No (stream)	Yes	Fast
Message Queue	Bi-dir	Any	No	Yes	Medium
Signals	N/A	Any	Signal-safe	N/A	Immediate

✅ Real-World Use Cases

- **Shared Memory** – Game engines, simulation data sharing
- **Message Queues** – Job queues, logging daemons
- **Signals** – Process termination, alarm timers
- **Pipes** – Shell command chaining (`ls | grep foo`)
- **FIFOs** – Simple IPC between unrelated CLI tools

```
# Shell pipe example:
ps aux | grep chrome | wc -l
```

```
# FIFO example
mkfifo /tmp/fifo
echo "hello" > /tmp/fifo &
cat /tmp/fifo
```

✅ Advanced IPC Mechanisms

🔹 Semaphores (System V vs POSIX)

🔸 Semaphores

A **synchronization primitive** used to control access to shared resources. Can be **binary (mutex)** or **counting (resource pool)**.

🔸 System V Semaphores

Legacy API, uses integer semaphore arrays with IDs.

Functions:

- `semget()` – create or get a semaphore set
- `semctl()` – control operations (initialize, remove)
- `semop()` – perform P/V (wait/signal) operations

Example:

```
int semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);
semctl(semid, 0, SETVAL, 1); // set initial value

struct sembuf op = {0, -1, 0}; // wait
semop(semid, &op, 1);
```

Pros:

- Supports arrays (multiple semaphores)
- Useful in System V environments

Cons:

- Verbose, complex

- Not thread-friendly

🔸 POSIX Semaphores

Modern API, supports **named and unnamed semaphores**.

Header: `<semaphore.h>`

Functions:

- `sem_init()` – for unnamed semaphores
- `sem_open()` – for named semaphores
- `sem_wait()`, `sem_post()`, `sem_destroy()`

Example:

```
 sem t sem;
sem_init(&sem, 0, 1);   // Binary semaphore

sem wait(&sem);         // lock
// critical section
sem_post(&sem);         // unlock
```

Named Example:

```
 sem t *sem = sem_open("/mysem", O_CREAT, 0666, 1);
sem wait(sem);
sem post(sem);
sem_close(sem);
```

Pros:

- Easier to use
- Thread-compatible
- File descriptor support for named semaphores

Cons:

- Named semaphores require cleanup (`sem_unlink`)

🔹 Sockets (UNIX and Network)

🔸 Sockets

Sockets provide **bidirectional communication** between processes over:

- Local (UNIX Domain)
- Network (TCP/IP)

🔸 UNIX Domain Sockets

Used for communication between processes **on the same machine**.

Header: `<sys/socket.h>`
Address: Filesystem path

Example:

```
int sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
struct sockaddr_un addr;
addr.sun_family = AF_UNIX;
strcpy(addr.sun_path, "/tmp/mysock");

bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
listen(sockfd, 5);
```

Pros:

- Low overhead
- Secure and fast

🔸 Network Sockets (TCP/UDP)

Used for communication **over network** (between machines or remote apps).

Types:

- `SOCK_STREAM` – TCP
- `SOCK_DGRAM` – UDP

Common Functions:

- `socket()`, `bind()`, `listen()`, `accept()`
- `connect()`, `send()`, `recv()`

TCP Example:

```
int sfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sfd, ...);
send(sfd, "Hello", 5, 0);
```

🔹 RPC (Remote Procedure Call)

🔸 Definition

A **mechanism that allows a program to call a function** in another address space (usually on a remote server), as if it were local.

Example:
Client calls `getTime()` which executes on a remote server.

🔸 Components:

1. **Client Stub** – Marshals parameters.
2. **Server Stub** – Unmarshals and calls the actual function.
3. **Binder** – Registers service location.

🔸 Flow:

```
Client → Client Stub → Request Packet → Network → Server Stub → Server
               ↑                                         ↓
         Response Packet ← Network ← Server Stub ← Server
```

🔸 Use Cases:

- NFS (Network File System)
- Microservices / gRPC
- Database clients (e.g., MongoDB driver)

🔹 Memory-Mapped Files

🔸 Definition

Maps a file or portion of a file into the **process's address space** for **direct memory-like access**.

API: `mmap()`

```
int fd = open("file.txt", O_RDWR);
char* map = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
strcpy(map, "edit in-place");
munmap(map, size);
```

Pros:

- Efficient I/O (no read/write syscall overhead)
- Useful for IPC (shared mapping)

Cons:

- Synchronization required for concurrent access
- Complex for large files or different mappings

🔹 Pipes vs FIFO vs Shared Memory

Feature	Pipes (Anonymous)	FIFO (Named Pipe)	Shared Memory
Direction	Unidirectional	Unidirectional	Bidirectional
Persistence	Temporary	Persistent in FS	Persistent until detached
Related Processes	Required	Not required	Not required
Synchronization Req.	No (stream-based)	No (stream-based)	Yes (mutex/sem)

Feature	Pipes (Anonymous)	FIFO (Named Pipe)	Shared Memory
Speed	Medium	Medium	Fastest
Setup Complexity	Low	Moderate	High
Use Case	Parent-child comm	Simple IPC tools	Large data, performance IPC

✅ Real-World Applications

- **POSIX Semaphores** – Thread pools, bounded buffer
- **UNIX Sockets** – IPC in daemons (e.g., Docker, systemd)
- **TCP Sockets** – Web clients, APIs, servers
- **RPC** – gRPC, distributed systems
- **Memory-mapped files** – Shared databases, multimedia tools

🧠 6. Memory Management

✅ Introduction

Memory management is a critical function of the operating system that handles the allocation, deallocation, protection, and translation of memory between **user processes** and **hardware**.

🔹 Logical vs Physical Address

🔸 Logical Address (Virtual Address)

- **Generated by the CPU** during program execution.
- Belongs to the process's own **logical view** of memory.
- Used by user-level programs.

🔸 Physical Address

- Actual location in **main memory (RAM)**.
- Managed by the **hardware (MMU)** and **OS**.

🔸 Mapping Process

Logical addresses must be translated to physical addresses using:

- **Base + Offset**
- **Page tables**
- **MMU hardware**

🔸 Example

Component	Value
Base Register	10000
Logical Address	300
Physical Address	10300

🔹 MMU (Memory Management Unit)

🔸 What is MMU?

MMU (Memory Management Unit) is a hardware component that **automatically translates virtual/logical addresses** to physical addresses.

🔸 Responsibilities:

- Address Translation
- Access Control (read/write/execute)
- Page table walking
- Caching translation via **TLB** (Translation Lookaside Buffer)

🔸 Diagram: Logical to Physical Translation

```
 +-------------+        +------------------+
 | Logical Addr| ----->  | MMU (Page Table) |
+-------------+         +------------------+
                               |
                               v
                       +---------------+
                       | Physical Addr |
                       +---------------+
```

🔸 Benefits of MMU:

- **Process isolation**: Each process has its own logical memory.
- **Protection**: Prevents access to unauthorized memory.
- **Virtual memory**: Enables more memory than physically available.

🔹 Swapping

🔸 What is Swapping?

Swapping is the process of **moving an entire process** between main memory and **disk (swap space)** to free up RAM.

🔸 Steps:

1. If RAM is full, OS chooses a process to **swap out** to disk.
2. When that process needs to run again, it is **swapped in**.

🔸 Use Cases:

- Overloaded systems with insufficient RAM
- Background/inactive processes

🔸 Swapping Area:

- Linux: `/swapfile`, `/dev/sdX`, `/proc/swaps`
- Windows: `pagefile.sys`

🔸 Pros:

- Allows execution of large programs
- Keeps active processes in memory

🔸 Cons:

- Slow (disk is much slower than RAM)
- Can lead to **thrashing** if overused

🔸 Example (Linux):

```
# Check swap usage
free -m

# Enable swap
sudo swapon /swapfile

# Disable swap
sudo swapoff /swapfile
```

✅ Comparison Table

Feature	Logical Address	Physical Address
Generated By	CPU	MMU/OS
Used By	User Programs	RAM Controller

Feature	Logical Address	Physical Address
Visibility	Not real location	Actual RAM location

Feature	MMU	Swapping
Layer	Hardware	OS-Level Memory Management
Purpose	Translate addresses	Free memory by disk swap
Speed	Very Fast (CPU cycle)	Very Slow (disk I/O)
Used In	Every memory access	Under memory pressure

🔹 Contiguous Memory Allocation & Fragmentation

✅ Contiguous Memory Allocation

Contiguous memory allocation assigns a **single continuous block** of memory to each process.

🔸 Types of Partitions

🔹 1. Fixed Partitions

- Memory is divided into **equal-sized blocks** at system boot.
- Each block can contain only one process.

Pros:

- Simple to implement
- No allocation overhead

Cons:

- **Internal fragmentation** (wasted space if process is smaller)
- Static partitioning → inefficient use of memory

🔹 2. Variable Partitions

- Memory is divided dynamically based on **process requirements**.
- Processes are allocated **exact-sized blocks**.

Pros:

- Less internal waste

Cons:

- **External fragmentation** over time
- May need compaction

🔶 Allocation Strategies

🔹 First Fit

- Allocates first block that is **big enough**.

Pros: Fast
Cons: Can cause early fragmentation near start

🔹 Best Fit

- Allocates the **smallest sufficient block**.

Pros: Less internal fragmentation
Cons: Leaves many small unusable gaps (external fragmentation)

🔹 Worst Fit

- Allocates the **largest available block**.

Pros: Leaves big blocks for future
Cons: Tends to break memory into unusable chunks

🔸 Example:

Given memory blocks: `100 KB, 500 KB, 200 KB, 300 KB, 600 KB`
Request: `212 KB`

Strategy	Allocation Result
First Fit	500 KB
Best Fit	300 KB
Worst Fit	600 KB

✅ Fragmentation

🔹 Internal Fragmentation

- Occurs when allocated memory **is larger than requested**.
- Wasted space inside allocated block.

Example:
Request: 212 KB
Block given: 300 KB
→ 88 KB wasted (internal)

🔹 External Fragmentation

- Occurs when **enough total free memory exists**, but it's **scattered in non-contiguous blocks**.

Example:
Free blocks: 100 KB, 200 KB, 150 KB
Request: 400 KB → Fails despite 450 KB total available

🔸 Visualization

```
 +--------------------+    +--------------------+
| Process A (180 KB) |    | Free (20 KB)       |
| Process B (300 KB) |    | Free (150 KB)      |
| Free (20 KB)       |    | Process C (200 KB) |
+--------------------+    +--------------------+
Internal Frag        External Frag
```

🔹 Compaction

Definition:
Shifting all memory contents to **remove external fragmentation** by merging free spaces.

Pros:

- Consolidates holes into one large block

Cons:

- CPU overhead
- Must pause running processes or use relocation

Used in:

- Systems with base-register relocation
- Manual/periodic OS triggers

✅ Summary Table

Term	Cause	Solution
Internal Fragmentation	Block > Request size	Variable partitioning
External Fragmentation	Scattered free memory	Compaction

Strategy	Description	Pros	Cons
First Fit	First available block	Fast	Fragmented start
Best Fit	Smallest suitable block	Efficient use	Leaves tiny holes
Worst Fit	Largest block available	Preserves size	Breaks large holes

---

🔹 Paging in Memory Management

✅ What is Paging?

Paging is a memory management scheme that eliminates the need for contiguous allocation by dividing **logical memory** into **pages** and **physical memory** into **frames** of the same size.

🔸 Terminologies

🔹 Page

- Fixed-size block of **logical (virtual) memory**
- Example size: 4 KB

🔹 Frame

- Fixed-size block of **physical memory**
- Same size as page (ensures mapping)

🔹 Page Table

- Maintains the mapping from **virtual pages** → **physical frames**
- Stored in main memory
- Indexed using **page number**

🔹 Offset

- Within-page byte index
- Final physical address = `frame_start + offset`

🔸 Address Translation Example

Suppose:

- Logical address = 16 bits
- Page size = 4 KB (2^12 bytes)

- Thus:
    - Page Number = upper 4 bits
    - Offset = lower 12 bits

Logical Address	Page #	Offset
0x1F3A	0x1	0xF3A

🔸 Address Translation Flow

```
 +---------------------+
| Virtual Address     |
+----------+----------+
| Page No. | Offset   |
+----------+----------+
      |         |
      |         V
      |   +--------------------+
      |   | Physical Frame No  |
      +-> | Page Table Lookup  |
          +--------------------+
                   |
                   V
        Physical Address = Frame * PageSize + Offset
```

🔹 TLB – Translation Lookaside Buffer

🔸 Definition

TLB is a **small, fast cache** in the MMU that stores **recent page table lookups** to speed up translation.

🔸 Operation

- TLB Hit → Fast translation
- TLB Miss → Access page table in memory (slow)

Typical Hit Rate: 90–99%

🔸 Example

Page #	Frame #
1	5
2	9

If virtual page 2 is in TLB:

- Physical address = `frame[2] * page_size + offset`

🔹 Multi-level Page Tables

🔸 Why Needed?

A flat page table can be **very large** for 32/64-bit address spaces.

🔸 Structure

Page tables are divided into **levels** (e.g., 2-level, 3-level) to reduce memory usage.

Example (2-level):

- First-level index → page directory
- Second-level index → actual frame

🔸 Diagram

```
Virtual Addr (32-bit):
+--------+-------+--------+
| DirIdx | TblIdx | Offset |
+--------+-------+--------+

DirIdx → Page Directory
TblIdx → Page Table Entry
Offset → within page
```

🔸 Pros & Cons

Pros	Cons
Space-efficient	Multiple memory accesses
Supports sparse space	Slower than flat table

🔹 Inverted Page Table

🔸 Definition

In an inverted page table, there's **one entry per physical frame**, not per process/page.

Each entry stores:

- Process ID
- Virtual Page Number
- Frame info

🔸 Benefits

- Reduces table size (1 entry per frame)
- Suitable for large address spaces

🔸 Limitation

- Slower lookups (requires hash or linear search)

🔹 Hashed Page Table

🔸 Concept

Uses a **hash function** on the virtual page number to quickly locate frame entries.

Each hash bucket contains:

- Linked list of (VPN, PID, Frame)

🔸 Benefits

- Efficient for **large address spaces**
- Used in **64-bit architectures**

✅ Comparison Summary

Feature	Flat Page Table	Multi-Level Page Table	Inverted Page Table	Hashed Page Table
Lookup Speed	Fast	Slower (multi-access)	Slow (search/hash)	Medium (hash-based)
Space Efficiency	Poor	Good	Excellent	Good
Scaling (64-bit)	Bad	OK	Good	Good
Per Process Table	Yes	Yes	Single (global)	Single (global)

✅ Real-World Insights

- Linux uses **multi-level paging** (4-level on x86-64).
- TLB is critical for performance — TLB misses cause **page walks**.
- Inverted tables are useful for **virtualized systems** (Hypervisors).

🔹 Segmentation, Memory Protection & Demand Paging

🔹 Segmentation

✅ What is Segmentation?

Segmentation is a memory management technique where a process is divided into **logical segments**:

- Code
- Stack
- Heap
- Data

Each segment has a **variable size** and a **base + limit** pair.

🔸 Segment Table

- Keeps track of all segments in a process.
- Each entry contains:
    - **Base**: Start physical address of the segment.
    - **Limit**: Length of the segment.

🔸 Address Translation

```
Logical Address = (segment number, offset)
if (offset < limit)
    Physical Address = base + offset
else
    → SEGMENTATION FAULT
```

🔸 Segment Table Example

Segment	Base	Limit
0 (code)	1000	400
1 (stack)	5000	300

Accessing `seg=0, offset=350` → OK → `1000 + 350 = 1350`

Accessing `seg=1, offset=500` → **Segmentation Fault**

🔹 Segment Fault

- Raised when offset exceeds segment limit.

- Common in stack overflows, buffer overflows, invalid pointer dereference.

🔷 Segmentation with Paging

🔸 Concept

Combines:

- **Segmentation**: For logical division of program.
- **Paging**: To manage memory within each segment efficiently.

🔸 Address Breakdown

```
Virtual Address → (Segment #, Page #, Offset)
```

- Segment Table maps to page tables.
- Each segment has its own **page table**.

🔸 Benefits

- Logical separation + flexible physical mapping
- Solves external fragmentation

🔸 Diagram

```
[Segment #] → Segment Table → Page Table of that Segment → Frame + Offset → Physical Address
```

🔷 Memory Protection and Access Control

🔸 Why?

To **prevent processes from accessing memory not assigned** to them (accidental or malicious).

🔸 Mechanisms

1. **Base & Limit Registers**
    - Hardware enforces memory boundaries.
2. **Access Bits** (R/W/X)

- Set by OS for each page/segment.
- MMU checks on every access.

3. **Page Table Permissions**

   - `PTE = {frame_no, read, write, exec, valid}`

4. **Segmentation**

   - Enforces boundary per logical segment.

🔸 Examples

- Read-only code segments
- No write access to shared libraries
- Stack overflow → segmentation fault

🔸 OS Support

- Linux uses **page-level access control** via `mprotect()` and PTE flags.
- Windows uses **DEP, ASLR** to prevent exploits.

🔸 Trap on Violation

If a process:

- Writes to a read-only page
- Accesses invalid address → MMU triggers **trap** → OS kills process

---

🔷 Demand Paging

✅ What is Demand Paging?

A technique where pages are **not loaded into memory until they are accessed**.

🔸 Process Lifecycle:

1. Process starts → only few essential pages are loaded.
2. Access to a missing page → **page fault**
3. OS fetches page from disk → memory → resume execution

🔸 Benefits

- Reduced startup time
- Better memory utilization
- Support for large address spaces

🔸 Page Fault Handling

```
if page in memory:
    proceed
else:
    trigger page fault
    OS checks validity
    load from disk (or kill if invalid)
    update page table + TLB
```

🔸 Valid-Invalid Bit

Used in Page Table:

Bit	Meaning
1	Page is in memory
0	Page is not loaded

🔸 Performance Factors

- **Page fault rate**
- **Disk I/O latency**
- **Page replacement algorithm**

✅ Summary Table

Concept	Key Component	Benefit
Segmentation	Segment Table	Logical division, flexibility
Paging	Page Table	Removes external fragmentation
Seg+Paging	Segment + Page Tbl	Combines flexibility + mapping
Memory Protection	R/W/X Bits	Security, Isolation
Demand Paging	Page Fault + Disk	Lazy loading, efficient memory

---

✅ Real-World

- **Linux**: Implements demand paging via `mmap()`, `copy-on-write`
- **Java JVM**: Uses segmentation for stack/code/heap layout
- **OS-level exploits**: Segfaults are enforced using protection bits

🧠 7. Virtual Memory

✅ What is Virtual Memory?

Virtual Memory is a memory management technique where processes are given the **illusion of having large contiguous memory**, even if the physical memory is small or fragmented.

🔹 Why Use Virtual Memory?

- **Isolation**: Each process gets its own secure memory space.
- **Efficiency**: Only needed pages are loaded (via demand paging).
- **Flexibility**: Can exceed physical RAM using disk (swap).
- **Simplified memory management** for processes and the OS.

🔸 Virtual Address Space (VAS)

🔹 Definition

The total range of memory addresses a process can use.

- **Logical addresses** generated by CPU.
- **Mapped to physical memory** via page tables and MMU.

🔹 Address Space Layout (Example – Linux x86-64)

Region	Typical Range	Description
Text (code)	0x00400000 – …	Executable instructions
Data	0x00600000 – …	Global/static variables
Heap	0x00800000 – …	`malloc`, `new` allocations
Stack	High → Low	Local variables, function calls
Mapped Files	Dynamic	Libraries, `mmap` regions

🔹 Diagram: Virtual Address Space

```
 +-------------------------+
| Stack (grows downward)  |
+-------------------------+
| Heap (grows upward)     |
+-------------------------+
| Data Segment            |
+-------------------------+
| Text Segment (code)     |
+-------------------------+
| NULL                    |
+-------------------------+
```

🔶 Demand Paging & Page Fault (Revisited)

🔹 Recap: Demand Paging

Only **accessed pages** are loaded into memory on demand.

Mechanism:

- Use **valid-invalid bit** in page table.
- Trigger **page fault** if accessed page is not present.

🔹 Page Fault Handling Steps

1. Trap raised by MMU.
2. OS checks if access is valid.
3. If valid:
   - Load page from disk to memory.
   - Update page table + TLB.
   - Resume process.
4. If invalid:
   - Segmentation fault → kill process.

🔹 Page Fault Rate

Let `p = page fault rate`
Effective access time (EAT):

```
EAT = (1 - p) × MemoryAccessTime + p × PageFaultTime
```

🔹 Example:

Assume:

- Memory access = 100ns
- Page fault (disk) = 10ms
- p = 0.001

```
EAT = 0.999 × 100ns + 0.001 × 10ms
    ≈ 0.0000999 + 0.01 = ~10µs
```

→ Even small `p` significantly increases access time.

🔶 Copy-on-Write (COW)

🔹 What is COW?

A technique used to **optimize memory use when forking** a process.

- Instead of copying all memory, parent and child share **same pages** marked as **read-only**.
- When either process writes to a page → page fault → OS **copies page**.

🔹 Used In:

- `fork()` **+** `exec()` **model**
- Virtual machines (VMs)
- `vfork()`, container engines (Docker)

🔹 Workflow:

1. `fork()` is called → child gets a copy of page table (not memory).
2. Pages are marked read-only.
3. If child or parent writes → **page fault**
4. OS duplicates the page for that process → updates page table

---

🔹 COW Diagram

```
 [Parent]            [Child]
    |                  |
+--+---+           +---+--+
|Page A| <--R/O--> |Page A|
+------+           +------+

→ Write access →
[Page Fault]
→ OS makes a copy

+------+           +------+
|Page A|           |Page A'|
+------+           +------+
```

🔹 Benefits

- Saves memory during fork-heavy operations.
- Improves performance by avoiding unnecessary copies.

🔹 Linux Example

Use `getrusage()` or `/proc/<pid>/status` to monitor `VmSize` vs `VmRSS`.

✅ Summary Table

Concept	Role
Virtual Address Space	Logical memory view for process

Concept	Role
Demand Paging	Lazy loading of memory pages
Page Fault	Triggered when page not in memory
Copy-on-Write	Optimizes memory for shared pages

✅ Real-World Usage

- **COW** used by `fork()` in Unix/Linux.
- Virtual memory enables **container isolation**.
- **Demand paging** improves system startup time.
- TLB + multi-level page tables optimize VAS access.

🔷 Page Replacement Algorithms & Working Set Model

✅ Why Page Replacement?

When a page fault occurs and **no free frame is available**, the OS must **replace an existing page** using a **Page Replacement Algorithm (PRA)**.

Goal: Minimize the number of **page faults** while ensuring good performance.

🔶 Common Page Replacement Algorithms

🔹 1. FIFO (First-In First-Out)

Idea: Remove the **oldest page** loaded into memory.

Implementation:

- Maintain a queue.
- On replacement, evict the page at the **front**.

Example:

Pages: 1 2 3 4 1 2 5
Frames: 3
Faults: 1 2 3 (evict 1) → 4 → evict 2 → 5

Pros: Simple
Cons: Can evict frequently used pages → causes **Belady's anomaly**

🔹 2. LRU (Least Recently Used)

Idea: Evict the **least recently accessed** page.

Implementation:

- Use a stack or timestamps to track access history.

Example:

Pages: 1 2 3 4 1 2 5
Evict page that hasn't been used for the longest time.

Pros: Good approximation of optimal
Cons: Costly to implement in hardware

🔹 3. Optimal (OPT or MIN)

Idea: Evict the page that **won't be used for the longest time in the future**.

Note: Theoretical – requires **future knowledge**.

Example:

Pages: 7 0 1 2 0 3 0 4 2
Frames: 3
→ Replace the page with the **farthest next use**.

Pros: Best performance
Cons: Not implementable in practice

🔹 4. LFU (Least Frequently Used)

Idea: Evict the page with **lowest access count**.

Pros: Captures usage frequency
Cons: Old pages may remain due to initial high usage

Countermeasure: Use aging or decay counters.

🔹 5. Second-Chance (Clock)

Improves upon FIFO using an **"access bit"**.

Mechanism:

- Use a circular queue (clock).
- Each page has a **reference bit**.
- If bit = 0 → replace.
- If bit = 1 → clear and give second chance.

Steps:

1. Clock hand points to page.
2. If bit = 0 → evict.
3. If bit = 1 → bit ← 0, move clock hand.

Pros: Simple and effective approximation of LRU
Cons: Slightly more overhead than FIFO

🔹 Clock Algorithm Diagram

```
Clock:
 [P1]* → [P2] → [P3] → [P4] → ...

Each frame has:
+--------+----------+
| Frame  | Ref Bit  |
+--------+----------+
|  Page1 |    1     |
|  Page2 |    0     | ← Evict this
|  Page3 |    1     |
+--------+----------+
```

✅ Comparison Table

Algorithm	Info Used	Pros	Cons
FIFO	Load time	Simple	Belady’s anomaly
LRU	Last used time	Good approximation	Expensive to implement
Optimal	Future usage	Best page fault rate	Not implementable
LFU	Frequency count	Tracks usage frequency	Stale pages may stay
Second-Chance	Reference bit	Approximates LRU	May rotate often
Clock	Circular version	Efficient and fair	Needs extra bit per frame

🔹 Working Set Model

✅ What is Working Set?

The **working set** of a process is the set of pages **actively used** during a **specific time window**.

🔸 Working Set Window (Δ)

- A fixed number of recent memory references.
- Pages referenced in the last Δ time units = working set.

🔸 Use Case

- Helps OS **decide how many frames** to allocate to a process.
- Helps prevent **thrashing** (frequent page faults).

🔸 Example:

Δ = 10, Recent references:
Pages accessed: 1 2 3 4 1 2 5 1 2 3
→ Working Set = {1, 2, 3, 4, 5}

🔸 Working Set Strategy

- Maintain working set size (WSS) for each process.
- Ensure:

  ```
  Σ WSSi ≤ Total number of frames
  ```

- If not → system is overcommitted → swap out or kill processes.

🔸 Thrashing

When too many processes compete for memory and generate **excessive page faults**.

- CPU utilization ↓
- Disk I/O ↑

✅ Summary

Topic	Purpose
FIFO	Evict oldest
LRU	Evict least recently used
Optimal	Evict based on future access
LFU	Evict least frequently used
Second-Chance	Approximate LRU
Clock	Circular version with ref bits
Working Set Model	Frame allocation + avoid thrashing

🔹 Belady’s Anomaly, Prepaging, TLB Miss Handling

🔹 Belady’s Anomaly

✅ What is Belady’s Anomaly?

Belady's Anomaly refers to the **counter-intuitive situation** where **increasing the number of page frames results in more page faults** in some page replacement algorithms.

🔸 Observed In:

- **FIFO** (First-In-First-Out)
- **Not** observed in LRU or Optimal algorithms

🔸 Example:

Reference string: `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`

Frame Count	FIFO Page Faults
3 frames	9
4 frames	10 ⬅️ more faults!

🔸 Why it happens?

FIFO doesn't consider **page usage patterns**, so adding frames can retain **irrelevant pages** longer, evicting useful ones sooner.

🔸 Visualization:

```
 Reference String: 1 2 3 4 1 2 5 1 2 3 4 5
3 Frames (FIFO):     9 faults
4 Frames (FIFO):    10 faults → Belady's anomaly
```

✅ Solution

Use smarter algorithms like **LRU**, **Optimal**, or **Stack algorithms**, which do not suffer from Belady's anomaly.

🔷 Prepaging

✅ What is Prepaging?

Prepaging is the process of **loading pages into memory before they are requested**, anticipating their use.

🔸 Motivation

- Reduce **page faults** at process start.

- Improve **startup performance** of programs.

🔸 Contrast with Demand Paging

Technique	When Pages Loaded
Demand Paging	When referenced (on fault)
Prepaging	Loaded in advance

🔸 Pros

- Fewer initial page faults
- Better performance for sequential memory access patterns

🔸 Cons

- Wastes memory if preloaded pages are never used
- Overhead if prediction is inaccurate

🔸 Real-World Use

- OS may load adjacent code/data pages when a page fault occurs (heuristics).
- Libraries or executables may trigger prepaging during `exec`.

🔷 TLB Miss Handling

✅ What is a TLB?

TLB (Translation Lookaside Buffer) is a small, fast cache inside the MMU that stores **recent virtual** → **physical page translations**.

🔸 What is a TLB Miss?

A **TLB miss** occurs when a virtual address **is not found** in the TLB. The translation must then be retrieved from the **page table** in main memory.

🔸 Types of Misses

1. **Soft Miss**: Entry found in page table → populate TLB.
2. **Hard Miss**: Entry not in page table → trigger **page fault**.

🔸 TLB Miss Handling Steps:

1. CPU issues virtual address → no match in TLB.
2. OS walks page table to find physical address.
3. TLB is updated with this mapping.
4. Retry instruction.

🔸 TLB Miss Overhead

- Adds delay due to page table access.
- Handled in **hardware** (fast) or **software** (slower).

🔸 TLB Replacement Policies

When TLB is full, OS/hardware uses replacement algorithms:

- **LRU** (common)
- **FIFO**
- **Random**

🔸 Optimization Techniques

- **Larger TLBs** with multiple levels (L1 TLB, L2 TLB)
- **TLB shootdown** for synchronization across cores (e.g., on context switch)
- **ASIDs** (Address Space Identifiers) to reduce TLB flushes during context switch

---

✅ Summary

Topic	Description
Belady's Anomaly	More frames → more faults (only in FIFO-like algorithms)
Prepaging	Load pages in advance to reduce future faults
TLB Miss	When address not found in TLB, access page table instead

---

🧠 8. File Systems

🔹 File Concepts

✅ What is a File?

A **file** is a named collection of related data stored on a non-volatile storage medium (e.g., disk), managed by the operating system.

🔸 File Metadata & Attributes

Metadata is information stored about a file (not the file's content itself):

Attribute	Description
File Name	Human-readable name
File Type	Regular, Directory, Symbolic Link, etc.
Size	Total bytes
Location	Disk block addresses
Owner / Group	User and group ownership
Permissions	Read, Write, Execute flags
Timestamps	Created, Modified, Accessed

🔸 File Types & Extensions

Type	Extension Examples
Text	`.txt`, `.md`, `.csv`
Binary	`.exe`, `.bin`, `.o`
Media	`.mp3`, `.mp4`, `.jpg`
System/Config	`.sys`, `.conf`, `.log`

Note: File extensions are **not enforced by OS**, but used by applications.

🔹 Access Modes

🔸 1. Sequential Access

- Data is accessed **in order**, from beginning to end.
- Most common (e.g., log files, video/audio).

Operations: `read_next()`, `write_next()`

🔸 2. Direct Access (Random Access)

- File can be accessed at **any position** using an offset.
- Used in databases, large datasets.

Operations: `seek(position)`, `read(position)`

🔸 3. Indexed Access

- Uses an **index block** or table to store pointers to blocks.
- Useful when files are too large or fragmented.

✅ Comparison Table

Access Type	Flexibility	Use Case Example
Sequential	Low	Log file, Tape storage
Direct	High	Databases, Video seek
Indexed	Moderate	File system tables

🔹 File Operations

🔸 1. `create()`

- Allocates space for a new file.
- Updates directory table.
- Initializes metadata (timestamps, owner, etc.)

🔸 2. `open(filename, mode)`

- Loads file metadata into memory (Open File Table).
- Sets mode (`r` , `w` , `a` , `rb` , etc.).
- Returns **file descriptor (FD)** or handle.

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

🔸 3. `read(fd, buffer, size)`

- Reads up to `size` bytes from file into `buffer` .
- Returns number of bytes actually read.

```
 char buf[100];
read(fd, buf, 100);
```

🔸 4. `write(fd, buffer, size)`

- Writes `size` bytes from buffer to file.

```
 char data[] = "Hello!";
write(fd, data, strlen(data));
```

🔸 5. `seek(fd, offset, whence)`

- Moves the file pointer to a new position.

```
lseek(fd, 0, SEEK_SET); // Move to beginning
```

🔸 6. `close(fd)`

- Removes file descriptor from process table.
- Updates metadata (e.g., modification time).

🔸 Additional Operations

Operation	Purpose
`delete()`	Remove file entry from directory
`chmod()`	Change permissions
`rename()`	Change file name
`stat()`	Fetch file metadata
`fsync()`	Flush file buffers to disk (for safety)

✅ File Descriptor Table (Per Process)

FD	File
0	stdin
1	stdout
2	stderr
3	user_fd

✅ Real-World

- **Linux/Unix** exposes files as descriptors; everything is a file (including sockets and pipes).
- **High-performance** systems use memory-mapped I/O and async I/O.
- **Buffered I/O** (`fopen` , `fread`) adds performance but requires flushing.

✅ Summary

Concept	Description
File	Collection of data on disk
Metadata	Attributes like size, permissions, times
Access Modes	Sequential, Direct, Indexed
Operations	`open`, `read`, `write`, `seek`, `close`

🔹 File Descriptor Table, Directory Structures, Mounting

🔹 File Descriptor Table

✅ What is a File Descriptor?

A **file descriptor (FD)** is a **non-negative integer** that uniquely identifies an open file within a process.

🔸 How It Works

- Each process maintains its own **File Descriptor Table**.
- The table stores:
  - File descriptor index (0, 1, 2, ...)
  - Pointers to entries in the **System-wide Open File Table**
  - Mode (read/write), file position, access flags

🔸 Standard File Descriptors

FD	Description
0	`stdin`
1	`stdout`
2	`stderr`

🔸 Example (C Code):

```
int fd = open("notes.txt", O_RDONLY);
read(fd, buffer, 100);
close(fd);
```

🔸 Diagram

```
[Per Process FD Table]
+----+-----------+
| FD | File Ptr  |
+----+-----------+
|  0 | → stdin   |
|  1 | → stdout  |
|  2 | → stderr  |
|  3 | → notes.txt (read mode)
+----+-----------+

→ [System-wide Open File Table]
```

🔷 Directory Structure

✅ What is a Directory?

A **directory** is a file that contains references to other files and directories (subdirectories), forming a **hierarchical file system**.

🔶 Types of Directory Structures

🔹 1. Single-Level Directory

- All files are in **one flat namespace**.

Pros: Simple
Cons: Name conflicts, no user separation

Example:

```
/file1
/file2
```

🔹 2. Two-Level Directory

- One directory per user.

Pros: Isolation between users
Cons: Cannot share files easily

Example:

```
/user1/file1
/user2/file2
```

🔹 3. Tree-Structured Directory (Common)

- Hierarchical nesting of directories.

Pros: Organized, scalable
Example:

```
/home/ayush/docs/resume.pdf
```

🔹 4. Acyclic Graph Directory

- Allows **sharing files or subdirectories** using links.

Example: `ln file1 file1_link`

Pros: File sharing
Cons: Needs cycle prevention

🔹 5. General Graph Directory

- **Hard links** can create **cycles**.
- Requires garbage collection or reference counts to delete safely.

✅ Comparison Table

Type	Supports Sharing	Prevents Cycles	Example Use
Single-Level	No	Yes	Early OS
Two-Level	No	Yes	Multi-user
Tree	No	Yes	Linux FS
Acyclic Graph	Yes	Yes	`ln` in UNIX
General Graph	Yes	No	Advanced FS

🔹 Mounting & Unmounting

✅ What is Mounting?

Mounting is the process of **attaching a new file system** (e.g., USB, CD, another disk) to the **existing directory tree**.

🔸 Mount Point

- A **directory** where the new file system is attached.
- Example: Mounting a USB at `/mnt/usb`

```
sudo mount /dev/sdb1 /mnt/usb
```

🔸 Unmounting

- Detaching the file system from the directory hierarchy.
- Ensures data is flushed and no corruption occurs.

```
sudo umount /mnt/usb
```

🔸 Mount Table

The OS maintains a mount table to track:

- Device ID
- Mount point
- File system type (ext4, NTFS, etc.)

🔸 Diagram: Mounting a File System

```
Before:
 /
 ├── home/
 └── var/

After mounting `/dev/sdb1` on `/mnt/usb`:
 /
 ├── home/
 ├── var/
 └── mnt/
      └── usb/  ← contents of new FS
```

🔸 Real-World Examples

- Mounting external drives, ISO files, or remote NFS shares.
- Auto-mounting configured via `/etc/fstab` in Linux.

✅ Summary

Concept	Description
File Descriptor	Index to open file in process table
Directory Structure	Organizes files hierarchically
Mounting	Integrate external file systems into main tree
Unmounting	Safely detach external file systems

🔹 File Allocation, Free Space Management & Inode Structure

🔹 File Allocation Methods

✅ What is File Allocation?

File allocation refers to how blocks on disk are assigned to a file's content.

🔸 Goals

- Efficient disk utilization
- Fast access
- Support for dynamic file growth

🔹 1. Contiguous Allocation

🔸 Description

- Allocate **consecutive blocks** to a file.

🔸 Pros

- Fast **sequential** and **direct access**
- Simple metadata: just base + length

🔸 Cons

- External fragmentation
- Difficult to grow files dynamically

🔸 Example

```
 File A: [Block 10–20]
Access Block 5 → Base + Offset = 10 + 5 = Block 15
```

🔹 2. Linked Allocation

🔸 Description

- Each file block contains a **pointer to the next block**

🔸 Pros

- No external fragmentation
- Easy to grow file

🔸 Cons

- Poor random access
- Pointer overhead per block

🔸 Example

```
Block 10 → Block 55 → Block 73 → NULL
```

🔹 3. Indexed Allocation

🔸 Description

- Use a separate **index block** containing all block addresses of the file.

🔸 Pros

- Fast **random access**
- No fragmentation

🔸 Cons

- Overhead of maintaining index blocks
- Limited file size unless multi-level indexing

🔸 Diagram

```
[File A]
→ Index Block: [12, 34, 56, 78, ...]
→ Each entry points to actual file block
```

✅ Comparison Table

Method	Random Access	Fragmentation	Growth Support	Metadata Overhead
Contiguous	Excellent	High	Poor	Low
Linked	Poor	None	Good	High
Indexed	Good	None	Good	Medium/High

🔹 Free Space Management

✅ What is Free Space Management?

Tracking which disk blocks are **available** for new allocations.

🔹 1. Bitmap (Bit Vector)

- One bit per block:
  - `0` → free
  - `1` → allocated

🔸 Pros

- Fast to check and allocate
- Compact storage

🔸 Cons

- Slower when bitmap is large
- Needs to scan bits to find free blocks

🔸 Example

```
Block Map: [0 1 0 0 1 1 0] → Free blocks at 0, 2, 3, 6
```

🔹 2. Linked List (Free List)

- Maintain a **linked list** of free disk blocks

🔸 Pros

- Easy to implement

🔸 Cons

- Slow to find contiguous space
- Overhead of pointers

🔸 Diagram

```
Free → 7 → 20 → 89 → NULL
```

🔹 3. Grouping

- Store addresses of **n free blocks in the first free block**.
- Next block contains pointers to the next n free blocks, and so on.

🔸 Pros

- Improves allocation time
- Reduces pointer overhead

🔹 Inode Structure (UNIX/Linux)

✅ What is an Inode?

An **inode** is a data structure that stores metadata about a file in Unix-like systems.

🔸 Contents of an Inode

Field	Description
File type	Regular, directory, symbolic link
Permissions	Owner, group, others
Timestamps	Created, modified, accessed
UID / GID	Owner and group
File size	In bytes
Block pointers	Addresses of disk blocks
Link count	Number of hard links

🔸 Inode Block Pointer Structure (EXT)

Index	Type
0–11	Direct blocks
12	Single indirect
13	Double indirect
14	Triple indirect

🔸 Diagram

```
[Inode]
 ├── Direct[0–11] → point directly to data blocks
 ├── Single Indirect → block of pointers → data blocks
 ├── Double Indirect → block → block → data
 └── Triple Indirect → block → block → block → data
```

🔸 Inode Number vs File Name

- File name is stored in **directory**.

- Inode number maps name → inode → file metadata.

✅ Real-World Usage

- UNIX `ls -li`: shows inode number
- File deletion only occurs when link count = 0
- Ext2/3/4, XFS, UFS use inode-based design

✅ Summary

Component	Description
Contiguous	Sequential block allocation
Linked	Blocks point to next
Indexed	Index block contains all addresses
Bitmap	Bit vector for free/used blocks
Inode	Metadata + block pointers for file

🔹 Journaling, Links, and FS Implementation Layers

🔹 Journaling and Crash Recovery

✅ What is Journaling?

Journaling is a mechanism used by modern file systems to **log metadata and/or data changes before applying them**, to ensure **crash consistency**.

🔸 Why Needed?

- Prevents **data corruption** during crashes or power loss.
- Ensures file system can **recover to a consistent state**.

🔸 How It Works

1. FS writes an **intent log (journal)** with the metadata change.
2. If system crashes before the change is committed:
    - **Replay or discard** incomplete operations using the journal.
3. If successful, commit changes to the actual file system.

🔸 Journaling Modes

Mode	Description
Writeback	Only metadata journaled (data may be lost)
Ordered	Metadata + ordered data writes
Journaled	Both metadata and data journaled (safest)

🔸 Real-World FS with Journaling

File System	Journaling Support
ext3/ext4	Yes
NTFS	Yes
XFS	Yes
FAT32	No

🔸 Example Workflow

```
 1. Write "A" to file.txt
2. Log metadata + data to journal
3. Flush journal
4. Apply to disk
5. Mark transaction complete
```

🔸 Crash Recovery Steps

- Read journal at boot time.
- Replay **uncommitted transactions**.
- Discard **partial or corrupt logs**.
- File system is restored to a **consistent state**.

🔹 Symbolic Links vs Hard Links

✅ Links Overview

A **link** provides an alternate name/path to access a file.

🔸 Hard Link

- **Points directly to the inode** of a file.
- Multiple filenames share the **same inode**.

- Changes to one link reflect in all.
- Cannot link to **directories** or across filesystems.

```
ln file1 file2  # Creates a hard link
```

🔸 Symbolic Link (Symlink)

- A special file that **contains a pathname** to another file.
- Works across different file systems.
- Can link to **directories** and non-existent targets.

```
ln -s file1 symlink1
```

✅ Comparison Table

Feature	Hard Link	Symbolic Link
Points to	Inode	File path
Cross-FS allowed	❌ No	✅ Yes
Can link dirs	❌ No	✅ Yes
Affected by move	❌ No	✅ Yes (can break)
File deletion	Removes link; inode survives	Breaks link

🔹 File System Implementation Layers

✅ Overview

A file system is **structured into layers**, each responsible for a specific part of file access and management.

🔹 1. Application Layer

- **User-facing layer**
- Invokes file system calls: `open()`, `read()`, `write()`, etc.
- Interfaces with **C library / libc**

🔹 2. Logical File System

- Handles **file system metadata and naming**
- Maintains file descriptors, inode mappings
- Implements **access control, security, file structure**

🔹 3. File-Organization Module

- Maps logical files to **physical blocks**
- Implements **block allocation**, free space management
- Handles indexed/contiguous/linked file layout

🔹 4. Basic File System

- Handles **actual I/O** between disk and OS
- Caches blocks, issues **read/write requests**
- Manages **buffers** and queues

🔹 5. I/O Control

- Includes **device drivers** and **interrupt handlers**
- Talks directly to disk hardware (controller, DMA)
- Responsible for physical **block I/O operations**

🔸 Diagram: File System Layered Architecture

```
 +----------------------+
|  Application Layer   | ← open(), read(), write()
+----------------------+
| Logical File System  | ← inode, permissions
+----------------------+
| File Organization    | ← block allocation, indexing
+----------------------+
| Basic File System    | ← block cache, scheduling
+----------------------+
| I/O Control Layer    | ← device driver, disk access
+----------------------+
| Disk Hardware        |
+----------------------+
```

✅ Summary

Component	Role
Journaling	Ensures consistency during crashes
Hard Link	Alias to same inode; no file path used
Symbolic Link	Pointer to filename; allows dir/cross-FS links
FS Layers	Structured abstraction from app to disk hardware

🧠 9. I/O Systems

🔹 I/O Hardware

✅ Key Components

Component	Description
I/O Devices	Input/Output units (keyboard, disk, printer)
Device Controller	Manages specific hardware device
Device Driver	OS software to control the controller
Registers	Control, status, and data registers
Interrupt Lines	Notify CPU of events
Data Bus	Transfers data between memory and devices

🔸 I/O Types

- **Block Devices**: transfer blocks of data (e.g., disk)
- **Character Devices**: stream data byte-by-byte (e.g., keyboard)

🔹 Polling vs Interrupt Driven I/O

🔸 Polling

- CPU repeatedly checks device status.

Pros:

- Simple to implement
Cons:
- Wastes CPU cycles
- Poor performance under load

🔸 Interrupt Driven I/O

- Device raises an **interrupt** to signal readiness.

Pros:

- Efficient CPU utilization
- Faster response to I/O
Cons:
- Requires interrupt handling logic

✅ Comparison

Feature	Polling	Interrupt-Driven
CPU usage	High (busy wait)	Low
Latency	High	Low
Implementation	Easy	Requires handlers
Usage	Simple devices	Real-time, disk, network

🔹 Interrupt Handling

✅ Process

1. Device sends **interrupt signal**.
2. CPU stops current task, saves context.
3. Jumps to **Interrupt Service Routine (ISR)**.
4. ISR handles device I/O (read/write).
5. CPU resumes previous task.

🔸 Nested Interrupts

- Higher-priority interrupts can interrupt an ISR.

🔸 Software Interrupts

- Generated by software (e.g., `int 0x80` syscall in Linux)

🔹 DMA (Direct Memory Access)

✅ What is DMA?

DMA allows devices to **transfer data directly to/from memory** without involving the CPU for each byte.

🔸 How It Works

1. CPU configures DMA controller.
2. DMA controller takes over the bus.
3. Performs block transfer between device ↔ memory.
4. DMA raises an **interrupt** on completion.

🔸 Pros

- Faster than programmed I/O
- Frees CPU to perform other tasks

🔸 Diagram

```
Device ↔ DMA Controller ↔ Main Memory
              ↑
       Configured by CPU
```

🔹 I/O Scheduling

✅ Goal

Optimize disk head movement to **minimize seek time** and improve **throughput**.

🔹 Disk Scheduling Algorithms

🔸 1. FCFS (First-Come, First-Served)

- Requests served in arrival order.

Pros: Simple
Cons: Poor performance

🔸 2. SSTF (Shortest Seek Time First)

- Selects request with **minimum seek distance**.

Pros: Improves average seek time
Cons: Can cause starvation

🔸 3. SCAN (Elevator Algorithm)

- Head moves in one direction, serves requests along the way.
- Reverses at the end.

Pros: Fair and efficient
Cons: Longer delay for end tracks

🔸 4. C-SCAN (Circular SCAN)

- Like SCAN but always **moves in one direction**.

- After reaching end, jumps back to start.

Pros: Uniform wait time
Cons: Slightly higher head movement

🔸 5. LOOK

- Like SCAN but **reverses direction early** (at last request).

🔸 6. C-LOOK

- Like C-SCAN but **wraps around only to the next request** instead of full jump.

✅ Gantt Chart Example (SSTF)

Requests: 40, 10, 22, 30, 5
Initial Head: 20

```
20 → 22 → 30 → 10 → 5 → 40
```

✅ Comparison Table

Algorithm	Directional?	Starvation?	Good For
FCFS	No	No	Low-load systems
SSTF	No	Yes	Minimizing seek time
SCAN	Yes	No	Balanced workloads
C-SCAN	Yes	No	Uniform wait time
LOOK	Yes	No	Shorter head travel
C-LOOK	Yes	No	Circular but efficient

✅ Summary

Concept	Description
I/O Hardware	Devices, controllers, registers, drivers
Polling	CPU checks device repeatedly
Interrupts	Device alerts CPU asynchronously
DMA	Device ↔ memory transfers w/o CPU

Concept	Description
I/O Scheduling	Algorithms to optimize disk seek operations

🔷 Disk Structure, RAID, Buffering, and Spooling

🔷 Disk Structure

✅ Components

Component	Description
Platters	Circular disks that store data magnetically
Tracks	Concentric circles on platters where data is stored
Sectors	Subdivisions of tracks (e.g., 512 bytes)
Cylinders	Set of tracks vertically aligned across platters
Head	Reads/writes data from/to platters
Arm	Moves the head across tracks

🔸 Diagram

```
 Side View of Disk:
+-------------------------+
| Platter 1 (Track/Sector) |
| Platter 2 (Track/Sector) |
+-------------------------+

Top View:
+-------------------------+
| Track                   |
|   +-- Sector            |
|       +-- Data Block    |
+-------------------------+

Cylinder = Set of aligned tracks across platters
```

🔸 Access Time = Seek + Rotational + Transfer

- **Seek Time**: Move head to correct track
- **Rotational Latency**: Wait for sector to rotate under head
- **Transfer Time**: Time to read/write the data

🔷 RAID Levels (0–6)

✅ What is RAID?

RAID (Redundant Array of Independent Disks) is a method to combine multiple disks for **performance, redundancy**, or both.

🔸 RAID Levels Overview

Level	Description	Redundancy	Performance	Min Disks
RAID 0	Striping	❌ No	✅ High	2
RAID 1	Mirroring	✅ Yes	Read ↑	2
RAID 2	Bit-level striping with ECC	✅ Rare	✅	Not used
RAID 3	Byte-level striping + parity	✅ Yes	Medium	≥3
RAID 4	Block-level striping + parity	✅ Yes	Medium	≥3
RAID 5	Block-level striping + dist. parity	✅ Yes	✅ High	≥3
RAID 6	RAID 5 + dual parity	✅ High	Slight ↓	≥4

🔸 Key Examples

- **RAID 0**: Fastest, no fault tolerance
- **RAID 1**: Safe, expensive (2× storage)
- **RAID 5**: Balanced (performance + fault tolerance)
- **RAID 6**: Can survive 2 disk failures

🔸 RAID Diagram: RAID 5

```
 Disk1: A1  A2  Parity(P1)
Disk2: A3  P2  A4
Disk3: P3  A5  A6
```

🔹 Buffering and Caching

✅ Buffering

A **buffer** is a temporary memory used to **hold data during transfer** between devices and processes.

- Helps **match speed difference** (e.g., disk ↔ RAM)
- May store input/output data temporarily
- Examples: Keyboard buffer, disk write buffer

✅ Caching

A **cache** stores frequently used data for **faster access**.

- Used to avoid repeated slow I/O operations
- Caching can be at:
  - File system level
  - Block level
  - Hardware level (disk cache)

🔸 Buffering vs Caching

Feature	Buffering	Caching
Purpose	Smooth data flow	Speed up access
Data usage	Used once, then discarded	Frequently reused
Scope	I/O pipelines	Memory–disk, CPU–RAM, etc.

🔷 Spooling (Simultaneous Peripheral Operations Online)

✅ What is Spooling?

Spooling is a technique where **I/O requests are queued** in secondary storage (usually disk), allowing the device to serve one job at a time.

🔸 Real-World Example

- **Printing**: Print jobs are stored in a spool directory; the printer processes them one-by-one.

🔸 Benefits

- **Asynchronous**: CPU doesn't wait for I/O
- Supports **multiprogramming**
- Allows **efficient device usage**

🔸 Difference from Buffering

Feature	Buffering	Spooling
Scope	Temporary memory	Disk (persistent)
Queueing	Typically not queued	Jobs queued

Feature	Buffering	Spooling
Example	Keyboard input	Printer jobs

✅ Summary

Topic	Key Concepts
Disk Structure	Platters, Tracks, Sectors, Cylinders
RAID	Data redundancy & performance (RAID 0–6)
Buffering	Temp memory to handle device speed mismatch
Caching	Memory for fast access of frequently used data
Spooling	Queuing I/O jobs (e.g., print) in disk for async ops

---

🧠 10. Security & Protection

🔹 Security Goals

✅ What is Security in OS?

Security in an OS refers to **protecting data and system resources** from unauthorized access, misuse, or compromise.

🔸 Three Core Goals (CIA Triad)

Goal	Description
Confidentiality	Ensuring data is accessed only by authorized users
Integrity	Ensuring data is **not altered** by unauthorized users or programs
Availability	Ensuring resources/services are **available** to authorized users when needed

---

🔸 Other Extended Goals

- **Authenticity**: Verifying identities
- **Accountability**: Tracking actions of users (audit logs)

🔹 Access Control

✅ What is Access Control?

Access control determines **who can access what** resources, and **what operations** they are allowed to perform.

🔸 Access Control Matrix (ACM)

A logical table showing permissions:

	File1	File2	Printer
UserA	R/W	R	No
UserB	No	R/W	W

🔸 Access Control List (ACL)

- Stored **per object** (e.g., file)
- List of users and their permissions

```
# Sample ACL for file.txt
UserA: read, write
UserB: read
```

🔸 Example (Linux)

```
setfacl -m u:john:rw file.txt
getfacl file.txt
```

🔸 Capability List

- Stored **per subject** (e.g., process/user)
- List of objects and permissions accessible to that subject

```
UserA: [File1: R/W, File3: R, Printer: W]
```

✅ ACL vs Capability List

Feature	ACL	Capability List
Stored With	Object	Subject (User/Process)
Easy to Audit	Who can access a file	What can a user access
Revoking Access	Easier	Harder

🔹 Authentication vs Authorization

🔸 Authentication

Who are you?

- Verifies the **identity** of the user
- Methods:
  - Username/password
  - Biometrics
  - OTP / 2FA
  - Certificates (X.509)

```
Login → "Ayush" + correct password → Access granted
```

🔸 Authorization

What can you do?

- Defines the **permissions** assigned to an authenticated user
- Happens **after authentication**

```
User "Ayush" → Authorized to read/write file1 but only read file2
```

✅ Comparison Table

Feature	Authentication	Authorization
Purpose	Identity verification	Permission validation
When	First step	After authentication
Methods	Password, biometric	ACLs, Role-based permissions
Example	Login screen	File access control

✅ Summary

Concept	Description
CIA Triad	Confidentiality, Integrity, Availability
ACL	Object-level access control
Capability List	Subject-level access permissions
Authentication	Verifying identity

Concept	Description
Authorization	Checking what an identity is allowed to do

◆ Protection Rings, Unix Permissions, and Malware

◆ Protection Rings

✅ What are Protection Rings?

Protection rings are hierarchical levels of privilege in which code can execute, primarily used in CPU and OS architecture to enforce security and isolation.

◆ Ring Levels

Ring	Name	Privilege Level	Example
0	Kernel Mode	Highest (full)	OS Kernel, Drivers
1	OS Services	Medium-High	Optional - OS modules
2	I/O Drivers	Medium	Some drivers, services
3	User Mode	Lowest	Applications, user programs

Most modern OSes use only **Ring 0** and **Ring 3**.

◆ Diagram

```
 +---------------------+
|  Ring 3: User Apps  |
+---------------------+
|  Ring 2: Drivers    |
+---------------------+
|  Ring 1: Services   |
+---------------------+
|  Ring 0: Kernel     |
+---------------------+
```

◆ Domain of Protection

✅ What is a Protection Domain?

A **domain** defines a set of **resources** and the **access rights** a process has over those resources.

- Each process operates in a **domain**
- Resources include: files, memory, I/O, CPU, etc.
- OS can **switch domains** on context switch

◆ Example

Domain	Allowed Actions
User Mode	Read user files, open programs
Admin Mode	Install apps, access system files
Kernel Mode	Access hardware, schedule tasks

◆ Unix File Permissions

✅ Permission Types

Each file has 3 categories of users and 3 permissions:

Category	Description
Owner	Creator of file
Group	Assigned group
Others	Everyone else

Permission	Symbol	Numeric
Read	r	4
Write	w	2
Execute	x	1

◆ Example

```
chmod 755 file.txt
```

- Owner: 7 → rwx
- Group: 5 → r-x
- Others: 5 → r-x

```
ls -l file.txt
-rwxr-xr-x 1 user user  23 Jul  file.txt
```

◆ `umask`

- Sets **default permission mask** when creating files.
- Common default: `umask 022` → files get `644`, dirs get `755`

🔹 Special Permission Bits

Bit	Name	Use Case
`suid`	Set User ID	Run file with file owner's privileges
`sgid`	Set Group ID	Inherit group ID of directory
`sticky`	Sticky Bit	Only owner can delete file in shared dir

```
 chmod u+s file      # sets suid
chmod g+s directory # sets sgid
chmod +t /tmp       # sticky bit
```

🔹 Malware Types

🔸 Trojan Horse

- A **disguised malicious program** that appears useful but compromises system security when run.

Example: Game that secretly sends your credentials

🔸 Virus

- **Self-replicating code** that attaches to other files/programs.
- Needs a **host file** and **manual execution** to spread.

🔸 Worm

- **Self-replicating program** that spreads across networks without a host file or user intervention.

Example: WannaCry ransomware worm

🔸 Rootkit

- Malware designed to **gain and hide privileged access** to the system.

Can:

- Replace system binaries
- Hook into kernel functions
- Hide files, processes, users

✅ Malware Comparison Table

Type	Needs Host	Spreads Automatically	Privilege Escalation	Detection Difficulty
Trojan	Yes	No	Possible	Medium
Virus	Yes	No	Possible	Medium
Worm	No	Yes	Yes	High
Rootkit	No	Yes (after infection)	Yes (Ring 0)	Very High

✅ Summary

Concept	Description
Protection Rings	Hardware-enforced privilege levels
Protection Domain	Access rights associated with a process
Unix Permissions	Read/write/execute controls via chmod, umask
suid/sgid/sticky	Special execution or delete permissions
Malware Types	Trojan, virus, worm, and rootkits

🔷 Encryption & Secure OS Design

🔷 Encryption Techniques

✅ What is Encryption?

Encryption is the process of converting **plaintext into ciphertext** using a key, so only authorized users can decrypt it back to the original message.

🔸 1. Symmetric Key Encryption

- **Same key** is used for both encryption and decryption.
- Fast, efficient, used for **bulk data encryption**

🔸 Examples

- AES (Advanced Encryption Standard)
- DES (Data Encryption Standard)
- RC4, Blowfish

🔸 Diagram

```
Plaintext --(Key)--> Ciphertext --(Key)--> Plaintext
```

🔸 Pros

- Fast, less computation
- Simple key management (if securely shared)

🔸 Cons

- Requires **secure key distribution**
- Not scalable for large user bases

🔸 2. Asymmetric Key Encryption

- Uses a **pair of keys**: Public Key + Private Key
- Data encrypted with one key can only be decrypted with the other

🔸 Examples

- RSA, ECC (Elliptic Curve), Diffie-Hellman

🔸 Diagram

```
Message → Encrypted with Public Key → Only Private Key can decrypt
```

🔸 Uses

- Secure key exchange
- Digital signatures
- SSL/TLS (HTTPS)

✅ Comparison Table

Feature	Symmetric	Asymmetric
Keys Used	Single shared key	Public/Private key pair
Speed	Fast	Slower
Use Case	Bulk encryption	Key exchange, digital sign
Key Distribution	Difficult	Easy (public key shared)

🔷 Secure OS Design

✅ What is Secure OS Design?

Designing an operating system with **built-in security policies**, **access control**, and **audit mechanisms** to reduce vulnerabilities.

🔹 SELinux (Security-Enhanced Linux)

- Developed by NSA and Red Hat
- Uses **Mandatory Access Control (MAC)** model
- Enforces strict policies on files, processes, sockets

🔸 Key Features of SELinux

Feature	Description
Type Enforcement	Access allowed based on subject & object types
Role-Based Access	Maps users to roles with limited privileges
MAC Enforcement	Access decisions based on security policy
Labeling	Each file/process has a security label

🔸 Example SELinux Policy

```
allow httpd_t httpd_sys_content_t : file { read getattr open };
```

- Allows Apache (`httpd_t`) to read content labeled `httpd_sys_content_t`

🔹 Other Secure OS Features

OS Feature	Description
AppArmor	Path-based MAC framework for Linux
TrustedBSD	MAC framework used in FreeBSD
TPM Integration	Hardware-based root of trust
Sandboxing	Restrict app/system calls (e.g., Chrome sandbox)
Capability System	Fine-grained privilege separation

🔸 Secure Boot and UEFI

- Prevents loading unsigned OS or kernel modules
- Ensures system boots from a **trusted source only**

✅ Summary

Topic	Description
Symmetric Crypto	One key used for both encryption/decryption
Asymmetric Crypto	Uses key pairs; public and private
SELinux	Linux MAC model enforcing strong security policies
Secure OS Design	Includes sandboxing, TPM, secure boot, audit logs

---

🧠 11. Deadlocks (In-Depth)

🔷 Four Coffman Conditions

✅ What is a Deadlock?

A **deadlock** occurs when a group of processes are **each waiting for resources** that are held by other processes in the group, resulting in a **circular wait** and **no progress**.

🔸 Coffman Conditions (All Must Hold)

Condition	Description
1. Mutual Exclusion	At least one resource must be held in a **non-shareable** mode
2. Hold and Wait	A process holds at least one resource and is **waiting** for more
3. No Preemption	Resources cannot be forcibly removed from a process
4. Circular Wait	A cycle of processes exists, each waiting for a resource held by the next

If **all four** conditions are satisfied → **deadlock is possible**.

🔸 Diagram

```
 P1 → holds R1 → waiting for R2
P2 → holds R2 → waiting for R3
P3 → holds R3 → waiting for R1
(Circular Wait → Deadlock)
```

🔷 Wait-For Graph (WFG)

✅ What is a Wait-For Graph?

A **Wait-For Graph** is used to model deadlocks among processes only.

- **Nodes**: Processes
- **Edges**: P1 → P2 means P1 is **waiting for** a resource held by P2

🔸 Deadlock Detection

- If WFG has a **cycle** → deadlock exists
- Used in systems where **resource request is dynamic**

🔸 Example

```
P1 → P2 → P3 → P1
⇒ Cycle → Deadlock
```

🔷 Resource Allocation Graph (RAG)

✅ What is RAG?

A graph model showing:

- **Processes** (circles)
- **Resources** (squares)
- Edges:
  - **Request edge** (→): Process → Resource (wants)
  - **Assignment edge** (←): Resource → Process (allocated)

🔸 RAG Symbols

Symbol	Meaning
P → R	Process P is requesting resource R
R → P	Resource R is allocated to P

🔸 Cycle Detection in RAG

- **Cycle without multiple instances**: **Deadlock exists**
- **Cycle with multiple instances**: **May or may not** be a deadlock

🔸 Example: RAG with Deadlock

```
P1 → R1 → P2 → R2 → P1
```

- Cycle exists → Deadlock possible

🔸 Gantt Chart Analogy (Optional)

Gantt charts are not typically used for deadlock detection but can be used to simulate resource allocation order and delay patterns due to blocking/waiting.

✅ Summary

Concept	Description
Coffman Conditions	4 necessary conditions for deadlock
Wait-For Graph	Detect cycles among processes
RAG	Models process-resource relationships
Cycle Detection	Cycle ⇒ Deadlock (single instance case)

🔹 Deadlock Prevention, Avoidance, and Recovery

🔹 Deadlock Prevention Techniques

✅ Goal

To **ensure at least one** of the four Coffman conditions **does not hold**, thereby **preventing deadlock**.

🔸 1. Hold and Wait Elimination

- Require processes to **request all resources at once** before execution begins.

Pros: Eliminates hold-and-wait
Cons: Leads to **low resource utilization** and potential **starvation**

🔸 2. Preemption

- If a process holding resources is blocked, forcibly **preempt** its resources and assign to others.

Pros: Avoids indefinite waiting
Cons: Complex implementation; not all resources are preemptible (e.g., printer)

🔸 3. Circular Wait Prevention (Resource Ordering)

- Impose a **global ordering** of resources. Processes must request resources in this order.

```
Order: R1 < R2 < R3

Valid: Request R1, then R2
Invalid: Request R2, then R1
```

Pros: Simple and practical
Cons: Needs careful resource classification

🔷 Deadlock Avoidance: Banker's Algorithm

✅ What is Banker's Algorithm?

A **deadlock avoidance** algorithm proposed by Dijkstra, which checks if granting a resource will leave the system in a **safe state**.

Used in systems where the **maximum resource need is known in advance**.

🔸 Data Structures

Structure	Meaning
`Available[]`	Number of available instances per resource
`Max[][]`	Max demand of each process
`Allocation[][]`	Resources currently allocated
`Need[][]`	`Max - Allocation`

🔸 Safety Algorithm Steps

1. Initialize `Work = Available`, `Finish[i] = false`
2. Find a process `P_i` such that `Need_i ≤ Work` and `Finish[i] = false`
3. If found, do:
    - `Work = Work + Allocation[i]`
    - `Finish[i] = true`
    - Repeat step 2
4. If all `Finish[i] = true` → **Safe state**
5. Else → **Unsafe (potential deadlock)**

🔸 Example

Given:

```
Available = [3, 3, 2]
Max       = [[7,5,3], [3,2,2], [9,0,2], [2,2,2], [4,3,3]]
Alloc     = [[0,1,0], [2,0,0], [3,0,2], [2,1,1], [0,0,2]]
```

Check if the system is in a safe state.

(Safety check performed as per the algorithm above)

🔹 Deadlock Recovery

✅ Used When:

- Deadlock **not prevented or avoided**
- Detected using Wait-For Graph
- Need to **recover system to usable state**

🔸 1. Process Termination

- **Kill All**: Terminate all processes in deadlock
- **Kill One-by-One**: Until deadlock is resolved (select based on priority, CPU time, resources held)

🔸 2. Resource Preemption

- Select victim process
- Preempt its resources
- Rollback and resume later

Requires: Rollback support and consistent state tracking

🔸 Trade-offs

Method	Pros	Cons
Kill All	Easy to implement	High cost, user data loss
Kill One-by-One	Less disruptive	Needs decision-making heuristics
Preemption	Reuses resources	Complex to implement safely

✅ Summary

Technique	Description
Prevention	Deny one Coffman condition

Technique	Description
Banker’s Algorithm	Grants only if resulting state is safe
Recovery	Kill or preempt processes to break deadlock

🧠 12. Linux Internals / System Programming

🔹 Process Creation and Management

✅ `fork()`

- Used to **create a new process** by duplicating the calling process.
- Returns:
  - `0` to the **child**
  - `PID` of child to the **parent**
  - `-1` on failure

🔸 **Example**

```
pid_t pid = fork();
if (pid == 0)
    printf("Child Process\n");
else if (pid > 0)
    printf("Parent Process\n");
```

✅ `exec()` Family

- Replaces current process image with **new program**
- Variants: `execl()`, `execv()`, `execvp()`, `execle()`

🔸 **Example**

```
execl("/bin/ls", "ls", "-l", NULL);
```

> After `exec()`, **previous code does not run** if successful.

✅ `wait()` and `waitpid()`

- Makes parent wait for **child to terminate**
- Returns **PID of child**, sets `exit_status`

```
int status;
pid_t pid = wait(&status);
```

✅ `exit()`

- Terminates the process, returning **status to parent**

```
exit(0);
```

🔹 Process ID Functions

✅ `getpid()` and `getppid()`

Function	Description
`getpid()`	Get **process ID**
`getppid()`	Get **parent process ID**

🔸 Example

```
printf("PID: %d, PPID: %d\n", getpid(), getppid());
```

🔹 Signal Handling

✅ `kill()`

- Sends a **signal** to a process

```
kill(pid, SIGTERM);  // send termination signal
```

✅ `signal()`

- Sets a **simple signal handler**

```
void handler(int signum) {
    printf("Caught signal %d\n", signum);
}
signal(SIGINT, handler);
```

✅ `sigaction()`

- Advanced alternative to `signal()`
- Supports flags and `siginfo_t`

```
struct sigaction sa;
sa.sa_handler = handler;
sigaction(SIGINT, &sa, NULL);
```

🔹 Nice Values: `nice()` and `renice()`

✅ `nice()`

- Sets the **initial priority (niceness)** of a process
- Lower value → higher priority

```
nice(10);  // make process lower priority
```

✅ `renice`

- Change priority of a **running process**

```
renice -n 5 -p 1234  # increase nice value for PID 1234
```

Range	Description
-20	Highest priority
19	Lowest priority

🔹 `/proc` and `/sys` File Systems

✅ `/proc`

- Virtual filesystem exposing **process and kernel info**
- Exists **in-memory**, not on disk

🔸 Common Paths

Path	Meaning
`/proc/<pid>/`	Info for a specific process
`/proc/cpuinfo`	CPU details
`/proc/meminfo`	Memory usage
`/proc/stat`	CPU statistics
`/proc/filesystems`	Supported FS types

🔸 Example

```
 cat /proc/uptime
cat /proc/self/status
```

✅ `/sys`

- Exports **kernel objects** (kobjects) and hardware details
- Used by **udev** and drivers

🔸 Example Paths

Path	Description
`/sys/class/`	Devices grouped by class
`/sys/block/`	Block devices (disks)
`/sys/devices/`	Physical hardware devices

✅ Summary

Concept	Description
`fork()`	Creates child process
`exec()`	Replaces process image
`wait()`	Parent waits for child to terminate
`kill()`, `signal()`	Signal handling and delivery
`nice()` / `renice()`	Process priority control
`/proc`, `/sys`	Virtual FS exposing runtime system metadata

🔹 Linux Processes, Monitoring Tools, and Scripting

🔹 Zombie vs Orphan Processes

✅ Zombie Process

- A process that has **terminated**, but **parent hasn't called** `wait()` to collect its exit status.
- Occupies entry in **process table** (PID stays).

🔸 Characteristics

Attribute	Value
State	`Z` (zombie)
Cleanup needed	By parent (`wait()`)
Resource usage	Minimal

```
ps aux | grep Z
```

🔸 Fix

- Parent should call `wait()`
- If parent terminates, **init (PID 1)** cleans up zombie

✅ Orphan Process

- A **child process whose parent has exited**.
- OS automatically assigns its parent to **init (PID 1)**.

🔸 Characteristics

Attribute	Value
Adopted by	`init` or `systemd`
Cleanup	Proper via `init`
Harmful?	❌ Usually safe

🔷 Monitoring Tools

✅ `strace`

- Traces **system calls** made by a program.

```
strace ./a.out
```

✅ `lsof`

- Lists **open files** by processes.

```
lsof -i :8080     # show processes using port 8080
```

✅ `top`

- Interactive real-time **process monitor**.

```
top
```

✅ `vmstat`

- Displays **virtual memory statistics**.

```
vmstat 1 5        # sample every 1s, 5 times
```

✅ `free`

- Shows **memory usage**.

```
free -h
```

✅ `iostat`

- Shows **CPU and I/O usage**.

```
iostat -xz 1
```

🔹 Bonus: Shell Implementation (Mini Project)

✅ Components of a Shell

1. **Prompt**: Show command line prompt
2. **Input**: Read user input using `getline()` or `fgets()`
3. **Parse**: Tokenize input using `strtok()`
4. **Fork & Exec**: Create child and run command
5. **Wait**: Parent waits for child

🔸 Sample C Implementation

```
while (1) {
    printf("shell> ");
    fgets(input, 1024, stdin);
    pid_t pid = fork();
    if (pid == 0) {
        execlp(token[0], token[0], NULL);
        exit(1);
    } else {
        wait(NULL);
    }
}
```

🔹 Bash Scripting Basics

✅ Shebang

```
#!/bin/bash
```

✅ Variables

```
name="Ayush"
echo "Hello, $name"
```

✅ Conditionals

```
if [ $age -gt 18 ]; then
  echo "Adult"
fi
```

✅ Loops

```
for i in 1 2 3; do
  echo $i
done
```

✅ Functions

```
greet() {
  echo "Hi, $1"
}
greet "Ayush"
```

✅ Arguments

```
./script.sh arg1 arg2

echo $1  # arg1
echo $2  # arg2
```

✅ Useful Built-ins

Command	Purpose
`read`	Read input from user
`basename`	Extract filename from path
`dirname`	Extract directory name
`source`	Run a script in current shell
`trap`	Catch signals

✅ Summary

Topic	Description
Zombie Process	Dead process not waited for by parent
Orphan Process	Child whose parent has died (reassigned to init)
strace/lsof/top	Monitoring tools
Shell Impl	Fork, parse, execute commands
Bash Basics	Variables, loops, conditionals, functions

🧠 13. Advanced Topics

🔹 Memory-Mapped I/O vs Port-Mapped I/O

✅ Memory-Mapped I/O (MMIO)

- I/O devices are mapped into **same address space** as memory.
- CPU uses **regular load/store instructions** to communicate with devices.

🔸 Example

```
#define LED_REGISTER *((volatile int*) 0xFF203020)
LED_REGISTER = 1; // turn LED on
```

✅ Port-Mapped I/O (PMIO)

- Uses **separate I/O address space**.
- Accessed via special CPU instructions (e.g., `IN`, `OUT` on x86)

```
OUT 0x64, AL    ; send AL to keyboard controller
```

✅ Comparison Table

Feature	Memory-Mapped I/O	Port-Mapped I/O
Address Space	Shared with memory	Separate I/O space
Access	Load/Store instructions	IN/OUT instructions
Portability	Higher	Lower (x86 only)

Feature	Memory-Mapped I/O	Port-Mapped I/O
Performance	Faster with caching	Slower due to limited space

🔹 NUMA (Non-Uniform Memory Access)

✅ What is NUMA?

- Architecture where a multi-core CPU has **multiple memory regions** with **different access times**.
- Each CPU is closer (faster access) to its own local memory.

🔸 Diagram

```
 +---------+    +--------------------+
| CPU 0   |----| Memory Node 0 (fast)
+---------+    +--------------------+
     |
     | (slow)
     ↓
+---------+    +--------------------+
| CPU 1   |----| Memory Node 1 (fast)
+---------+    +--------------------+
```

🔸 Benefits

- Improves scalability on multi-socket systems.
- Reduces memory latency with **CPU-local memory access**.

🔸 OS NUMA Awareness

- Linux uses `numactl` to control memory affinity.
- Schedulers try to place tasks near their memory.

```
numactl --cpunodebind=0 --membind=0 ./a.out
```

🔹 Kernel Preemption and Latency

✅ Kernel Preemption

- The ability to **preempt (interrupt)** a running kernel task to schedule something else.
- Important for **low latency**, **real-time** systems.

🔸 Types of Kernels

Type	Description
Non-Preemptive	Kernel runs to completion
Preemptive	Allows context switches during kernel execution
RT Preempt Patch	Full preemptibility with bounded latency

🔸 Latency Considerations

- **Latency** is time between event and its processing.
- Important for **real-time audio**, robotics, etc.

🔸 Tools

```
latencytop          # Monitor sources of latency
cat /proc/sys/kernel/preempt_max_latency_us
```

🔹 Multi-Core CPU Scheduling

✅ Challenges

- Cache affinity: Avoid moving processes across cores (costly)
- Load balancing: Evenly distribute tasks
- Synchronization: Protect shared data with minimal overhead

🔸 Linux CPU Scheduling Policies

Policy	Description
CFS	Completely Fair Scheduler (default)
FIFO	Real-time: strict ordering
RR	Round-robin real-time
SCHED_DEADLINE	For periodic real-time tasks

🔸 Processor Affinity

- Pin processes to specific cores using `taskset`

```
taskset -c 0 ./my_app
```

- Kernel uses **load balancing** between CPUs
- NUMA-aware scheduling also applies

✅ Summary

Concept	Description
MMIO vs PMIO	Device access via memory vs special I/O ports
NUMA	CPU-memory locality model for performance
Kernel Preemption	Interrupt kernel tasks for lower latency
Multi-Core Scheduling	Efficient task handling across cores & caches

---

🔹 Advanced OS Concepts (Part 2)

🔹 Cache Coherency and False Sharing

✅ Cache Coherency

In multi-core CPUs, **each core has its own cache**. Cache coherency ensures that **all cores see the same value for a shared variable**.

🔸 Example Issue

- Core A updates `x = 5`, Core B sees `x = 2` (old value)
- Fixed by **cache coherence protocols** (e.g., MESI)

🔸 Coherence Protocols

Protocol	Description
MESI	Modified, Exclusive, Shared, Invalid
MOESI	Adds Owned state to MESI
MSI	Simpler, lacks exclusive/owned

✅ False Sharing

Occurs when **multiple threads modify different variables** that lie on the **same cache line**, causing unnecessary cache invalidations.

🔸 Example

```
struct {
    int a; // Thread 1
```

```
    int b; // Thread 2
} s;
```

Even though `a` and `b` are independent, writing to them on different threads can cause **false sharing**.

🔸 Fix

- **Padding** structure members to separate cache lines
- Use `alignas(64)` or compiler-specific directives

🔹 Lock-Free and Wait-Free Data Structures

✅ Lock-Free

- Guarantees **system as a whole makes progress**
- Uses atomic operations like `CAS (Compare-And-Swap)`

✅ Wait-Free

- Stronger: Guarantees **every thread makes progress** in finite time

---

🔸 Use Cases

- High-performance queues, stacks, hash tables
- Used in **real-time** or **low-latency** environments

🔸 Code Snippet (Lock-Free Stack Push)

```
void push(int val) {
    Node* newNode = new Node(val);
    do {
        newNode->next = top;
    } while (!CAS(&top, newNode->next, newNode));
}
```

🔹 Page Coloring

✅ What is Page Coloring?

Technique to control **which physical pages are mapped** to virtual addresses to **minimize cache conflicts**.

🔸 Why?

- Cache is set-associative → multiple virtual pages can map to same cache set
- OS uses "color" metadata to **distribute pages uniformly across cache sets**

🔸 Benefit

- Improves cache hit rate
- Reduces performance degradation due to conflict misses

🔷 IOMMU (Input-Output Memory Management Unit)

✅ What is IOMMU?

- Manages memory access for **DMA-capable devices**
- Translates **device-visible addresses** → physical memory

🔸 Benefits

- **Memory protection** from faulty devices
- Enables **device pass-through** in virtualization
- Reduces need for bounce buffers

🔸 Used in

- Virtualization (e.g., QEMU/KVM with VFIO)
- High-speed NICs, GPUs

🔷 CPU Affinity

✅ What is CPU Affinity?

- Binding a process/thread to run on **specific CPU cores**
- Avoids overhead of migration and improves **cache locality**

🔸 Tools

```
taskset -c 0,1 ./program  # Bind to cores 0 and 1
```

🔸 Types

Affinity Type	Description
Soft Affinity	Hint, OS may override
Hard Affinity	Strict binding to core

🔹 Real-Time Scheduling

✅ What is Real-Time Scheduling?

Schedulers designed to **meet timing constraints**. Often used in embedded or industrial systems.

🔸 Algorithms

Algorithm	Description
EDF	Earliest Deadline First (dynamic)
Rate Monotonic	Fixed-priority, shorter period = higher prio

🔸 POSIX Policies

Policy	Description
SCHED_FIFO	First-in, first-out real-time
SCHED_RR	Round-robin real-time
SCHED_DEADLINE	Deadline-based scheduling

```
chrt -f 50 ./realtime_task
```

🔹 OS Boot Process (BIOS → Bootloader → Kernel)

✅ BIOS/UEFI

- Initializes hardware
- Loads **Bootloader (MBR/GRUB)** from disk

✅ Bootloader

- Loads and executes **kernel image**
- Sets up basic memory layout, kernel parameters

✅ Kernel

- Initializes devices, mounts root filesystem
- Starts `init` or `systemd` (PID 1)

🔸 Flow

```
 Power ON
   ↓
BIOS/UEFI → Load Bootloader
   ↓
Bootloader → Load Kernel (bzImage, vmlinuz)
   ↓
Kernel → Mount RootFS, start init/systemd
```

✅ Summary

Concept	Description
Cache Coherency	Ensures all cores see same value of shared var
Lock/Wait-Free	Progress guarantees without locks
Page Coloring	Avoid cache conflicts via memory page placement
IOMMU	Secure DMA & virtual address translation
CPU Affinity	Bind process to CPU cores for locality
Real-Time Sched	Timely execution using policies like EDF/RM
OS Boot	BIOS → Bootloader → Kernel → Init

🧠 14. Virtualization and Containers

🔹 Hypervisor Types

✅ What is a Hypervisor?

A **hypervisor** is software or firmware that creates and runs **virtual machines (VMs)** by abstracting the underlying hardware.

🔸 Type 1 Hypervisor (Bare Metal)

- Runs **directly on hardware** (no host OS)

- Offers better performance and isolation
- Used in **data centers** and **enterprise servers**

🔸 **Examples:**

- VMware ESXi
- Microsoft Hyper-V (native)
- Xen
- KVM (with Linux as a minimal host)

🔸 **Diagram:**

```
 +------------------------+
|   Virtual Machine 1    |
+------------------------+
|   Virtual Machine 2    |
+------------------------+
|     Type 1 Hypervisor  |
+------------------------+
|     Hardware           |
```

🔸 Type 2 Hypervisor (Hosted)

- Runs **on top of a host OS**
- Easier to install and manage
- Slightly lower performance due to added layer

🔸 **Examples:**

- VirtualBox
- VMware Workstation
- QEMU (w/o KVM)

🔸 **Diagram:**

```
 +------------------------+
|   Virtual Machine      |
+------------------------+
|   Type 2 Hypervisor    |
+------------------------+
|   Host Operating System|
+------------------------+
|     Hardware           |
```

✅ Comparison Table

Feature	Type 1 Hypervisor	Type 2 Hypervisor
Runs On	Bare-metal (no OS)	Host OS
Performance	High	Medium
Use Case	Production/Data Centers	Dev/Test/Desktop

Feature	Type 1 Hypervisor	Type 2 Hypervisor
Examples	ESXi, Xen, Hyper-V	VirtualBox, VMware Player

🔹 Virtual Machines vs Containers

✅ Virtual Machines

- Emulate entire **hardware stack**
- Includes its own **guest OS**
- **Heavyweight**, slower startup, higher resource usage

🔸 Characteristics

Attribute	Value
Isolation	Strong (full kernel)
OS Overhead	High (each VM has full OS)
Startup Time	Slower
Portability	Lower (bigger image size)

✅ Containers

- Share **host OS kernel**, isolate only userspace
- Lightweight and faster to boot
- Popular in **microservices** and **CI/CD pipelines**

🔸 Characteristics

Attribute	Value
Isolation	Process-level (namespace/cgroup)
OS Overhead	Minimal
Startup Time	Fast
Portability	High (image layers)

🔸 Comparison Table

Feature	Virtual Machines	Containers
OS Requirement	Full guest OS per VM	Share host kernel

Feature	Virtual Machines	Containers
Resource Usage	High	Low
Boot Time	Seconds	Milliseconds
Isolation	Stronger (via hypervisor)	Weaker (via namespaces)
Management	Complex	Easier with tools like Docker
Performance	Lower	Near-native

🔸 Tools/Technologies

Area	VMs	Containers
Tools	VirtualBox, VMware, KVM	Docker, Podman, containerd
Orchestration	-	Kubernetes, Docker Swarm

✅ Summary

Concept	Description
Type 1 Hypervisor	Runs directly on hardware, high performance
Type 2 Hypervisor	Runs on host OS, easier for dev use
VMs	Full OS virtualization, more overhead
Containers	Lightweight OS abstraction, fast, efficient

🔹 Container Internals, Orchestration, and Virtualization Tools

---

🔹 Docker Internals

✅ Namespaces

- Provide **process-level isolation** in Linux.
- Each container gets its own **namespace instance**.

Namespace	Isolates
`pid`	Process IDs

Namespace	Isolates
`net`	Network interfaces
`mnt`	Mount points
`uts`	Hostname and domain
`ipc`	Inter-process communication
`user`	UID/GID mappings

```
 lsns        # list namespaces
unshare -p -f bash   # create new pid namespace
```

✅ cgroups (Control Groups)

- Restrict and monitor **resource usage** (CPU, memory, I/O)

Resource	Controlled Via
CPU	`cpu,cpuacct`
Memory	`memory`
I/O	`blkio`

```
cat /sys/fs/cgroup/memory/docker/<container-id>/memory.limit_in_bytes
```

✅ Union File Systems (UnionFS)

- Docker uses layered file systems to create images.

🔸 Common Types:

Type	Description
AUFS	Advanced multi-branch union FS (older)
OverlayFS	Default in most distros (modern)

🔸 Layer Concept

```
 Image Layers:
---------------
| App Layer     |
| Python Layer  |
| OS Base Layer |
---------------
```

Each `RUN`, `COPY`, or `ADD` adds a new **immutable layer**.

🔷 KVM, QEMU, Xen

✅ KVM (Kernel-based Virtual Machine)

- Linux kernel module that turns Linux into a **Type 1 hypervisor**.
- Works with **QEMU** for full virtualization.

```
kvm-ok     # check if CPU supports KVM
```

✅ QEMU

- **Hardware emulator** and **virtualizer**
- Can emulate **CPU, memory, devices**
- Works in:
  - Pure emulation mode (slow)
  - With KVM acceleration (fast)

```
qemu-system-x86_64 -hda disk.img -m 1G
```

✅ Xen

- Bare-metal hypervisor (Type 1)
- Supports **para-virtualization** and **full virtualization**

🔸 Xen Architecture:

```
 +-------------+
| Dom0 (Host) |
+-------------+
| DomU (Guest VMs) |
+------------------+
| Xen Hypervisor   |
+------------------+
| Hardware         |
```

🔷 Container Scheduling & Orchestration (Kubernetes Overview)

✅ Kubernetes (K8s)

An open-source system for **automating deployment**, **scaling**, and **management** of containerized apps.

🔸 Core Concepts

Term	Description
Pod	Smallest unit — 1 or more containers
Node	Physical/VM that runs pods
Cluster	Group of nodes
Deployment	Declarative definition of pod lifecycle
Service	Abstracts access to pods (load-balancing)
Scheduler	Assigns pods to nodes

🔸 Basic Architecture

```
 +------------------------+
|   kubectl (CLI)        |
+------------------------+
|   API Server           |
|   Controller Manager   |
|   Scheduler            |
+------------------------+
|   Worker Nodes         |
|   - kubelet            |
|   - container runtime  |
|   - pods               |
+------------------------+
```

🔸 Benefits

- Automated rollout & rollback
- Self-healing (restart, reschedule, etc.)
- Horizontal scaling
- Secrets & config management
- Persistent storage

🔹 Cloud-Native OS Behavior

✅ What is a Cloud-Native OS?

- OS designed for **containers**, **virtualization**, and **scalability**
- Often **immutable**, minimal footprint

🔸 Examples

OS	Description
CoreOS	Designed for container clusters
Bottlerocket	AWS's container-optimized OS
RancherOS	Lightweight, container-based OS
Flatcar Linux	Secure, auto-updating server OS

🔸 Features

- Immutable root filesystem
- Built-in support for container runtimes (Docker, containerd)
- Automatic updates
- Minimal attack surface

✅ Summary

Concept	Description
Namespaces	Isolate kernel resources per container
cgroups	Control CPU, mem, IO usage of containers
UnionFS	Layered FS used in Docker images
KVM/QEMU/Xen	Virtualization tools/hypervisors
Kubernetes	Manages pods/services across nodes
Cloud-native OS	Minimal, container-first OS designs

🧠 15. Distributed Operating Systems

🔹 Characteristics and Goals

✅ What is a Distributed Operating System?

A **Distributed Operating System (DOS)** manages a **collection of independent computers** and makes them appear to the user as a **single coherent system**.

🔸 Key Characteristics

Feature	Description
Transparency	Hides complexity from users/applications
Openness	Extensibility via standards
Scalability	Support for large, growing systems
Fault Tolerance	Continue functioning despite partial failures
Resource Sharing	Efficient use of system-wide hardware/software

---

🔸 Goals

- **Location Independence**
- **Load Balancing**
- **Efficient IPC** (Remote Communication)
- **Improved Reliability and Availability**

🔹 Transparency Types

Type	Description
Access	Hide difference in local/remote resource access
Location	Hide where resource resides
Migration	Hide relocation of resources/processes
Replication	Hide that multiple copies exist
Concurrency	Hide coordination between concurrent users
Failure	Hide failure and recovery mechanisms

🔹 Clock Synchronization

✅ Need for Synchronization

Distributed systems have **independent clocks**, causing drift. This affects:

- Timestamps
- Consistency
- Coordination

🔸 Network Time Protocol (NTP)

- Synchronizes clocks with **high-precision reference servers**

- Uses **round-trip delay** and **offset estimation**

🔸 **Equation:**

```
 Offset = ((T2 - T1) + (T3 - T4)) / 2
Delay = (T4 - T1) - (T3 - T2)
```

(T1-T4: timestamped communication events)

🔸 Berkeley Algorithm

- Elects **a leader node** to compute **average time**
- Sends adjustment to all participants

```
Leader polls others → Collects times → Averages → Sends delta adjustments
```

🔹 Election Algorithms

✅ Bully Algorithm

- Highest-ID process becomes coordinator.

🔸 **Steps:**

1. Node notices coordinator failure
2. Sends "election" to higher-ID nodes
3. If no response → becomes coordinator
4. Else → waits for "coordinator" message

✅ Ring Algorithm

- Nodes are arranged in a **logical ring**

🔸 **Steps:**

1. Node detects failure
2. Passes token with its ID
3. Each node appends its ID
4. Highest ID becomes new coordinator

🔹 Mutual Exclusion in Distributed Systems

✅ Requirements

- **Mutual exclusion** (1 process in CS at a time)
- **No starvation**
- **Fairness**

🔸 Algorithms

Algorithm	Description
Ricart-Agrawala	Uses timestamps; requires 2n–1 messages
Token Ring	Token circulates in logical ring
Maekawa's	Quorum-based voting among processes

🔸 Ricart-Agrawala Example:

```
 P1 sends request(timestamp) to all
Others reply if not in CS or lower timestamp
P1 enters CS if all replies received
```

🔹 Google's Cluster OSes: Borg, Omega, Kubernetes

✅ Borg

- Internal cluster manager at Google
- **Static allocation + optimistic task scheduling**
- Basis for Kubernetes

✅ Omega

- Designed after Borg to allow **multiple concurrent schedulers**
- Shared state + optimistic concurrency control

✅ Kubernetes

- Open-source orchestration built on Borg/Omega principles
- Enables **declarative container orchestration**

🔸 Comparison Table

Feature	Borg	Omega	Kubernetes
Internal Use	Yes (Google)	Yes	No (OSS)
Open Source	❌	❌	✅

Feature	Borg	Omega	Kubernetes
Scheduler Type	Centralized	Distributed	Pluggable
Resource Model	Rigid	Shared State	Declarative

✅ Summary

Topic	Key Points
Distributed OS Goals	Transparency, Fault Tolerance, Resource Sharing
Transparency Types	Access, Location, Replication, Failure, Concurrency
Clock Sync	NTP (precision), Berkeley (average-based)
Election Algorithms	Bully (highest ID wins), Ring (token passing)
Mutual Exclusion	Ricart-Agrawala, Token Ring, Maekawa
Cluster OS	Borg (static), Omega (concurrent schedulers), K8s (open)

🧠 16. Miscellaneous / System Design-Adjacent

🔹 Page Cache vs Buffer Cache

✅ Page Cache

- Caches **file contents** in **virtual memory pages**
- Speeds up file I/O by avoiding repeated disk reads
- Used when using `read()` / `write()` operations

🔸 Example

```
int fd = open("file.txt", O_RDONLY);
read(fd, buf, size);  // fetched into page cache
```

✅ Buffer Cache

- Caches **disk blocks** for **block-level devices**
- Operates at **block device layer**
- Often used with **raw block access**

🔸 Comparison

Feature	Page Cache	Buffer Cache
Layer	Filesystem (VFS)	Block I/O layer
Used By	read(), write(), mmap()	Raw block device I/O
Data Unit	Virtual memory pages	Disk blocks
Performance	High for file-backed access	High for raw device access

---

🔹 mmap() vs read()

✅ `read()`

- Copies data from **kernel space → user buffer**
- Traditional way of doing I/O
- No shared memory: slower for large datasets

```
read(fd, buffer, size);
```

✅ `mmap()`

- Maps file into **user-space virtual memory**
- No system call needed for each access
- Supports **lazy loading** and **page-level access**

```
char *data = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
```

🔸 Use Cases

Use Case	Best Approach
Small file, read-once	`read()`
Large file, random access	`mmap()`
Inter-process shared memory	`mmap()`

🔸 Performance

- `mmap()` may be faster due to **page cache sharing**
- Avoids user/kernel copies
- Allows memory-level manipulation

🔸 Caveats

- `mmap()` is harder to manage, especially with signals
- `read()` is simpler and portable

🔷 Thread vs Coroutine

✅ Threads

- OS-level entities
- Preemptively scheduled
- Can run on multiple cores (parallelism)

```
std::thread t1(func);
```

✅ Coroutines

- **User-level** cooperative routines
- Manually yield control to next coroutine
- Lightweight: no kernel context switch
- Cannot run in parallel (only concurrency)

🔸 Comparison Table

Feature	Thread	Coroutine
Scheduling	Preemptive (by OS)	Cooperative (manual yield)
Stack	Separate per thread	Often shared or segmented
Context Switch	Kernel → user space	User-space only (faster)
Performance	Higher overhead	Lightweight
Parallelism	Yes (multi-core)	No (single thread concurrency)
Language Support	C++, Java, POSIX	Python (asyncio), C++20, Lua

---

✅ Coroutine Example (Python)

```
async def main():
    await task1()
    await task2()
```

✅ Coroutine Example (C++20)

```
task<> async_operation() {
    co_await some_async_thing();
```

```
}
```

✅ Summary

Concept	Description
Page vs Buffer Cache	Page cache for file I/O, buffer cache for block I/O
mmap() vs read()	mmap maps file into memory; read copies to buffer
Thread vs Coroutine	Thread = OS-managed; Coroutine = user-level

---

🧠 16. Miscellaneous / System Design-Adjacent (Part 2)

🔹 Epoll vs Select vs Poll

✅ `select()`

- Monitors multiple file descriptors for readiness.
- Limited by `FD_SETSIZE` (usually 1024).
- Inefficient for large FDs (O(n)).

```
fd_set readfds;
select(nfds, &readfds, NULL, NULL, &timeout);
```

✅ `poll()`

- Overcomes `select`'s FD limit using a list.
- Still linear O(n) scan.

```
struct pollfd fds[2];
poll(fds, 2, timeout);
```

✅ `epoll` (Linux only)

- Scalable (O(1) event detection).
- Uses event-based callbacks.
- Efficient for thousands of FDs (e.g., servers).

```
int epfd = epoll_create1(0);
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
epoll_wait(epfd, events, maxevents, timeout);
```

🔸 Comparison Table

Feature	select()	poll()	epoll()
FD Limit	~1024	High	Very High
Performance	O(n)	O(n)	O(1)
Edge Triggered	❌	❌	✅
Kernel Copy	Entire FD set	Entire set	Internal tracking

🔹 File Descriptor Table Limits

✅ File Descriptors

- Every process has a **File Descriptor Table** (array)
- Global system-wide & per-process limits apply

🔸 Check Current Limits

```
ulimit -n             # soft limit
cat /proc/sys/fs/file-max  # system-wide
```

🔸 Increase Limit

```
ulimit -n 65535       # temp increase
```

To make permanent (Linux):

```
/etc/security/limits.conf
```

🔹 Load Average (1m, 5m, 15m)

✅ What is Load Average?

- Shows **average number of processes in runnable state** (not sleeping).
- Displayed by `top`, `uptime`, etc.

```
top
# load average: 0.72, 1.18, 1.43
```

🔸 Meaning

Time Window	Meaning
1 min	Short-term CPU demand
5 min	Medium-term load
15 min	Long-term system pressure

If **load average > number of CPUs**, system is overloaded.

🔹 Swappiness

✅ What is Swappiness?

- Linux kernel parameter controlling **tendency to swap** pages to disk.

Value	Behavior
0	Avoid swapping aggressively
60	Default in most distros
100	Swap as much as possible

🔸 Check & Set

```
cat /proc/sys/vm/swappiness
sysctl vm.swappiness=10
```

To persist:

```
echo "vm.swappiness = 10" >> /etc/sysctl.conf
```

🔸 Use Case

- Lower value for **performance-critical apps** (databases).
- Higher for **memory-constrained systems**.

🔹 Out-of-Memory Killer (OOM Killer)

✅ What is OOM Killer?

- When Linux runs out of memory, the **OOM Killer** selects a process to kill to **free memory**.

🔸 Scoring

- Processes with **higher memory usage**, **low priority**, or **less critical** roles are killed first.
- Controlled via:

```
/proc/<pid>/oom_score
/proc/<pid>/oom_adj
```

🔸 Avoid OOM Kill

```
echo -1000 > /proc/<pid>/oom_score_adj  # lower score → avoid kill
```

🔸 Logs

- OOM events logged in `/var/log/syslog` or `dmesg`

🔸 Real-world Example

```
dmesg | grep -i "oom"
```

```
Out of memory: Kill process 12345 (node) score 987 or sacrifice child
```

✅ Summary

Concept	Key Points
epoll/select/poll	epoll scales best for many FDs
FD Limits	`ulimit -n` controls open files per process
Load Average	Indicates runnable process count over time
Swappiness	Balances RAM usage vs swap tendency
OOM Killer	Frees memory when exhausted, kills low-priority processes

---

🧠 16. Miscellaneous / System Design-Adjacent (Part 3)

🔹 Memory Pressure Detection

✅ What is Memory Pressure?

- Occurs when **available free memory is low**.
- Triggers the OS to **reclaim**, **swap**, or **kill** processes.

🔸 Detection Tools

Tool / File	Purpose
`/proc/meminfo`	Check free, cached, swap memory
`vmstat`	View active/inactive page stats
`sar -B`, `free -m`	Track memory usage patterns
`dmesg`	Detect OOM and page reclaim messages
`ps`, `top`, `htop`	Identify high memory consumers

🔸 Example

```
 vmstat 1
free -h
```

🔷 Reclaiming Memory (kswapd)

✅ kswapd

- **Kernel thread** that frees memory under pressure.
- Reclaims pages by:
  - Dropping page cache
  - Swapping out memory
  - Evicting clean/dirty pages

🔸 When does kswapd trigger?

- When the amount of free pages drops below `low watermark`
- Kswapd tries to reach `high watermark`

🔸 Relevant Files

```
 cat /proc/sys/vm/min_free_kbytes
cat /proc/zoneinfo
```

🔹 HugePages and Transparent HugePages

✅ HugePages

- Memory pages larger than **default 4KB**, usually 2MB or 1GB.
- Reduces **TLB misses**, improves performance for **memory-intensive apps** (e.g., DBs, VMs)

🔸 Enabling HugePages

```
echo 128 > /proc/sys/vm/nr_hugepages
```

Use `hugetlbfs` to allocate.

✅ Transparent HugePages (THP)

- Linux kernel feature that **automatically backs memory allocations with HugePages**
- No code changes required

```
cat /sys/kernel/mm/transparent_hugepage/enabled
```

🔸 Comparison

Feature	HugePages	THP
Setup	Manual	Automatic
Flexibility	Fixed allocations	Dynamic at runtime
Apps Supported	Databases, HPC apps	General purpose

🔹 Memory Leaks and Valgrind

✅ Memory Leak

- Happens when memory is **allocated but never freed**
- Common in C/C++ when `malloc()` or `new` is used without `free()` or `delete`

🔸 Symptoms

- Gradual memory consumption growth
- No corresponding release
- Application slows or crashes

✅ Valgrind

- Tool for **memory debugging**, **leak detection**, **profiling**
- Simulates a CPU and tracks memory allocations

🔸 Install

```
sudo apt install valgrind
```

🔸 Usage

```
valgrind --leak-check=full ./my_app
```

🔸 Sample Output

```
==1234== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==1234==    at 0x....: malloc (vg_replace_malloc.c:309)
==1234==    by 0x....: main (main.c:5)
```

✅ Fixing Leaks

- Use `free()` after `malloc()`
- Track ownership of dynamically allocated objects
- Use smart pointers in C++ (`unique_ptr` , `shared_ptr`)

✅ Summary

Concept	Key Points
Memory Pressure	Detected via `/proc/meminfo` , `vmstat` , `kswapd` logs
kswapd	Kernel process to reclaim pages on low memory
HugePages	Bigger memory pages for better TLB performance
THP	Auto HugePages without code change
Valgrind	Leak detector and memory profiler

🧠 16. Miscellaneous / System Design-Adjacent (Part 4)

🔹 Scheduling Tuning (`/proc/sched_debug`)

✅ What is `/proc/sched_debug`?

- A **Linux kernel interface** that provides **detailed runtime info** about the **scheduler state**.
- Useful for analyzing **task priorities**, **runtimes**, and **CPU affinities**.

🔸 How to View

```
cat /proc/sched_debug
```

🔸 Example Output Snippet

```
 Sched Debug Version: v0.11, 4.15.0-112-generic
sysctl_sched_latency                    : 12000000
cpu#0, 4600.000 MHz
  .nr_running                    : 2
  .load                          : 1024
  .tg_load_avg                   : 2048
...
```

🔸 Key Fields Explained

Field	Meaning
`nr_running`	Number of runnable tasks on CPU
`load`	CPU load (load balancing decisions)
`tg_load_avg`	Load average for the scheduling group
`sysctl_sched_latency`	Max delay to schedule a task (ns)
`sysctl_sched_min_granularity`	Minimum granularity for time slice (ns)

✅ How to Tune Scheduler (CFS Parameters)

You can change scheduling behavior dynamically via `/proc/sys/kernel/` or using `sysctl`.

🔸 Parameters

Parameter	Description
`sched_latency_ns`	Target latency to run all tasks once
`sched_min_granularity_ns`	Minimum time slice any task gets

Parameter	Description
`sched_wakeup_granularity_ns`	Wakeup preemption threshold

🔸 Check Current Values

```
cat /proc/sys/kernel/sched_latency_ns
cat /proc/sys/kernel/sched_min_granularity_ns
```

🔸 Modify Values

```
echo 6000000 > /proc/sys/kernel/sched_latency_ns
```

To make changes persistent:

```
sudo nano /etc/sysctl.conf

# Add:
kernel.sched_latency_ns = 6000000
kernel.sched_min_granularity_ns = 2000000
```

Then apply:

```
sudo sysctl -p
```

✅ Use Cases for Tuning

Scenario	Tuning Strategy
Latency-sensitive app (gaming, HFT)	Lower `sched_latency_ns`, smaller granularity
Throughput workload (batch jobs)	Increase granularity to reduce context switch
Real-time simulations	Pin tasks with CPU affinity, tune wakeup time

---

✅ Tools for Scheduler Analysis

Tool	Purpose
`schedtool`	Manually assign scheduler and priority
`chrt`	Set real-time policies (`SCHED_FIFO`, etc)
`htop`	Show CPU usage and priorities
`perf sched`	Record and visualize scheduling latency

✅ Summary

Concept	Key Info
`/proc/sched_debug`	Shows real-time kernel scheduler diagnostics
`sched_latency_ns`	Controls time frame for fair scheduling
`sched_min_granularity_ns`	Minimum guaranteed CPU slice
Tuning Tools	`sysctl`, `schedtool`, `chrt`, `perf`
Use Cases	Useful in optimizing latency vs throughput tradeoffs