⭐ UNIT 1 — INTRODUCTION TO OPERATING SYSTEMS (Super Simple English + Compact Notes)

1. What is Software?

Software is of two types:
1) Application Software: Programs used for daily tasks. They help the user directly. Examples: WhatsApp, Chrome, MS Word, YouTube, Games.
2) System Software: Programs that help the computer run properly. They support application software. Examples: Operating System, Device Drivers, Compiler.

2. What is an Operating System (OS)?

An Operating System is software that controls the whole computer. It manages CPU, memory, files, input/output devices, security, and running programs. It hides hardware complexity so the user can work easily.
Simple Line: OS = Computer ka manager.
Examples: Windows, Android, iOS, Linux, MacOS.

⭐3. Why Do We Need an Operating System?

Without an OS, the computer cannot work properly.

Problem 1: Apps become very complicated

Every app must talk to hardware by itself. Example: Chrome must write code to connect to RAM, keyboard, display, and storage. Apps become big, slow, and hard to develop.

Problem 2: One app can take all CPU and RAM

Without OS control, one app can use 100% CPU and all RAM. Other apps stop working → system hangs or crashes.

Problem 3: No memory protection

Apps can read or change each other's memory. Example: WhatsApp reading Chrome data, a game overwriting another program, viruses easily damaging data.

Conclusion: Without an OS → Computer becomes unsafe, slow, unstable, and unusable. So an OS is necessary.

⭐4. What is an OS Made Up Of?

OS is a collection of many small programs: Process Manager, Memory Manager, File Manager, Device Manager, Security Manager, I/O Manager. These work together to run the computer smoothly.

⭐5. What Does an Operating System Do?

1) Provides Access to Hardware

OS controls CPU, RAM, Hard Disk, Keyboard, Mouse, Printer, Camera. Apps cannot directly use hardware; OS helps them. Example: You click "Print" → OS sends the command to the printer.

2) Interface Between User and Hardware

OS provides an interface like GUI (Windows, Android) or CLI (Linux terminal). Without OS, user cannot interact with hardware.

3) Resource Management

OS manages CPU time, RAM, storage, files, devices, and security. It decides which app gets CPU first, how much memory each app gets, and who can access which file. Example: Chrome + VS Code + YouTube all run smoothly because OS divides CPU fairly.

4) Hiding Hardware Complexity

OS hides hardware details. Example: App says "save file" → OS handles all complex work internally.

5) Program Execution & Protection

OS loads apps into RAM, starts them, stops them, protects their data, handles errors, and manages crashes. Example: If one app crashes, OS stops it but system keeps working.

⭐6. How OS Sits Inside the Computer?

User → Application Programs → Operating System → Hardware
Meaning: User uses apps, apps talk to OS, OS talks to hardware

What is an Operating System?

An Operating System (OS) is software that controls the entire computer and helps applications run. It works like a manager between the user, applications, and hardware.

Why OS is Needed?

Keeps CPU busy, avoids long waiting (no starvation), runs important tasks first, manages files, memory, CPU, devices, and keeps the system safe. Without OS, the computer cannot work properly.

Goals of OS

Maximum CPU utilization → CPU should not sit idle.
Less starvation → Every program gets a chance.
High priority first → Important tasks run earlier.

Types of Operating Systems

1) Single-Process Operating System

Only one program runs at a time. If one app is running, another cannot start. Used earlier when computers were slow.
Example: **MS-DOS (1981)**

2) Batch Processing Operating System

Jobs are grouped into batches and executed one batch at a time.
Steps: User gives job → Operator groups jobs → CPU runs the batch → Next batch starts.
Problems: No priority, long waiting, CPU idle during I/O.
Example: **Old punch-card systems (1960s)**

3) Multiprogramming Operating System

Many programs stay in memory together. If one waits for I/O, CPU runs another. Goal: CPU must always stay busy.
Key: Single CPU + Context switching.
Example: **THE System (1960s), modern Linux.**

4) Multitasking Operating System

One CPU but many tasks appear to run together because CPU switches very fast. Makes system responsive.
Example: **Windows, macOS, Android.**

5) Multiprocessing Operating System

Computer has more than one CPU. Different CPUs handle different tasks.

Benefits: Higher speed, less waiting, system still works if one CPU fails.
Example: **Windows NT, Linux servers, multi-core CPUs.**

6) Distributed Operating System

Many computers connected by network behave like one big computer. User does not feel they are separate machines.
Used for cloud servers and large systems.
Example: **LOCUS, modern distributed OS concepts.**

7) Real-Time Operating System (RTOS)

System must give results within strict timing. Even small delay is not allowed.
Used in robots, medical devices, air-traffic control, self-driving cars.
Example: **VxWorks, QNX.**

MULTITASKING and MULTITHREADING

Program:
A program is a file stored on disk that contains instructions to do a specific job. It is compiled and ready to run.

Process:
A program that is currently running. It lives in RAM (primary memory).

Thread:
A thread is a small, lightweight part of a process.
Each thread has its own path of execution.
Threads share the same memory of the process.
Used for parallel work inside a single process.

Examples:
A browser with many tabs → each tab is a thread.
A text editor → typing, spell-check, auto-save work in different threads together.

Difference: Multitasking vs Multithreading

Multitasking:
CPU runs more than one process by switching between them.
Each process gets separate memory and protection.
Used when multiple applications run together (Chrome + VS Code + Music Player).
Needs at least 1 CPU.

Multithreading:
A single process is divided into multiple threads.

Threads share the same memory.
Used to perform multiple tasks inside one program.
Better performance with more than 1 CPU/core.

KEY DIFFERENCES

Multitasking:
- **More than one process runs (CPU switches between processes).**
- **Needs memory isolation for each process.**
- **OS allocates separate memory to each process.**
- **Slow context switching.**
- **CPU cache is flushed.**
- **Works even with 1 CPU.**

Multithreading:
- **One process has many threads.**
- **No isolation; threads share memory.**
- **OS gives memory to the process, threads share it.**
- **Fast context switching.**
- **CPU cache is preserved.**
- **Better if CPU ≥ 1 (multi-core is best).**

Thread Scheduling

OS gives each thread a small time slice of CPU.
Threads run based on priority.
Even if they belong to the same process, OS decides which thread runs next.

Context Switching

Thread Context Switching:
OS switches from one thread to another inside the same process.
No change in memory space.
Fast because threads share memory.

Process Context Switching:
OS switches from one process to another.
Memory address space changes completely.
Slower because CPU must load new memory, flush cache, reload registers.

Components of Operating System

Operating System do major parts mein divided hota hai: Kernel (core, kernel mode) + User Space (apps run here). OS ka main kaam hardware ko control karna aur user programs ko smooth environment dena.

🔥 1) Kernel (Heart of OS)

Kernel = core part jo directly CPU, RAM, Disk, Display, Keyboard, Network Card se interact karta hai.
Ye bootloader ke baad sabse pehle load hota hai, kernel mode mein run karta hai aur pure system ko control karta hai. Normal apps hardware ko directly access nahi kar sakti because security risk, isliye kernel unko protect karta hai.

🔥 2) User Space (Apps Run Here)

User Space wo area jaha GUI (Windows, Icons, Menus), CLI (Terminal, Bash, PowerShell), aur Applications (Chrome, VS Code, Games) run hoti hain.
User space hardware se directly baat nahi kar sakta → kernel ke through request send hoti hai.

🔥 3) Shell (Inside User Space)

Shell = command interpreter. User ke commands ko kernel tak forward karta hai.
Types: CLI (Bash, PowerShell, Zsh), GUI Shell (Windows Explorer, Gnome Shell).
Example: `mkdir folder` → shell → kernel → folder create.

⭐ Kernel Ke 4 Major Functions

🎯 1) Process Management

Kernel process ko create, delete, schedule karta hai. Kaun CPU use karega? → Scheduling.
Context Switching se multiple apps parallel chalti hain.
IPC se processes data exchange karti hain.
Example: Game + Spotify dono ko CPU time kernel deta hai.

🎯 2) Memory Management

Kernel RAM ko manage karta hai. Memory allocate/deallocate, free/used blocks track karta hai.
Virtual Memory + Page Table handle karta hai.
Example: Chrome open → 400MB allocate. Close → free.

🎯 3) File Management

Kernel poora file system (NTFS, FAT32, ext4) control karta hai.
Files/folders create-delete, permissions set, disk read/write sab kernel karta hai.
Example: Game progress save → kernel disk pe write karta hai.

🎯 4) I/O Management

Kernel saare Input/Output devices handle karta hai: keyboard, mouse, display, HDD, pen drive, network card, printer etc.

I/O Techniques

Spooling: Slow device ke liye queue (example: printer queue).
Buffering: Temporary memory for single job (example: YouTube buffer).
Caching: Fast memory for frequently used data (browser cache, disk cache).

⭐ Types of Kernels

1) Monolithic Kernel

Sab services kernel ke andar: Process + Memory + File + I/O + Drivers.
Fast but bulky. Agar ek service crash → pura OS affected.
Examples: Linux, Unix, MS-DOS.
Analogy: Poori team ek hi kitchen me — ek galti se sab ruk jata.

2) Microkernel

Kernel me sirf essential parts: Process mgmt, Memory mgmt, IPC.
Baaki services user space me.
Small, reliable, more stable but thoda slow due to extra switching.
Examples: MINIX, L4, Symbian.

3) Hybrid Kernel

Monolithic + Microkernel ka mix.
Fast + secure + modular.
Used in: Windows NT/7/10/11, macOS (XNU).
Modern OS isliye hybrid use karte because speed + stability dono chahiye.

4) Nano Kernel / Exokernel

Bahut minimal kernel. Direct hardware access apps ko deta.
Used in research/special systems. High performance.

⭐ User Mode ↔ Kernel Mode Communication

User programs hardware ko directly access nahi kar sakti → IPC (Inter-Process Communication) ka use hota hai.

IPC Methods:

Shared Memory: Fast but needs synchronization.
Message Passing: Safe; kernel messages forward karta hai.
Real examples:
Browser → Network Driver, Game → GPU, WhatsApp → Internet.

SYSTEM CALLS

System Call = special request jo user program kernel ko send karta hai jab program koi aisa kaam karna chahta hai jo directly allowed nahi hota. User programs User Mode me, kernel Kernel Mode me run hota — isliye system calls bridging ka kaam karte hain aur hardware access ko secure banate hain.

⭐ Why System Calls Needed?

User program directly hardware access, process create, memory allocate, file read/write, IPC, kuch nahi kar sakta.
Direct access = security risk.
Isliye system call = safe entry gate from User Space → Kernel Space.

⭐ How Apps Interact with Kernel?

Process:
User program instruction → needs OS service → system call → software interrupt → CPU switches User Mode→Kernel Mode → kernel task performs → result return → back to User Mode.
👉 System call is the ONLY path to enter kernel mode.

⭐ Example: mkdir myfolder

`mkdir` command system call nahi hota — yeh sirf wrapper hai.
`mkdir()` system call → kernel file mgmt run → directory create → control return.

⭐ Example: Creating a Process

Chrome click → launcher uses fork(), exec(), CreateProcess() → kernel new process create karta, memory allocate karta → user space me control return.

⭐ System Calls are C Functions

Most system calls C me likhe hote because low-level, fast, hardware access friendly.

⭐ Types of System Calls

🔥 1) Process Control System Calls

Kaam: process create, terminate, load, wait, priorities set, memory allocate/free.
Examples: fork(), exec(), exit(), abort(), wait(), waitpid()
Real Use: Chrome open → process creation. Antivirus kill → termination. Game level load → exec().

🔥 2) File Management System Calls

Kaam: file/dir create, delete, open, close, read, write, seek, permissions.
Examples: open(), close(), read(), write(), chmod(), chown()
Real Use: Copy-paste movie → read/write. VS Code save → write(). Permissions change → chmod().

🔥 3) Device Management System Calls

Kaam: device request, release, read, write, attach/detach, attributes change.
Examples: ioctl(), read(), write()
Real Use: Printer request. USB read. Screen brightness → ioctl().

🔥 4) Information Maintenance System Calls

Kaam: get/set time, system data, process/file/device attributes.
Examples: getpid(), alarm(), sleep()
Real Use: Clock app time fetch, Task Manager process list, CPU temp read.

🔥 5) Communication System Calls

Kaam: create/delete communication channel, send/receive message, shared memory, mmap, status transfer.
Examples: pipe(), shmget(), mmap()
Real Use: WhatsApp → network driver. Chrome → plugins. Client/server apps.

⭐ Mode Transition (User → Kernel → User)

System call execute hote hi:
User mode → system call instruction → software interrupt → CPU kernel mode shift → kernel operation → return to user mode.
Isse security & privilege control maintain hota hai.

⭐ Windows vs Unix System

Category	Windows System Calls	Unix/Linux System Calls
Process Control	CreateProcess(), ExitProcess(), WaitForSingleObject()	fork(), exit(), wait()
File Management	CreateFile(), ReadFile(), WriteFile(), CloseHandle(), SetFileSecurity()	open(), read(), write(), close(), chmod(), chown()
Device Management	SetConsoleMode(), ReadConsole(), WriteConsole()	ioctl(), read(), write()
Info Maintenance	GetCurrentProcessID(), SetTimer(), Sleep()	getpid(), alarm(), sleep()
Communication	CreatePipe(), CreateFileMapping(), MapViewOfFile()	pipe(), shmget(), mmap()

⭐ LEC-6: WHAT HAPPENS WHEN YOU TURN ON YOUR COMPUTER? i) PC Power ON

You press the power button → electricity flows → CPU becomes active.

ii) CPU Loads BIOS/UEFI

CPU looks for firmware stored on the motherboard's ROM chip.
Old systems use BIOS (Basic Input Output System).
Modern systems use UEFI (Unified Extensible Firmware Interface) → more advanced and faster.

iii) BIOS/UEFI Runs POST

POST checks whether hardware is working:
• RAM detected?
• Keyboard/mouse detected?
• CPU/GPU present?
• Hard disk/SSD connected?
If any major component fails → system shows error/beep → boot stops.
UEFI works like a mini operating system (more powerful than BIOS).

iv) BIOS/UEFI Loads Bootloader

After successful POST, BIOS searches bootloader in storage.
BIOS checks MBR (Master Boot Record) → first 512 bytes of disk.
UEFI checks EFI System Partition (ESP).
Control is handed over to the bootloader.

v) Bootloader Loads Operating System

Bootloader is a small startup program responsible for loading the kernel.
Examples:
• Windows → Windows Boot Manager (bootmgr.exe)
• Linux → GRUB
• macOS → boot.efi
Bootloader loads the Kernel, and kernel loads user space (desktop, apps, services).

FULL BOOT SEQUENCE (ONE LINE):
Power ON → BIOS/UEFI → POST → Bootloader → Kernel → OS Starts

⭐ LEC-7: 32-BIT vs 64-BIT OPERATING SYSTEM

1. Memory Addressing Ability

32-bit OS → can access 2^{32} = 4GB RAM max
64-bit OS → can access 2^{64} = 17 billion GB (virtually unlimited)

2. CPU Processing Capacity

32-bit CPU processes 32 bits (4 bytes) per instruction.
64-bit CPU processes 64 bits (8 bytes) per instruction.
More bits → faster processing of large calculations.

3. Performance Difference

64-bit OS/CPU is faster because:

- **Larger registers**
- **Can handle heavy apps**
- **Can use more RAM**
- **Better multitasking**

32-bit slows down with large data and many apps.

4. RAM Usage

32-bit OS → no matter how much RAM you install, only 4GB usable.
64-bit OS → uses all installed RAM.

5. Compatibility Rules

- **64-bit CPU → supports both 32-bit and 64-bit OS**
- **32-bit CPU → supports only 32-bit OS**

Software:

- **64-bit OS → runs 32-bit + 64-bit apps**
- **32-bit OS → cannot run 64-bit apps**

6. Graphics & Performance

64-bit processors handle 8-byte calculations, which boosts:

- **Gaming performance**
- **Video editing**
- **Rendering**
- **Heavy software reliability**

ONE-LINE SUMMARY:

64-bit OS = faster, supports more RAM, better for modern apps
32-bit OS = outdated, limited to 4GB, used only in old systems

⭐ LEC-8: STORAGE DEVICES BASICS

A computer uses 4 types of memory to store and access data: Registers, Cache, RAM, Secondary Storage.

1) Register (Fastest Memory)
Registers are inside the CPU and are the smallest but fastest memory. They store: instructions, memory addresses, temporary data, and small values needed during execution. Purpose: provide extremely fast data access so CPU can execute instructions without delay.

2) Cache Memory (Second Fastest)
Cache is a high-speed memory between CPU and RAM. It stores frequently used data/instructions so CPU does not repeatedly fetch from RAM. This increases system speed. Cache has levels:
• L1 → smallest and fastest
• L2 → larger but slower
• L3 → largest but slowest among caches
Still much faster than RAM.

3) Main Memory (RAM)
RAM is the primary working memory used by CPU for running programs. It is volatile, meaning data is lost when the PC is switched off. Any program you open first loads into RAM and then runs. RAM provides medium-speed access and is larger than registers and cache.

4) Secondary Storage (HDD/SSD/Pen Drive)
This is permanent, long-term storage. It is non-volatile, meaning data stays safe even when the PC is off. Used to store files, software, operating systems, videos, etc. Examples include HDD, SSD, pen drive, SD card. Secondary storage is slowest but offers very large capacity.

🆚 MEMORY COMPARISON (CONGESTED FORMAT)

Cost:
Registers (highest) > Cache > RAM > Secondary storage (cheapest)

Speed (Fast → Slow):
Register → Cache → RAM → HDD/SSD

Storage Size (Small → Large):
Register → Cache → RAM → HDD/SSD

Volatility:
Registers, Cache, RAM → volatile
Secondary storage → non-volatile

⭐ LEC-9: INTRODUCTION TO PROCESS

1) Program: A program is code stored on disk; it is not running and has no RAM, CPU time, stack, heap, or I/O. Program = passive entity. Examples: chrome.exe, a.out, .class file.

2) Process: A process is a running program. It is active and has its own RAM area (stack, heap, data, code), CPU time, registers, and I/O devices/files. Process = program + execution context. Example: chrome.exe opens → becomes a running process.

3) How OS Creates a Process:
Step A: OS loads code segment + global/static variables into RAM.
Step B: OS creates a private stack for local variables, function calls, return addresses.
Step C: OS creates a heap for dynamic memory (objects, arrays, linked list nodes).
Step D: OS initializes I/O (file descriptors, keyboard/screen connection, libraries).
Step E: OS jumps to main() and process starts executing.

4) Process Memory Layout:
CODE = machine instructions (read-only).
DATA = global/static vars + constant strings.
HEAP = dynamic memory (grows upward).
STACK = local vars + function calls (grows downward).

5) Process Attributes:
OS stores info in the Process Table; each entry = PCB.
PCB contains: PID, state (new, ready, running, waiting, terminated), program counter (next instruction), CPU registers (ACC, general registers, base pointer, stack pointer), memory info (stack start, heap start, code address), scheduling

info (priority, time slice), I/O status (open files/devices), accounting info (CPU usage + run time). PCB = complete identity + saved state of a process.

6) Why PCB Stores Registers (Context Switching):
When CPU is running a process and its time slice ends, OS performs context switch.
Step 1 (Save Old): CPU ke sare register values + program counter + pointers OS old process ke PCB me save kar deta hai. Old process fully paused but its state safely stored.
Step 2 (Load New): OS new process ka PCB padhta hai aur uske stored registers, program counter, pointers CPU me load kar deta hai. Ab CPU exactly new process ki last saved position par aa jata hai.
Next switch: new → old reverse hoga (new save hoga, old load hoga).
Final: CPU always resumes any process exactly from its last executed instruction because PCB stores complete snapshot of process state.

⭐ LEC-10: PROCESS STATES & PROCESS QUEUES (ULTRA-CONGESTED NOTES)

1) PROCESS STATES (Lifecycle of a Process)
NEW: OS picks program, creates PCB, allocates memory (stack+heap). Process is being created. Example: Clicking Chrome → Chrome process in NEW.
READY: Process fully loaded in RAM, waiting for CPU. "I am ready, give me CPU!" READY ≠ RUNNING.
RUNNING: CPU currently executing this process; instructions are running.
WAITING/BLOCKED: Process is not using CPU; waiting for I/O (file read/write, keyboard input, disk, network). CPU given to another process.
TERMINATED: Process finished; OS deletes PCB, deallocates memory, removes from process table.

State Diagram: NEW → READY → RUNNING → TERMINATED; RUNNING → WAITING → READY.

---

2) PROCESS QUEUES (Managed by OS)
Job Queue: Located in secondary storage; contains all NEW state processes.

Managed by Long-Term Scheduler (LTS). LTS loads selected processes into memory, controls degree of multiprogramming.
Ready Queue: Located in RAM; contains all READY processes waiting for CPU. Managed by Short-Term Scheduler (CPU Scheduler). Scheduler picks the best process and sends it to CPU.
Waiting Queue: Located in RAM; contains processes in WAITING (BLOCKED) state due to I/O operations (file, keyboard, disk, network). After I/O completes, process returns to Ready Queue.

WAITING (Simple Meaning): Process was RUNNING → needs I/O → moves to WAITING → I/O complete → moves back to READY.

Brief way

A) Job Queue (NEW State Processes)
Where: HDD/SSD (secondary storage).
What: All programs that have just started but are NOT loaded into RAM yet are kept here.
Managed By: Long-Term Scheduler (LTS).
LTS Work:
• Select which program should be moved from HDD → RAM.
• Control multiprogramming (too many processes = slow, too few = CPU idle).
Example: You open Chrome, VS Code, a Game → all first enter Job Queue → LTS loads some of them into RAM.

B) Ready Queue (READY State Processes)
Where: RAM (main memory).
What: All processes that are ready to run and waiting for CPU. They have memory but not CPU time.
Managed By: Short-Term Scheduler (CPU Scheduler).
STS Work:
• Pick the best process from Ready Queue.
• Give CPU to that process.
• Handle context switching.
Example: Chrome, WhatsApp Desktop, Spotify, VS Code → all sitting in Ready Queue waiting for their CPU turn.

C) Waiting Queue (I/O BLOCKED State Processes)
Where: RAM.
What: Processes waiting for some I/O operation (not using CPU).

Examples of I/O Wait: File read/write, keyboard input, mouse input, disk access, network packet.
When I/O completes: Process moves from Waiting Queue → back to Ready Queue.

---

3) DEGREE OF MULTIPROGRAMMING
Definition: Number of processes present in main memory (RAM) at a time.
Controlled by LTS.
High multiprogramming → better CPU utilization + good throughput; but more memory usage + more context switching.

---

4) DISPATCHER (VERY IMPORTANT)
Dispatcher = OS component that actually starts the process chosen by CPU Scheduler.
Functions:
• Context Switch: old process registers saved, new process registers loaded.
• Switch to User Mode: kernel → user mode.
• Jump to Program Counter: starts/resumes process from its last saved instruction.
Without dispatcher, CPU scheduling cannot work.

⭐ LEC-11: Swapping | Context Switching | Orphan Process | Zombie Process

(VERY SIMPLE ENGLISH + CONGESTED FORMAT)

1) Swapping
Swapping means moving a process out of RAM and placing it into disk (HDD/SSD) to free memory. Later the same process can be brought back into RAM and continue from the same point.
Why needed: RAM becomes full, too many processes, to balance CPU-bound and I/O-bound programs.
Controlled by: Medium-Term Scheduler (MTS).
Steps:
• Swap-Out: process goes from RAM → disk

• Swap-In: process comes from disk → RAM
Summary: Swapping = temporary memory shifting (RAM ↔ Disk).

2) Context Switching
Context Switching means changing the CPU from one process to another because CPU can run only one process at a time.
Steps:
• OS saves the current process state (registers + program counter) into its PCB.
• OS loads the next process state from its PCB into the CPU.
• CPU starts running the new process.
Important:
Context switching produces no useful work.
Speed depends on number of registers, RAM speed, and hardware support.

3) Orphan Process
An orphan process is a child process whose parent has already finished or terminated, but the child is still running.
What happens:
• When the parent ends, the child becomes "orphan".
• The operating system assigns this orphan child to the init process (PID 1).
• Init becomes the new parent so the child can finish normally and be cleaned up.
Example:
Parent process ends early → child is still working → child becomes orphan → init adopts it.

4) Zombie Process (Defunct Process)
A zombie process is a process that has already completed execution but its entry is still present in the process table.
Why does this happen:
• The child finished execution.
• But the parent did not call wait() to collect the child's exit status.
When removed:
• When the parent finally calls wait(), the exit status is collected and the zombie entry is removed (reaped).
Why zombies appear:
• Child finishes quickly.
• Parent is busy or poorly designed and does not read the exit status in time.