

OPERATING SYSTEMS

UNIT 1 — Detailed Notes

Introduction to Operating Systems

Topics covered: Concept of OS • Generations of OS • Types of OS • OS Services • System Calls • OS Structure (Layered / Monolithic / Microkernel) • Concept of Virtual Machine

1. Concept of Operating System

1.1 What is an Operating System? (Definition)

An Operating System (OS) is a system software program that sits between the computer hardware and the user. It manages all the hardware resources (CPU, memory, storage, input/output devices) and provides a friendly environment where the user can run application programs (like browser, games, editors) without worrying about the hardware details.

In simple words: the OS is like a translator and manager. The user speaks a simple language (click, type, run a program) and the OS converts that into instructions the hardware can understand, while also making sure everything is shared fairly and safely.

1.2 Why do we need an Operating System?

- **Hardware is too complex:** Writing a program that directly controls the CPU, memory chips, hard disk, and printer is extremely hard. OS gives simple functions (read a file, print, allocate memory) so programmers don't need to know hardware details.
- **Resource sharing:** A computer has limited CPU, memory and devices, but many programs want to run. OS decides who gets the CPU when, and for how long.
- **Convenience:** Users want a simple way to run programs, save files, and use devices — OS gives a clean interface (CLI or GUI) for this.
- **Protection and security:** OS makes sure one program cannot damage another program's data, and unauthorized users cannot access the system.
- **Efficient use of resources:** OS makes sure CPU and memory are not wasted — for example, when one program is waiting for input, OS gives CPU to another program.

1.3 How does an Operating System work internally?

The core part of the OS is called the Kernel. The kernel always stays in the main memory (RAM) once the computer boots up. It runs in a special protected mode called Kernel Mode (or Supervisor Mode), which has full access to hardware. Normal application programs run in User Mode, which has restricted access.

When a user program needs something from the hardware (like reading a file), it cannot touch the hardware directly. It must ask the OS using a System Call. This request causes the CPU to switch from user mode to kernel mode, the OS performs the task safely, and then control is returned back to the user program. (System Calls are explained in detail in Topic 5.)

 **Diagram Suggestion (search on Google Images):** Draw a simple 3-layer box diagram: Top box = 'Application Programs (Word, Browser, Games)', Middle box = 'Operating System (Kernel)', Bottom box = 'Hardware (CPU, Memory, Disk, I/O devices)'. Draw arrows going down from Application to OS to Hardware, and label the OS box as 'sits between user and hardware'. Search Google Images: "operating system layers diagram hardware kernel application".

1.4 Main Functions / Roles of an Operating System

Function	Explanation
Process Management	Creating, scheduling, and terminating processes; deciding which process uses CPU and when.

Function	Explanation
Memory Management	Allocating and freeing RAM to processes, keeping track of which part of memory is used or free.
File Management	Creating, deleting, reading, writing files and organizing them into folders/directories.
Device Management	Controlling I/O devices (keyboard, mouse, printer, disk) through device drivers.
Security & Protection	Preventing unauthorized access and stopping one process from interfering with another.
User Interface	Providing CLI (Command Line Interface) or GUI (Graphical User Interface) so users can interact with the system.

1.5 Operating System vs Other Software (How it is different)

Type	Category	Purpose	Example
Operating System	System software	Manages hardware and provides platform to run other software	Windows, Linux, Android
Application Software	User-level software	Performs specific user tasks using OS services	MS Word, Chrome, VLC
Firmware	Low-level software stored in ROM/Flash	Controls hardware at the most basic boot level, loads OS	BIOS, UEFI

1.6 Real World Analogy

Think of a big hotel. The hotel building, kitchen, rooms are the hardware. The guests are the application programs who want service. The Hotel Manager (the Operating System) takes requests from guests, decides which room is free, manages the staff (CPU time), keeps the accounts (accounting), and makes sure one guest doesn't disturb another guest's room (protection). The guest never talks to the kitchen staff directly — everything goes through the manager.

1.7 Importance / Use

Without an OS, every application would need to include code to directly control hardware, making software development extremely difficult, unsafe, and non-portable. The OS gives a stable, uniform platform so software runs efficiently across different machines and multiple users/programs can safely share one computer.

2. Generations of Operating Systems

Operating Systems have evolved over decades along with changes in computer hardware technology (from vacuum tubes to modern chips). Each generation solved the limitations of the previous one.

Generation	Hardware Used	Key Development	Examples
0th Gen (1940s)	Vacuum tubes, plugboards	No OS existed. Machines were programmed manually using wires and switches. Only one problem could be solved at a time.	ENIAC
1st Gen (1950s)	Vacuum tubes, punch cards	Introduced Batch Processing OS. Jobs with similar needs were grouped (batched) and run one after another automatically by an operator, using punch cards.	GM-NAA I/O
2nd Gen (1955-1965)	Transistors	Batch systems improved with Spooling (Simultaneous Peripheral Operation On-Line) and early Multiprogramming — CPU idle time was reduced by loading multiple jobs.	IBM 1401
3rd Gen (1965-1980)	Integrated Circuits (ICs)	Multiprogramming became common. Time-Sharing Systems allowed many users to use the computer at once (each got a small CPU time slice). OS became a separate field of study.	UNIX, Multics, IBM OS/360
4th Gen (1980-present)	VLSI, Microprocessors, Personal Computers	OS with GUI (Graphical User Interface) became common. Networking and Distributed OS emerged, allowing computers to communicate.	MS-DOS, Windows, Linux, macOS
5th Gen (Present/Future)	Mobile devices, Cloud, AI	OS for smartphones, cloud computing platforms, and OS that use AI for smart resource management and voice-based interfaces.	Android, iOS, Cloud OS

2.1 Why studying generations matters

Understanding generations helps you see WHY certain OS concepts (like multiprogramming or time-sharing) were invented — each was a solution to a real hardware limitation of that time, such as wasted CPU time or single-user restriction. This context makes later topics (like OS types) much easier to understand instead of memorizing them.

3. Types of Operating Systems

As computing needs changed, different types of OS were designed for different situations — some for speed, some for many users, some for real-time control. Below is a deep look at each type.

3.1 Batch Operating System

Definition: Jobs with similar requirements are grouped together into a 'batch' and executed one after another without user interaction during execution.

- **How it works:** An operator collects similar jobs, loads them together. The OS executes them in sequence, and results are collected later (like a printed report).
- **Why needed:** In early computers, setting up a job (loading tape/cards) took a long time compared to actual execution — batching many jobs reduced this setup overhead per job.
- **Drawback:** No interaction with the program while it runs; if a job has an error, the CPU may sit idle till the operator notices.
- **Example:** Payroll processing, bank statement generation.

3.2 Multiprogramming Operating System

Definition: Multiple programs are kept in the main memory at the same time, and the CPU switches between them so that CPU is never idle.

- **How it works:** When the currently running program needs to wait for I/O (like reading from disk), the OS immediately switches CPU to another ready program instead of letting CPU sit idle.
- **Why needed:** In batch systems, CPU used to sit idle whenever a job did I/O — multiprogramming keeps CPU utilization high.
- **Example:** Early IBM mainframe systems.

3.3 Multitasking Operating System

Definition: An extension of multiprogramming where CPU switches between programs so fast (using time slices) that it feels like multiple programs are running at the same instant.

- **How it works:** Uses a technique called Time-Sharing internally — each process gets a small slice of CPU time (like 10-100 milliseconds) in round-robin fashion.
- **Example:** You can type in MS Word while music plays in the background and a file downloads — this is multitasking.

3.4 Time-Sharing Operating System

Definition: A logical extension of multiprogramming where the CPU time is divided (shared) among multiple users who are connected through terminals, so each user gets the feeling that the computer is dedicated only to them.

- **How it works:** Each user's program gets a quantum (fixed small time) of CPU. After the quantum ends, CPU moves to the next user's process. Because the quantum is very small, response feels instant.
- **Example:** UNIX systems used by many students at once from different terminals.

3.5 Multiprocessing Operating System

Definition: An OS that uses two or more CPUs (processors) within a single computer, and can execute multiple processes at literally the same time (true parallelism), not just switching quickly.

- **Types:** Symmetric Multiprocessing (all CPUs are equal and share the same OS copy) and Asymmetric Multiprocessing (one master CPU controls others).
- **Advantage:** More reliable (if one CPU fails, others continue) and higher throughput.
- **Example:** Modern multi-core computers running Windows/Linux.

3.6 Distributed Operating System

Definition: Manages a group of physically separate computers (connected via a network) and makes them appear to the user as a single unified computer system.

- **How it works:** Resources (files, processing power) are spread across multiple machines. The OS handles communication and coordination between them using message passing.
- **Advantage:** If one node/machine fails, the rest of the system can still keep working (fault tolerance); resources can be shared across locations.
- **Example:** Google's server clusters, LOCUS.

3.7 Network Operating System (NOS)

Definition: Runs on a server and gives the ability to manage data, users, groups, security, and other networking functions over a Local Area Network (LAN).

- **Key difference from Distributed OS:** In NOS, users know which machine (server) they are connecting to and computers keep their own independent identity, unlike distributed OS where the whole network looks like one single computer.
- **Example:** Novell NetWare, Windows Server.

3.8 Real-Time Operating System (RTOS)

Definition: An OS designed to process input and give output within a strict, guaranteed time limit (deadline). The correctness of the system depends not only on the result but also on the time at which the result is produced.

Type	Meaning	Example
Hard Real-Time	Missing a deadline causes total system failure or is unacceptable.	Airbag deployment system, Pacemaker, Missile control system
Soft Real-Time	Missing a deadline is undesirable but system still keeps working (with reduced quality).	Online video streaming, Video games

3.9 Embedded Operating System

Definition: A specialized, lightweight OS built into a device to perform a specific, dedicated task, unlike a general-purpose OS (like Windows) that runs many kinds of programs.

- **Why needed:** Devices like washing machines, ATMs, or cars have limited memory/CPU, so they need a small, efficient OS designed only for their fixed job.
- **Example:** Embedded Linux in a smart TV, OS inside a washing machine controller, ATM machine OS.

3.10 Mobile Operating System

Definition: An OS specifically designed for smartphones and tablets, optimized for touch input, small screens, and battery efficiency.

- **Example:** Android, iOS.

3.11 Quick Comparison Table of OS Types

OS Type	User Interaction	Core Idea	Complexity
Batch	No	Multiple similar jobs, no interaction	Low
Multiprogramming	No (background)	Keep CPU busy using multiple jobs in memory	Medium
Time-Sharing	Yes	Many users share CPU with quick switching	Medium-High
Multiprocessing	Yes	True parallel execution with multiple CPUs	High
Distributed	Yes	Multiple computers look like one system	High
Real-Time	Yes (strict deadline)	Must respond within a fixed time	Very High (time-critical)

4. OS Services

OS Services are the useful functions that an Operating System provides to users and to the programs running on it, so that both find the computing environment convenient and safe.

4.1 Why does OS provide services?

Programs and users need common operations again and again — running a program, reading/writing files, using devices, detecting errors. If every application had to write its own code for these from scratch, it would be repetitive, error-prone and unsafe. So the OS provides these as ready-made, trusted services.

4.2 List of OS Services (Detailed)

- **Program Execution:** OS loads a program into memory, allocates required resources, runs it, and terminates it (either normally or due to an error).
- **I/O Operations:** Since user programs cannot directly access I/O devices for safety and hardware-control reasons, the OS provides a way to perform input/output through system calls.
- **File System Manipulation:** OS allows programs to create, delete, read, write, search files and manage permissions on them.
- **Communication:** OS allows information exchange between processes, either on the same computer (using shared memory or message passing) or over a network.
- **Error Detection:** OS constantly checks for possible errors in CPU, memory hardware, I/O devices, or user programs (like divide-by-zero) and takes corrective action.
- **Resource Allocation:** When multiple users or jobs run at the same time, OS allocates resources (CPU, memory, storage) to each of them fairly and efficiently.
- **Accounting:** OS keeps track of which users/programs use how much and which kind of resources, useful for billing or usage statistics.
- **Protection and Security:** Protection ensures controlled access to resources (so one process cannot interfere with another); Security ensures the whole system is safe from outside/unauthorized users.
- **User Interface:** OS provides Command Line Interface (CLI, text commands) or Graphical User Interface (GUI, icons/windows) so users can interact with the system easily.

4.3 Classification of Services

OS Services can be broadly grouped into two types:

Category	Examples of services
Services that help the USER	Program execution, I/O operations, File manipulation, Communication, Error detection, User Interface
Services that help EFFICIENT operation of the SYSTEM itself	Resource allocation, Accounting, Protection and Security

 **Diagram Suggestion (search on Google Images):** Draw a central circle labelled 'Operating System' with 9 arrows/spokes going outward to small boxes labelled: Program Execution, I/O Operations, File Manipulation, Communication, Error Detection, Resource Allocation, Accounting, Protection & Security, User Interface. This looks like a wheel/star diagram. Search Google Images: "operating system services diagram wheel".

5. System Calls

5.1 What is a System Call? (Definition)

A System Call is the programming interface (a special function) through which a user program requests a service from the Operating System's kernel. It is the only way for a user program to safely access hardware or kernel-level resources such as files, memory, or devices.

5.2 Why do we need System Calls?

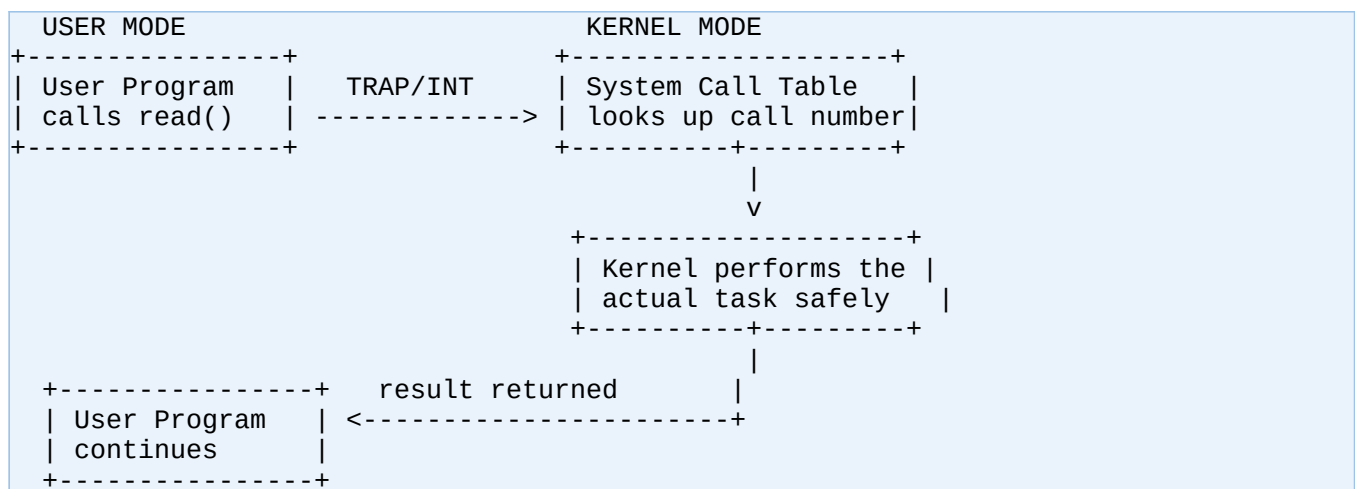
The CPU works in two modes for safety:

Mode	Description	Example
User Mode	Restricted mode. Normal application programs run here. Cannot directly access hardware or critical memory.	Running a calculator app, a text editor
Kernel Mode (Supervisor Mode)	Privileged mode with full hardware access. Only the OS kernel runs here.	Reading a file from disk, allocating memory

If a user program is allowed to directly touch hardware, a buggy or malicious program could crash the system or steal another program's data. So whenever a user program needs a critical task done (like opening a file), it must request the kernel using a system call — the kernel checks permissions and does the task safely on the program's behalf.

5.3 How a System Call Works Internally (Step by Step)

- 1. The user program calls a library function (e.g., `read()` in C), which internally prepares a system call.
- 2. A special CPU instruction called TRAP (or software interrupt) is executed. This instruction switches the CPU from User Mode to Kernel Mode.
- 3. The System Call Number (an ID identifying which service is needed, e.g., 'open a file') is placed in a CPU register.
- 4. The kernel looks up this number in a System Call Table, which points to the correct kernel function to execute.
- 5. The kernel executes the requested task (like actually reading data from the disk).
- 6. Control switches back to User Mode, and the result is returned to the user program.



5.4 Types of System Calls

Type	Purpose	Example (Unix/C)
Process Control	Create, terminate, load, execute, or wait for a process.	fork(), exec(), exit(), wait()
File Management	Create, delete, open, close, read, write a file.	open(), read(), write(), close()
Device Management	Request/release a device, read/write from a device.	ioctl(), read(), write()
Information Maintenance	Get or set system data like time, date, process ID.	getpid(), time(), alarm()
Communication	Send/receive messages between processes, create/use network connections.	pipe(), send(), recv()

5.5 C Code Walkthrough — System Call in Action

Below is a simple C program that uses system calls `open()`, `write()`, and `close()` to write text into a file. Notice these functions look like normal functions, but internally they trigger a mode switch to the kernel:

```
#include <fcntl.h>    // for open()
#include <unistd.h>    // for write(), close()

int main() {
    int fd;
    char *msg = "Hello Operating System!\n";

    // System call: ask kernel to open/create a file
    fd = open("notes.txt", O_WRONLY | O_CREAT, 0644);

    // System call: ask kernel to write data to the file
    write(fd, msg, 25);

    // System call: ask kernel to close the file
    close(fd);

    return 0;
}
```

Explanation: `open()`, `write()`, and `close()` are NOT ordinary C functions that just run in user space — they are wrappers that internally issue a TRAP instruction to switch to kernel mode, because only the kernel is allowed to actually talk to the file system and hard disk. Once the kernel finishes the task, control returns back to your program.

5.6 System Calls vs Library Functions (Common Confusion)

Term	Meaning	Example
System Call	Direct request to the kernel; causes mode switch (user to kernel mode)	open(), fork(), read()
Library Function	Normal function that runs in user mode; may internally use a system call, or may not need one at all	printf(), strlen(), sqrt()

Important point: `printf()` is a library function, but internally it eventually calls the `write()` system call to actually display characters on the screen. `strlen()` however never needs the kernel — it just counts characters in memory using user-mode code only.

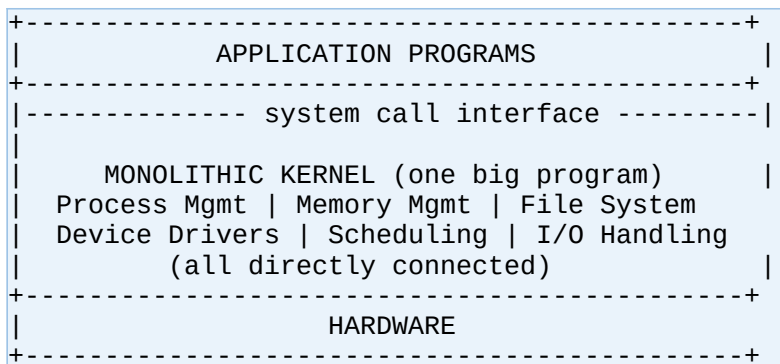
6. Structure of an Operating System

The 'structure' of an OS means how its internal components (process manager, memory manager, file system, device drivers, etc.) are organized and how they communicate with each other. A good structure makes the OS easier to design, more reliable, and easier to modify.

6.1 Monolithic Structure (Simple Structure)

Definition: In a monolithic structure, the entire operating system runs as one single large program in kernel mode. All the OS services (process management, memory management, file system, device drivers) are compiled together into one big kernel, and every part can directly call every other part.

- **How it works:** There is no strict boundary between different OS components internally — it is one big blob of code with direct function calls between modules, all running with full kernel privileges.
- **Advantage:** Very fast, because components call each other directly without overhead.
- **Disadvantage:** Hard to maintain and debug — a bug (like a bad device driver) in any part can crash the entire OS, because everything shares the same memory space and privilege level.
- **Example:** Original UNIX, MS-DOS, and much of the traditional Linux kernel.



6.2 Layered Structure

Definition: The OS is broken down into a number of layers (levels), stacked on top of each other. Layer 0 is the hardware, and the topmost layer is the user interface. Each layer can only use the functions and services of the layer directly below it.

- **How it works:** This is based on the concept of Abstraction / Information Hiding — a layer does not need to know HOW the layer below it works, it just needs to know WHAT services that layer provides.
- **Advantage:** Easy to design, debug, and update — you can modify one layer without touching others, as long as the interface between layers stays the same.
- **Disadvantage:** Slower than monolithic, because a request may have to pass through many layers one by one; also, careful layer design is needed since a layer can use only layers below it (not above).
- **Example:** THE Operating System (by Dijkstra) — one of the first layered OS designs.

```
Layer 5: User Programs
Layer 4: I/O Management
Layer 3: Operator-Process Communication
Layer 2: Memory Management
Layer 1: CPU Scheduling
```

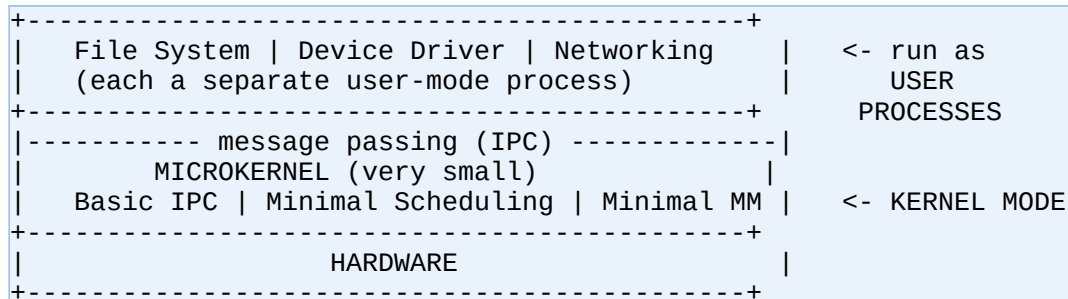
Layer 0: Hardware

(Each layer can call only the layer directly below it)

6.3 Microkernel Structure

Definition: In a microkernel structure, only the most essential functions (like basic inter-process communication, minimal process scheduling, and basic memory management) are kept inside the kernel. All other services (file system, device drivers, networking) are moved OUT of the kernel and run as separate user-mode processes.

- **How it works:** Since most services now run in user mode as separate processes, they communicate with each other and with the tiny kernel using message passing instead of direct function calls.
- **Advantage:** More reliable and secure — if a device driver (running in user mode) crashes, it does not bring down the entire OS, because it is isolated from the kernel; also easier to extend the OS with new services.
- **Disadvantage:** Slower performance compared to monolithic, because of the extra overhead of message passing between user-mode services and the kernel.
- **Example:** QNX, early Mac OS X (Mach-based), Minix.



6.4 Comparison: Monolithic vs Layered vs Microkernel

Feature	Monolithic	Layered	Microkernel
Basic Idea	One big kernel program	Kernel divided into levels	Only essential services in kernel; rest as user processes
Communication	Direct function calls	Layer calls layer below	Message passing (IPC)
Speed	Fastest	Medium	Slowest (due to IPC overhead)
Reliability	Low (one crash can affect all)	Medium	High (isolated services)
Ease of modification	Difficult	Easy (per layer)	Easy (add/remove services)
Example	Original UNIX, MS-DOS	THE OS	QNX, Minix

7. Concept of Virtual Machine

7.1 What is a Virtual Machine? (Definition)

A Virtual Machine (VM) is a software-based simulation of a complete physical computer. It gives the illusion to each user/program that they have their own separate, dedicated machine (with its own CPU, memory, and I/O devices), even though multiple virtual machines may actually be sharing the same underlying physical hardware.

7.2 Why do we need Virtual Machines?

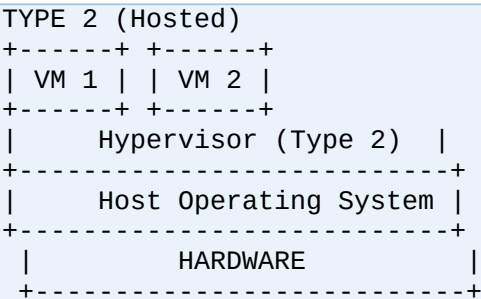
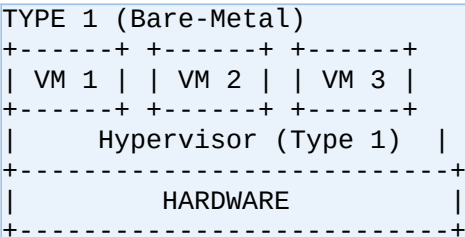
- **Run multiple OS on one machine:** You can run Windows and Linux at the same time on a single physical computer, without needing two separate machines.
- **Isolation and safety:** If something crashes or gets a virus inside one VM, other VMs and the real physical machine remain unaffected.
- **Testing and development:** Developers can test software on many different 'virtual' computer setups without buying extra hardware.
- **Efficient hardware usage:** Businesses/cloud providers can run many virtual servers on one big physical server, saving cost and power.

7.3 How Virtual Machines Work Internally

The key software that creates and manages virtual machines is called the Virtual Machine Monitor (VMM) or Hypervisor. It sits between the physical hardware and the virtual machines, and divides the real hardware's resources (CPU time, memory, disk space) among several virtual machines, giving each one the illusion of owning the entire computer.

7.4 Types of Hypervisors

Type	Description	Notes	Example
Type 1 (Bare-Metal)	Runs directly on the physical hardware, without a host OS underneath it.	Faster and more efficient, used in servers/data centers.	VMware ESXi, Microsoft Hyper-V, Xen
Type 2 (Hosted)	Runs as an application on top of an existing host operating system.	Easier to install/use, but slightly slower due to extra host OS layer.	VMware Workstation, Oracle VirtualBox



7.5 Virtual Machine vs Operating System (Key Difference)

Aspect	Operating System	Virtual Machine
Purpose	Manages real physical hardware directly	Simulates a complete computer on top of real

Aspect	Operating System	Virtual Machine
		hardware
Runs on	Directly on hardware	On a hypervisor, which itself runs on hardware (or on a host OS)
Can host what	Runs application programs	Can run a full separate guest Operating System inside it

7.6 Real Examples

Oracle VirtualBox and VMware Workstation let a student install Windows inside their Linux laptop (or vice versa) for practice. Cloud platforms like AWS, Google Cloud, and Microsoft Azure rent out virtual machines to companies instead of physical servers, so companies can scale up or down easily without buying real hardware.

Quick Revision Summary

Topic	One-Line Recap
Concept of OS	OS is the system software that manages hardware and gives a convenient, safe platform to run programs.
Generations of OS	OS evolved from no-OS (0th gen) to batch, multiprogramming, time-sharing, PC/GUI, and now mobile/cloud/AI-based systems.
Types of OS	Different OS designs (Batch, Multiprogramming, Time-Sharing, Multiprocessing, Distributed, Network, Real-Time, Embedded, Mobile) suit different needs like speed, multiple users, or strict deadlines.
OS Services	OS gives ready-made services like program execution, I/O, file handling, communication, error detection, resource allocation, accounting, protection/security, and user interface.
System Calls	The only safe way for a user program to request kernel services; causes a switch from user mode to kernel mode via a TRAP instruction.
Monolithic Structure	Entire OS is one big program in kernel mode — fast but hard to maintain and less reliable.
Layered Structure	OS divided into layers, each using only the layer below it — easy to design/debug but slower.
Microkernel Structure	Only essential services stay in the kernel; rest run as separate user processes using message passing — reliable but slower due to IPC overhead.
Virtual Machine	A software simulation of a full computer, managed by a Hypervisor/VMM, allowing multiple OS to run on one physical machine.

15 Exam-Style Questions — Unit 1

- Q1.** Define Operating System. Explain why an Operating System is needed with suitable examples.
- Q2.** Explain the functions of an Operating System in detail.
- Q3.** Describe the various generations of Operating Systems with their key characteristics.
- Q4.** What is a Batch Operating System? Explain its working with advantages and disadvantages.
- Q5.** Differentiate between Multiprogramming, Multitasking, and Multiprocessing Operating Systems.
- Q6.** What is a Time-Sharing Operating System? How is it different from a Batch Operating System?
- Q7.** Explain Distributed Operating System and Network Operating System with suitable examples. How are they different?
- Q8.** What is a Real-Time Operating System? Explain Hard Real-Time and Soft Real-Time systems with examples.
- Q9.** List and explain any six services provided by an Operating System.
- Q10.** What is a System Call? Explain how a system call switches the CPU from user mode to kernel mode.
- Q11.** Explain the different types of system calls with suitable examples for each type.
- Q12.** Differentiate between a System Call and a Library Function with examples.
- Q13.** Explain the Monolithic structure of an Operating System with a suitable diagram. State its advantages and disadvantages.
- Q14.** Explain the Layered approach and Microkernel structure of an Operating System with diagrams, and compare them.
- Q15.** What is a Virtual Machine? Explain the concept of Hypervisor and differentiate between Type 1 and Type 2 hypervisors.