CSE 3204

OPERATING SYSTEMS

Chapter 1: Introduction to Operating Systems & Structures

Comprehensive Teaching Notes with Case Studies & Real-World Scenarios

Designed for students with no prior knowledge of Operating Systems

Based on: Operating System Concepts by Silberschatz, Galvin and Gagne, 7th Edition

Department of Computer Science & Engineering
Adama Science & Technology University (ASTU)
January 2026

How to Use These Notes

Welcome to CSE 3204 - Operating Systems. These notes are designed to guide you through every concept from the ground up, assuming you have no prior experience with operating systems. Each section builds on the previous one, and throughout you will find:

Symbol	Meaning
📋 CASE STUDY	A real-world example of how the concept is applied in practice
🌍 REAL-WORLD SCENARIO	A scenario from everyday life that connects to the concept
💡 KEY CONCEPT	An important definition or idea you must understand
⚠️ COMMON MISTAKE	A frequent misunderstanding to watch out for
🧠 THINK ABOUT IT	A question to prompt your own thinking before reading on

1. What Is an Operating System?

1.1 The Big Picture — Starting From Zero

Imagine you just bought a brand-new laptop. You turn it on. You see a screen, maybe a desktop with icons, and you can open apps like Chrome or Word. But how does clicking an icon translate into running a program? How does typing on your keyboard result in text appearing on screen? Who is managing all of this behind the scenes?

The answer is the Operating System (OS). It is the invisible layer of software that makes your computer usable. Without it, you would need to know exactly which memory address to write to, which electrical signals to send to the disk, and how to coordinate every hardware component yourself — in machine code. The OS handles all of that so you (and your applications) do not have to.

> 💡 **KEY CONCEPT: What is an Operating System?**
> An Operating System is a program that acts as an intermediary between a user of a computer and the computer hardware. Its goals are: (1) to execute user programs and make solving user problems easier, (2) to make the computer system convenient to use, and (3) to use the computer hardware in an efficient manner.

1.2 Two Roles of an Operating System

The OS plays two fundamental roles that may seem to be in tension with each other:

Role 1: Resource Allocator

A modern computer has many resources: the CPU (processor), memory (RAM), storage (hard disk or SSD), network cards, printers, and more. Multiple programs and users all want to use these resources. The OS is the manager that decides who gets what resource, when, and for how long.

> 🌍 **REAL-WORLD SCENARIO: Airport Traffic Control**
> Think of the OS as an air traffic controller at a busy airport. Multiple planes (programs) all want to land and take off (use the CPU and memory). The controller (OS) ensures that planes do not collide, assigns runways (resources) fairly, and handles emergencies (errors). Without the controller, there would be chaos — just as a computer without an OS would be unusable.

Role 2: Control Program

The OS also acts as a supervisor or referee. It controls the execution of programs to prevent errors and improper use of the computer. If a program tries to read memory that belongs to another program, the OS steps in and stops it.

📋 **CASE STUDY: Google Chrome Sandbox**

When you open a malicious website in Chrome, the browser's JavaScript code could theoretically try to access files on your hard drive. The OS prevents this by enforcing memory boundaries — Chrome's tab process runs in a restricted 'sandbox' and cannot access memory regions assigned to other processes or to the OS itself. This is the OS acting as a control program.

1.3 The Kernel — The Heart of the OS

💡 **KEY CONCEPT: The Kernel**

The kernel is 'the one program running at all times on the computer.' It is the core of the OS. Everything else is either a system program (ships with the OS, like the file manager or shell) or an application program (installed by users, like Chrome or Word). The kernel is always in memory and always running.

When you hear about the Linux kernel, the Windows NT kernel, or the macOS XNU kernel, they are referring to this fundamental, always-running core. The kernel is the part of the OS that has direct access to hardware.

1.4 Where the OS Lives in the System Stack

The computer system has four layers, stacked from bottom to top:

Layer	Examples & Description
1. Hardware (bottom)	CPU, RAM, hard disk, keyboard, mouse, GPU — the physical components
2. Operating System	Windows, Linux, macOS, Android — manages hardware and provides services
3. System Programs	Shells (bash, cmd), compilers (gcc), file managers — ship with the OS
4. Application Programs (top)	Chrome, Word, Spotify, games — installed by users to solve problems

Users interact with application programs. Application programs interact with the OS (through system calls). The OS interacts with hardware. This layering is what makes modern computers usable without needing to understand electronics.

🌍 **REAL-WORLD SCENARIO: Restaurant Analogy**

Think of the OS like the kitchen management system in a restaurant. Customers (users) only interact with the menu and waiters (application programs). Waiters send orders to the kitchen (OS). The kitchen manages the chefs, ovens, and ingredients (hardware) and produces results. Customers never need to know how the oven works or which chef is available.

2. Computer System Structure

2.1 The Four Components

A complete computer system has four components that work together. Understanding how they relate to each other is foundational:

Component	Description
User	The person, machine, or other computer that needs the system to do work
Application Programs	Software (compiler, text editor, database, browser) that solves specific user problems
Operating System	Controls and coordinates hardware use among various users and applications
Computer Hardware	Physical components: CPU, memory (RAM), I/O devices (disk, keyboard, monitor)

The key insight here is the hierarchy: users talk to apps, apps talk to the OS, the OS talks to hardware. No layer skips another — this discipline is what keeps the system stable and secure.

> 📋 **CASE STUDY: Microsoft Word and the File System**
>
> When you click 'Save' in Microsoft Word (an application), Word does NOT directly write to the hard disk. Instead, it calls an OS service: 'Please write these bytes to this file.' The OS (say, Windows) then uses its file system driver to find where on the physical disk to write, manages buffering the data, and coordinates with the disk controller hardware. Word never touches the hardware directly. This separation is why Word works on both a spinning hard drive and an SSD — it just asks the OS, and the OS handles the differences.

2.2 Computer System Organization

Inside a computer, multiple components need to share resources, particularly the system memory (RAM). Here is how this is organized:

- One or more CPUs and multiple device controllers are all connected through a common bus (a shared communication channel).
- Each device controller is responsible for a specific type of device: the disk controller manages disks, the USB controller manages USB devices, the graphics adapter manages the screen, etc.
- All these components share access to the main memory (RAM) over the bus.
- CPUs and I/O devices can execute concurrently — they run simultaneously, competing for memory access.

💡 KEY CONCEPT: Device Controller

A device controller is a small hardware component that acts as the 'manager' for a specific device. It has its own local buffer (a small, fast memory). When the CPU wants to read from a disk, it tells the disk controller what to fetch, and the controller goes off and does the work. When it is done, it notifies the CPU via an interrupt. This allows the CPU to do other work instead of waiting.

🌍 REAL-WORLD SCENARIO: Hotel Concierge

The CPU is like a hotel manager who has many requests to handle. Instead of personally running errands (fetching data from disk, waiting for a print job), the manager delegates to concierges (device controllers), each responsible for a specific service. The concierge handles the task and sends a notification (interrupt) when done. The manager can handle other guests in the meantime — this is how concurrency improves efficiency.

3. Computer System Operation

3.1 How a Computer Starts — The Bootstrap

Ever wondered what actually happens in those few seconds between pressing the power button and seeing your desktop? That process is called bootstrapping (or booting).

> 💡 **KEY CONCEPT: Bootstrap Program**
> The bootstrap program is a small piece of code loaded at power-up or reboot. It is stored in ROM (Read-Only Memory) or EEPROM, which is why it persists even when the power is off. This type of storage is called firmware. The bootstrap program: (1) initializes all aspects of the system (checks RAM, detects hardware), (2) finds the OS kernel on disk, and (3) loads the kernel into memory and starts its execution.

> 📋 **CASE STUDY: BIOS/UEFI on a PC**
> On every modern PC, when you press the power button, the CPU begins executing code from a chip on the motherboard called BIOS (older systems) or UEFI (modern systems). This is the firmware/bootstrap. It performs a Power-On Self Test (POST) — checking that RAM is working, disks are detected, etc. It then searches for a bootloader (e.g., GRUB on Linux, Windows Boot Manager on Windows) on a disk, loads it into RAM, and hands over control. The bootloader then loads the OS kernel. This whole sequence is why your computer takes a few seconds to start.

3.2 Interrupts — How the CPU Stays Efficient

Here is a fundamental problem: the CPU is extremely fast (billions of operations per second), but I/O devices like disks and network cards are relatively slow. If the CPU had to wait idle for each I/O operation to finish, it would waste most of its time doing nothing. Interrupts solve this problem.

> 💡 **KEY CONCEPT: Interrupt**
> An interrupt is a signal from a hardware device or software that tells the CPU: 'Something needs your attention.' When a device finishes its work (e.g., disk read complete, key pressed), it sends an interrupt signal to the CPU. The CPU pauses its current task, handles the interrupt, and then resumes. This allows the CPU to execute other instructions while an I/O operation is in progress, dramatically improving efficiency.

Type of Interrupt	Description & Example
Hardware Interrupt	Triggered by a device. Example: You press a key on the keyboard — the keyboard controller sends an interrupt to signal the CPU.
Software Interrupt (Trap)	Triggered by a program via a special instruction (system call) or by an error like division by zero.
Exception	An internal interrupt caused by an error — illegal memory access, overflow, etc.

How Interrupts Work — Step by Step

1. CPU is executing Program A.
2. CPU tells disk controller: 'Read sector 1042 and put the data in buffer X.'
3. CPU continues executing Program A (or switches to Program B) — it does NOT wait.
4. Disk controller finishes reading. It sends an interrupt signal to the CPU.
5. CPU saves its current state (so it can resume later), then jumps to the Interrupt Service Routine (ISR) — a special OS function that processes the interrupt.
6. ISR processes the event (e.g., copies data from buffer to the right place).
7. CPU restores its saved state and resumes the previous program.

💡 KEY CONCEPT: Interrupt Vector

The interrupt vector is a table in memory that maps each type of interrupt to the address of its Interrupt Service Routine (ISR). When interrupt #14 fires (say, a page fault), the CPU looks up entry 14 in the vector to find the address of the ISR for page faults, and jumps to it. This lookup takes only nanoseconds.

💡 KEY CONCEPT: Trap

A trap (also called a software interrupt or exception) is an interrupt caused by software — either by a deliberate system call (a program asking the OS for a service) or by an error (like dividing by zero). Traps allow user programs to request OS services in a controlled way.

🌍 REAL-WORLD SCENARIO: Hospital Emergency System

Think of interrupts like a hospital's paging system. A doctor (CPU) is treating a patient (running a program). When an emergency occurs in another room (I/O completes or error), the paging system buzzes (interrupt). The doctor saves notes about the current patient, handles the emergency, then returns. Without this system, the doctor would have to stand at the door waiting for emergencies instead of treating patients.

4. Operating System Structure

4.1 The Problem: One CPU, Many Programs

In the early days of computing, a computer ran only one program at a time. You submitted a job, it ran to completion, then the next job started. This approach (called serial processing) wasted enormous amounts of expensive CPU time whenever a job was waiting for I/O.

Think about it: if a program is reading a large file from disk, the CPU might be idle for 50-100 milliseconds. At modern speeds, the CPU could have executed millions of instructions in that time. How do we fix this?

4.2 Multiprogramming

> 💡 **KEY CONCEPT: Multiprogramming**
> Multiprogramming increases CPU utilization by keeping multiple jobs (programs) in memory simultaneously. When one job has to wait (e.g., for I/O), the OS switches the CPU to another job. The CPU is never left idle as long as any job is ready to run. A job scheduler selects which jobs are brought into memory; the CPU scheduler selects which ready job gets the CPU next.

Visually: memory is divided between the OS and several jobs. At any moment, one job is running. When it blocks (waits), another takes its place:

Memory Region	Contents
0 — low address	Operating System (kernel always resident in memory)
Above OS	Job 1 (e.g., spreadsheet computation)
Above Job 1	Job 2 (e.g., file download)
Above Job 2	Job 3 (e.g., email client checking server)
Top of memory (512M+)	Job 4 (e.g., background virus scan)

> 📋 **CASE STUDY: Unix on a 1970s Mainframe**
> When Ken Thompson and Dennis Ritchie developed Unix at Bell Labs in the early 1970s, the system needed to support multiple researchers sharing a single PDP-11 minicomputer. Multiprogramming was essential: when researcher A's data analysis was waiting for tape I/O, the CPU would switch to researcher B's compilation job. This allowed a $500,000 machine to serve a whole department efficiently — a revolutionary improvement over earlier batch systems where one job ran alone.

4.3 Timesharing (Multitasking)

💡 KEY CONCEPT: Timesharing

Timesharing is a logical extension of multiprogramming in which the CPU switches among jobs so rapidly that each user perceives they have their own interactive computer. The response time is typically less than 1 second. Since each user's command takes little CPU time, many users can share the system simultaneously. Each user has at least one process in memory.

Timesharing introduces several new requirements and concepts:

Concept	What It Solves / Means
Process	Each user program executing in memory is called a process. A process is a program in execution.
CPU Scheduling	When multiple processes are ready simultaneously, the OS must decide which one runs next.
Swapping	If processes do not fit in memory, some are temporarily moved to disk and brought back when needed.
Virtual Memory	Allows a process to run even if its full code/data is not in RAM — only the needed parts are loaded.
Response Time < 1 sec	The goal of timesharing — users expect immediate feedback when they type a command.

🌍 REAL-WORLD SCENARIO: Modern Smartphone

Your smartphone runs dozens of apps 'simultaneously' — Spotify playing music, WhatsApp checking for messages, Maps tracking your location, and Instagram refreshing your feed. In reality, your phone has a limited number of CPU cores. The OS constantly switches between these processes, giving each a tiny slice of time (hence 'timesharing'). The switching is so fast that everything appears to run simultaneously. When you switch from WhatsApp to Maps, the OS swaps WhatsApp's state to storage and loads Maps' state — this is swapping in action.

5. Operating System Operations

5.1 Why Protection Is Needed

Imagine if any program could do anything it wanted — read all files, overwrite memory used by the OS, or loop forever consuming all CPU time. A malicious or buggy program could crash the entire system or steal data from other users. We need mechanisms to protect the system and users from each other.

5.2 Dual-Mode Operation

> 💡 **KEY CONCEPT: Dual-Mode Operation**
> To protect itself and other system components, the OS (with hardware support) operates in two modes:User Mode (mode bit = 1): Execution is done on behalf of a user program. Certain dangerous instructions (privileged instructions) are NOT allowed in this mode.Kernel Mode / Monitor Mode (mode bit = 0): Execution is done on behalf of the OS. ALL instructions, including privileged ones, are allowed.The hardware has a single bit called the mode bit. When the CPU is in kernel mode, mode bit = 0. In user mode, mode bit = 1.

Examples of privileged instructions (only allowed in kernel mode):

- Accessing I/O devices directly
- Modifying memory protection tables
- Halting the CPU
- Changing the mode bit itself

5.3 The System Call — Crossing the Boundary

When a user program needs the OS to do something (read a file, allocate memory, send data over a network), it makes a system call. Here is exactly what happens:

8. User process is executing in user mode (mode bit = 1).
9. It executes a special system call instruction, which triggers a trap (software interrupt).
10. Hardware switches the mode bit to 0 (kernel mode).
11. The OS kernel executes the requested service (e.g., reads from disk).
12. When finished, the OS executes a return-from-system-call instruction.
13. Hardware resets mode bit to 1 (user mode). Control returns to user program.

> 📋 **CASE STUDY: Python's open() Function**
> When you write 'f = open("data.txt", "r")' in Python, you are ultimately making an OS system call — on Linux it becomes the 'open()' syscall, on Windows it becomes 'CreateFile()'. Python's file handling

code is a high-level API that wraps the low-level system call. The OS then checks if you have permission to read the file, finds it on disk, and returns a file descriptor. Your Python code (user mode) never directly touches the disk — the OS (kernel mode) does that safely.

5.4 Timer — Preventing CPU Hogging

What if a program accidentally (or deliberately) enters an infinite loop? It would consume the CPU forever, and no other program could run. How does the OS regain control?

> 💡 **KEY CONCEPT: Timer Interrupt**
>
> Before scheduling any process, the OS sets a hardware timer to interrupt after a specific time period (e.g., 10 milliseconds). The timer counts down. When it reaches zero, it generates a hardware interrupt, transferring control back to the OS. The OS can then decide whether to give the same process more time or switch to another. This is how the OS ensures no process monopolizes the CPU.

> 🌍 **REAL-WORLD SCENARIO: Kitchen Timer in a Restaurant**
>
> A chef (OS) manages multiple cooking orders (processes). Each item gets a specific oven time (timer). When the timer rings (interrupt), the chef checks if the dish is done. If not, it might get more time. But the oven (CPU) never just waits for one dish forever — the timer ensures all dishes get attention. Without timers, one dish could burn the kitchen (infinite loop crashing the system).

6. Computing Environments

A computing environment is a collection of computer hardware, software, machines, and networks that support the processing and interaction of electronic information. There are four major environments:

6.1 Traditional Environment

The original computing environment where office PCs were connected to a network, and terminals were attached to mainframes or minicomputers. These provided both batch processing (offline, automated jobs) and timesharing.

- Office environment: PCs connected to a LAN (Local Area Network)
- Now extended with portals — web interfaces that give networked access to resources from anywhere
- Home networks evolved from single machines, to dial-up modems, to broadband with firewalls and routers

> 📋 **CASE STUDY: University Computer Lab (1990s)**
>
> Students in a 1990 university computer lab sat at dumb terminals — screens and keyboards with no processing power. These terminals were wired to a central VAX mainframe running VMS or Unix. The mainframe's OS (timesharing) gave each student an interactive session. All processing happened on the mainframe. Students shared the CPU without knowing it. This is the classic traditional computing environment.

6.2 Client-Server Environment

> 💡 **KEY CONCEPT: Client-Server**
>
> In a client-server environment, machines are divided into two roles:- Clients: make requests for services- Servers: respond to those requestsTwo types of servers: (1) Compute-server — provides computation services (databases, scientific calculations), (2) File-server — provides file storage and retrieval services.

> 📋 **CASE STUDY: Gmail**
>
> When you open Gmail, your laptop/phone (client) sends an HTTP request to Google's mail servers (servers). The mail servers process your request, fetch your emails from a database, and send back the data. Your client just displays it. The heavy computation and storage happens on Google's server farm — your client just sends requests and shows results. This is client-server architecture at massive scale.

6.3 Peer-to-Peer (P2P) Environment

💡 **KEY CONCEPT: P2P**

In P2P, there is no distinction between clients and servers. Every node (peer) can act as both a client and a server. To join, a peer either registers with a central lookup service or uses a discovery protocol to broadcast its availability. P2P is resilient — there is no single point of failure.

📋 **CASE STUDY: BitTorrent**

When you download a file via BitTorrent (e.g., a Linux ISO), you connect to a 'swarm' of peers. Each peer shares pieces of the file they already have, and also downloads pieces they do not have. Your machine is simultaneously a client (downloading) and a server (uploading). There is no central server holding the file. This is P2P — robust, distributed, and efficient.

📋 **CASE STUDY: Bitcoin**

Bitcoin's blockchain is a P2P system. Every node stores the full transaction history and validates new transactions. There is no central bank. Transactions are broadcast to all peers, validated by consensus, and added to the blockchain. The OS on each Bitcoin node manages the networking, disk storage, and CPU needed for this P2P process.

6.4 Web-Based Environment

The web has transformed how we think about computing environments. Key developments:

- PCs became the predominant client devices, but are increasingly joined by phones, tablets, smart TVs, IoT devices
- Load balancers emerged as a new category of device to distribute web traffic among multiple servers
- OS evolution: from simple client OSes (Windows 95) to full-featured OSes that can be both clients and servers (Linux, Windows Server)

🌍 **REAL-WORLD SCENARIO: Netflix Streaming**

When you watch Netflix, your device (phone, TV, laptop) runs an OS that handles: (1) your browser or Netflix app running in user mode, (2) network stack (system calls to send/receive data), (3) video decoder (hardware-accelerated via OS drivers), (4) audio system. Netflix's servers — running Linux — use load balancers to distribute millions of simultaneous viewers across thousands of servers worldwide. Your OS and Netflix's OS are both critical parts of your viewing experience.

7. Special Purpose Systems

7.1 Real-Time Embedded Systems

> 💡 **KEY CONCEPT: Real-Time System**
> A real-time system has strict timing constraints — operations must complete within a fixed time deadline, or the system fails. They are used to monitor and control physical devices and usually have minimal or no user interface.

Type	Example
Hard Real-Time	Anti-lock braking system (ABS) in a car — if the brake response is 10ms late, the car crashes
Soft Real-Time	Video streaming — a late frame causes a visual glitch but not a crash
Embedded Controllers	Microwave oven, washing machine, thermostat, factory robot arms

> 📋 **CASE STUDY: Mars Rover — VxWorks RTOS**
> NASA's Mars rovers (Curiosity, Perseverance) run VxWorks, a Real-Time Operating System. On Mars, communication with Earth takes 5-20 minutes each way. The rover's OS must make immediate, autonomous decisions — 'there is a rock in the path, stop now' — with hard real-time guarantees. Missing a deadline by even 100ms could mean the rover drives off a cliff. Traditional OSes like Windows would not be acceptable here due to unpredictable delays.

7.2 Multimedia Systems

Multimedia systems handle audio and video data in addition to conventional data like text and spreadsheets. They require:

- Strict timing — video frames must arrive at 24 or 60 frames per second
- High bandwidth — HD video can require 100+ Mbps data throughput
- Special compression/decompression algorithms (codecs) often hardware-accelerated

7.3 Handheld / Mobile Systems

PDAs, smartphones, and tablets run specialized embedded operating systems:

- Limited battery — OS must aggressively manage power
- Limited memory — OS manages RAM carefully, may terminate background apps
- Touch input — OS handles gesture recognition as a core service
- Examples: Android (based on Linux kernel), iOS (based on XNU kernel)

📋 CASE STUDY: Android OS Architecture

Android phones run a Linux kernel at the bottom (device drivers, memory management, security). Above it is a Hardware Abstraction Layer (HAL). Above that is the Android Runtime (ART) which runs app code. Your apps (WhatsApp, Camera) run in the application layer. When WhatsApp wants to send a photo, it calls Android APIs, which call system services, which call the Linux kernel via system calls, which talk to the network hardware. This is the complete OS stack in your pocket.

8. Operating System Views

We can understand an OS from four different perspectives, each revealing different aspects of what it is and how it works:

View	The Question It Answers
Functional View	What does the OS do? (Its responsibilities)
Component View	What pieces is the OS made of? (Designer's perspective)
Services View	What does the OS offer to users and programmers?
Structure View	How is the OS implemented? (Architectural approach)

8.1 Functional View — What the OS Does

The OS performs the following functions:

Function	Description
Program Execution	Starting programs, managing their execution, communicating results
I/O Operations	Initiating and managing all I/O (disk reads, keyboard input, screen output)
File System Management	Creating, maintaining, manipulating files and directories
Communication	Between processes of the same user, and between different users/systems
Exception Detection & Handling	Detecting errors, power failures, printer out of paper, and responding
Resource Allocation	Processor scheduling, I/O scheduling, memory management
Accounting	Tracking resource usage per user for billing and performance analysis
Protection	Maintaining data integrity, keeping unauthorized users out, logging failed access attempts

8.2 Component View — What the OS Is Made Of

Processes

> 💡 **KEY CONCEPT: Process**
> A process is a program in execution. A program on disk is a passive entity (like a recipe in a book). A process is an active entity (like actually cooking the recipe). The OS creates a process when you run a program. Each process has its own memory space, open files, and execution state. A computer has

many concurrent processes: user processes (your apps) and OS (system) processes (background services).

Process management operations the OS must perform:

- Creation and termination of processes
- Suspension (pausing) and resumption of processes due to interrupts or context switches
- Synchronization — ensuring processes that share data or resources do so correctly
- Communication — inter-process communication (IPC) so processes can cooperate
- Deadlock detection and prevention — ensuring processes do not wait for each other forever

📋 **CASE STUDY: A Shell Running a Compiler**

When you type 'gcc main.c' in a terminal, here is what happens in terms of processes: (1) The shell (a process) reads your command, (2) The shell calls fork() to create a new child process, (3) The child process calls exec() to replace itself with the gcc compiler process, (4) gcc (now its own process) reads main.c, compiles it, and writes the output, (5) gcc terminates and sends its exit status back to the shell process, (6) The shell resumes and shows you the next prompt. Six distinct process management operations for one simple command.

Storage Management

- Managing main memory (RAM): allocating to active processes, tracking free vs. used regions, de-allocating when a process ends
- Managing secondary storage (disk): managing free sectors/tracks, allocating to programs, scheduling disk access requests

I/O Management

- Device drivers: accept I/O requests and invoke the correct device-specific routines
- Buffering: temporarily storing data in memory while it is being transferred between devices
- Caching: keeping frequently used data in faster memory for quicker access
- Spooling: overlapping I/O of one job with computation of others (classic example: print spooler)

File System

💡 **KEY CONCEPT: File**

A file is a named collection of related information defined by its creator. The OS provides the file system — an abstraction that maps logical files (names, directories) to physical storage locations on disk. Without the file system, you would have to remember exact disk sector numbers to find your data.

Networking

The OS provides the networking stack — protocols like TCP/IP, FTP, HTTP — that allow processes to communicate over a network. Every time your browser fetches a web page, the OS's network subsystem is handling the actual transmission and reception of data packets.

Protection

The OS controls access of programs, processes, and users to resources. It enforces permissions, handles authentication, and maintains logs of access attempts.

8.3 Services View — What the OS Offers

1. Command Interpreters (Shells)

A shell is a program that provides a user interface to the OS kernel. It reads commands from the user and executes them by making system calls.

Shell	Platform
bash (Bourne Again Shell)	Linux, macOS, Git Bash on Windows
zsh	macOS (default since Catalina), Linux
PowerShell	Windows (modern, also available on Linux)
cmd.exe	Windows (older, classic command prompt

📋 **CASE STUDY: Bash Script in CI/CD Pipeline**

Modern software teams use bash scripts in their Continuous Integration (CI) pipelines. When code is pushed to GitHub, a GitHub Actions runner (a Linux VM) executes bash scripts: checking out code, running tests, building Docker images, deploying to servers. Every command in that script is a shell interacting with the OS — creating processes (fork/exec), managing files (read/write system calls), setting up pipelines (|) that pass data between processes via inter-process communication. The shell is not just for beginners — it is a critical production tool.

2. System Calls

💡 **KEY CONCEPT: System Call**

A system call is a programming interface to the services provided by the OS. It is the mechanism by which a user program requests OS services. System calls are typically written in C/C++ but are accessed via high-level Application Programming Interfaces (APIs) in most programming languages. The three most common APIs are: Win32 API (Windows), POSIX API (Unix/Linux/macOS), and the Java API (JVM).

Category	Examples
Process Control	fork(), exec(), exit(), wait(), kill()
File Management	open(), read(), write(), close(), delete(), create()
Device Management	ioctl(), read(), write() (on device files)
Information Maintenance	getpid(), gettime(), alarm()
Communication	socket(), send(), recv(), pipe(), shmget()
Protection	chmod(), chown(), setuid(), umask()

🌍 REAL-WORLD SCENARIO: System Calls in Go

As a Go developer, when you write 'os.ReadFile("config.json")', Go's standard library internally calls the 'read' system call on Linux (or 'ReadFile' on Windows). You write clean Go code; the OS handles the hardware. Go's os package is an abstraction layer over system calls — exactly what the POSIX API is designed to provide. Understanding system calls helps you understand what happens underneath every file, network, and process operation in your programs.

3. System Programs

System programs come with the OS and provide a convenient environment for program development and execution:

- File manipulation: ls, cp, mv, mkdir, rm, find
- Status information: ps, top, df, date, who
- File modification: text editors (nano, vim, notepad)
- Programming language support: compilers (gcc), interpreters (python), linkers
- Program loading and execution: the loader, dynamic linker
- Communications: ssh, ftp, curl, ping

8.4 Structure View — How the OS Is Built

Different OSes are structured differently. Here are the four main architectural approaches, each with real-world examples:

Structure 1: Simple / Monolithic (MS-DOS approach)

> 💡 **KEY CONCEPT: Simple Structure**
> Early OSes like MS-DOS were written to provide maximum functionality in minimum space. They were not divided into modules — everything was in one layer. Interfaces and levels of functionality were not well separated, making the system vulnerable to buggy or malicious programs that could crash the entire system.

- No clean separation between user and kernel code
- Any program error could corrupt OS data
- Simple to develop initially, but difficult to maintain or extend

> 📋 **CASE STUDY: MS-DOS and Viruses**
> MS-DOS had no protection between user programs and the OS. This is precisely why early PC viruses (Brain, Stoned, Michelangelo in the late 1980s-90s) were so destructive — a program could directly write to the boot sector or overwrite OS code. There was no mode bit, no memory protection, no system call boundary. The virus ran in the same privilege level as the OS. This vulnerability motivated the design of protected-mode operating systems like Windows NT.

Structure 2: Layered Approach

> 💡 **KEY CONCEPT: Layered OS Structure**
> The OS is divided into N layers. Layer 0 is the hardware. Layer N (highest) is the user interface. Each layer can only use functions from lower layers. This strict discipline means bugs are localized — a bug in layer 3 cannot corrupt layer 1. But it adds overhead: a request may pass through many layers.

UNIX used a simplified layered approach. The kernel sits between the system-call interface and the physical hardware, providing: file system, CPU scheduling, memory management, device drivers, and much more — all in one large kernel layer.

> 📋 **CASE STUDY: Network Protocol Stack (TCP/IP)**
> The TCP/IP networking model is a perfect example of the layered approach applied to networking. Application layer (HTTP) -> Transport layer (TCP) -> Network layer (IP) -> Data Link layer (Ethernet) -> Physical layer (cables). Each layer only communicates with the layer directly above and below it. A web browser does not need to know about Ethernet frames; it just hands data to TCP. TCP does not

care if the underlying network is Ethernet or WiFi. This layering is why the same browser works on any network.

Structure 3: Microkernel

💡 KEY CONCEPT: Microkernel

A microkernel moves as much OS functionality as possible OUT of the kernel and into user-space processes. The kernel itself only handles: basic memory management, CPU scheduling, and inter-process communication (message passing). Everything else (file system, device drivers, network stack) runs as regular user-space processes.Benefits: More reliable and secure (less kernel code = fewer kernel bugs), easier to port to new hardware, easier to extend.Drawbacks: Performance overhead from user-to-kernel mode switches for each message.

Monolithic Kernel	Microkernel
File system in kernel	File system as user-space server process
Network stack in kernel	Network stack as user-space server process
All device drivers in kernel	Most device drivers in user space
Fast (no message passing)	Slower but more reliable (message passing)
Examples: Linux, Windows NT	Examples: Mach, MINIX, QNX, L4

📋 CASE STUDY: macOS X (Mach Microkernel)

macOS is built on a hybrid kernel combining the Mach microkernel (from Carnegie Mellon University) and BSD Unix. The Mach layer provides: IPC (message passing between processes), virtual memory management, and thread management. The BSD layer provides: POSIX API compatibility, file system, networking. Above both sits Apple's proprietary OS services. This is why macOS can run iOS apps (same kernel) and why it is highly stable — device driver crashes do not bring down the system.

Structure 4: Modular Kernel

💡 KEY CONCEPT: Modular Kernel

Most modern OSes implement loadable kernel modules (LKMs). The kernel has a small core, and additional functionality (file systems, device drivers, schedulers) can be loaded and unloaded dynamically at runtime without rebooting. It is like the layered approach but more flexible — modules can talk to each other, not just to the layer below.

📋 CASE STUDY: Linux Kernel Modules

Linux uses loadable kernel modules extensively. When you plug in a USB WiFi adapter, Linux automatically loads the correct driver module (e.g., rtlwifi for a Realtek chip). When you unplug it, the

module can be unloaded. You can see loaded modules with 'lsmod' and load/unload them with 'insmod'/'rmmod'. This modularity is why Linux can run on everything from a Raspberry Pi to a supercomputer with the same kernel codebase — just different modules loaded.

Solaris uses a classic modular approach where the core Solaris kernel connects to modules for: scheduling classes, file systems, loadable system calls, device and bus drivers, STREAMS modules, executable formats, and miscellaneous modules.

Structure 5: Virtual Machines

💡 KEY CONCEPT: Virtual Machine

A virtual machine (VM) takes the layered approach to its logical conclusion: it treats both the hardware and the OS kernel as if they were hardware. A virtual machine monitor (VMM or hypervisor) sits on the real hardware and creates the illusion of multiple separate machines. Each VM has its own virtual CPU, virtual memory, virtual devices, and runs its own OS independently.

Key characteristics of VMs:

- Each VM is completely isolated from all other VMs — a bug or crash in one VM cannot affect another
- The physical hardware resources are shared and multiplexed among all VMs by the hypervisor
- CPU scheduling creates the illusion that each VM has its own processor
- Perfect environment for OS research and development — you test a new OS without risking the real system
- The main challenge: exact duplication of underlying machine behavior is technically complex

📋 CASE STUDY: AWS EC2 — Cloud Computing

Amazon Web Services (AWS) EC2 is built entirely on virtual machine technology. When you launch an EC2 instance, AWS is creating a VM on a physical server in one of their data centers. That physical server (running Linux + Xen or Nitro hypervisor) hosts dozens of customer VMs simultaneously. Each customer's VM is completely isolated — Netflix's VM cannot access Airbnb's data. The hypervisor schedules VMs on physical CPUs, allocates RAM, and provides virtual disks backed by actual SSDs. This is how AWS can offer 'your own server' to millions of customers from a fixed set of physical hardware.

📋 CASE STUDY: VMware Workstation / VirtualBox

On a laptop running Linux (host OS), you can run VMware Workstation or VirtualBox (the hypervisor). Inside it, you can run Windows XP, Windows 10, and another Linux simultaneously as VMs. Each VM thinks it has its own hardware. This is used for: (1) software testing across OS versions, (2) running legacy software, (3) creating isolated development environments (before Docker became popular),

(4) malware analysis in a sandboxed environment. If the malware destroys the VM, you just restore from a snapshot — the host OS is unaffected.

Feature	Non-Virtual Machine vs. Virtual Machine
Hardware access	OS directly controls hardware | Hypervisor provides virtual hardware to each OS
Isolation	Only process-level isolation | Complete OS-level isolation between VMs
Number of OSes	One OS per machine | Multiple OSes on one machine simultaneously
Crash impact	OS crash = machine unusable | VM crash = only that VM affected
Examples	Regular laptop | VMware, VirtualBox, AWS EC2, Azure VMs, Docker (containers)

The Java Virtual Machine (JVM)

The JVM is a special type of virtual machine — not a machine-level VM like VMware, but a language-level VM. Java programs are compiled to bytecode (not native machine code). The JVM is a program that interprets and executes this bytecode on the real machine. This gives Java its famous 'write once, run anywhere' property: the same .class file runs on Windows, Linux, and macOS because the JVM abstracts the underlying OS and CPU.

📋 CASE STUDY: Android Dalvik / ART

Android used to run Java code on the Dalvik Virtual Machine (a JVM variant). Modern Android uses ART (Android Runtime), which compiles bytecode to native machine code ahead of time (AOT compilation) for better performance. The principle is the same: your Java or Kotlin code is compiled to bytecode, and the runtime (acting as a virtual machine layer) executes it on top of the Linux kernel. This is why the same Android APK works on phones with ARM chips from Qualcomm, Samsung, and MediaTek — ART (the VM) handles the differences.

9. Reading Assignments & Study Guide

The following topics from your lecture must be studied independently. Use these notes as a foundation, then deepen your understanding with the Silberschatz textbook (Chapters 1-2).

Assignment	What to Focus On
1. Functional and Service Views	OS services for users: user interface, program execution, I/O, file system, communications, error detection. OS services for system: resource allocation, accounting, protection and security.
2. System Calls and Programs	What are the 6 categories of system calls? Give 2 examples from each category. What do system programs do that system calls don't?
3. Cache Memory	What is cache? Why is it needed (speed gap between CPU and RAM)? What is the cache hierarchy (L1, L2, L3)? What are the advantages and performance implications?
4. OS Services (Two Categories)	Study each service: (a) User-facing: UI, program execution, I/O ops, file manipulation, communications, error detection. (b) System-facing: resource allocation, accounting, protection and security.
5. OS Generations	Serial processing (no OS, operator loads jobs manually), Simple Batch Systems (monitor, automatic job sequencing), Multiprogrammed Batch Systems, Time-Sharing Systems — characteristics, advantages, and limitations of each.

9.1 Self-Test Questions

Use these questions to test your understanding before your exam:

14. Explain the difference between a program and a process. Give a real-world analogy.
15. Why is dual-mode operation necessary? What would happen if it did not exist?
16. Describe the sequence of events when you save a file in a text editor. Name every OS component involved.
17. Compare monolithic kernels and microkernels. Which would you choose for a medical device OS, and why?
18. Explain how AWS EC2 uses virtual machine concepts from this lecture. Which VM structure (hypervisor type) does it most resemble?
19. Why do real-time embedded systems need special operating systems? What would go wrong if an ABS braking system ran on Windows?
20. Trace the path of an interrupt from hardware event to program resumption, naming each component involved.

10. Key Terms Glossary

All critical terms from this chapter, in one place for quick reference:

Operating System	Software that manages hardware, runs user programs, and provides a convenient user interface. Acts as both resource allocator and control program.
Kernel	The one program always running on the computer. The core of the OS that has direct hardware access.
Bootstrap	The first code executed at power-up, stored in firmware (ROM/EEPROM), that loads the OS kernel into memory.
Interrupt	A signal to the CPU that an event needs attention, causing the CPU to save state and execute an Interrupt Service Routine (ISR).
Trap	A software-generated interrupt, caused by a program error (exception) or deliberate service request (system call).
Multiprogramming	Keeping multiple jobs in memory so the CPU always has something to execute, maximizing CPU utilization.
Timesharing	Rapidly switching the CPU among processes to give each user an interactive experience with <1 second response time.
Dual Mode	Hardware mechanism with user mode (mode bit=1, restricted) and kernel mode (mode bit=0, privileged) to protect the OS.
System Call	The interface through which user programs request OS services. Triggers a mode switch from user to kernel mode.
Process	A program in execution. An active entity with memory space, CPU state, and associated resources.

Virtual Machine	An abstraction that gives each OS or program the illusion of its own dedicated hardware. Enables multiple OSes on one physical machine.

Microkernel	An OS design that minimizes kernel size by moving services like file systems and drivers to user space, communicating via message passing.

Real-Time OS	An OS with strict timing guarantees, used in embedded systems where missing a deadline causes system failure.

End of Chapter 1 — Introduction to Operating Systems