/* Most of the topics have been written in class, This note has been compiled from various sources and can be taken as reference only*/

Chapter 1 Introduction

1.1 Introduction to operating system - Written in Class
1.2 OS as and extended machine and resource manager - Written in Class
1.3 History of OS
1.4 Type of OS: Mainframe, Server, Personal, Smartphone and handheld, IOT and embedded, real time, smart card
1.5 OS Components: Kernel, shell, Utilities, applications
1.6 Types of OS Kernel: Monolithic, Micro, Nano, Layered, Hybrid, Exokernel
1.7 System calls, shell commands, shell programming
1.8 POSIX standard ***Assignment I***
1.9 Bootloader, MBR/GPT, UEFI and legacy Boot

1.3 History of Operating Systems

- **1940s – Early Computers (No OS):**
  - Computers ran a single program at a time.
  - Input/output handled manually using switches and punch cards.
  - No operating system existed; programmers managed all machine operations.
- **1950s – Batch Processing Systems:**
  - Introduction of **batch systems** to process jobs in groups without user interaction.
  - Jobs submitted using punch cards and executed sequentially.
  - **IBM 701** and **UNIVAC** systems used this model.
    - ⇨ The UNIVAC (Universal Automatic Computer) I, introduced on March 31, 1951, stands as the first commercial electronic computer, marking a pivotal moment in computing history.
  - Basic I/O System (BIOS) started appearing.
- **1960s – Multiprogramming and Time-Sharing:**
  - **Multiprogramming** allowed multiple programs in memory to improve CPU utilization.
  - **Time-sharing systems** (e.g., CTSS, MULTICS) allowed multiple users to interact with the computer simultaneously.
    - ⇨ The Compatible Time-Sharing System (CTSS) was an influential operating system that was not only an early time-sharing operating system, it also influenced the design and programmers of the UNIX and MULTICS operating systems.
    - ⇨ Multics ("Multiplexed Information and Computing Service) was a mainframe time-sharing operating system that influenced many later systems.
  - Emergence of system calls and user interfaces.
  - Birth of early OSs like **UNIX** (developed at Bell Labs in 1969).

- **1970s – UNIX and Portability:**
  - UNIX introduced concepts like hierarchical file system, multiuser support, and device independence.
  - Rewritten in C language, making UNIX portable across different hardware.
- **1980s – Personal Computer OSs:**
  - Rise of personal computers led to user-friendly operating systems like:
    - **MS-DOS** (Microsoft Disk Operating System)
    - **Mac OS** (Apple's graphical OS)
      - ⇨ The original **Mac OS** was released by **Apple Inc.** on **January 24, 1984**, alongside the launch of the first **Apple Macintosh** computer.
      - ⇨ The name "Mac OS" officially came into use starting with **Mac OS 7.6** in **1997**
  - Introduction of GUI-based systems.

- **1990s – GUI and Networking:**
  - **Windows 95/98** integrated graphical user interface and plug-and-play hardware support.
  - Networking and internet capabilities became standard.
  - **Linux** emerged as a free, open-source UNIX-like OS.
- **2000s – Modern OS Evolution:**
  - Growth of mobile OSs: **Android**, **iOS**.
  - Enhanced security, multitasking, and support for multiple processors.
  - Development of server and enterprise-grade OSs (e.g., Windows Server, Red Hat Enterprise Linux).
- **2010s – Virtualization and Cloud OSs:**
  - Rise of **virtual machines** and hypervisors (e.g., VMware, Hyper-V).
  - Operating systems adapted for cloud computing and containerization (e.g., Docker).
    - ⇨ **What is Docker?**
    - ⇨ **Docker** is a **containerization platform** that allows developers to **package applications and their dependencies** into lightweight, portable units called **containers**.
    - ⇨ It was released in **2013** and has since become the industry standard for container-based development and deployment.
    - ⇨ ---
    - ⇨ **What is Containerization?**
    - ⇨ **Containerization** is a method of **virtualizing an operating system**, allowing multiple containers to run on the same host OS while being **isolated** from each other.
    - ⇨ Unlike virtual machines, containers **share the host OS kernel**, making them more efficient and faster to start.
  - Lightweight OSs for IoT devices.

- **2020s – OS Trends:**
    - Focus on **AI integration**, **security**, **cloud-native environments**, and **edge computing**.
    - Increased popularity of **open-source** and **cross-platform** systems.

Linux Operating System

Introduction

- **Linux** is an **open-source**, **UNIX-like** operating system.
- It was created by **Linus Torvalds** in **1991** while he was a student at the University of Helsinki, Finland.
- The core of Linux is the **Linux kernel**, which manages system resources and hardware.

History and Development

- Inspired by **MINIX**, a small UNIX-based OS used for education.
- First Linux kernel version (0.01) released in **September 1991**.
- Developed under the **GNU General Public License (GPL)** – free to use, modify, and distribute.
- Supported by a global community of developers.

Key Features

- **Multitasking:** Can run multiple processes simultaneously.
- **Multiuser:** Multiple users can access system resources independently.
- **Portability:** Runs on various hardware platforms (PCs, servers, mobile, embedded).
- **Security:** File permissions, user roles, and advanced security modules (like SELinux).
- **Modularity:** Kernel modules can be loaded/unloaded as needed.
- **Networking:** Strong built-in networking capabilities.
- **Shell and Command Line Interface (CLI):** Powerful scripting and automation tools.

Structure of Linux

1. **Kernel** – Core of the OS, manages CPU, memory, and devices.
2. **System Libraries** – Provide functions to programs and interfaces with the kernel.
3. **System Utilities** – Essential tools for system management.
4. **User Interface** – CLI (like Bash) or GUI (like GNOME, KDE).

Popular Linux Distributions (Distros)

- **Ubuntu** – User-friendly, widely used for desktops and servers.
- **Debian** – Stable and free OS, basis for many other distros.
- **Fedora** – Cutting-edge features, community-supported.
- **Red Hat Enterprise Linux (RHEL)** – Commercial, enterprise-grade.
- **CentOS** – Free RHEL alternative (now replaced by CentOS Stream).
- **Arch Linux** – Lightweight, rolling release for advanced users.

Where is Linux Used?

- Web servers and cloud computing (Apache, Nginx, AWS, Google Cloud).
- Supercomputers (over 90% run on Linux).
- Android smartphones (Android is based on the Linux kernel).
- IoT devices, routers, smart TVs, and embedded systems.
- Development and programming environments.

Advantages of Linux

- Free and open-source.
- Highly customizable.
- Stable and secure.
- Large community support.
- Suitable for developers, servers, and enterprise environments.

Types of Operating Systems

1. Batch Operating System

- Executes a **batch of jobs** without manual intervention.
- Jobs with similar needs are grouped and processed together.
- **No direct interaction** between user and computer.
- Example: IBM OS/360

2. Time-Sharing Operating System

- Also known as **Multitasking OS**.
- Allows **multiple users** to use the system simultaneously by giving each user a time slice.
- Improves responsiveness and resource utilization.
- Example: UNIX, Multics

3. Distributed Operating System

- Controls a group of **independent computers** and makes them appear as a **single system**.
- Resources and tasks are distributed across multiple machines.
- Example: LOCUS, Amoeba

4. Network Operating System

- Manages **network resources** like shared files, printers, and user access.
- Provides services to computers connected over a network.
- Example: Novell NetWare, Windows Server

5. Real-Time Operating System (RTOS)

- Responds to inputs or events within a **strict time constraint**.
- Used in **embedded systems**, robotics, medical devices, etc.
- Types:
    - **Hard Real-Time OS**: Strict timing requirements (e.g., pacemakers).
    - **Soft Real-Time OS**: Less strict (e.g., audio/video streaming).

- Example: VxWorks, RTLinux

6. Multiprogramming Operating System

- Multiple programs **reside in memory simultaneously**.
- CPU is switched among programs to improve utilization.
- Enhances CPU efficiency by reducing idle time.
- Example: UNIX

7. Multitasking Operating System

- Allows a **single user** to run **multiple applications** simultaneously.
- Each task gets a small CPU time slice.
- Example: Windows, macOS

8. Multiuser Operating System

- Enables **multiple users** to access a computer system **at the same time**.
- Resources are shared and managed securely among users.
- Example: UNIX, mainframe OSs

9. Single-User Operating System

- Designed for **one user** at a time.
- Found in personal computers and mobile devices.
- Example: MS-DOS, Windows 10 (single-user mode)

10. Mobile Operating System

- Specifically designed for **mobile devices** like smartphones and tablets.
- Optimized for touch input, power management, and mobile hardware.
- Example: Android, iOS

---

1. Multiprogramming Operating System

Definition:

- Allows **multiple programs** to reside in **main memory** at the same time.
- The **CPU executes one program at a time**, but switches to another when the current one waits for I/O.

Goal:

- **Maximize CPU utilization** by reducing idle time.

Working:

- When one program is waiting (e.g., for I/O), the OS switches the CPU to another ready program.

Example:

- Early UNIX systems, IBM OS/360.

Key Point:

- No user interaction is required. Switching is done by the OS automatically when processes are blocked.

---

2. Multitasking Operating System

Definition:

- Allows a **single user** to run **multiple applications simultaneously** (e.g., browser + media player).

Goal:

- Improve **user convenience and responsiveness**.

Working:

- The CPU switches rapidly between tasks (processes), giving the **illusion that all are running at once**.

Example:

- Windows, macOS, Linux (desktop use).

Key Point:

- A special case of multiprogramming focused on **interactive user experience**.

---

3. Time-Sharing Operating System

Definition:

- An **extension of multiprogramming** with a **focus on multi-user access**.
- Each user/program gets a **time slice** of the CPU.

Goal:

- Provide **interactive access** to multiple users at once.

Working:

- The CPU switches between users' processes quickly, ensuring fair CPU time.

Example:

- UNIX (multi-user mode), mainframe systems.

Key Point:

- Allows **multiple users to use the system interactively**, often via terminals.

Comparison Table

Feature	Multiprogramming OS	Multitasking OS	Time-Sharing OS
Main Focus	CPU utilization	User convenience	Multi-user interaction
Users	Single	Single	Multiple
Type of Tasks	Programs	User applications	User sessions/tasks
CPU Allocation	When I/O waits	Rapid switching	Time slices (quantum)
Examples	OS/360, early UNIX	Windows, macOS	UNIX, MULTICS

Types of Operating Systems Based on Platform

---

1. Mainframe Operating Systems

Description:

- Designed to **handle large-scale processing** for enterprises.
- Support **hundreds or thousands of simultaneous users and jobs**.
- Provide high **throughput**, **reliability**, and **I/O handling**.

Key Features:

- Multiuser and multitasking.
- Advanced job scheduling and resource allocation.
- Extensive support for batch and transaction processing.

Examples:

- **IBM z/OS**
- **UNIVAC Exec**
- **MVS (Multiple Virtual Storage)**

---

2. Server Operating Systems

Description:

- Used on **network servers** to manage client requests and backend processing.
- Run continuously and are optimized for **performance, security, and uptime**.

Key Features:

- Support for **network services** (file, print, database, web, etc.).
- **Multiuser** support.
- Robust **security** and **resource management**.

Examples:

- **Windows Server**
- **Red Hat Enterprise Linux (RHEL)**
- **Ubuntu Server**
- **FreeBSD**

---

3. Personal Computer Operating Systems

Description:

- Designed for **single-user desktops or laptops**.
- Optimized for **ease of use**, **application compatibility**, and **multitasking**.

Key Features:

- GUI-based user interface.
- Support for **multitasking**, multimedia, and plug-and-play hardware.
- Less emphasis on heavy networking or multiuser control.

Examples:

- **Microsoft Windows**
- **macOS**
- **Linux (Ubuntu Desktop, Fedora, etc.)**

---

4. Mobile/Smartphone Operating Systems

Description:

- Tailored for **mobile devices** like smartphones and tablets.

- Optimized for **touch interface**, **battery life**, and **mobile hardware**.

Key Features:

- Lightweight and energy-efficient.
- Integrated with mobile hardware (camera, GPS, etc.).
- App-based ecosystem.

Examples:

- **Android** (based on Linux kernel)
- **iOS** (Apple)
- **HarmonyOS** (Huawei)

---

5. Smart Card Operating Systems

Description:

- Designed for **secure and minimal resource environments** like SIM cards, credit cards, and ID cards.
- Focus on **security and data protection**.

Key Features:

- Small memory footprint.
- Often use **real-time or custom microkernel**.
- Highly secure and tamper-resistant.

Examples:

- **Java Card OS**
- **MULTOS**
- **Proprietary OS by card manufacturers**

---

6. Embedded Operating Systems

Description:

- Run on **embedded systems**—devices dedicated to specific functions (e.g., microwave ovens, car control systems).
- Usually **real-time OS** with limited resources.

Key Features:

- Lightweight, fast booting.
- Real-time capabilities.
- Minimal user interface (often no GUI).

Examples:

- **VxWorks**
- **FreeRTOS**
- **Embedded Linux**
- **QNX**

---

7. IoT (Internet of Things) Operating Systems

Description:

- Power **smart devices** connected to the internet (e.g., smart thermostats, wearable tech).
- Require **low power**, **scalability**, **wireless communication**, and **real-time** features.

Key Features:

- Very lightweight.
- Network stack integration (e.g., MQTT, CoAP).
- Security and remote update capability.

Examples:

- **RIOT OS**
- **Contiki**
- **TinyOS**
- **Zephyr OS**
- **Amazon FreeRTOS**

Summary Table

Platform	Example Devices	OS Examples	Key Traits
Mainframe	Enterprise servers	z/OS, MVS	High throughput, batch, multiuser
Server	Web, DB, mail servers	RHEL, Windows Server	Network services, performance
Personal	PC, Laptops	Windows, macOS, Ubuntu	GUI, multitasking
Mobile	Smartphones, Tablets	Android, iOS	Touch UI, app-based, mobile hardware
Smart Card	SIM, bank cards	Java Card OS, MULTOS	Security, small memory
Embedded	Microwaves, Cars	VxWorks, Embedded Linux	Real-time, task-specific
IoT	Smart home, wearables	RIOT OS, Zephyr, Contiki	Low power, real-time, connected

---

Components of Operating System

1. **Kernel**
    - The core part of the OS.
    - Manages CPU, memory, and device communication.
    - Provides low-level services like process scheduling, interrupt handling, and system calls.
    - Operates in **privileged mode** (kernel mode).

←----------

What is a Kernel?

Definition:

The **kernel** is the **core component of an operating system**. It acts as a **bridge between applications and the hardware**.

- Runs in a **privileged mode** (kernel mode).
- Manages system resources and controls hardware.

Key Responsibilities:

Function	Description
Process Management	Handles creation, scheduling, and termination of processes.
Memory Management	Allocates and deallocates memory for processes.
File System	Manages reading/writing files and directories.
Device Management	Controls hardware devices using drivers.
System Calls Handling	Responds to user requests via system calls.
Security and Protection	Controls access to system resources.

Kernel vs Operating System:

Term	Scope
Kernel	Core part of OS, runs at low level
Operating System	Entire software system including kernel, shell, UI, utilities

---

2. **Process Management**
    - Responsible for creating, scheduling, and terminating processes.
    - Handles **process synchronization** and **interprocess communication (IPC)**.
    - Manages CPU allocation using scheduling algorithms.
3. **Memory Management**
    - Manages the system's RAM.
    - Keeps track of each byte in memory—what is free and what is allocated.
    - Handles **paging**, **segmentation**, and **virtual memory**.
    - Allocates memory to processes and protects memory spaces.
4. **File System**
    - Organizes and manages data storage on disks and other storage devices.
    - Controls file creation, deletion, reading, and writing.
    - Maintains file metadata (permissions, timestamps).
    - Implements directories and access control.
5. **Device Management (I/O Management)**
    - Manages input and output devices (keyboards, disks, printers, etc.).
    - Uses **device drivers** as an interface between hardware and the OS.
    - Handles buffering, caching, and spooling.
6. **User Interface (UI)**
    - Allows user interaction with the OS.

- Can be **Command-Line Interface (CLI)** or **Graphical User Interface (GUI)**.
- Translates user commands into system calls.

7. **System Utilities**
   - Programs that perform specialized, system-related tasks.
   - Examples: disk cleanup, antivirus software, system monitoring tools.

What is the Shell?

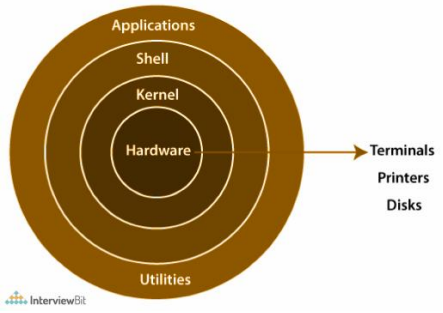
- The **shell** is a **command interpreter** that takes user commands and translates them into actions by making system calls to the kernel.
- It provides a **user interface**, usually a **Command-Line Interface (CLI)**, although some shells also support scripting and automation.
- Examples: **Bash (Linux shell)**, **PowerShell (Windows)**, **sh, csh, zsh**.

How it fits in OS architecture:

Layer	Description
Kernel	Core OS managing hardware and resources
Shell	Interface that interprets user commands
User Applications	Programs run by users

Shell is a user-level program that provides command interpretation and scripting capability, serving as an interface to the OS kernel.

"Bash, sh, csh, zsh, and PowerShell are examples of shells that act as command interpreters, enabling users to interact with the operating system through commands and scripts.



A shell is a program that provides an interface for users to interact with the operating system's kernel. The kernel is the core of the operating system, responsible for managing system resources. The shell interprets user commands and translates them into requests that the kernel can execute. Linux operating systems come with a default shell called Bash and in windows, Command Prompt is default shell program.

Types of Operating System Kernels

---

1. Monolithic Kernel

- All core services (CPU scheduling, memory management, file system, drivers) run in **kernel space**.
- Single large program.

Pros:

- High performance (fast communication between components).

Cons:

- Difficult to maintain/debug; any bug can crash the whole system.

Examples:

- **Linux**
- **Unix**
- **MS-DOS**

---

2. Microkernel

- Only **essential services** (like CPU scheduling, memory management, IPC) run in kernel space.
- Other services (device drivers, file systems) run in **user space**.

Pros:

- Better security and modularity.
- Easier to extend and maintain.

Cons:

- Slightly lower performance due to frequent user-kernel switches.

Examples:

- **MINIX**
- **QNX**
- **L4**

---

3. Nano Kernel

- A very minimal kernel responsible only for **basic hardware control** like interrupt handling and context switching.
- Often used in real-time systems.

Pros:

- Extremely lightweight.

Cons:

- Requires higher-level components to function.

Use Case:

- **Embedded** and **real-time systems**

---

4. Layered Kernel

- Kernel is designed in **layers**, with each layer built on top of a lower one.
- Each layer performs a specific task and communicates only with adjacent layers.

Pros:

- Well-structured and easy to debug.

Cons:

- Reduced performance due to strict layer-to-layer communication.

Example:

- **THE OS** (early layered design)
- **Parts of Windows NT**

---

5. Hybrid Kernel

- Combines aspects of **monolithic** and **microkernel** designs.
- Some services run in kernel space (for performance), others in user space (for modularity).

Pros:

- Balanced performance and flexibility.

Examples:

- **Windows NT kernel**
- **macOS (XNU kernel)**
- **Modern Linux (modular)**

---

6. Exokernel

- A **minimalist kernel** that gives applications direct access to hardware.
- Provides **no abstractions**—applications build their own.

Pros:

- Maximum efficiency and flexibility.

Cons:

- Complex application development.

Example:

- **ExOS** (MIT research)

---

7. Other Related Types

Kernel Type	Description	Example
Modular Kernel	Core kernel + dynamically loadable modules	Modern **Linux**
Real-Time Kernel	Designed to meet **strict timing constraints**	**FreeRTOS**, **VxWorks**
Secure Kernel	Focuses on **isolation and secure execution**	Used in **military OS**

Summary Table

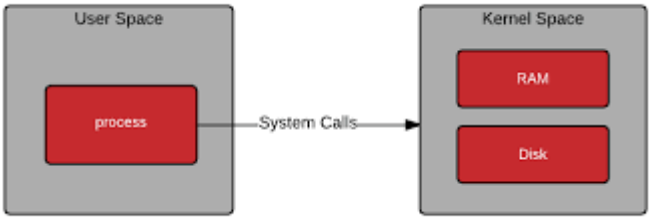
Kernel Type	Key Feature	Example OS
Monolithic	All services in one kernel space	Linux, Unix

Kernel Type	Key Feature	Example OS
Microkernel	Minimal core, services in user space	MINIX, QNX
Nano Kernel	Only handles lowest-level hardware control	Embedded RTOS
Layered	Organized in logical layers	THE OS
Hybrid	Mix of monolithic and microkernel	Windows, macOS
Exokernel	Minimal kernel, apps manage resources	ExOS (research)
Modular	Kernel with loadable modules	Modern Linux

System Call – Brief Explanation

A **system call** is a **programmatic way for user programs to interact with the operating system**.

- It acts as a **bridge between user space and kernel space**.



- User applications **cannot access hardware or kernel functions directly**, so they use system calls to request services from the OS.

🔹 **Why Are System Calls Needed?**

- To perform **privileged operations** like:
  - File manipulation
  - Process control
  - Memory allocation
  - Device access

Example Analogy:

Think of the **Operating System as a manager**, and **system calls as formal requests** made by employees (programs). The manager (OS) approves and performs the requested actions.

Common Types of System Calls:

Type	Examples
Process Control	fork(), exec(), exit()
File Management	open(), read(), write(), close()
Device Management	ioctl(), read(), write()
Memory Management	mmap(), brk()
Information Maintenance	getpid(), alarm(), time()
Communication	pipe(), shmget(), send(), recv()

1. System Call

Definition:

A **system call** is a way for a program to request a service from the operating system's kernel, such as file operations, process control, or communication.

Why Use System Calls?

User-level programs **cannot access hardware directly**, so they use system calls to:

- Create or terminate processes
- Read/write files
- Allocate memory
- Communicate between processes

Common System Call Categories:

Category	Example Functions
Process Control	`fork()`, `exec()`, `exit()`, `wait()`
File Management	`open()`, `read()`, `write()`, `close()`
Device Management	`ioctl()`, `read()`, `write()`
Information Maintenance	`getpid()`, `alarm()`, `sleep()`
Communication	`pipe()`, `shmget()`, `msgsnd()`

Example:

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

int main() {
    pid_t pid = fork(); // System call to create a new process
    if (pid == 0)
        printf("Child process\n");
    else
        printf("Parent process\n");
    return 0;
}
```

---

2. Shell Commands

Definition:

A **shell command** is a command entered in a terminal or shell to perform an operation, such as listing files, copying files, or viewing content.

What is a Shell?

The **shell** is a command-line interface that allows users to interact with the OS.

Examples of Common Shell Commands:

Command	Description
`ls`	Lists files in a directory
`cd`	Changes the current directory
`pwd`	Prints the current directory
`cp`	Copies files or directories
`mv`	Moves or renames files
`rm`	Removes files or directories
`cat`	Displays content of a file
`echo`	Prints text to terminal
`man`	Displays manual pages for commands

---

3. Shell Programming

☑ *Definition:*

Shell programming (or shell scripting) is writing a series of shell commands in a file to automate tasks.

File Type:

Shell scripts are usually saved with `.sh` extension.

How to Create and Run a Shell Script:

1. Create a file: `nano script.sh`
2. Add script code (see example below)
3. Make it executable: `chmod +x script.sh`
4. Run it: `./script.sh`

Example Shell Script:

```
bash
#!/bin/bash
# My first shell script

echo "Hello, $USER"
```

```
echo "Today is: $(date)"
echo "Your current directory is: $(pwd)"
```

Common Uses of Shell Scripts:

- Automating backups
- Installing software
- Monitoring system logs
- Repeating command-line tasks

Summary

Concept	Key Idea
System Call	Interface between user program and kernel
Shell Command	A single instruction to the OS via terminal
Shell Script	A file containing a sequence of shell commands