### **1.1 Introduction to Operating System (Expanded Version)**

#### **Interaction of OS and Hardware**

The interaction between the **operating system** (OS) and hardware is fundamental to computer operations. An OS is the software layer that allows users and applications to interact with hardware resources like **CPU**, **memory**, and **I/O devices** (input/output).

The OS manages all aspects of the computer hardware. For instance, when you type on the keyboard, the OS captures this input and forwards it to the appropriate application. It schedules CPU time, allocates memory, and controls access to data storage devices. By acting as an intermediary, it simplifies the complexity of interacting directly with the hardware.

Hardware sends **interrupts** to the OS when it needs to be serviced. For example, when a device like a printer is ready to receive data, it sends an interrupt, and the OS responds by executing the required instructions. This ensures that hardware can operate concurrently with other tasks.

#### **Goals of OS**

The primary goals of an operating system are to:

1. **Convenience**: The OS provides a user-friendly interface, making the system easier to use. This goal focuses on shielding users from the complexities of the hardware. With a GUI (Graphical User Interface) or CLI (Command-Line Interface), users can execute tasks easily.
2. **Efficiency**: An OS ensures that hardware resources are used efficiently, maximizing performance and throughput. This goal involves resource allocation strategies that minimize idle time for components like the CPU.
3. **Ability to Evolve**: Modern OSes must adapt to new hardware and software technologies. An evolving OS can support new hardware devices, security updates, and optimizations without requiring major overhauls.

#### **Basic Functions of OS**

An OS performs several basic functions to manage the system efficiently:

1. **Process Management**: A process is a program in execution. The OS handles process creation, scheduling, and termination. It allocates resources like memory and CPU time to each process, ensuring smooth execution.
2. **Memory Management**: The OS manages the system's memory by allocating memory space to processes and deallocating it once the processes are done. It keeps track of which parts of memory are being used by which processes and optimizes memory usage.
3. **File Management**: Files and directories are stored in secondary storage (like hard drives). The OS manages file creation, deletion, and organization, making it easy for users and programs to store and retrieve information.
4. **I/O Management**: The OS manages input and output devices, controlling data transfer between the CPU, memory, and peripherals like keyboards and printers.
5. **Security**: Protecting the system's resources from unauthorized access is critical. The OS enforces security measures by controlling user access to resources and data.

#### **OS Services**

Operating systems provide various services to users and applications:

1. **Program Execution**: The OS loads programs into memory and allows them to execute by managing resources like CPU and memory.
2. **I/O Operations**: The OS controls input and output devices, handling operations like reading from a file or writing to a printer.
3. **File-System Manipulation**: The OS provides functions for file creation, deletion, reading, writing, and manipulation of directories.
4. **Communication**: It manages communication between processes (Inter-Process Communication - IPC), enabling processes to exchange data.
5. **Error Detection**: The OS continuously monitors system operations for errors, allowing it to respond to potential issues.

#### **System Calls**

**System calls** are used by applications to request services from the OS. They act as a bridge between user-level programs and the OS kernel.

**Types of system calls include:**

1. **Process Control**: Handles process creation and management (e.g., fork(), exec(), exit()).
2. **File Management**: Enables programs to create, open, read, and write files (e.g., open(), read(), write()).
3. **Device Management**: Manages interactions with hardware devices (e.g., ioctl()).
4. **Information Maintenance**: Provides functions to retrieve and set process and system information (e.g., getpid(), alarm(), sleep()).
5. **Communication**: Facilitates data exchange between processes (e.g., pipe(), send(), recv()).

### **1.2 Types of Operating Systems (Expanded Version)**

Operating systems come in various types depending on how they manage processes, resources, and user interactions. Here are the different types of OS:

#### **Batch OS**

Batch operating systems were among the earliest types of OS. In this system, users submit jobs to be processed, and the OS collects these jobs into batches, executing them one by one. There is **no direct interaction** between the user and the system during job execution.

Batch OS is designed to maximize CPU utilization by executing jobs sequentially. Jobs are processed in the order of submission, and the system executes each job fully before moving on to the next. This method was efficient for tasks like payroll processing, where jobs could run without user intervention.

However, batch systems suffered from **slow response time**, as users had to wait until all jobs in the batch were processed. Modern systems do not use pure batch processing, but the concept of batch processing is still used in some specialized fields.

#### **Multiprogramming OS**

Multiprogramming operating systems allow **multiple programs** to reside in memory at the same time. By keeping several jobs in memory, the OS can **switch between them** when one job is waiting (e.g., for I/O). This ensures that the CPU is always working on some task.

In a multiprogramming system, the OS maximizes CPU utilization by executing processes in an overlapping manner. For example, while one process is waiting for an I/O operation to complete, the CPU can execute another process. This leads to **better resource utilization** and **faster processing** compared to batch systems.

#### **Time-sharing OS**

Time-sharing systems are a more advanced form of multiprogramming. In a time-sharing system, the CPU allocates short time slices to each task, allowing **multiple users to interact** with the system simultaneously. The OS switches between tasks so quickly that it creates the illusion of **parallel execution**.

In these systems, the CPU switches tasks frequently, ensuring that all users get a response within seconds. This makes time-sharing ideal for environments where users need **real-time interaction** with the system. Examples of time-sharing systems include **MULTICS** and modern **Unix** systems.

#### **Parallel OS**

Parallel operating systems are designed to handle **multiple processors** working together. By dividing tasks among processors, parallel systems can execute programs faster and more efficiently. Each processor can handle a portion of a task, leading to **increased throughput**.

Parallel systems are often used in **scientific computing** and environments where tasks are too large for a single processor to handle. By breaking down tasks into smaller pieces, these systems provide **fault tolerance** and **scalability**.

#### **Distributed OS**

In a distributed operating system, the **processing power is distributed** across multiple computers (nodes) connected by a network. Each node operates independently, but they all work together to provide a cohesive system. This allows users to access resources and services as if they were all located on a single machine.

Distributed OS provides **resource sharing**, load balancing, and improved **fault tolerance**. Systems like **Amoeba OS** and **Plan 9** are examples of distributed operating systems. These systems are commonly used in **cloud computing** and **networked environments**.

#### **Real-Time OS**

Real-time operating systems are designed for systems that require tasks to be completed within **strict time constraints**. These systems are used in environments like embedded systems, where tasks must be completed in a **predictable timeframe** to ensure proper operation.

Real-time OS can be further classified into:

1. **Hard Real-Time Systems**: Where missing a deadline can result in catastrophic failure (e.g., airbag systems).
2. **Soft Real-Time Systems**: Where missing a deadline results in degraded performance but not system failure (e.g., video streaming).

Examples of real-time systems include **VxWorks** and **QNX**.

### **1.3 Structures of Operating Systems (Expanded Version)**

Operating systems can be structured in various ways depending on how components are organized. The structure of an OS affects its **modularity**, **performance**, and **maintainability**.

#### **Monolithic OS**

In a monolithic system, the entire OS runs as a **single large program** in kernel mode. All OS services (e.g., process management, file management, device drivers) are tightly integrated into the kernel. The advantage of this structure is its simplicity and **high performance**, as all components can directly communicate.

However, monolithic OSes are difficult to **debug** and **maintain** because any change in one part of the system can affect other parts. Modern examples of monolithic OS include **Linux** and **Unix**.

#### **Layered OS**

Layered systems divide the OS into multiple **layers**, with each layer performing specific functions. The outermost layer interacts with the user, while the innermost layer interacts with the hardware. Each layer only communicates with its neighboring layers, which simplifies debugging and system development.

The **bottom layer** typically handles **hardware interactions**, while the upper layers handle **process management**, **memory management**, and **file systems**. Layered systems improve modularity and maintainability. An example of a layered system is the **THE Operating System**.

#### **Virtual Machines**

Virtual machines (VMs) provide an abstraction of the physical machine, allowing multiple OS instances to run on a single physical system. Each VM operates as if it were a separate physical machine, with its own operating system and applications.

The OS manages resources like CPU and memory, ensuring that each VM has enough resources to function. VMs are commonly used in **server consolidation**, where multiple servers are run on a single machine, and in **cloud computing** environments. Examples include **VMware**, **VirtualBox**, and **KVM**.

#### **Microkernels**

Microkernel systems minimize the size of the kernel by running only essential OS services (e.g., process communication, memory management) in kernel mode. Other services, like device drivers and file systems, run in **user space**, outside of the kernel.

This separation improves the OS's **modularity** and **security**, as faults in user-space services do not affect the kernel. Microkernels are commonly used in systems that prioritize reliability, like **Minix** and **Mach**.

### **1.4 Modern UNIX Systems (Expanded Version)**

**UNIX** was first developed in the late 1960s and remains one of the most influential operating systems. It introduced several core concepts that are still in use today, such as **multi-user** and **multi-tasking** capabilities. Modern UNIX systems include **Linux**, **Solaris**, **AIX**, and **BSD**, among others.

#### **General Characteristics of UNIX**

1. **Multi-user & Multi-tasking**: UNIX supports multiple users who can run multiple tasks simultaneously. This capability makes it ideal for servers, development environments, and multi-user workstations.
2. **Stability**: UNIX systems are known for their robustness and stability, making them a popular choice for **mission-critical** applications.
3. **Large Number of Applications**: UNIX supports a wide range of applications, from web servers to development tools. Both commercial and open-source applications are available.
4. **Less Resource Intensive**: UNIX systems are often more efficient in their use of system resources, allowing them to run effectively on older hardware.
5. **Security**: UNIX was designed with security in mind. It uses a permission-based model to control access to files and processes, which has made it a popular choice for **secure environments** like **servers** and **network appliances**.

#### **Components of UNIX OS**

1. **Kernel**: The kernel is the heart of the UNIX operating system. It manages hardware resources, processes, memory, and files. The kernel also handles **system calls** from user programs, managing **process scheduling** and **I/O operations**.
2. **Shell**: The shell is the command interpreter for UNIX. It provides the user interface, accepting commands from the user and passing them to the kernel for execution. Popular UNIX shells include **Bash**, **Zsh**, and **Csh**.
3. **Utilities**: UNIX includes a range of built-in utilities that simplify system administration. These utilities handle tasks like file management, process control, and network configuration.

#### **Modern UNIX Systems**

Modern UNIX systems have evolved from the original UNIX design, incorporating new features and enhancements while maintaining core functionality.

1. **Linux**: An open-source UNIX-like operating system. It is one of the most popular OSes in use today, known for its flexibility, community support, and wide range of distributions (e.g., **Ubuntu**, **Fedora**, **Red Hat**).
2. **BSD**: A family of UNIX-like operating systems known for their performance, scalability, and advanced networking features. **FreeBSD**, **OpenBSD**, and **NetBSD** are common BSD variants.
3. **Solaris**: Originally developed by Sun Microsystems, Solaris is known for its scalability and support for enterprise applications.