Operating System

An operating system (OS) is the fundamental software that makes your computer tick. It acts as a middleman between you and the computer's hardware, managing all the resources and programs running behind the scenes.

Characteristics / Functions of Operating Systems

1. Resource Management:

- Handles allocation and deallocation of resources like CPU, memory, storage, and I/O devices.
- Ensures efficient utilization by multiple programs running concurrently.
    - Example: Allocating more CPU power to a video editing software compared to a web browser.

2. Process Management:

- Controls the execution of programs (processes).
- Handles multitasking (running multiple programs at once) and scheduling (determining which process gets CPU time).
    - Example: Switching between a web browser and a word processor while they are both open.

3. Memory Management:

- Keeps track of memory usage by different processes.
- Allocates memory space to running programs and manages virtual memory (using storage to extend RAM).
    - Example: Warning message when you have too many programs open and memory is running low.

4. File Management:

- Organizes files and folders on storage devices.
- Provides mechanisms for creating, deleting, accessing, and modifying files.
    - Example: Saving a document with a specific filename in a designated folder.

5. Device Management:

- Acts as an intermediary between software and hardware devices.
- Handles communication with devices like printers, scanners, and disk drives.
    - Example: Selecting a printer from a list of available devices when printing a document.

6. Security:

- Controls access to system resources and protects against unauthorized access.
- Provides mechanisms for user authentication and authorization.
    - Example: Entering a password to log in to your computer.

7. User Interface (UI):

- Provides a way for users to interact with the operating system.
- Can be a command-line interface (CLI) or a graphical user interface (GUI).
    - Example: Clicking icons or using menus to navigate through the operating system.

Evolution of Operating Systems

- **Early Days (1940s & before):** No OS, complex codes directly controlled hardware.
- **Batch Processing (1940s-1950s):** Basic job scheduling, programs executed sequentially.
- **Multiprogramming (1950s-1960s):** Multiple programs in memory, CPU switches for efficiency.
- **Time-Sharing (1960s-1970s):** Multiple users share a computer, illusion of simultaneous operation.
- **GUI Revolution (1970s-1980s):** User-friendly graphical interfaces with icons and menus.
- **Networking Era (1980s-1990s):** Communication and resource sharing between networked computers.
- **Modern OS (1990s-Present):** Continuous improvement in features, security, and user experience for various devices.

Types of Operating Systems

Operating systems come in various flavors, each suited for different purposes.

1. Batch Operating System:

- Jobs (programs) are submitted in batches (groups) and processed one after another.

- No direct user interaction.
- Example: Early mainframe systems.

2. Multiprogramming Operating System:

- Allows multiple programs to be loaded into memory at once.
- CPU rapidly switches between programs, creating the illusion of multitasking (but only one program runs at a time).
    - Example: IBM OS/360.

3. Multiprocessing Operating System:

- Takes advantage of computers with multiple processors (CPUs).
- Distributes tasks among multiple CPUs for faster processing.
    - Example: Modern server operating systems like Windows Server.

4. Multitasking Operating System:

- Enables true multitasking, allowing users to run multiple programs concurrently.
- Programs share system resources like CPU and memory.
    - Examples: Windows, macOS, Linux.

5. Time-Sharing Operating System:

- Similar to multitasking, but with a focus on multiple users on a single computer system.
- Rapidly allocates CPU time to each user, giving the illusion of individual use.
    - Example: Early operating systems for multi-user mainframes.

6. Real-Time Operating System (RTOS):

- Designed for systems with strict timing requirements.
- Guarantees predictable response times for critical tasks.
    - Examples: Operating systems in industrial control systems, medical equipment.

7. Distributed Operating System:

- Spans multiple computers across a network, appearing as a single system to users.
- Distributes tasks and data across the network for improved performance and scalability.
    - Example: Network File System (NFS).

8. Network Operating System (NOS):

- Manages resources on a network, like file sharing, security, and user permissions.
- Provides centralized administration for network devices.
    - Examples: Windows Server, Linux with Samba.

Concept of System Calls

System calls are the messengers between your programs and the operating system. Here's a quick breakdown:

- **Function:** System calls are the programmatic way a program requests services from the operating system's kernel, the core of the OS.
- **Why Use Them?** Programs can't directly access hardware or perform privileged operations (like managing memory) on their own. System calls provide a safe and controlled way to interact with these resources.

How it Works:

- Program makes a system call using a specific function provided by the OS.
- This triggers a switch from user mode (program execution) to kernel mode (privileged OS execution).
- The kernel handles the request, accessing hardware or performing the operation.
- The kernel returns control and results back to the program.

Types of System Calls:

- **Process Control:** Manage the creation, termination, and scheduling of processes.
- **File Management:** Create, read, write, and delete files and directories.
- **Device Management:** Interact with hardware devices like printers, disks, and networks.
- **Information Maintenance:** Get information about the system, like memory usage or process status.
- **Communication:** Allow processes to exchange data and synchronize their actions.

Benefits:

- **Security:** System calls restrict programs from directly accessing hardware, preventing accidental or malicious damage.
- **Protection:** Kernel mode has higher privileges, ensuring programs don't interfere with core system operations.

- **Abstraction:** System calls provide a standardized interface for accessing OS services, making programming easier and more portable across different systems.

Process Management

Process management is a fundamental function of an operating system (OS) that deals with the creation, scheduling, and termination of processes. A process is essentially a program that's actively being executed.

Process Concept

- **Process:** An active program in execution. A process is an instance of a program that's currently being executed by an operating system (OS). (Think of a recipe being cooked, not just the recipe itself)
- **Key Difference from Program:** A program is a passive entity, while a process is actively executing instructions. A process is an instance of a program.
- **Process Creation:** When you run a program, the operating system creates a process for it.
- **Process Management:** The operating system is responsible for creating, scheduling, and terminating processes.

Structure of a Process:

- **Program Code:** Instructions to be executed.
- **Data Section:** Stores variable values.
- **Stack:** Manages function calls and local variables.
- **Heap:** Dynamically allocated memory during runtime.
- **Program Counter:** Keeps track of the next instruction to execute.
- **Registers:** Store temporary data for the CPU.

Process State: A process can be in various states like:

- **New:** Just created, waiting for resources (e.g., opening a new program).
- **Ready:** Waiting for the CPU to be assigned (e.g., program waiting in line for execution).
- **Running:** Actively executing instructions (e.g., program currently using the CPU).
- **Waiting/Blocked:** Needs a resource (like I/O) before continuing (e.g., waiting for user input, waiting for data from disk).
- **Terminated:** Finished execution (e.g., program has completed its task).

Example: When you open multiple web browsers, each browser is a separate process running the same program.

Process Control Block (PCB)

- A data structure used by the Operating System (OS) to manage processes.
- Created for each process when it's initiated.
- Holds all relevant information about a process for the OS to track its state and manage resources.

Information stored in a PCB (Examples):

- **Process State:** Running, waiting, ready, terminated (e.g., "Running" for a web browser actively displaying a webpage).
- **Process ID (PID):** Unique identifier for the process (e.g., Task Manager on Windows shows PIDs for running processes).
- **Program Counter (PC):** Memory address of the next instruction to be executed (like keeping your place in a book).
- **CPU Registers:** Values stored in CPU registers specific to the process (e.g., temporary data used during calculations).
- **Memory Management Information:** Memory allocated to the process (e.g., how much RAM a game is using).
- **I/O Status Information:** Open files and devices associated with the process (e.g., a file being edited in a word processor).

Importance of PCB:

- Enables process scheduling (determining which process gets CPU time).
- Facilitates context switching (saving and restoring process state during switching).
- Ensures efficient resource allocation and management.

Context Switching

- **What it is:** In computing, context switching is the process of saving the state (current information) of a running program or task (process) and then switching to a different process. This allows a single CPU to efficiently handle multiple programs at the same time.

Why it's important:

- **Multitasking:** Enables a single CPU to appear to run multiple programs concurrently.
- **Process management:** Allows the operating system to manage multiple processes and prioritize tasks.
- **Responsive systems:** Keeps the system feeling responsive even when multiple programs are running.

The Process:

- **Save State:** The CPU registers, memory addresses, and other critical data of the current process are saved.
- **Switch Context:** The saved state of a new process is loaded into the CPU.
- **Resume Execution:** The new process picks up where it left off and starts running.

CPU Scheduling

- **Concept:** CPU scheduling is the process by which the operating system decides the order in which processes will be allocated to the CPU for execution. This ensures efficient utilization of CPU resources in a multiprogramming environment.

- **What it is:** The act of managing how processes are allocated time on the CPU in a multitasking operating system.

- **Why it matters:** Keeps the CPU busy and improves overall system responsiveness by ensuring processes don't wait idly.

Types of Scheduling:

- **Preemptive:** OS can interrupt a running process to give the CPU to a higher priority process. (Ex: Handling a critical system task)
- **Non-preemptive:** A process keeps the CPU until it finishes its burst (CPU usage cycle) or voluntarily yields it. (Ex: Simple animation playing uninterrupted)

Types of Process Schedulers:

1. Long-Term Scheduler (Job Scheduler):

- Selects processes from secondary storage (like a hard disk) and admits them to the main memory's ready queue.
- Controls the degree of multiprogramming (number of processes in ready state).

2. Short-Term Scheduler (CPU Scheduler):

- Selects a process from the ready queue and allocates it to the CPU for execution.
- Employs scheduling algorithms (like FCFS, priority) to make decisions.

3. Medium-Term Scheduler (Swapper):

- Swaps processes between main memory and secondary storage to manage memory usage.
- Less frequent than short-term scheduling.

Scheduling Criteria

When an operating system manages multiple processes, it needs a way to decide which process gets to use the CPU at any given time. This decision-making process is called scheduling, and various criteria are used to evaluate and choose the most suitable process. Here's a breakdown of some key scheduling criteria:

1. CPU Utilization:

- Measures how busy the CPU is.
- Ideally, we want to keep the CPU busy to maximize productivity.
    - **Example:** The First Come First Served (FCFS) algorithm generally leads to high CPU utilization but can lead to long waiting times for shorter processes.

2. Throughput:

- Represents the number of processes completed per unit time (e.g., processes completed per second).
- A higher throughput indicates the system is processing tasks efficiently.
    - **Example:** The Shortest Job First (SJF) algorithm often improves throughput as it prioritizes shorter processes, leading to faster completion times.

3. Turnaround Time:

- The total time taken for a process to finish execution, from submission to completion.
- Lower turnaround time signifies faster process execution.
    - **Example:** Priority scheduling can prioritize critical processes, reducing their turnaround time but potentially increasing it for lower-priority ones.

4. Waiting Time:

- The time a process spends waiting in the ready queue for its turn on the CPU.
- Ideally, we want to minimize waiting times to ensure processes get processed promptly.
    - **Example:** Round Robin (RR) scheduling allocates CPU time in short slices (quantum), ensuring processes don't wait indefinitely but may lead to some overhead due to context switching.

5. Response Time:

- The time it takes for a process to start responding after submitting a request.
- A shorter response time indicates faster initial processing, especially for interactive tasks.
    - **Example:** Priority scheduling with high priority for interactive processes can improve response time but may affect turnaround time for other processes.

Types of Scheduling Algorithms

Scheduling algorithms determine the order in which processes are executed by the CPU. Here are some common types:

1. First Come First Served (FCFS):

- Processes are served in the order they arrive.
- Simple to implement, but can lead to starvation for short processes behind long ones.
    - Example: Waiting in line at a coffee shop.

2. Shortest Job First (SJF):

- Prioritizes processes with the shortest execution time.
- Minimizes average waiting time, but requires knowing process execution times upfront (often not realistic).
    - Example: Scheduling short emails before long reports.

3. Longest Job First (LJF):

- Prioritizes processes with the longest execution time.
- Can improve throughput (total work completed) but can lead to starvation for short processes.
    - Example: Scheduling a long software update to install overnight.

4. Priority Scheduling:

- Assigns priorities to processes. Higher priority processes are served first.

- Preemptive vs Non-preemptive: Preemptive allows higher priority processes to interrupt running ones (e.g. emergency surgery).
    - Example: Important system tasks getting priority over background tasks.

5. Round Robin (RR):

- Processes are allocated a short time slice (quantum).
- After the slice, the process is preempted and placed at the back of the queue.
- Ensures fairness for short processes and provides good response time.
    - Example: Sharing a game console among friends, each gets a turn for a set time.

Critical Section Problem

- A section of code in a multi-threaded program where shared resources are accessed. Concurrent execution in this section can lead to inconsistencies and errors (like race conditions).
- A critical section is a section of code that accesses a shared resource and must be executed by one process at a time to avoid race conditions and ensure consistency.

Key Properties for Solutions:

- **Mutual Exclusion:** Only one thread can be in the critical section at a time.
- **Progress:** If no thread is in the critical section, a waiting thread should eventually enter.
- **Bounded Waiting:** There should be a limit on how long a thread waits to enter.

Inter-Process Communication (IPC)

- **What it is:** Mechanisms that allow processes to communicate and exchange data with each other.
- **Why it's important:** Enables processes to cooperate and share resources, leading to more efficient and complex programs.

Types of IPC:

- **Shared Memory:** Processes share a designated memory region to exchange data directly. (Imagine two roommates using a grocery list on the fridge)
- **Message Passing:** Processes send and receive messages through queues or mailboxes. (Think of sending notes back and forth)
- **Pipes:** Processes communicate through unidirectional data streams. (Like using a talking tube)

- **Semaphores:** Used to coordinate access to shared resources and ensure that only one process can access a shared resource at a time

Semaphores

- **What they are:** Synchronization tools used to control access to shared resources between processes.

- **Not for direct communication:** Semaphores don't transfer data, they coordinate access to prevent conflicts. (Think of taking turns on a swing set)

- **Operations:**
    - **P(wait):** Decrements the semaphore value. If the value is zero, the process blocks until another process signals. (Like waiting your turn)
    - **V(signal):** Increments the semaphore value. If any processes are blocked waiting, one is woken up. (Like signaling someone it's their turn)

- **Types:**
    - **Binary Semaphores (Mutexes):** Only two values (0 or 1) - ensures only one process accesses the resource at a time. (Like a single swing)
    - **Counting Semaphores:** Value can range beyond 1 - controls access to resources with multiple instances (e.g., a pool of printer connections). (Think of multiple swings at a park)