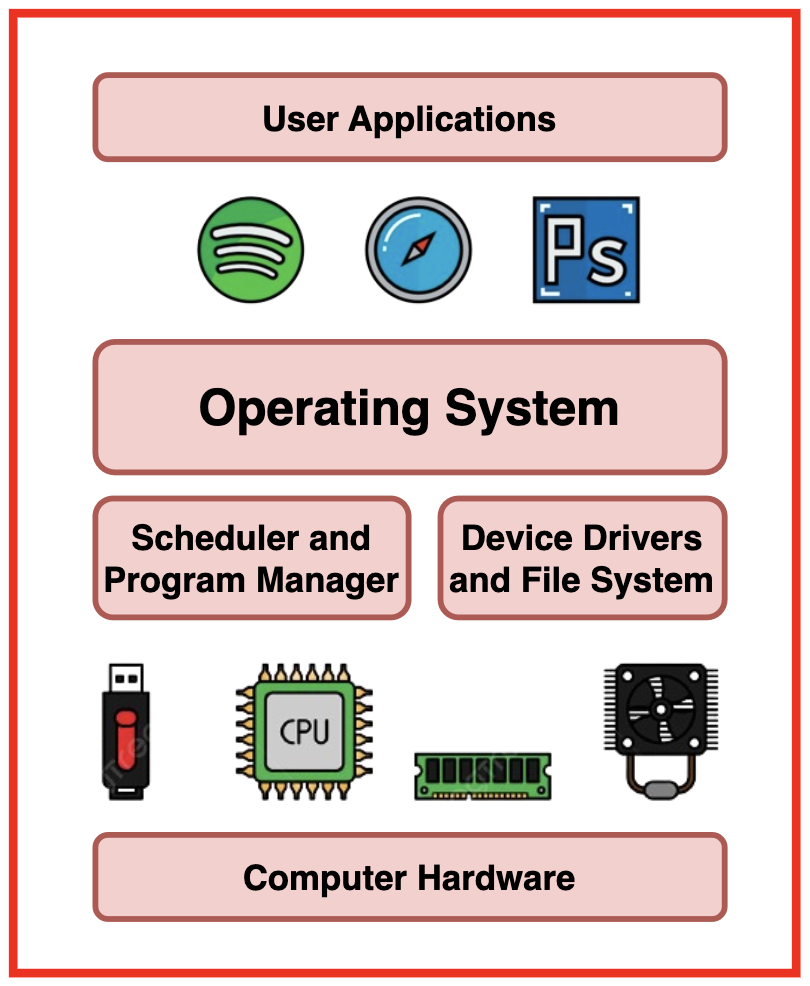
INTRODUCTION TO OS

1. **OS and Main Functions**

A computer is made up of two parts: software programs and the hardware they run on, like electronic chips and the motherboard. The operating system connects these two parts, providing a platform for software to interact with hardware.

The operating system acts as a manager, coordinating and controlling the computer's resources. Whenever you send a command, press a key, or move the mouse, the operating system ensures the right changes happen on the hardware side.



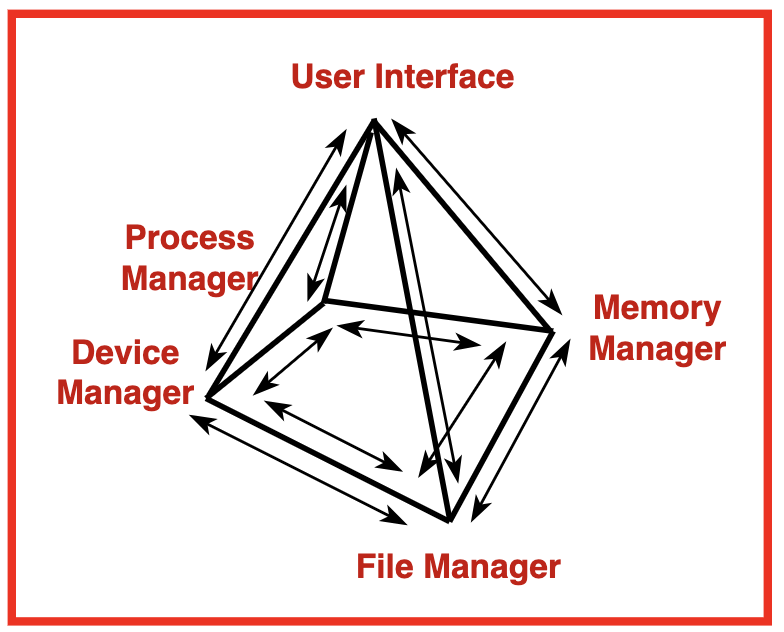


**Purpose and Functions of an Operating System**

* **Process Manager:** Manages the programs running on the computer, ensuring they get enough CPU time and can communicate properly with input/output devices.
* **Memory Manager:** Allocates and deallocates memory for each process, allowing multiple processes to run simultaneously without interruption.
* **Device Manager:** Controls communication between hardware devices (like keyboards and printers) and the software commands you run.
* **File Manager:** Organizes and provides access to files, folders, and directories on your computer and external storage devices.
* **User Interface:** Provides a way for users to interact with the computer, either through a graphical interface or a command line.

**What Happens When You Execute a Command?**

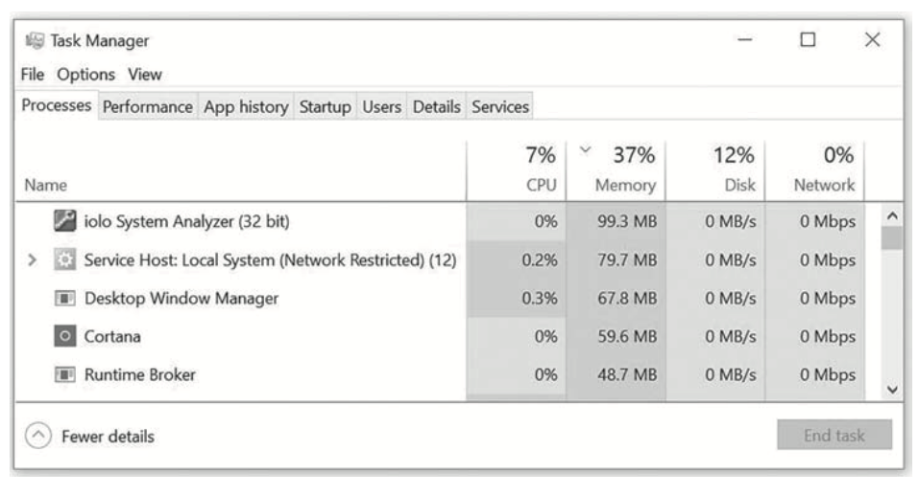
When you execute a command (like clicking the mouse or pressing a key), several processes occur:



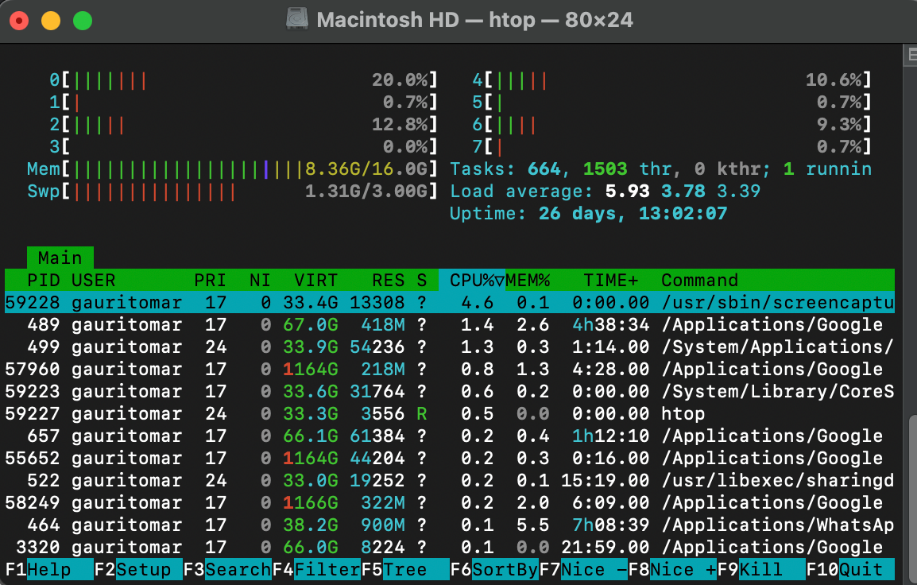
* The **Device Manager** detects the electrical signal from your action.
* The **Process Manager** acknowledges that the command has been received.
* The **Process Manager** determines what resources the command needs (like memory or external devices).
* If necessary, it calculates the file location on disk and communicates with the **Device Manager** to retrieve it.
* The required files are loaded into the **RAM** so the **CPU** can execute the program.
* After execution, the **Process Manager** sends outputs to the appropriate managers to display results or produce sounds.
* If needed, outputs are stored in secondary memory via the **File Manager**.

**Examples of Popular Operating Systems**

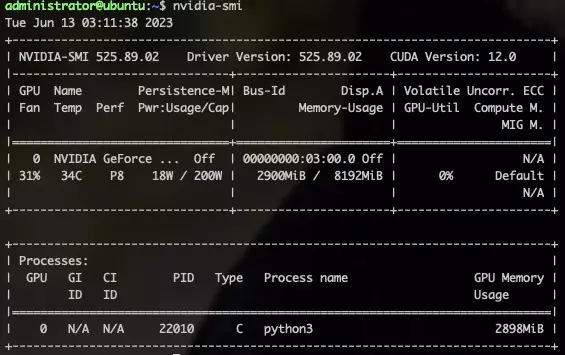
* **Windows:** Developed by Microsoft, it is the most widely used operating system for personal computing, evolving from Windows 1.0 to Windows 10.



* **macOS:** A proprietary operating system designed for Apple computers, known for its user-friendly interface and integration with Apple hardware.



* **Linux:** An open-source operating system kernel, developed by Linus Torvalds, used in various devices from servers to embedded systems. It has several distributions like Ubuntu and Fedora.



1. **Process,Programs and Threads**

Program:

* Set of Instructions written to do something
* Eg: A word document app is a program

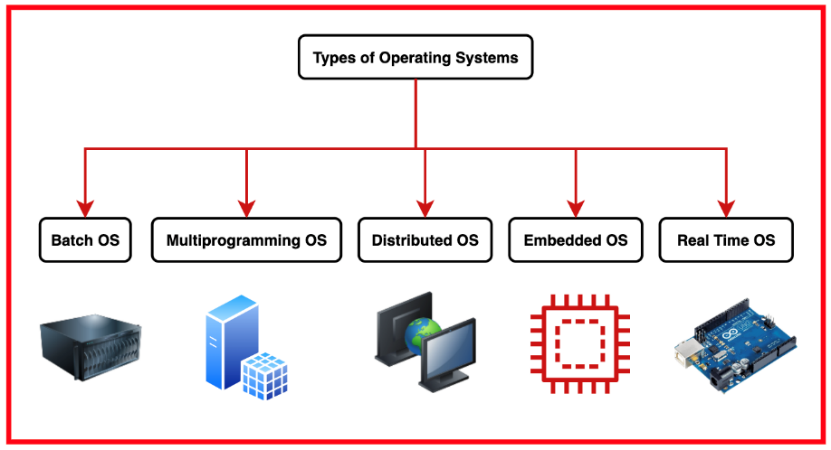
Process:

* What happens when program starts running
* Eg:When you open the word app it becomes a process and runs

Thread:

* Helper working inside the process.
* They divide the work of a process into smaller ones
* Eg: In a browser one thread loads a webpage another handles user clicks another runs videos

1. **Type of OS**



**1. Batch Operating Systems**

Batch Operating Systems execute processes as complete batches, where jobs are entered one at a time. Once a job begins processing, no other job can start until the current job finishes executing. These jobs are executed sequentially without user interaction.

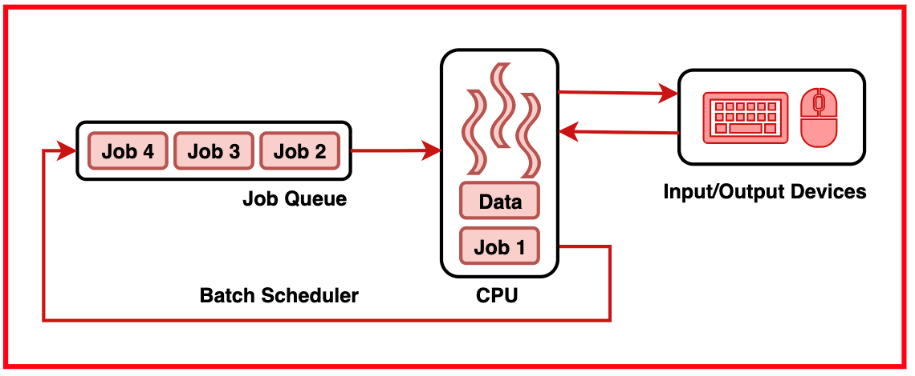
They are designed to process large volumes of data efficiently and automate repetitive tasks such as payroll processing, report generation, and data processing.

The efficiency of a batch system is measured in throughput, which is the number of jobs completed in a given amount of time.

**2. Multiprogramming Operating Systems**

Multiprogramming Operating Systems allow multiple programs to run simultaneously by sharing the CPU and other system resources. These systems quickly switch between programs, giving the appearance that all programs are executing at the same time.

This increases CPU utilization and throughput, allowing the system to handle tasks more efficiently and improve response time and user productivity by reducing idle time.



**3. Real-Time Operating Systems**

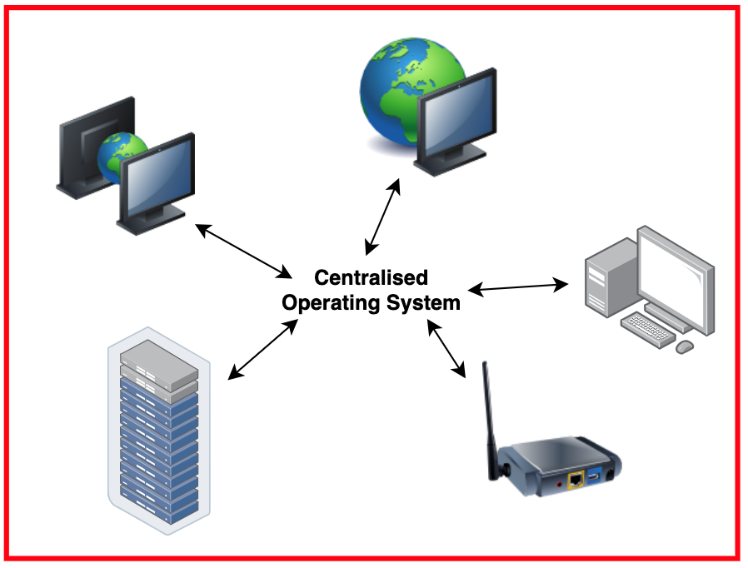
Real-Time Operating Systems (RTOS) are used in time-critical environments where reliability and deadlines are important. They are designed to guarantee a timely response to input events within a specified time frame.

These are used in applications that require deterministic behavior and strict timing constraints, such as process control, robotics, avionics, and medical devices.

**4. Distributed Operating Systems**

Distributed Operating Systems manage resources and coordinate tasks across multiple interconnected computers and nodes. They support distributed file systems, process management, and communication protocols across nodes.

This provides the user an interface to hide the underlying complexity and a simple way to execute programs across multiple machines and storage devices.



**5. Embedded Operating Systems**

Embedded Operating Systems are specialized operating systems designed to run on embedded systems, which are computer chips embedded into larger devices or machinery. These systems are resource-constrained and optimized for specific tasks and applications.

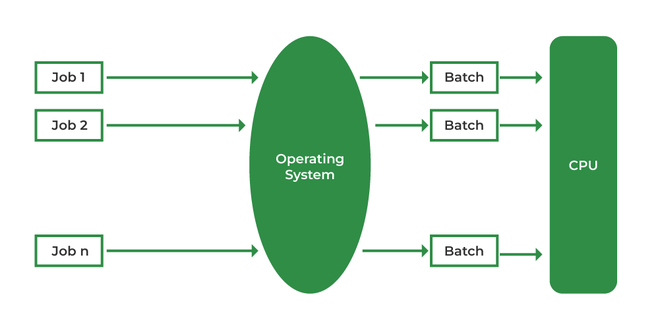
They are used in a wide range of embedded devices across various industries, including consumer electronics, automotive systems, industrial automation, medical devices, and IoT (Internet of Things) devices.

**Applications of Different Types of Operating Systems**

| **Operating System Type** | **Application** | **Example** |
| --- | --- | --- |
| Batch OS | Payroll Processing, Report Generation, Data Processing | IBM z/OS |
| Multiprogramming OS | Time Sharing Systems, Web Servers, Database Servers | Linux, macOS, Windows NT, MySQL, Nginx |
| Real-Time OS | Process Control, Robotics, Aerospace Systems | VxWorks, RTLinux, ROS |
| Distributed OS | Distributed File System, Cloud Computing, Peer-to-peer networks | Google’s File System, Apache Hadoop, BitTorrent |
| Embedded OS | Consumer Electronics, Automotive Systems, Industrial Automation, Internet of Things (IoT) devices | FreeRTOS, Contiki, AUTOSAR (Automotive Open System Architecture), QNX, Android (used in smartphones), iOS (used in iPhones) |

1. **Batch OS**

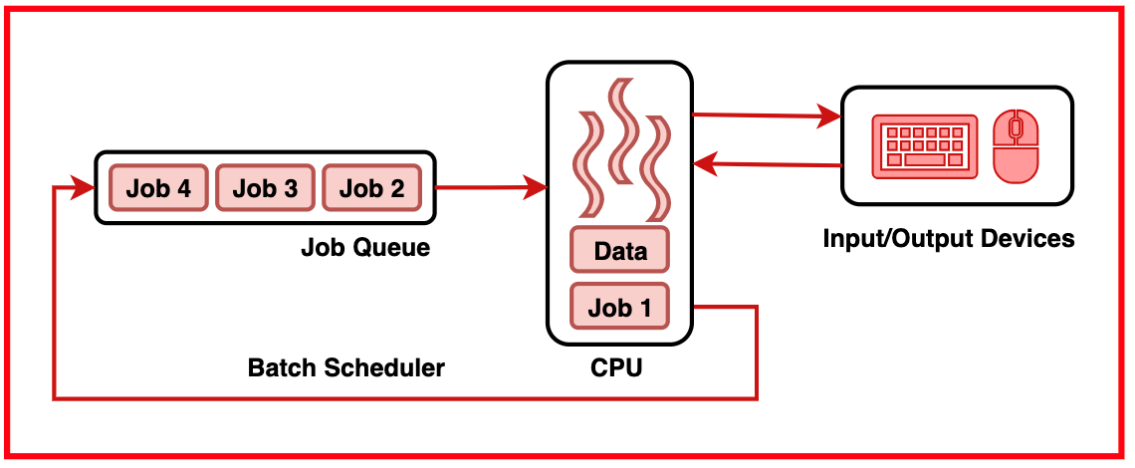
In batch operating system the jobs were performed in batches. This means Jobs having similar requirements are grouped and executed as a group to speed up processing. Users using batch operating systems do not interact with the computer directly. Each user prepares their job using an offline device for example a punch card and submits it to the computer operator. Once the programmers have left their programs with the operator, they sort the programs with similar needs into batches.



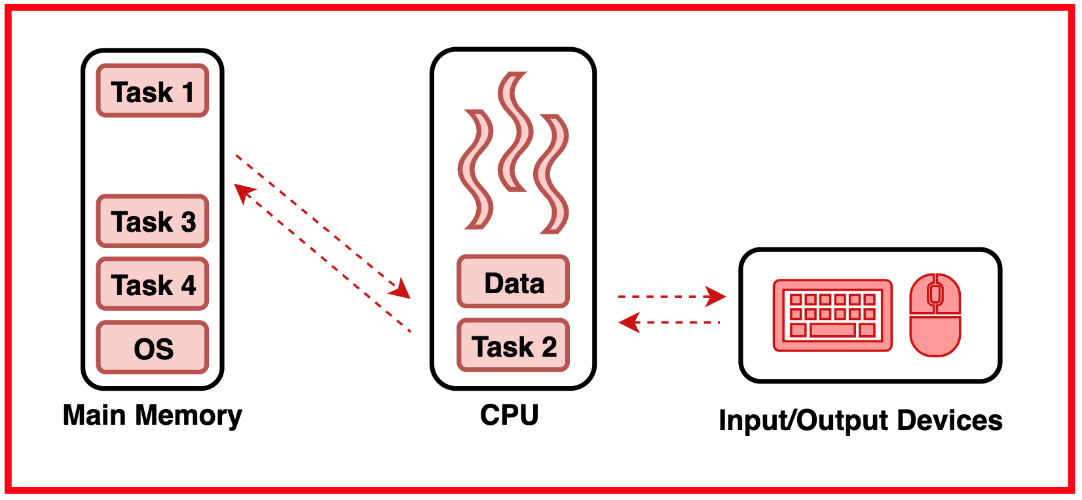
1. **Multiprogramming and Multitasking OS**

In a **Multiprogramming Operating System**, CPU utilisation is maximised by keeping multiple jobs in **main memory** simultaneously. If one job is occupied with **Input/Output (I/O) operations**, the CPU is assigned to another job.

Multiple jobs are organised in the main memory, and the CPU **switches between jobs** as needed. This approach reduces CPU idle time and increases overall system efficiency.



In a **Multitasking Operating System**, multiple tasks share common resources like the CPU and memory. The tasks are loaded into the CPU and executed by **switching rapidly** among them using a **small time quantum**. This switching occurs so quickly that users feel they are interacting with all tasks at the same time.



**Benefits of Multiprogramming and Multitasking**

* **Higher CPU Utilisation:** By allowing the CPU to switch between executing multiple programs or tasks, the system minimises idle time and maximises throughput.
* **Improved Responsiveness:** Users experience better responsiveness as multiple tasks can run concurrently, reducing waiting times and creating a smoother user experience.
* **Fault Isolation:** Programs or tasks run in **separate memory spaces**, so if one encounters an error or crashes, others remain unaffected, leading to greater reliability and stability.
* **Efficient Use of Hardware:** Sharing hardware resources among multiple programs reduces the need for extra hardware and supports scalability, allowing systems to handle growing workloads efficiently.

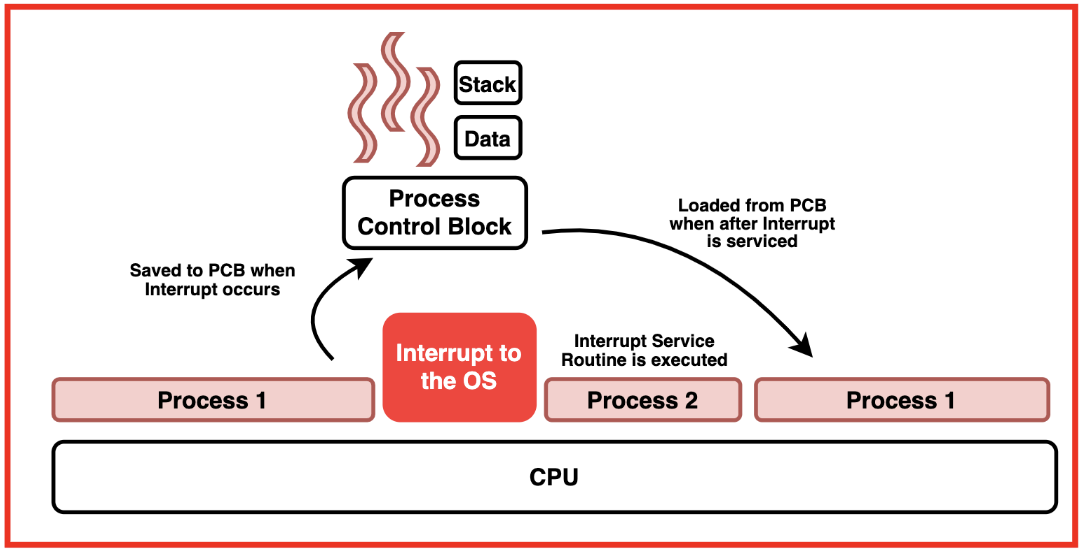
**Challenges and Limitations**

* **Resource Contention:** Multiple programs or tasks may compete for shared resources like CPU, memory, and I/O devices, leading to **resource starvation** for some programs.
* **Overhead of Context Switching:** The process of saving and restoring the state of a task during a switch consumes CPU cycles and resources. Increased context switching can negatively impact system performance.
* **Complex Synchronisation:** Managing concurrent access to shared resources and maintaining data consistency is difficult. Without proper mechanisms, issues like **race conditions** and **deadlocks** can occur.
* **Scalability Challenges:** Managing thousands or millions of concurrent tasks is hard. As systems grow, ensuring performance, reliability, and scalability becomes more complex.

**Context Switching**

**Context Switching** is the process of saving the state of a process or thread so that it can be paused and resumed later, allowing the CPU to switch between multiple tasks efficiently. It involves storing and restoring information like CPU registers and memory pointers.

Context switching enables multitasking operating systems to handle multiple tasks concurrently, creating the illusion of simultaneous execution.





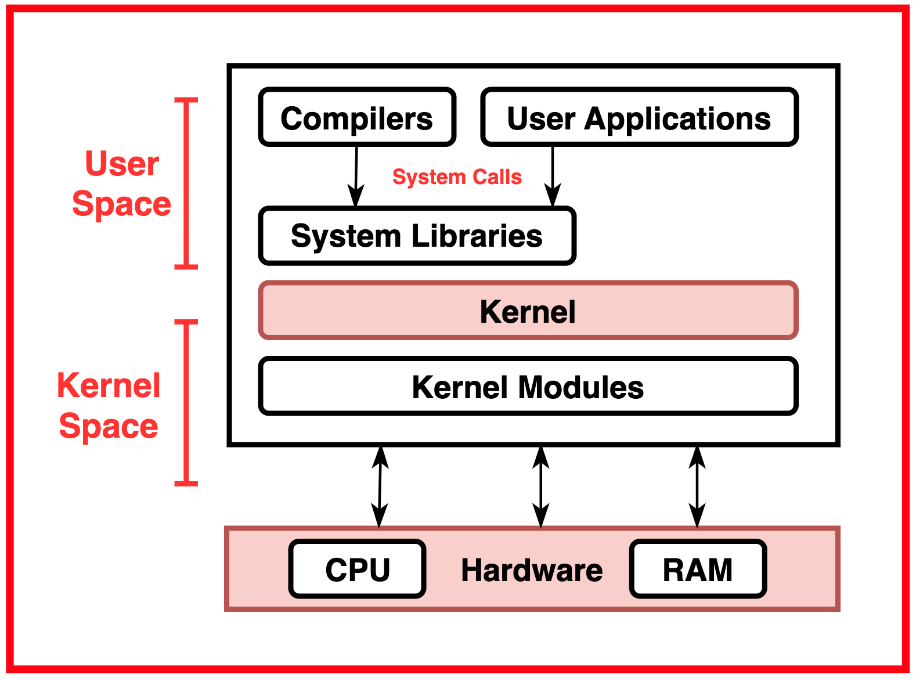
**Use Cases of Multiprogramming and Multitasking**

* **Example 1 - User Applications:** A user can run a word processor while listening to music and downloading files. Each application runs as a separate task, and the OS manages their execution efficiently.
* **Example 2 - Servers:** A web server uses multitasking to handle multiple user requests at the same time, ensuring each client gets a response without delay.
* **Example 3 - Background Processes:** System updates or backups run in the background while a user continues working on other applications, thanks to the multitasking capabilities of the OS.

1. **Kernel and user modes**

The **kernel** is the core component of the operating system that acts as a bridge between the hardware and software layers of a computer system. It is responsible for managing system resources, such as CPU, memory, and Input/Output devices, and providing essential services to run user applications.

As the central control program, the kernel governs the operations of the entire operating system.





**Kernel Mode**

The kernel operates in **privileged mode**, allowing it to access hardware resources and perform critical system-level tasks. It interacts directly with the hardware components of the computer and provides a layer of abstraction to shield user applications from the complexities of hardware management.

**User Mode**

**User Mode** is a restricted execution environment where user applications and processes run. In User Mode, programs have limited access to system resources and cannot perform privileged operations, such as accessing hardware or modifying system settings.

User Mode provides a protected environment that isolates user applications from the critical components of the operating system, ensuring system stability and security. User mode programs communicate with the kernel through controlled mechanisms, such as **system calls**, to request access to shared resources or perform privileged operations.

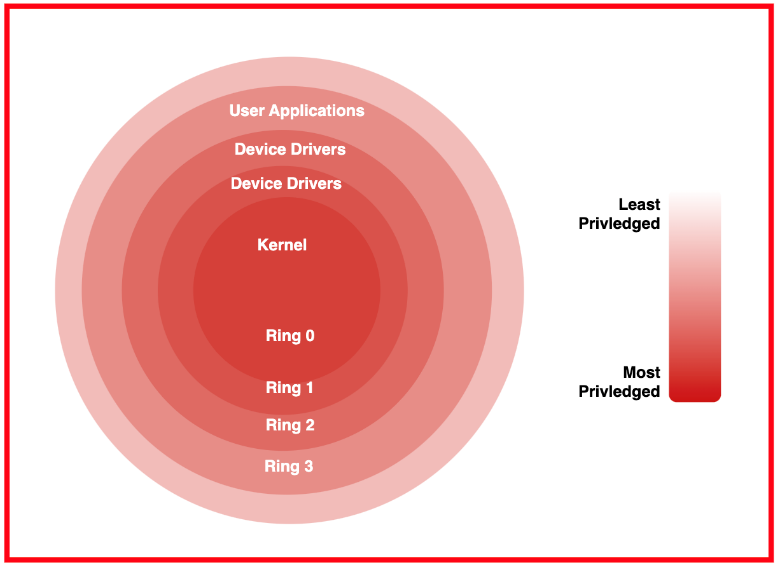
**Privileged Mode**

**Privileged Mode**, also known as Supervisor Mode or Kernel Mode, is an execution environment in which the operating system kernel runs with unrestricted access to system resources. It can perform critical operations like accessing hardware directly, modifying system settings, and managing system memory.

The kernel operates in privileged mode to execute essential system tasks, such as process management, memory allocation, and hardware control, which require elevated privileges beyond what user mode programs can access.

**Protection Rings**

**Protection rings** are a mechanism used by processors to enforce access control and privilege levels within the system. Modern processors typically support multiple protection rings, with **Ring 0** being the most privileged (kernel mode) and higher-numbered rings being less privileged (user mode).

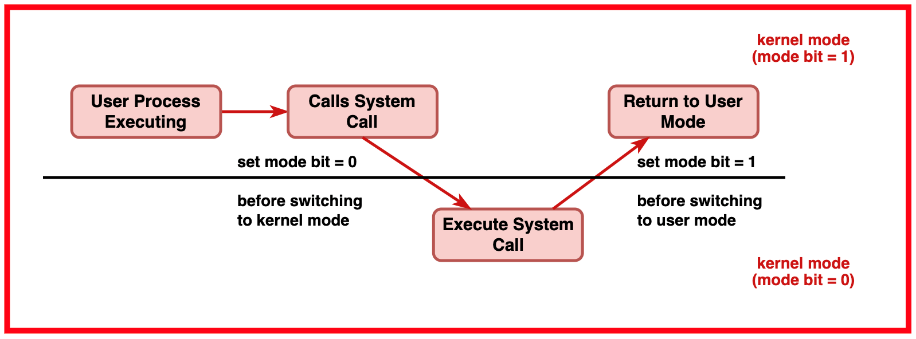




The operating system uses protection rings to define different privilege levels for executing code and accessing system resources. User mode programs operate within the least privileged protection ring, while the kernel operates within the most privileged ring, allowing the kernel to control access to system resources and ensure system security.

**Transition between Kernel and User Modes**

A computer system switches execution context from running in privileged mode (kernel mode) to running in restricted mode (user mode) to maintain security and stability while allowing user programs to access privileged resources through controlled means.





When a computer system boots up, it typically starts executing user programs in user mode. In user mode, programs have restricted access to system resources and cannot perform privileged operations directly.

When a user program needs to perform a privileged operation or access protected resources (such as reading from or writing to disk), it makes a request to the kernel through a mechanism known as a **system call**.

Upon receiving a system call request, the CPU switches execution context from user mode to kernel mode. This transition involves changing the processor's execution mode and switching to a separate stack (kernel stack) reserved for handling kernel-level operations.

In kernel mode, the operating system kernel gains full access to system resources and can perform privileged operations on behalf of the user program. It validates the system call request, checks for necessary permissions, and executes the requested operation.

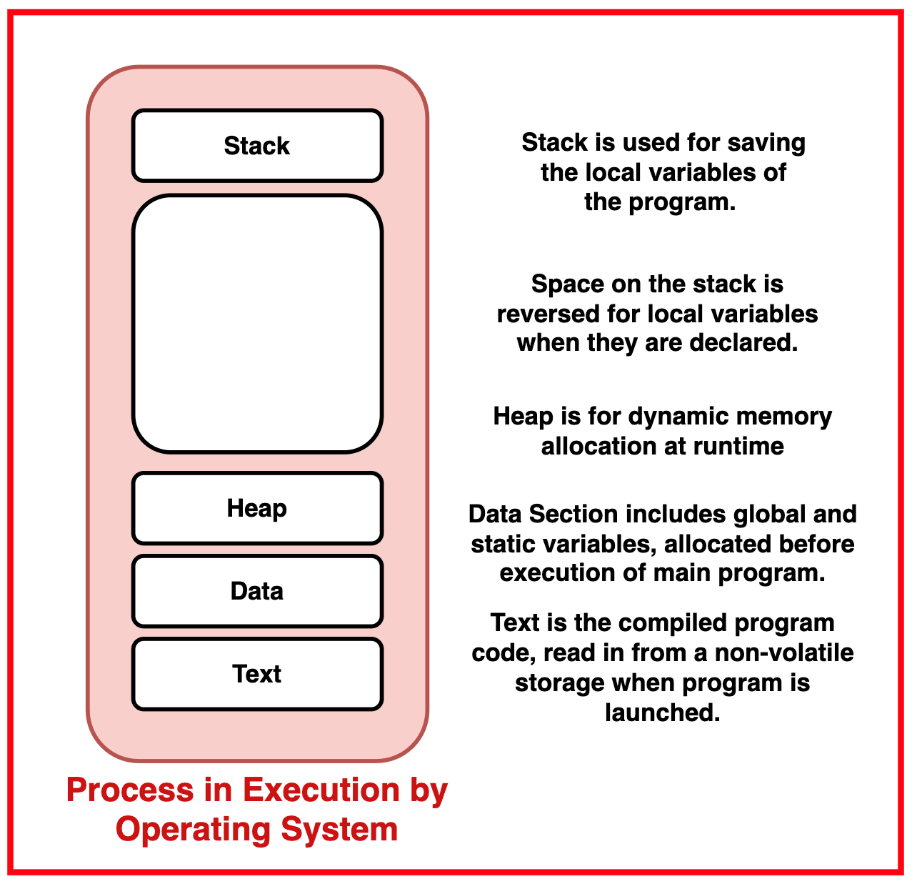
After completing the requested operation, the kernel prepares the result (if any) and copies it back to the user space. It also updates the CPU registers to restore the user program's execution state.

Once the system call handling is complete, the CPU switches execution context back to user mode. The program resumes execution from the point where the system call was invoked, continuing its normal flow with the result of the system call operatio

1. **Process and its states**

**Process** is an active task on a computer system that requires a set of resources, including memory, CPU time, and special registers to perform its functions. It is an instance of a program in execution.

A process can be considered as a unit of work that has been submitted by the user. It consists of an executable program along with the associated data and resources required for its execution.



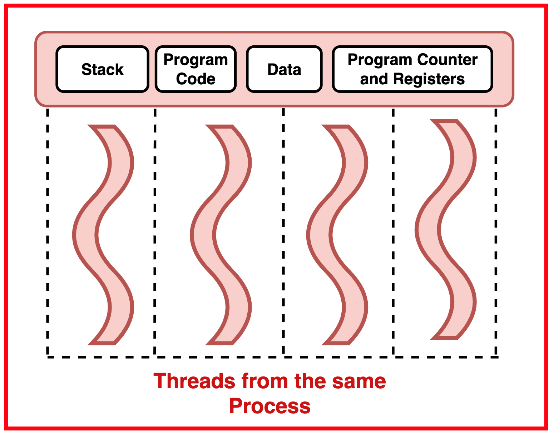


**Process Management** involves the activities related to the creation, manipulation, scheduling, and termination of processes in an operating system. It includes functions such as process creation, process scheduling, process synchronisation, and inter-process communication.

The primary goal of process management is to allocate resources to processes in a fair and efficient manner to ensure optimal utilisation of system resources.

**Multithreading** is the capability of an operating system to support multiple threads within a single process. A thread is a lightweight process that shares the same memory space and resources with other threads within the same process. Multithreading allows concurrent execution of multiple threads within a process, enabling better utilisation of CPU resources and enhanced responsiveness in applications.

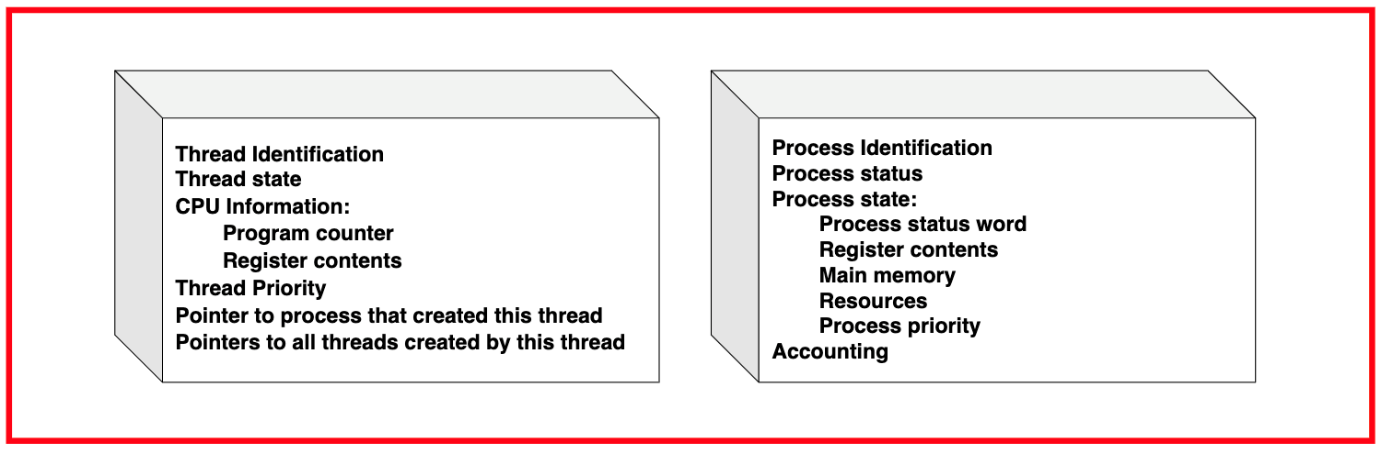
Each thread within a process can perform independent tasks and share data and resources with other threads, facilitating efficient parallelism and concurrency in program execution.





**Process Control Block (PCB) and Thread Control Block (TCB)**

The **Process Control Block (PCB)** and **Thread Control Block (TCB)** are data structures used by operating systems to manage processes and threads, respectively. They contain essential information about the state, resources, and execution context of processes and threads, allowing the operating system to effectively manage and control their execution.





**Process Control Block (PCB)**

PCBs are created by the operating system when a process starts and are updated throughout its lifetime. They are assigned a unique identifier for distinguishing them from other processes and contain status information indicating if the process is on hold, ready, running, or waiting.

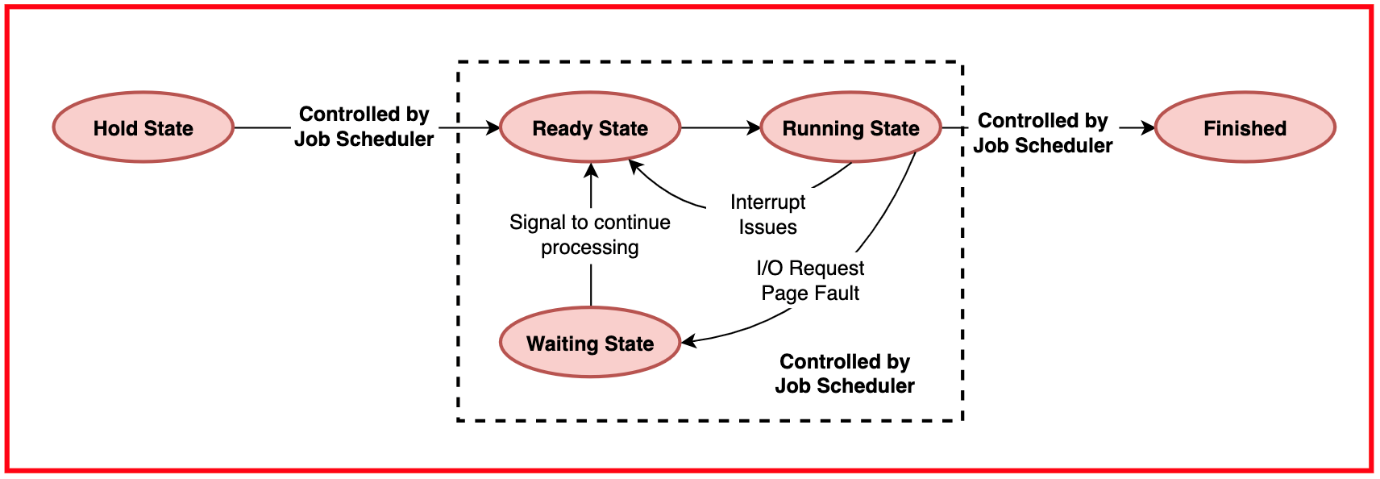
It stores process state, including the instruction counter and register contents when not running, and holds the CPU register contents if interrupted. PCB includes details of memory allocation, such as address and virtual memory mapping, and lists all allocated resources like hardware units or files. They are used by priority scheduling algorithms for determining execution order and hold accounting information for billing and performance measurement.

**Thread Control Block (TCB)**

TCBs are created by the operating system when a thread is initiated and are updated during its execution. Each TCB is assigned a unique identifier to distinguish it from other threads. It contains information about the thread's current state, indicating if it's ready, running, or waiting.

The TCB stores the execution context, including details about the current instruction being executed and the data being used. Additionally, it includes scheduling information used by the scheduler to manage thread execution, such as priority levels and scheduling parameters.

**Process States**





A process undergoes various states through the Operating System, from being initially placed on hold to its eventual completion. In a multiprogramming system, where the CPU can be allocated to many jobs, each with numerous processes, managing processor resources becomes quite complex.

Initially, when a user submits a job into the system, it enters the hold state. This state is controlled by the Job Scheduler, which places the job in a queue for further processing. At this stage, characteristics of each job, such as CPU time estimate, priority, and resource requirements, are noted down in a table by the job spooler or disk controller.

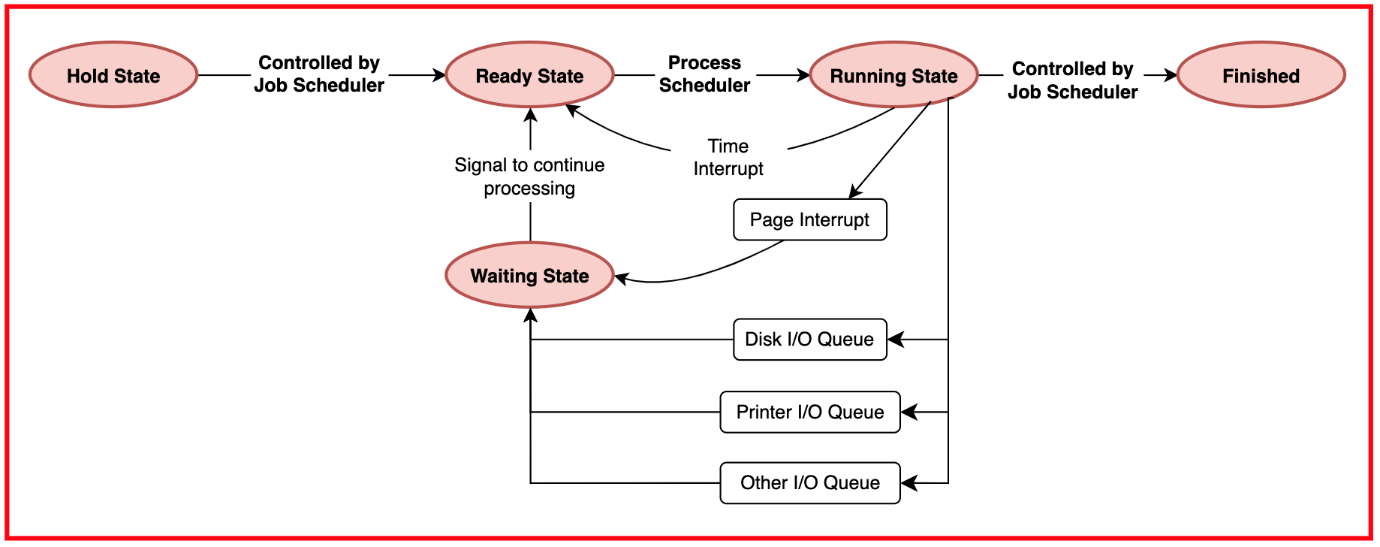
Once the interrupts related to the job are resolved, it transitions to the ready state. The running state signifies that the job is actively being processed by the CPU. In a single-processor system, only one job or process can be in the running state at any given time.

If a job encounters a situation where it cannot proceed, such as waiting for a specific resource allocation or an I/O operation to finish, it enters the waiting state. Upon resolving the waiting condition, the job transitions back to the ready state, indicating its readiness to resume execution.

Finally, when a job completes its execution, it enters the finished state and is returned to the user. Throughout these transitions, the Job Scheduler and Process Scheduler play crucial roles. The Job Scheduler initiates transitions between hold, ready, and finished states based on predefined policies, while the Process Scheduler manages transitions between ready, running, and waiting states based on predefined algorithms.

**Process Creation and Termination**

When a process is created, the operating system allocates the necessary resources, such as memory space, registers, and system data structures like the Process Control Block (PCB). This involves initialising the process state, assigning a unique identifier, and setting up the execution environment. The creation process typically involves invoking system calls or library functions, which may include specifying the executable program, defining command-line arguments, and configuring process attributes.



On the other hand, process termination involves releasing all resources associated with the process and reclaiming memory and system structures. This includes deallocating memory space, closing open files, releasing hardware resources, and updating relevant data structures like process tables. Additionally, the operating system may perform cleanup tasks, such as notifying parent processes, updating accounting information, and handling any child processes. Proper process termination is essential for maintaining system stability and resource efficiency, ensuring that system resources are efficiently managed and available for allocation to other processes.

1. **Normal function call vs System Calls in OS**

**Function Calls**

Function calls are used to execute a specific block of code like a function or subroutine within a program. They run **within the same execution context** or address space as the calling program.

Function calls are suitable for tasks that are part of the program itself, such as:

* Performing calculations
* Manipulating data structures
* Invoking utility functions

**Parameters** and data are passed to functions through the program's stack or registers, depending on the programming language and system architecture.

Function calls are usually resolved **at compile-time or link-time**, which means they have a lower execution overhead compared to system calls.

**System Calls**

System calls provide a way for applications to **request services from the operating system**. Unlike function calls, system calls run in a different execution context or address space, as they need to interact with the privileged kernel mode of the OS.

System calls are used to access operating system features like:

* File I/O (e.g., reading or writing files)
* Process management (e.g., creating or terminating processes)
* Memory allocation
* Inter-process communication

They abstract the details of the OS implementation, providing a standard way to access its services.

**Parameters** for system calls are passed in a different way compared to function calls, as they have to transition between **user space and kernel space**. This can involve passing parameters in registers, memory blocks, or the stack.

System calls are typically implemented as software interrupts or traps, which **transfer control to the kernel**. This transition causes system calls to have higher execution overhead compared to function calls.

**Difference between Function Calls and System Calls**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Function Calls** | **System Calls** |
| **Execution Context** | Within the same execution context/address space as the calling program | Operate in a different execution context/address space, interacting with the OS kernel |
| **Purpose** | Execute specific blocks of code within the program | Request services from the operating system |
| **Usage** | Used for tasks like calculations, data manipulation, and utility functions | Used for accessing OS functionalities like file I/O, process management |
| **Parameters Passing** | Passed via the program's stack or registers | Passed differently, often bridging user space and kernel space |
| **Resolution Time** | Resolved at compile-time or link-time | Resolved at runtime when invoked by the program |
| **Overhead** | Lower execution overhead | Higher execution overhead due to context switching and kernel interaction |
| **Example** | int sum = add(2, 3); | int fd = open("file.txt", O\_RDONLY); |

**Example Application: Calculator**

**Function Calls:** In a calculator application, function calls like add, subtract, multiply, and divide are used to perform arithmetic operations. These functions are called whenever the user performs an operation like adding two numbers.

**System Calls:** If the calculator needs to **store the calculation history** in a file, it would use system calls. For example, it could use system calls to write the results to a text file or read the stored history.