## What is operating system?

Operating System lies in the category of system software. It basically manages all the resources of the computer. The operating system works as a communication channel between system hardware and system software. The operating system helps an application with the hardware part without knowing about the actual hardware configuration. It is one of the most important parts of the system and hence it is present in every device, whether large or small device.

Once installed, the operating system relies on a vast library of device drivers to tailor OS services to the specific hardware environment. Thus, every application may make a common call to a storage device, but the OS receives that call and uses the corresponding driver to translate the call into actions (commands) needed for the underlying hardware on that specific computer.

Overall all the hardware related stuff, like mostly memory management, and underlying virtual memory managements and file management etc are handled by OS, on which all application runs without worrying about these stuff

## What are the functions of an operating system?

So as we defined what is OS, we can define it’s functions as:

-Resource Allocation, de-allocation, management

-File management, memory management

-Abstraction: Hides underlying complexity for Applications

**Types of OS**

Types like batch and single processing OS are not relevant so we will not discuss that

**Multi programming** - One process at a time with context switching

**Multi tasking** - One process at a time with context switching based on time sharing

**Multi processing** - One process at a time with context switching based on time sharing and multiple processors

**Context switching:** The OS keeps several jobs in memory simultaneously. When one job is waiting for I/O, the OS can switch to another job.

In real sense multiple process runs on only multiple processing type of OS as it has multiple core of the CPU. While other types just gives delusion of execution of multiple process though in reality it’s just one process at a time but with fair share amount of time as context switch happens

**Distributed OS:** Distributed OS is an operating system that runs on multiple interconnected computers and provides a unified and transparent interface to the user as if it were running on a single machine

**Components of OS**

There are mainly Two components of OS: user space and kernel. **User space** where application runs isolated from kernel and they do not have direct access to hardware. They makes system calls to request service from the kernel.

The **kernel** is the core component of the OS, responsible for managing hardware, processes, memory, and system calls, acting as a bridge between hardware and user space. It’s basically a Main character of the OS story which is a layer between hardware and User space.

Both user space and kernel space exist in the same physical RAM but are separated logically by the operating system and the hardware (through memory management units and privilege modes).

**System Calls**: Facilitate communication between a user space application and the kernel. They are primarily used when an application needs to request services from the kernel.

Ex.We write mkdir <folder-name> we are in user space. It makes a system call to kernel, which does the file management work by creating folder in disk. And after that switching to user space we see a folder icon with given name.

**IPC**: Refers to mechanisms that enable communication between multiple processes in user space. These mechanisms (like pipes, sockets, or shared memory) allow processes to exchange data and coordinate without direct interaction with the kernel for each communication (though the kernel facilitates the initial setup of IPC).

Kernel has some types. One of them is **Monolithic** where all of the works we mentioned above done by kernel only. Monolithic kernels tend to become large and complex because all system services run in kernel space. While this approach can lead to faster performance due to direct access to hardware, it can also make the kernel harder to maintain and more prone to bugs. Chad OS like Linux/Unix adopted this approach.

Another type is **Micro kernel** where Important services are kept in kernel while other services like shift to User space. while certain services are moved to user space (like file management), the micro kernel still has to manage communication between these services, which introduces overhead and can lead to slower performance.

So some big brain people made out of box solution: **Hybrid kernel** where we combine elements of both monolithic and micro kernels by placing essential services in the kernel while moving less critical services to user space. This aims to balance performance and modularity, as seen in systems like Windows and macOS."

**Computer Booting Process**

**Power On**:

* The computer is powered on by the electrical supply or battery.

**BIOS/UEFI Initialization**:

* The CPU loads the BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface).
* BIOS/UEFI retrieves system settings from CMOS (Complementary Metal-Oxide Semiconductor) and performs the Power-On Self-Test (POST) to check hardware functionality.

**Finding and Loading the Bootloader**:

* BIOS/UEFI searches for a bootable device according to the specified boot order.
* On older systems, the bootloader is located in the Master Boot Record (MBR) at the 0th sector of the disk.
* On modern systems, the bootloader is located in the EFI System Partition (ESP) on the disk.

**Executing the Bootloader**:

* The bootloader is loaded into RAM and executed.
* It then loads the operating system kernel from the disk into RAM.

**Loading the Operating System**:

* The operating system kernel initializes hardware, sets up the system environment, and begins running system services, completing the boot process.

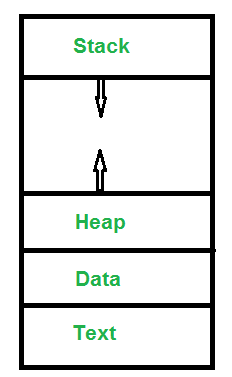
1. **bit vs 64-bit OS**

Memory in computers is organized into cells or blocks, each with a unique address. The size of the address determines the maximum amount of addressable memory.

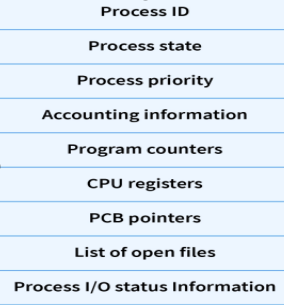
In a 32-bit operating system, the address size is 32 bits, which allows for addressing up to 2^32 bits memory locations, or 4 GB of RAM. However, practical constraints often limit usable memory to around 3 GB to 3.5 GB due to system reservations and hardware limitations. In contrast, a 64-bit operating system uses 64-bit addresses, theoretically allowing for 2^64 addressable locations, which equates to 16 exabytes of memory. While this theoretical limit is extremely high, real-world systems are typically designed to handle up to several terabytes of RAM. Most consumer systems, for example, support up to 1 TB or more, demonstrating the significant advantage of 64-bit systems in managing large amounts of memory.

**Process management**  
**Process creation:**

A program, such as an .exe file, becomes a process when it is executed. This transition involves several stages. First, the program is loaded into memory (RAM) along with its static data, which includes local variables, function arguments, and return value variables. This static data resides in the stack, a specific memory region allocated for the program. Next, the heap is allocated for dynamic data storage, which will be used while the process is running. Following this, I/O and error handles are set up. Once these preparations are complete, the main() function of the program is called, marking the completion of process creation and the start of the process’s execution.

**Process Architecture:**  
  
In the typical process architecture, the **Text** segment contains the program code, while the **Data** segment holds global variables. The **Heap** is reserved for runtime memory allocation, which grows as the process runs, while the **Stack** contains function variables, return values, and function call information. Both the stack and heap are dynamic, meaning they can grow as needed, but they are also subject to certain limits, which can result in stack overflow or maximum heap memory errors.  


**Characteristics of Process:**  
All the data related to a process is stored in a structure known as the Process Control Block (PCB). The PCB includes several key components:



**Process ID**: A unique identifier for each process.

**Process State**: The current state of the process, allowing the OS to resume execution at the correct point after a context switch.

**Process Priority**: Determines the scheduling of the process.

**Accounting Information**: Tracks resources used by the process.

**Program Counter**: Points to the next instruction to be executed.

**CPU Registers**: Save the current state of the process during context switching.

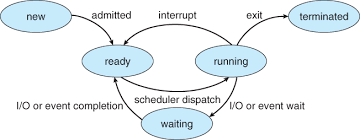
**PCB Pointers**: May link to other PCBs or related structures.

**List of Open Files**: Tracks files currently in use by the process.

**Process I/O Status Info**: Records the status of I/O operations.

While the structure of the PCB can vary, these core concepts remain consistent across different operating systems.

**States of Process:**



When a program is executed and converted into a process, it enters the **New** **state**, where it is placed in the job queue, awaiting further action. The Job Scheduler, or **Long Term Scheduler (LTS)**, is responsible for admitting the process from the job queue into the **Ready state**. In the Ready state, the process is queued up, waiting to be executed on the CPU. The Process Scheduler, or **Short Term Scheduler (STS)**, manages the transition of processes from the Ready state to the **Running state**, where they are actively executed by the CPU. This transition is governed by various scheduling algorithms, such as First Come First Serve (FCFS), Round Robin, or Priority Scheduling, depending on the OS design.

While in the Running state, the process can transition to different states based on its execution flow. If the process completes its execution, it moves to the **Terminated state**, where its resources are released back to the system. However, the process might not always run to completion in one go. It may be moved to the **Waiting state** if it requires I/O operations or specific events to occur. For example, if the process needs to read data from a file, it will wait until the I/O operation is complete before resuming execution.

In a multi-processing operating system, a process might be preempted, meaning it is **temporarily paused** to allow another process to use the CPU. This happens through a mechanism known as time-sharing, which ensures that all processes get fair access to the CPU. When a process is preempted, its current state, including register values, program counter, and other important information, is saved in its Process Control Block (PCB). This information is used to restore the process when it is returned to the Running state, a process known as **context switching**.

Once a process in the Waiting state completes its required I/O operations or other events, it is admitted back into the Ready state, where it can be dispatched for execution again. In some operating systems, there is also a **Mid-Term Scheduler (MTS)**, which temporarily swaps processes from the Ready queue to secondary storage (like a hard disk) to free up memory for other processes. This is done to improve system efficiency, especially when the Ready queue is overcrowded.

**Orphan and zombie process:**

In an operating system, every process has a **parent process** that initiated its creation. The very first process in the system is called the **"init" process** (often referred to as PID 1), which is the ancestor of all other processes, forming a hierarchical tree-like structure.

When a parent process is terminated, either due to completion or forceful termination, its child processes can become orphaned. An **orphan process** is one that continues to run even though its parent has been terminated. The operating system, specifically the kernel, takes care of these orphaned processes by reassigning them to the "init" process. This reassignment ensures that these processes still have a parent, maintaining the integrity of the process hierarchy and preventing them from being left unmanaged.

On the other hand, a **zombie process** is a different concept. When a process finishes execution, it doesn’t immediately disappear from the system. Instead, it returns an exit status to its parent process and enters the "terminated" state, but remains in the process table. This state is known as a "zombie" state. The process will stay in this state until its parent process reads the exit status using the wait() system call. Once the **parent acknowledges** the child's termination by reading the **exit status**, the system removes the zombie process from the process table. If the parent process fails to do this, the zombie process can linger in the system, though it no longer consumes any significant resources other than its entry in the process table.

In summary, while an orphan process is one that loses its parent and gets adopted by the init process, a zombie process is one that has completed execution but remains in the process table until its parent process retrieves its exit status.

**CPU scheduling Algorithms:**

Process scheduling is the method by which an operating system decides which process should move from the ready queue to the running state. This decision-making is crucial for optimizing CPU utilization and ensuring fair process handling. Scheduling algorithms fall into two categories: non-preemptive and preemptive. Non-preemptive algorithms do not allow a process to be interrupted once it starts executing; the process runs to completion before the next process is scheduled. This approach can be less efficient in a time-sharing system, as it does not support fair allocation of CPU time among processes. In contrast, preemptive algorithms enable the operating system to suspend and resume processes, allowing for better CPU time distribution and responsiveness, particularly in systems that require multitasking and fairness.

Key timings in process scheduling include **Arrival Time (AT)**, which marks when a process enters the ready queue from the job queue; **Completion Time (CT)**, the moment when a process finishes its execution; **Turnaround Time (TAT)**, calculated as the difference between CT and AT, representing the total time a process takes from arrival to completion; **Wait Time (WT)**, which is the total time a process spends waiting in the ready queue; and **Burst Time (BT)**, the time required by the process for CPU execution.

The efficiency of scheduling algorithms is often measured by average wait time and average turnaround time. A lower average wait time indicates that processes are not waiting excessively before receiving CPU time. Lower average turnaround time suggests that processes complete their execution more quickly. Preemptive algorithms typically perform better in terms of these metrics, as they can dynamically allocate CPU time to ensure fair and efficient processing.

CPU scheduling algorithms manage how processes are selected for execution, balancing efficiency and fairness. **Non-preemptive algorithms** such as **First-Come, First-Served (FCFS)** schedule processes in the order they arrive, which can cause inefficiencies like the convoy effect, where a long process delays shorter ones. **Shortest Job First (SJF)** prioritizes processes with the shortest execution times, reducing average waiting time but potentially leading to the starvation of longer processes.

In contrast, **preemptive algorithms** allow the system to interrupt and resume processes, enhancing responsiveness. **Round Robin (RR)** assigns fixed time slices to each process, ensuring fairness through time-sharing but possibly increasing turnaround times if the time quantum is not optimally chosen. **Priority Scheduling** allocates CPU time based on process priorities, which can lead to lower-priority tasks being starved of resources.

**Multilevel Queue Scheduling** categorizes processes into different queues based on their priority and type, applying distinct scheduling algorithms to each queue. This approach balances diverse needs but adds complexity in managing multiple queues. **Multilevel Feedback Queue Scheduling** builds on this by allowing processes to move between queues based on their behavior and past execution. This adaptability helps in better handling varying process requirements and workloads, improving responsiveness and efficiency.

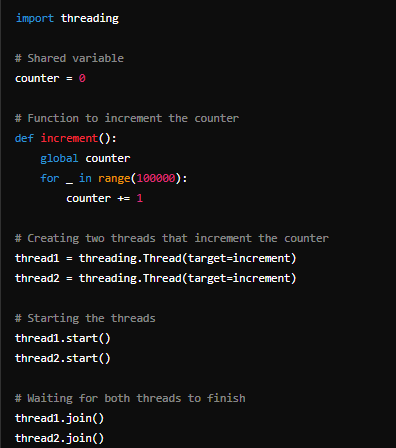
In practice, modern operating systems typically use **Multilevel Feedback Queue Scheduling** to handle a variety of workloads efficiently. This approach optimizes process management by dynamically adjusting priorities and queue placements, ensuring fair and effective CPU utilization across diverse applications.

**Concurrency:**

Concurrency refers to a system's ability to manage multiple tasks simultaneously, allowing them to progress without waiting for one to finish before starting another. This involves coordinating tasks efficiently to make optimal use of resources like CPU, memory, and I/O. Concurrency can occur both within a single application and across multiple applications running on a system.

When discussing concurrency, two key concepts are multiprocessing and multi-threading. Multiprocessing involves running multiple processes concurrently, each with its own memory space, and is typically executed on systems with multiple CPU cores. This allows for parallel execution, enhancing performance by utilizing the full processing power available. On the other hand, multi-threading occurs within a single process where multiple threads run concurrently, sharing the same memory space but with separate execution stacks. This can improve efficiency as threads handle different parts of a task simultaneously, but it also requires careful management to prevent conflicts and ensure proper synchronization since threads share resources and can interfere with each other.

In multi-threading, all threads execute concurrently while sharing the same memory, which can lead to issues such as race conditions. A race condition occurs when multiple threads attempt to access and modify shared data at the same time, leading to inconsistent or incorrect results. That’s why shared memory called critical section.Although processes have their own separate memory space, race conditions can still occur when they interact with shared resources such as files, databases, or inter-process communication (IPC) mechanisms like shared memory or message queues. If multiple processes try to access or modify a shared resource simultaneously without proper synchronization, a race condition can occur.

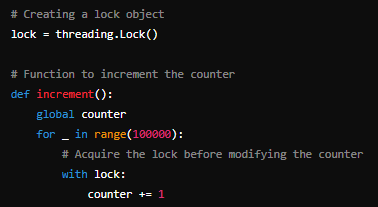


Ex:For instance, if two threads try to update a global counter variable simultaneously,

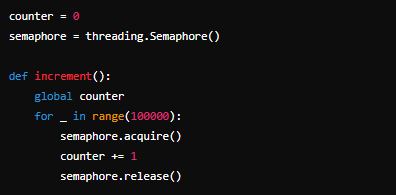
they might both read the same value and write back the incremented value, causing updates to be lost.

See in simpler terms, thread1 executing this line, counter = counter + 1. what if context switch happen to thread2 while, just counter + 1 got executed but didn’t assigned back to counter. So counter didn’t get updated.

To handle these issues, synchronization mechanisms like locks and semaphores are used. **Locks** ensure **mutual exclusion** by allowing only one thread to access a critical section of code at a time. When a thread acquires a lock, other threads must wait until the lock is released before they can access the critical section.



**Semaphores**, on the other hand, allow a limited number of threads to access a critical section concurrently. While semaphores manage access effectively, they don’t inherently guarantee data consistency, so locks are still necessary to prevent data inconsistencies in critical sections.

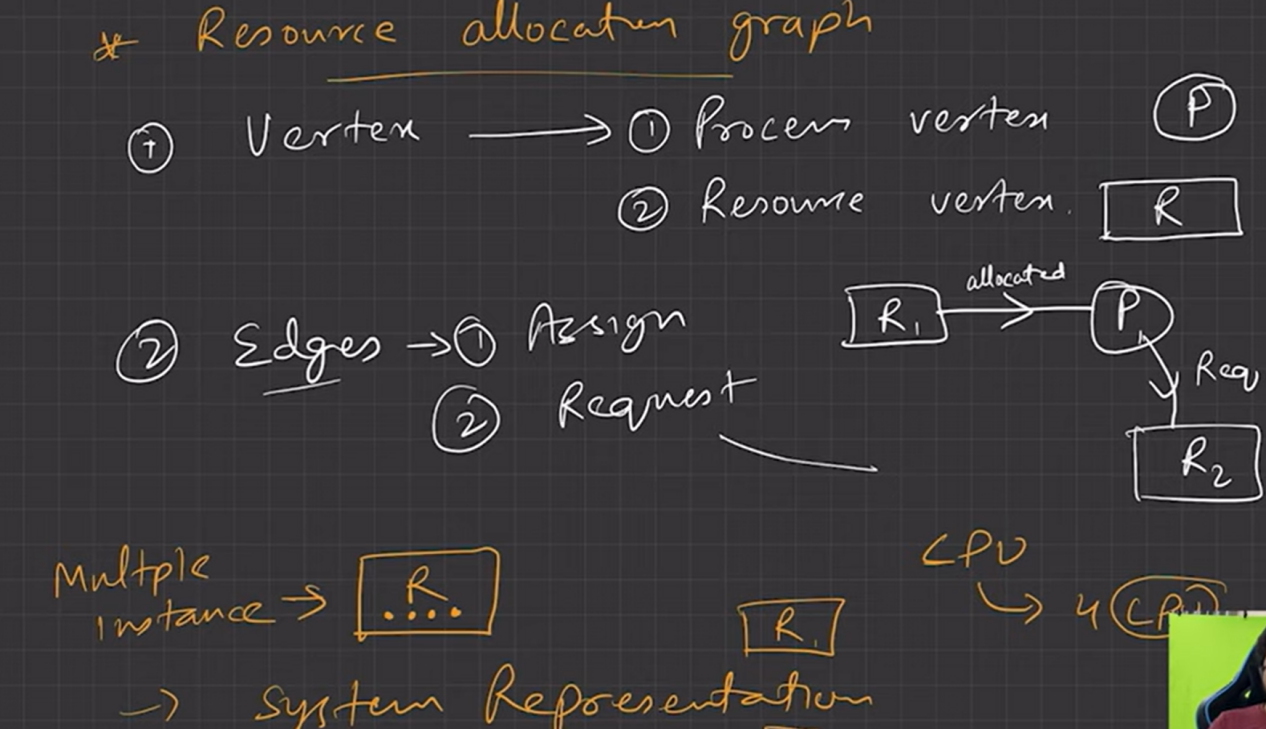


Thus, semaphores help manage access to shared resources, while locks ensure that only one thread modifies the critical section at any given time.

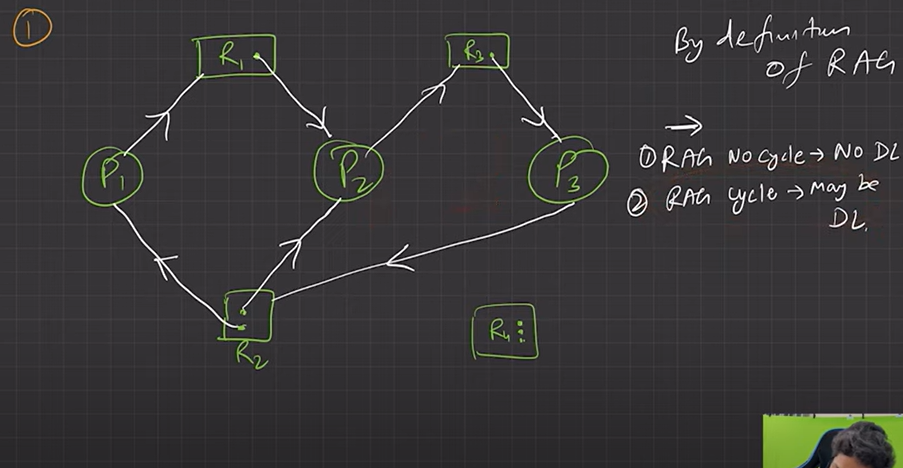
While any thread accessing critical section, thread acquires the lock. It might be possible due to bad code, that lock didn’t get released this condition is reffered as deadlock. This is also true for processes. In multiprocessing, the same concept of lock used. So here we will use term thread and process interchangebally. A process p1 acquires resource r1. process p2 acquires resource r2. p1 now waiting for r2 resource. P2 now waiting for r2. but both are circularly acquired by each other, so it became circular dependency and result into a deadlock situation.

So if process in condition of hold and wait, means holding/acquiring resource and waiting for resource at same time, and acquire is fully in control of process to release it, hold and wait condition is in some circular fashion, then it’s **deadlock!!!**

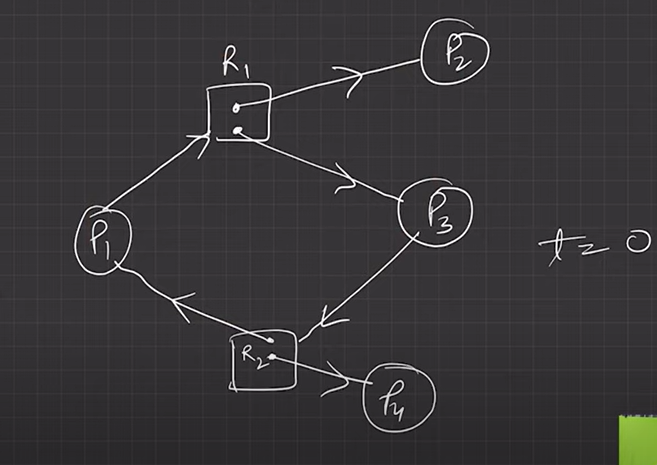
**RAG -** Resource Allocation Graph, is way to visualize the deadlock situation. processes are represented as nodes, and resources are shown as other nodes. An edge from a process to a resource indicates that the process is requesting the resource, while an edge from a resource to a process indicates that the resource is allocated to the process.



Now if there is any cycle in graph it’s may be deadlock, depending upon the instance of resources acquired by process, and process dependency on other resource. As presented in pic below, in cycle, r3 and r2 all instance of both resources are acquired by some process, and that process also waiting for the resources which are acquired by other. So it’s circular dependency which makes the deadlock.



In pic below, p2 acquiring R1 instance, but not waiting for another resource, same for p4 -r4. which will lead them to release r1 and r4 instances at some time because of no circular dependency.



To solve this deadlock problem we have multiple option. Deadlock Prevention/avoidance, detection, ignorance. Prevention basically means including code that prevents deadlock conditions to occur. Hold & wait will never occur if we allow process to request resources only when it releases the resources its holding. The power we give process to being preemptive, means no other process can acquire the resource until it releases, this power we have to take it from it. So when process request any resource which is currently holding by another process, process must release the resources acquired by it, it can be done interupt. When resources will be available it will be assigned back to it and process will lock/acquire them again. Since no process is hardly acquiring the resources there will be no such case as resource unavailability forever. So there will be no deadlocks. We have current state of system which has processes and resources, we schedule and allocate the process and resources in such way that deadlock will not occur is safe state and will occur is unsafe state. So scheduling and allocation in specific order will also avoid the deadlock. Banker algorithm is famous for doing that.

Other solution is to detect deadlocks, basically OS detects if deadlock arises, and then recover them by actions such as such as forcibly terminating one or more processes involved in the deadlock.

Jeezzz guess what, all of the above is mess. We have to ensure this and that condition and write bunch of code and implement some alien algorithms to avoid algorithm, and worst if deadlock detection. Every time OS will run deadlock detection algo. Where there is in real life 1 in 1000 case where deadlock arise and your system have to monitor system constantly to detect that 1 freaking deadlock. Guess what? Ignore the deadlock is best solution, no overhead of this and that just trust the OS, that’s how windows and Linux does and it really works pretty well right.

**Memory management**

Memory management is one of the most critical functions of an operating system (OS). It ensures that each running process has the necessary memory resources to function, while optimizing the use of the system’s physical memory (RAM) and secondary storage (such as hard disks). In this article, we'll explore the role of memory management in operating systems, how virtual memory works, and how modern systems handle memory allocation using techniques such as paging, swapping, and demand paging.

### ****1. What is Memory Management?****

Memory management refers to the way an operating system handles the allocation, tracking, and deallocation of memory for processes. The main objectives of memory management include:

* **Efficient allocation** of physical memory (RAM) to running processes.
* **Isolation** between processes to ensure they don’t interfere with each other's memory.
* **Maximization** of system performance by making effective use of RAM and secondary storage.
* **Security** by protecting memory spaces from unauthorized access by other processes.

The operating system uses several layers of abstraction and techniques to manage memory. This ensures that processes can run efficiently, even on systems with limited physical memory.

### ****2. Virtual Memory: The Core Abstraction Layer****

At the heart of modern memory management is the concept of **virtual memory**. Virtual memory is an abstraction that allows the operating system to present each process with the illusion of having access to a large, contiguous block of memory, even if the system’s physical RAM is limited.

#### ****What is Virtual Memory?****

Virtual memory is a **software-managed address space** that abstracts the actual physical memory (RAM) and allows the OS to handle memory in a flexible and efficient way.

* **Virtual memory is real**, but it is not physical memory. It is an **indirection layer** that maps the virtual address space seen by processes to actual physical locations in RAM or on disk.
* Each process gets its own **isolated virtual address space**. This means that different processes can operate without interference, even if they use the same virtual addresses.

#### ****Why Do We Need Virtual Memory?****

The primary purposes of virtual memory are:

* **Multiprogramming**: Multiple processes can be run concurrently, even if their combined memory needs exceed the physical RAM available.
* **Security**: Virtual memory provides isolation between processes, preventing one process from reading or corrupting another’s memory.
* **Efficiency**: Only the portions of memory that are actively being used are kept in RAM. This allows systems to run processes larger than the available physical memory.

### ****3. Memory Allocation: From Program to Process****

Memory management begins the moment a program starts running and becomes a **process**. Here's how it works:

#### ****a) Before Execution: Program on Disk****

Before a program is executed, it exists as a static file on **secondary storage** (like a hard disk or SSD). This file contains the program’s code and data but is not yet associated with any memory in the system. At this point, the program is simply a file and has no virtual memory associated with it.

#### ****b) Process Creation: Assignment of Virtual Memory****

When a program is executed (for example, when you run a Python script), the operating system creates a **process** and assigns it a **virtual address space**. This virtual memory consists of various segments, such as:

* **Code segment**: Contains the executable instructions.
* **Data segment**: Stores static data, like global variables.
* **Heap**: Used for dynamic memory allocation.
* **Stack**: Stores function call information and local variables.

At this stage, the **entire process** has been given a **virtual address space**, but it doesn’t yet have any direct mapping to physical memory (RAM). The operating system has essentially reserved virtual memory for the process, but this virtual memory doesn’t correspond to any specific location in RAM until the process actually runs and accesses it.

### ****4. Virtual to Physical Address Translation****

As a process runs, it needs to access memory to read data, execute code, or store variables. The addresses it uses to refer to memory are **virtual addresses**. These virtual addresses must be translated into **physical addresses** in RAM by a component called the **Memory Management Unit (MMU)**.

#### ****How Does Address Translation Work?****

When the CPU tries to access a memory location, it uses a **virtual address**. The MMU, with the help of the OS, translates this virtual address into a **physical address** using a data structure called the **page table**. This page table holds the mapping between virtual pages (small fixed-size chunks of the process’s virtual memory) and physical page frames (blocks of RAM).

The steps are as follows:

1. **Process accesses a virtual address**.
2. The **MMU** consults the **page table** to find the corresponding physical address in RAM.
3. If the page is in RAM, the translation is immediate, and the process continues execution.

#### ****What if the Page is Not in RAM?****

If the required page is not in RAM, a **page fault** occurs. This is a signal that tells the operating system that the requested page is not currently in physical memory. The OS then:

1. **Pauses the process** and loads the required page from **disk** (secondary storage) into RAM (if it's not already there).
2. **Updates the page table** with the new mapping between the virtual address and the physical page frame in RAM.
3. **Resumes the process**, which now has access to the data in RAM.

This technique, called **demand paging**, ensures that only the necessary pages are loaded into memory, making efficient use of RAM.

### ****5. Paging and Page Faults****

Modern operating systems divide both **virtual memory** and **physical memory** into small, fixed-size blocks called **pages** and **page frames**, respectively. The size of a page (usually 4KB) is the same in both virtual and physical memory.

#### ****What is Paging?****

* **Paging** is the process of dividing memory into pages and mapping virtual memory pages to physical memory page frames.
* The page table maintains the mapping of virtual pages to physical page frames.
* The OS only loads pages into RAM as they are needed, which is why it's called **demand paging**.

#### ****Page Fault Handling****

A **page fault** happens when a process tries to access a page that is not currently in RAM. Page faults are common and are an essential part of memory management:

* The OS handles the fault by loading the page from disk into RAM.
* If RAM is full, the OS may need to **swap out** a page to free up space for the new one. This process of replacing pages is called **page replacement**, and various algorithms like **Least Recently Used (LRU)** or **First In First Out (FIFO)** are used to decide which page to swap out.

### ****6. Swapping and Page Replacement****

When the system runs low on physical memory, the OS may need to **swap** some pages from RAM back to disk to make space for more active pages. This is done using a part of the disk called the **swap space**.

#### ****How Swapping Works:****

* If a page in RAM has not been used for a while, the OS can move it to **swap space** on the disk to free up RAM.
* When that page is needed again, the OS **swaps** it back into RAM, potentially replacing another page.

Swapping is slower than accessing memory in RAM because disk I/O (input/output) is much slower than accessing RAM. However, it allows systems to run more processes than can physically fit in RAM.

### ****7. Conclusion****

**Memory management** is a fundamental aspect of modern operating systems, providing processes with a **virtual memory space** while optimizing the use of limited physical memory (RAM). Through the concepts of **paging**, **demand paging**, **page faults**, and **swapping**, operating systems ensure that multiple processes can run efficiently, even when physical memory is scarce.

By assigning each process its own **virtual memory space**, the OS ensures **process isolation**, **security**, and **efficient memory usage**. The dynamic translation of **virtual addresses** to **physical addresses** allows the system to seamlessly handle memory requests, bringing in pages from secondary storage (disk) as needed.

Memory management techniques like **paging** and **swapping** are key to ensuring the smooth operation of multitasking environments, allowing modern systems to maximize performance while maintaining memory efficiency.