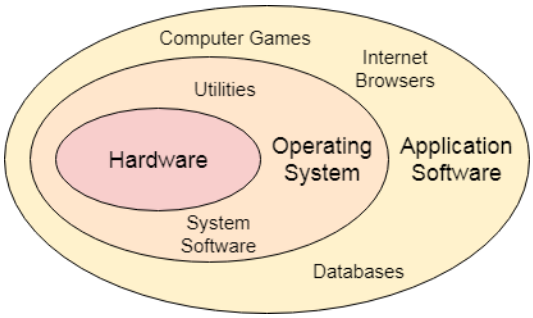
Operating System

PART 01: Introduction of OS and Types of OS

Definition:

In the Computer System (comprises of Hardware and software), Hardware can only understand machine code (in the form of 0 and 1) which doesn't make any sense to a naive user.

We need a system, which can act as an intermediary and manage all the processes and resources present in the system.



An **Operating System** can be defined as an **interface between user and hardware.** It is responsible for the execution of all the processes, Resource Allocation, CPU management, File Management and many other tasks.

The purpose of an operating system is to provide an environment in which a user can execute programs in convenient and efficient manner.

Application software performs specific task for the user.
System software operates and controls the computer system and provides a platform to run application software.

An **operating system** is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

Why OS?

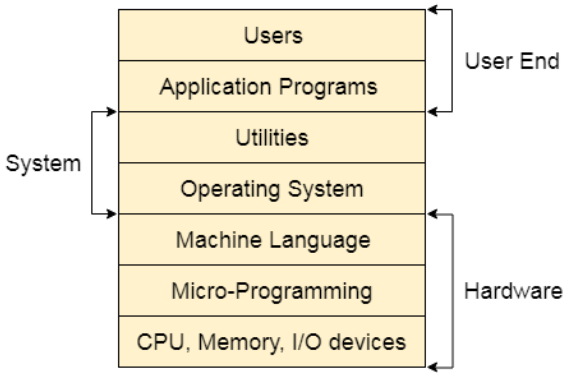
1. **What if there is no OS?**
    a. Bulky and complex app. (Hardware interaction code must be in app's code base)
    b. Resource exploitation by 1 App.
    c. No memory protection.

2. **What is an OS made up of?**
    a. Collection of system software.

Structure of a Computer System:

A Computer System consists of:

- Users (people who are using the computer)
- Application Programs (Compilers, Databases, Games, Video player, Browsers, etc.)
- System Programs (Shells, Editors, Compilers, etc.)
- Operating System (A special program which acts as an interface between user and hardware)
- Hardware (CPU, Disks, Memory, etc.)



Why need Operating System (OS):

What does Operating System do?

1. Process Management
2. Process Synchronization
3. Memory Management
4. CPU Scheduling
5. File Management
6. Security

Goals of Operating System:

- **Maximum CPU Utilization**.
- **Less process starvation**, (process starvation means not stuck in only one process. It is better to handle the multiple processes continuously).
- **Higher priority job execution**, (means, example: suppose when you insert any hard drive then it scan the virus immediately → it is a high priority execution).

Types of Operating System:

1. Single process operating system → [MS DOS, 1981]
2. Batch process operating system → [ATLAS, Manchester Univ., late 1950s – early 1960s]
3. Multi-programming operating system → [THE, Dijkstra, early 1960s]
4. Multi-tasking operating system → [CTSS, MIT, early 1960s]
5. Multi-processing operating system → [Windows NT]
6. Distributed operating system → [LOCUS]
7. Real time operating system → [ATC]

Single Process Operating System:

A Single-Process Operating System (SPOS) is a basic operating system design where **only one process can be executed at any given time.** In other words, the CPU (Central Processing Unit) is dedicated to running a single program, and there is no concept of multitasking. When that program finishes execution or is terminated, the operating system can then load and execute another program.

In a Single-Process Operating System:

- **Sequential Execution:** Programs are executed one after another in a sequential manner. When a program finishes execution, the operating system loads the next program into memory and executes it. This sequential execution model is straightforward but not efficient for modern computing needs where multitasking is essential for optimal system utilization.

- **Limited Resource Utilization:** Only one program is allowed to use system resources at any given time. This means that CPU cycles, memory, and other resources are dedicated to the running program. If a program requires extensive computational resources, other programs or tasks have to wait until it completes.

- **No Process Scheduling:** There is no need for complex process scheduling algorithms because there is only one process to manage. The operating system simply loads and executes that single program.

Single-Process Operating Systems are extremely rare in modern computing. Most contemporary operating systems, including Windows, macOS, Linux, and various mobile operating systems, are designed to support multitasking, allowing multiple processes or programs to run concurrently, providing a much more efficient and responsive computing environment.

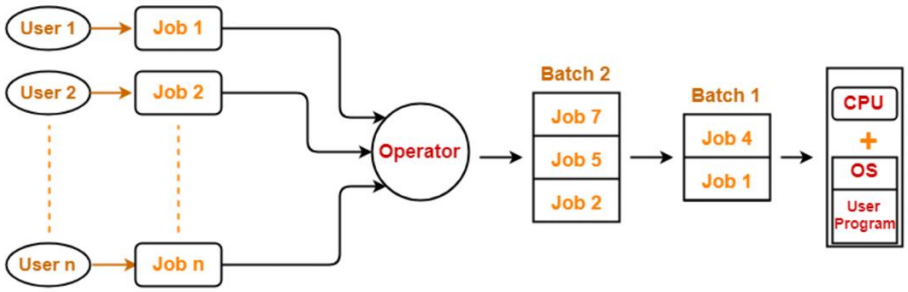
Batch Process Operating System:

In Batch operating system, access is given to more than one person; they submit their respective jobs to the system for the execution.

The system put all of the jobs in a queue based on first come first serve and then executes the jobs one by one. The users collect their respective output when all the jobs are executed.

The purpose of this operating system was mainly to transfer control from one job to another as soon as the job was completed. It contained a small set of programs called the resident monitor that always resided in one part of the main memory. The remaining part is used for servicing jobs.

- Firstly, user prepares his job using punch cards.
- Then, he submits the job to the computer operator.
- Operator collects the jobs from different users and sort the jobs into batches with similar needs.
- Then, operator submits the batches to the processor one by one.
- All the jobs of one batch are executed together.



Advantages of Batch OS:

- The use of a resident monitor improves computer efficiency as it eliminates CPU time between two jobs.

Disadvantages of Batch OS:

- Priorities cannot be set, if a job comes with some higher priority.
- May lead to starvation. (A batch may take more time to complete)
    - For Example: There are five jobs J1, J2, J3, J4, and J5, present in the batch. If the execution time of J1 is very high, then the other four jobs will never be executed, or they will have to wait for a very long time. Hence, the other processes get starved.

- CPU may become idle in case of I/O operations.

Multi-Programming Operating System:

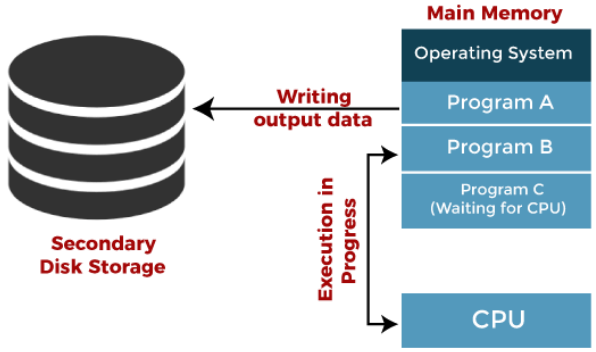
Multiprogramming is an extension to batch processing where the CPU is always kept busy. Each process needs two types of system time: CPU time and IO time.

In a multiprogramming environment, when a process does its I/O, The CPU can start the execution of other processes. Therefore, multiprogramming improves the efficiency of the system.

- Single CPU
- Context switching for processes.
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

PCB (Process Control Block):

Store the current state of a particular process from CPU to PCB. And send another process information to CPU for complete the process (it is called **context switching**).



Jobs in multiprogramming system

Advantages of Multi-Programming OS:

- Throughout the system, it increased as the CPU always had one program to execute.
- Response time can also be reduced.

Disadvantages of Multi-Programming OS:

- Multiprogramming systems provide an environment in which various systems resources are used efficiently, but they do not provide any user interaction with the computer system.

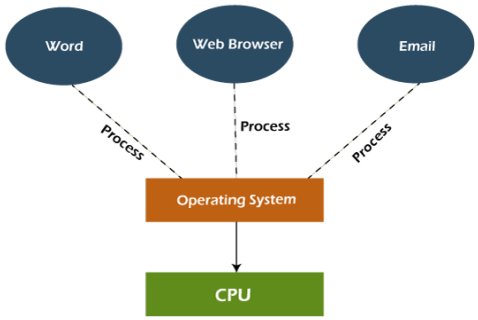
Multi-Tasking Operating System:

The multitasking operating system is a logical extension of a multiprogramming system that enables **multiple programs execute simultaneously**. It allows a user to perform more than one computer task at the same time.

Multi-Tasking Operating System (MTOS) is an operating system that allows multiple tasks or processes to run concurrently. Unlike a Single-Process Operating System (SPOS), where only one program can be executed at a time, multitasking operating systems enable users to run multiple applications simultaneously. This concurrent execution of multiple tasks enhances system efficiency, responsiveness, and user productivity.

- Single CPU.
- Able to run more than one task simultaneously.
- Context switching and time sharing used.
- Increased the responsiveness.

- CPU idle time is further reduced.



There are two main types of multitasking:

1. **Pre-emptive Multitasking:**
   In preemptive multitasking, the operating system has control over the execution of tasks. It can interrupt a currently executing task and switch to another task. The operating system uses a scheduler to determine which task to execute next. Preemptive multitasking ensures that no single task monopolizes the CPU, allowing for fair allocation of resources among multiple processes. This method is commonly used in modern operating systems like Windows, macOS, and various flavors of UNIX and Linux.

2. **Cooperative Multitasking:**
   In cooperative multitasking, tasks voluntarily yield control to other tasks. Each task is responsible for relinquishing control to allow other tasks to execute. If a task does not voluntarily yield control (for example, due to a programming error or infinite loop), it can cause the entire system to become unresponsive. Cooperative multitasking requires careful programming and can be less robust than preemptive multitasking. It was more common in early operating systems and is still used in certain specialized applications.

Key Features of Multi-Tasking Operating Systems:

- **Task Scheduling:** The operating system's scheduler decides which task to execute next based on priority, time slices, or other criteria. This scheduling ensures that tasks are executed fairly and in a timely manner.

- **Process Management:** Multi-tasking OS manages multiple processes, allocating system resources such as CPU time, memory, and I/O devices to each process. It keeps track of the state of each process and switches between them efficiently.

- **User Interaction:** Users can interact with multiple applications simultaneously, switching between them seamlessly. This capability enhances user productivity by allowing tasks like browsing the internet, listening to music, and working on documents to occur concurrently.

- **Resource Protection:** Multi-tasking operating systems provide isolation between processes to prevent one misbehaving or malfunctioning application from crashing the entire system. Each process operates in its protected memory space.

- **Improved System Utilization:** By allowing the CPU to switch between tasks, multi-tasking operating systems make more efficient use of system resources, ensuring that the CPU is almost always in use, leading to higher throughput and responsiveness.

Multi-tasking operating systems are the norm in modern computing environments, supporting the complex and varied needs of users and applications in today's digital world.

Advantages of Multi-Tasking OS:

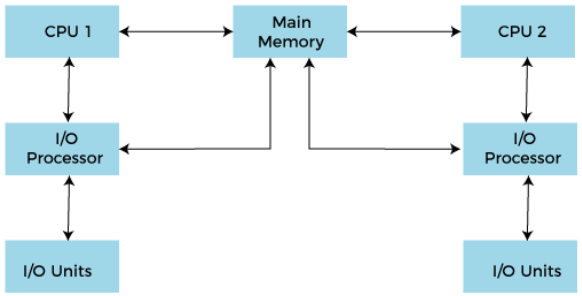
- This operating system is more suited to supporting multiple users simultaneously.
- The multitasking operating systems have well-defined memory management.

Disadvantages of Multi-Tasking OS:

- The multiple processors are busier at the same time to complete any task in a multitasking environment, so the CPU generates more heat.

Multi-Processing Operating System:

In Multiprocessing, Parallel computing is achieved. There are **more than one processors present in the system** which can execute more than one process at the same time. This will increase the throughput of the system.
In Multiprocessing, Parallel computing is achieved. More than one processor present in the system can execute more than one process simultaneously, which will increase the throughput of the system.



Working of Multiprocessor System

- More than 1 CPUs in a single computer (CPU 8 core: It means there are 8 logical processors are present. That can handle different tasks continuously. If one processor failed than another can still works).
- Increase the reliability, 1 CPU fails, other can work.
- Better throughput.
- Lesser process starvation (if one CPU is working on some process, other can be executed on other CPU).

What is Logical CPUs (CPU 8 Core):

- Logical CPUs refer to the number of virtual processors or threads that a processor can handle simultaneously. In the context of an 8-core CPU, it means that the physical CPU has 8 independent processing cores. Each core is capable of executing its own set of instructions.

- The term "**logical CPUs**" comes into play when hyper-threading or simultaneous multithreading (SMT) is used. Hyper-threading allows each physical core to handle multiple threads concurrently, essentially creating virtual or logical CPUs. For example, if you have an 8-core CPU with hyper-threading enabled, you might see 16 logical CPUs in the operating system. Each physical core can handle two threads simultaneously, making it appear as if there are more processors available.

A Multi-Processing Operating System (MPOS) is an operating system that can efficiently utilize and manage multiple central processing units (CPUs) within a computer system. Unlike multitasking, which involves running multiple processes or tasks on a single CPU, multiprocessing involves running multiple processes on multiple CPUs simultaneously. This parallel processing capability enhances system performance and allows for faster execution of tasks and applications. Here are the key features and benefits of multiprocessing operating systems:

Key Features of Multi-Processing Operating Systems:

- **Parallel Execution:** MPOS can execute multiple processes in parallel across multiple CPU cores. Each core can handle its own set of instructions independently, allowing for simultaneous processing of tasks.

- **Improved Performance:** By distributing tasks across multiple CPUs, MPOS can significantly improve the overall system performance. CPU-intensive tasks, such as complex calculations or multimedia processing, can be divided among multiple processors, reducing the time required for their completion.

- **Task Distribution:** MPOS efficiently distributes tasks or processes among available CPU cores. It can balance the workload dynamically, ensuring that all CPUs are utilized optimally.

- **High Throughput:** Multiprocessing allows the system to handle a large number of tasks simultaneously, leading to high throughput. This is particularly beneficial for servers, scientific simulations, video editing, and other computationally intensive applications.

- **Fault Tolerance:** Some multiprocessing systems offer fault tolerance features. If one CPU fails, the system can often continue functioning using the remaining CPUs, ensuring that critical tasks can still be executed.

- **Scalability:** Multiprocessing provides a scalable solution for increasing computational power. As the demands of applications grow, additional CPUs can be added to the system to enhance its processing capabilities.

- **Multicore Support:** With the prevalence of multicore processors, modern MPOS are designed to fully utilize the multiple cores available on each CPU chip. These systems can handle both multiple CPUs and multiple cores within each CPU.

- **Simultaneous Multithreading (SMT):** MPOS can support simultaneous multithreading, a technology that allows each CPU core to execute multiple threads concurrently. This further enhances multitasking capabilities and overall system performance.

Popular operating systems like Windows, Linux, and macOS have multiprocessing support built into their kernels, allowing them to take advantage of systems with multiple CPUs or multicore processors effectively. Multiprocessing is essential for handling the demands of modern applications, ranging from scientific simulations and data analysis to virtualization and server tasks, where efficient parallel processing can greatly enhance performance and responsiveness.

Advantages of Multi-Processing OS:

- **Increased reliability:** Due to the multiprocessing system, processing tasks can be distributed among several processors. This increases reliability as if one processor fails, the task can be given to another processor for completion.

- **Increased throughout:** As several processors increase, more work can be done in less

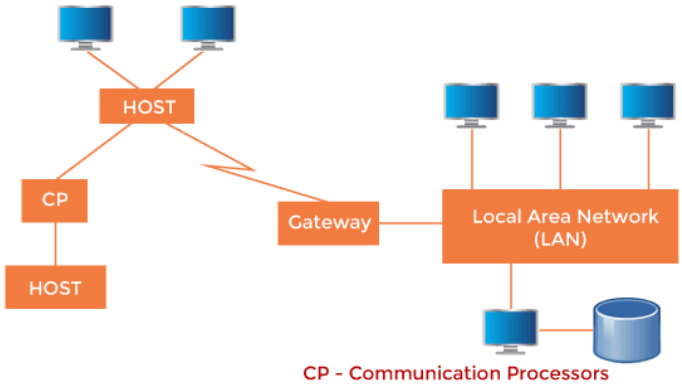
Disadvantages of Multi-Processing OS:

- Multiprocessing operating system is more complex and sophisticated as it takes care of multiple CPUs simultaneously.

Distributed Operating System:

The Distributed Operating system is not installed on a single machine, it is divided into parts, and these parts are loaded on different machines. A part of the distributed Operating system is installed on each machine to make their communication possible. Distributed Operating systems are much more complex, large, and sophisticated than Network operating systems because they also have to take care of varying networking protocols.

- OS manages many bunches of resources, >=1 CPUs, >=1 memory, >=1 GPUs, etc.
- **Loosely connected autonomous,** interconnected computer nodes.
- Collection of independent, networked, communicating, and physically separate computational nodes.



A Typical View of a Distributed System

Advantages of Distributed OS:

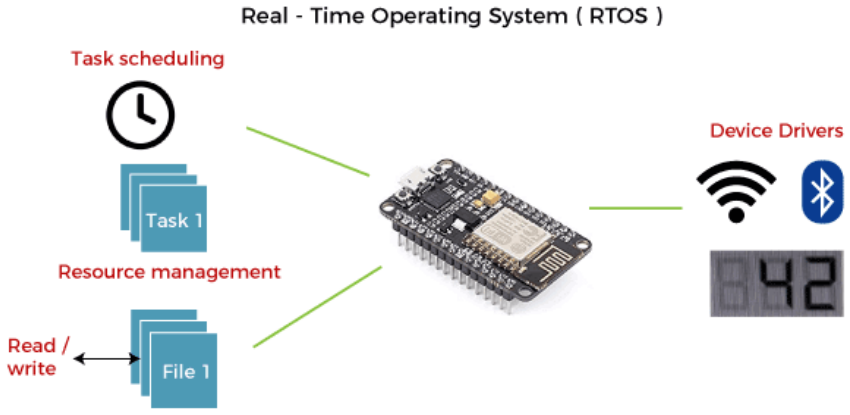
- The distributed operating system provides sharing of resources.
- This type of system is fault-tolerant.

Disadvantages of Distributed OS:

- Protocol overhead can dominate computation cost.

Real Time Operating System:

In Real-Time Systems, each job carries a certain deadline within which the job is supposed to be completed, otherwise, the huge loss will be there, or even if the result is produced, it will be completely useless.



The Application of a Real-Time system exists in the case of military applications, if you want to drop a missile, then the missile is supposed to be dropped with a certain precision.

- **Real time** error free, computations within tight-time boundaries.
- Air Traffic Control System (ATC), ROBOTS etc.

Advantages of Real Time OS:

- Easy to layout, develop and execute real-time applications under the real-time operating system.
- In a Real-time operating system, the maximum utilization of devices and systems.

Disadvantages of Real Time OS:

- Real-time operating systems are very costly to develop.
- Real-time operating systems are very complex and can consume critical CPU cycles.

PART 02: Multi-Tasking, Multi-Threading, kernel Space, User Space, System Call, etc.

Program:

A Program is an **executable file**, which contains a certain set of instructions written to complete the specific job or operation on your computer.

- It is a compiled code. Ready to be executed.
- Stored in Disk

".exe file": It is an executable file that helps to run the process, where process means program under execution. When you double click on **".exe"** file then process will execute.

Process:

In the context of operating systems and computer science, a **process** is a program in execution. It is an instance of a computer program that is running on a computer system. When a program is executed, the operating system creates a process in the computer's memory to represent the program's execution. This process contains the program code, its current activity, and the data associated with the program.

Here are some key aspects of processes:

- **Program in Execution:** A process represents a program that is currently running. It includes the program's code and its current activity.
- **Memory Space:** Each process has its own memory space, including the program's instructions and data. This memory space is protected from other processes to ensure data integrity and security.
- **Resources:** Processes have their own set of resources allocated by the operating system, including CPU time, memory, file handles, and I/O devices. The operating system manages these resources and ensures that processes do not interfere with each other.
- **Lifecycle:** Processes have a lifecycle, which includes creation, execution, suspension, and termination. The operating system is responsible for managing the transitions between these states.
- **Multitasking:** Modern operating systems support multitasking, allowing multiple processes to run concurrently. The operating system schedules processes and switches between them to give the illusion of simultaneous execution, even on systems with a single CPU core.

- **Process Control Block (PCB):** Each process is represented by a data structure called a Process Control Block (PCB). PCB contains information about the process, including its program counter (the address of the next instruction to be executed), CPU registers, and state (running, ready, blocked, etc.). PCBs are managed by the operating system.

- **Inter-Process Communication (IPC):** Processes may communicate with each other using Inter-Process Communication mechanisms provided by the operating system. IPC allows processes to exchange data and synchronize their activities.

Processes are fundamental to the functioning of modern operating systems and are crucial for enabling multitasking, parallelism, and the execution of multiple applications and tasks on a computer system. Operating systems use various algorithms and techniques to manage processes efficiently and ensure that they can coexist and run concurrently on the same hardware.

Thread:

In computer science, a **thread** is the smallest unit of execution within a process (smaller part of process or subset of process). A process, as mentioned earlier, is an instance of a program in execution, and each process can have one or multiple threads. Threads within a process share the same resources, such as memory space and file handles, but they run independently and allow for parallel execution of tasks.

- Single sequence stream within a process.
- An independent path of execution in a process.
- Light-weight process.
- Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
- E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spellchecking, formatting of text and saving the text are done concurrently by multiple threads.)

Here are some key points about threads:

- **Lightweight:** Threads are lightweight compared to processes. Creating a new thread requires fewer resources than creating a new process. Threads within the same process share the process's resources, which makes them more efficient in terms of memory and time overhead.

- **Concurrency:** Threads allow for concurrent execution of tasks. Multiple threads within a single process can run in parallel on multi-core processors. This concurrency enables applications to perform multiple tasks simultaneously, enhancing performance and responsiveness.

- **Shared Resources:** Threads within the same process share the process's memory space, which means they can read and modify the same data. While this shared memory simplifies communication between threads, it also requires careful synchronization to avoid data inconsistencies (race conditions).

- **Multithreading:** The practice of using multiple threads in a single program or process is known as multithreading. Multithreading allows applications to be more responsive to user input and perform tasks concurrently, leading to better utilization of CPU resources.

- **Thread States:** Threads have different states, such as running, ready, blocked, or terminated. The operating system or the program itself manages the transitions between these states based on events and scheduling decisions.

- **User-Level Threads and Kernel-Level Threads:** Threads can be managed by the operating system (kernel-level threads) or by the application itself without the operating system's direct involvement (user-level

threads). Kernel-level threads are generally more efficient because they can take advantage of multiple CPU cores, while user-level threads rely on a single kernel-level thread for execution.

- **Thread Safety:** Ensuring thread safety is essential when multiple threads access shared resources concurrently. Techniques like locks, semaphores, and murexes are used to synchronize access to shared data structures and prevent race conditions.

Multi-Tasking:

Multitasking is the concurrent execution of two or more tasks (also known as processes or threads) by a computer's central processing unit (CPU). It enables a computer to handle multiple tasks and applications simultaneously, giving users the perception that tasks are being executed in parallel, even though the CPU is rapidly switching between tasks. Multitasking can be achieved through both hardware and software means.

Multi-Threading:

Multithreading is a programming and execution technique that allows multiple threads within a single process to run concurrently. Each thread represents a separate flow of control within the same program. Multithreading enables programs to perform multiple tasks simultaneously, making better use of modern multicore processors and enhancing overall system performance and responsiveness.

Here are the key aspects of multithreading:

- **Threads:** A thread is the smallest unit of a CPU's execution. Within a process, multiple threads can be created, each executing a specific task independently. Threads within the same process share the same resources, such as memory space and file handles, but they run independently and can perform different tasks simultaneously.

- **Concurrency:** Multithreading allows different threads to execute concurrently, meaning they can run in parallel on multicore processors. This concurrency leads to faster task execution and improved system responsiveness, especially in applications that involve multiple I/O operations or require extensive computations.

- **Shared Resources:** Threads within the same process share the process's resources. While this shared memory simplifies communication between threads, it also requires careful synchronization to avoid data inconsistencies, known as race conditions. Proper synchronization mechanisms, such as locks and semaphores, are used to coordinate access to shared data.

- **Responsiveness:** Multithreading is often used in applications where responsiveness is critical. For example, in graphical user interfaces, one thread can handle user input and respond to user actions, while another thread performs background tasks like file downloads or data processing. This ensures that the user interface remains responsive even when the application is performing other tasks in the background.

- **Efficient Resource Utilization:** Multithreading makes efficient use of CPU resources, especially in systems with multiple processor cores. By dividing tasks into smaller threads, applications can utilize available CPU cores, leading to better system utilization and improved overall performance.

- **Types of Multithreading:**
  - **User-Level Threads:** Managed entirely by the application without kernel support. Operating systems treat each thread as if it were a separate process.

  - **Kernel-Level Threads:** Supported and managed by the operating system. The kernel schedules threads for execution, allowing true parallelism on multicore processors.

- **Complexity and Simplicity:** Multithreading can make software more complex due to the need for synchronization and the potential for race conditions. However, when implemented correctly, it simplifies the development of applications that require concurrent and parallel processing.

Multithreading is widely used in various applications, such as web servers, database management systems, multimedia applications, and video games, to enhance performance and responsiveness by leveraging the capabilities of modern hardware.

Difference between Multi-Tasking and Multi-Threading:

SL No.	Multi-Tasking	Multi-Threading
1.	In multitasking, users are allowed to perform many tasks by CPU.	While in multithreading, many threads are created from a process through which computer power is increased.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
4.	The multitasking component involves multiprocessing.	While the multithreading component does not involve multiprocessing.
5.	No. of CPU 1	No. of CPU >= 1. (Better to have more than 1)
6.	Multitasking is slow compared to multithreading.	While multithreading is faster.
7.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
8.	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently
9.	Multiple processes or tasks run simultaneously, sharing the same processor and resources	Multiple threads within a single process share the same memory space and resources
10.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process

Thread Scheduling:

Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.

Thread Context Switching:

Thread context switching refers to the process where a computer's CPU switches its focus from executing the instructions of one thread to executing the instructions of another thread. This switch allows multiple threads to share a single CPU core, enabling the appearance of concurrent execution. Context switching is managed by the operating system's scheduler, which determines the order in which threads are executed and ensures fair access to CPU time for each thread.

When a context switch occurs, the operating system saves the context, which includes the current state of the CPU registers and program counter, for the currently running thread. Then, it loads the saved context for the next thread to be executed. This process allows the CPU to seamlessly switch between threads, giving the illusion of parallel execution, even on systems with a single CPU core.

Here's an example to illustrate thread context switching:
Let's consider a simple multithreaded application with two threads: Thread A and Thread B. Both threads are executing different tasks within the same program. The operating system's scheduler decides to perform a context switch, transitioning from Thread A to Thread B.

1. **Thread A (Before Context Switch):**
   - Thread A is currently running and executing its instructions.
   - CPU registers hold the current state of Thread A, including the program counter (the memory address of the next instruction to be executed) and other relevant data.

2. **Context Switch:**
   - The **operating system scheduler** decides to switch from Thread A to Thread B.
   - Thread A's context is saved. This involves storing the current state of CPU registers and the program counter, allowing Thread A to resume its execution later.
   - The scheduler loads Thread B's context. This includes the saved state of CPU registers and the program counter for Thread B.

3. **Thread B (After Context Switch):**
   - Thread B's instructions can now be executed by the CPU.
   - Thread B continues its execution from where it left off during its previous time slice.

4. **Context Switch Back to Thread A:**
   - After some time, the scheduler decides to switch back to Thread A.
   - Thread B's context is saved, and Thread A's saved context is loaded.
   - Thread A continues its execution from where it left off.

This process of context switching allows Thread A and Thread B to appear as if they are executing simultaneously. However, it is important to note that on systems with a single CPU core, only one thread is truly executing at any given moment. The context switching mechanism gives the illusion of parallelism and allows systems to efficiently handle multiple tasks and maintain responsive user experiences.

Components of Operating System (OS):

- **Kernel Space**
- **User Space**

In computing, especially in operating systems, the **kernel** and **user space** are two distinct areas of memory and resources where different parts of a computer program, including the operating system itself, execute and operate. Understanding the division between these spaces is crucial for system stability, security, and resource management.

Kernel Space:
Kernel space is the protected memory space where the operating system's kernel resides. It is the core part of the operating system that has direct access to the computer's hardware and resources. The kernel performs essential tasks, such as managing memory, processes, device drivers, and system calls.

1. **Privileged Access:** Code running in the kernel space has full, unrestricted access to the computer's hardware and can execute privileged instructions. It can perform operations that normal user programs cannot, such as managing hardware interrupts, modifying memory page tables, and interacting directly with device drivers.

2. **Critical Operations:** The kernel space is responsible for managing system resources and ensuring the stability and security of the computer. It handles tasks like creating and terminating processes, managing memory allocation and deallocation, and controlling input/output operations.

3. **Isolation:** The kernel space is isolated from user space to prevent user programs from directly interfering with or damaging critical system resources. Any attempt by a user program to access kernel space directly results in a protection fault or segmentation fault, ensuring system stability.

User Space:

User space is the memory space where user applications and programs run. It is the area of memory where non-privileged user processes operate, executing tasks requested by users and interacting with the operating system through system calls.

Types of user Space:

- **GUI** (**Graphical User Interface**)
- **CLI** (**Command Line Interface**)

1. **Limited Access:** Code running in the user space does not have direct access to hardware resources or perform privileged operations. It relies on the operating system's kernel to perform tasks that require higher privileges, such as reading from or writing to hardware devices.

2. **User Programs:** User space contains all user-level applications, such as word processors, web browsers, games, and other software installed on the computer. These applications run in user space and interact with the kernel through defined interfaces, usually in the form of system calls.

3. **Resource Access:** User space programs access system resources (such as files, network connections, and input/output devices) through system calls provided by the operating system's kernel. These system calls act as gateways between user space and kernel space, allowing user programs to request services from the kernel.

In summary, the division between kernel space and user space ensures the security and stability of a computer system. The kernel space provides a controlled environment for managing hardware and system resources, while user space allows applications to run without interfering with critical system operations. Communication between user space and kernel space occurs through well-defined interfaces, maintaining the integrity of the system as a whole.

Shell: A shell, also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed.

So I can say in that way like, **User Space** is a memory space where users can communicate with programs, and **Kernel Space** is a space where it convert the program to process and allocate the memory to the process.

Function of Kernel:

1. **Process management:**
    a. Scheduling processes and threads on the CPUs.
    b. Creating & deleting both user and system process.
    c. Suspending and resuming processes
    d. Providing mechanisms for process synchronization or process communication.

2. **Memory management:**
    a. Allocating and deallocating memory space as per need.
    b. Keeping track of which part of memory are currently being used and by which process.

3. **File management:**
    a. Creating and deleting files.
    b. Creating and deleting directories to organize files.
    c. Mapping files into secondary storage.
    d. Backup support onto a stable storage media.

4. **I/O management:** to manage and control I/O operations and I/O devices
    a. Buffering (data copy between two devices), caching and spooling.

        i. **Spooling:** store the information into a particular storage. For example, suppose there are 10 users, each user provide 10 docs to printer for printing, and then the printer store this information in a particular storage for process the printing operation continuously.

            1. Within differing speed two jobs.
            2. E.g. Print spooling and mail spooling.

        ii. **Buffering:**
            1. Within one job.
            2. E.g. YouTube video buffering

        iii. **Caching:**
            1. Memory caching, Web caching etc.

Types of Kernels:

1. **Monolithic kernel**
2. **Micro Kernel**
3. **Hybrid kernel**
4. **Nano / Exo kernel, etc.**

Monolithic Kernel:

A **monolithic kernel** is an operating system architecture where all the operating system services run in the same address space, i.e., they execute in the kernel mode. In other words, all the essential operating system components, including device drivers, file systems, and system calls, are part of a single, unified kernel program. Monolithic kernels contrast with microkernels, where these components run as separate processes in user space, communicating through inter-process communication mechanisms.

- All functions are in kernel itself.
- Bulky in size.
- Memory required to run is high.
- Less reliable, one module crashes -> whole kernel is down.
- High performance as communication is fast. (Less user mode, kernel mode overheads)

Examples of Operating Systems with Monolithic Kernels:

1. **Linux:** The Linux kernel is a prominent example of a monolithic kernel. It provides core operating system services and interacts directly with hardware components. Additional functionalities, such as device drivers and file systems, are loaded into the kernel space as kernel modules when needed.

2. **UNIX:** Traditional UNIX operating systems, including versions like SunOS, AIX, and HP-UX, often used monolithic kernel architectures.

Advantages of Monolithic Kernels:

- **Performance:** Monolithic kernels can be more efficient than microkernels because they involve fewer context switches and inter-process communication overhead. In a monolithic kernel, components can communicate directly, which can lead to faster system performance.

- **Simplicity:** Monolithic kernels are conceptually simpler and easier to design, implement, and debug compared to microkernels. All components operate in the same address space, simplifying data sharing and function calls.

- **Low Latency:** Due to direct communication between components, monolithic kernels can provide low-latency responses, making them suitable for real-time applications.

- **Unified Address Space:** Since all components share the same address space, data sharing and communication between different parts of the kernel are more straightforward.

Disadvantages of Monolithic Kernels:

- **Lack of Modularity:** Monolithic kernels can become large and complex as all functionalities are bundled together. Adding new features or modifying existing ones can be challenging without affecting the entire system.

- **Limited Fault Isolation:** If a component within a monolithic kernel crashes, it can potentially bring down the entire system. There is less isolation between components compared to microkernel architectures.

- **Difficulty in Maintenance:** Large monolithic kernels can be challenging to maintain and extend, especially as the codebase grows. Modifying one component might inadvertently affect others, leading to unexpected behaviour.

- **Scalability Issues:** As the size of the kernel increases, it might consume more memory, which can be a concern for systems with limited resources or embedded systems.

In summary, monolithic kernels, offer high performance and low overhead but can be challenging to maintain and modify due to their lack of modularity. Choosing between monolithic and microkernel architectures often depends on specific use cases and trade-offs between performance, flexibility, and ease of maintenance.

Micro Kernel:

A **microkernel** is an operating system architecture in which the core functionality, such as process scheduling, file system management, and inter-process communication, is implemented as a minimal, highly protected kernel. Additional services, which in monolithic kernels are typically part of the kernel, are moved out of the kernel space and run as separate processes in user space. These user-level components communicate with each other via message passing, inter-process communication mechanisms.

- Only major functions are in kernel.
    - Memory mgmt.
    - Process mgmt.

- File mgmt. and IO mgmt. are in User-space.
- Smaller in size.
- More Reliable
- More stable
- Performance is slow.
- Overhead switching b/w user mode and kernel mode.

Examples of Operating Systems with Microkernels:

1. **QNX:** QNX is a real-time, Unix-like microkernel operating system used in embedded systems, automotive systems, and various other applications. It has a microkernel architecture where essential services run in user space.

2. **Minix:** Minix is a Unix-like operating system used for educational purposes and research. It was designed as a microkernel architecture and serves as an example of such a system.

Advantages of Microkernels:

- **Modularity:** Microkernels are highly modular. Services such as device drivers, file systems, and networking protocols run as separate processes, allowing for easy extension and modification without affecting the core kernel.

- **Reliability and Fault Isolation:** Since most components run in user space, if a user-level component crashes, it does not affect other parts of the system. This design enhances system reliability and fault tolerance.

- **Security:** Microkernels enforce strong process isolation. Critical parts of the operating system are protected, reducing the attack surface and making it harder for malicious code to compromise the system.

- **Flexibility:** Microkernels allow for a high degree of customization. Users can tailor the operating system by selecting and running only the necessary components, conserving system resources.

Disadvantages of Microkernels:

- **Performance Overhead:** The use of message passing between user-level components can introduce performance overhead compared to direct function calls in monolithic kernels. Context switches and inter-process communication can be more time-consuming.

- **Complexity:** While the microkernel itself is minimal, the overall system can be more complex due to the interactions between user-level components. Managing these interactions and ensuring proper communication can be challenging.

- **Resource Utilization:** Microkernels often require more memory and computational resources because of the need for inter-process communication and context switching. In resource-constrained environments, this can be a significant drawback.

- **Limited Hardware Support:** Some hardware architectures may not be well-supported by microkernel-based operating systems, limiting their applicability in certain environments.

In summary, microkernels offer modularity, reliability, and security advantages but may suffer from performance overhead and complexity. The choice between microkernels and monolithic kernels often depends on the specific requirements of the system being developed, considering factors such as resource availability, security needs, and the level of customization desired.

Hybrid Kernel:

A **hybrid kernel** is an operating system architecture that combines elements of both **monolithic kernels** and **microkernels**. In a hybrid kernel design, certain essential components (such as device drivers and file systems) traditionally found in the kernel space of monolithic kernels are moved to the user space, following the microkernel approach. However, unlike pure microkernels, some critical services remain in the kernel space for performance reasons. This combination of approaches allows for flexibility, modularity, and performance optimization.

Example of an Operating System with a Hybrid Kernel:

1. **Windows NT/2000/XP/7/8/10:** Microsoft Windows operating systems from the NT line (including Windows 2000, XP, 7, 8, and 10) use a hybrid kernel architecture. While the Windows kernel includes components running in kernel mode, many device drivers and system services operate in user mode.

2. **Mac OS**

Advantages of Hybrid Kernels:

- **Modularity:** Hybrid kernels offer a high degree of modularity. Many components can run in user space, allowing for easier customization, extension, and maintenance without affecting the kernel's core functionality.

- **Performance:** By retaining essential components in kernel mode, certain critical operations can be performed more efficiently, leading to better overall system performance compared to pure microkernels. Direct access to hardware and shared memory space can enhance performance for essential tasks.

- **Reliability:** By isolating drivers and some services in user space, the reliability and stability of the core kernel are maintained. If a user-level component fails, it does not necessarily crash the entire system.

- **Compatibility:** Hybrid kernels often provide compatibility with existing monolithic kernel designs and device drivers, allowing for a smoother transition from older operating systems.

Disadvantages of Hybrid Kernels:

- **Complexity:** Hybrid kernels can be more complex to design, implement, and maintain due to the combination of different architectures. The interaction between user-level components and kernel space requires careful management.

- **Resource Utilization:** While more lightweight than monolithic kernels, hybrid kernels can still consume significant memory and computational resources due to the necessity of supporting both user-level and kernel-level components.

- **Potential for Bugs:** The complexity of hybrid kernels can lead to increased possibilities of bugs and security vulnerabilities, requiring rigorous testing and validation processes.

- **Learning Curve:** Developing and debugging applications for hybrid kernels might require a deep understanding of both user-level and kernel-level interactions, making it potentially more challenging for developers.

In summary, hybrid kernels provide a balance between modularity, performance, and compatibility. They offer advantages in terms of customization and reliability while maintaining essential performance benefits from kernel-mode operations. However, the complexity of managing both user-level and kernel-level components can pose challenges in development and maintenance. The choice of kernel architecture depends on the specific requirements and trade-offs deemed essential for the target operating system.

How user-mode and kernel-mode are communicate each other:

Inter process communication (IPC).

1. Two processes executing independently, having independent memory space (Memory protection), But some may need to communicate to work.
2. Done by shared memory and message passing.

Communication between user mode and kernel mode in an operating system is essential for various tasks, such as system calls, hardware interactions, and other privileged operations. Operating systems ensure that user mode processes cannot directly access kernel mode memory or execute sensitive instructions. Instead, they provide controlled interfaces for communication.

Here are the main mechanisms for communication between user mode and kernel mode:

1. **System Calls:**
   System calls are functions provided by the operating system that allow user mode processes to request services from the kernel. Examples include file operations, process control, and network communication. To make a system call, a user mode process invokes a specific software interrupt or instruction (like `int 0x80` on x86 architectures) that transfers control to a predefined location in the kernel. The kernel then executes the requested operation and returns the result to the user mode process.

2. **Interrupts and Exceptions:**
   Hardware interrupts and software exceptions (like divide-by-zero or page faults) can trigger the operating system to switch from user mode to kernel mode. When an interrupt or exception occurs, the CPU transfers control to a specific interrupt handler or exception handler routine in the kernel. These handlers can perform necessary operations and communicate back to user mode, either directly or indirectly through the interrupted process's context.

3. **I/O Operations:**
   User mode processes often require input and output operations involving devices like disks, keyboards, and network interfaces. Device drivers in the kernel mode manage these devices. User mode processes can communicate with devices through system calls, and the kernel mode device drivers handle the actual device interactions on behalf of user mode processes.

4. **Shared Memory:**
   In some cases, user mode processes and kernel mode components can communicate through shared memory regions. However, this method requires careful synchronization to prevent data corruption and race conditions. Shared memory is typically used in performance-critical applications where direct memory access can improve communication efficiency.

5. **Controlled Interfaces:**
   Operating systems provide controlled interfaces for user mode applications to interact with kernel mode components. These interfaces include system calls, IOCTL (input/output control) requests for device drivers, and other well-defined communication channels. Direct access to kernel memory and resources is prohibited to maintain system stability and security.

6. **Call-back Functions:**
   Some operating systems allow user mode processes to register callback functions with the kernel. These callbacks can be invoked by the kernel in specific situations, enabling asynchronous communication between user mode and kernel mode.

7. **Signals and Events:**
   Operating systems often provide mechanisms for user mode processes to receive signals or events from the kernel, indicating events such as process termination or asynchronous I/O completion. User mode processes can handle these signals to respond to specific events initiated by kernel mode components.

In summary, communication between user mode and kernel mode occurs through controlled and well-defined interfaces, including system calls, interrupts, I/O operations, shared memory, and event mechanisms. These mechanisms ensure that user mode processes can request services from the kernel while maintaining system stability, security, and isolation between user mode and kernel mode components.

System Call:

A **system call** is a mechanism used by programs to request services from the operating system's kernel. It acts as a bridge between user-level applications and the low-level hardware of a computer system. User programs, which run in user mode, do not have direct access to hardware resources or the privileged instructions required to perform tasks such as I/O operations, managing files, or creating processes. System calls provide a secure way for user programs to interact with the kernel and request these privileged operations.

A system call is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform. User programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs.

E.g.: Mkdir laks

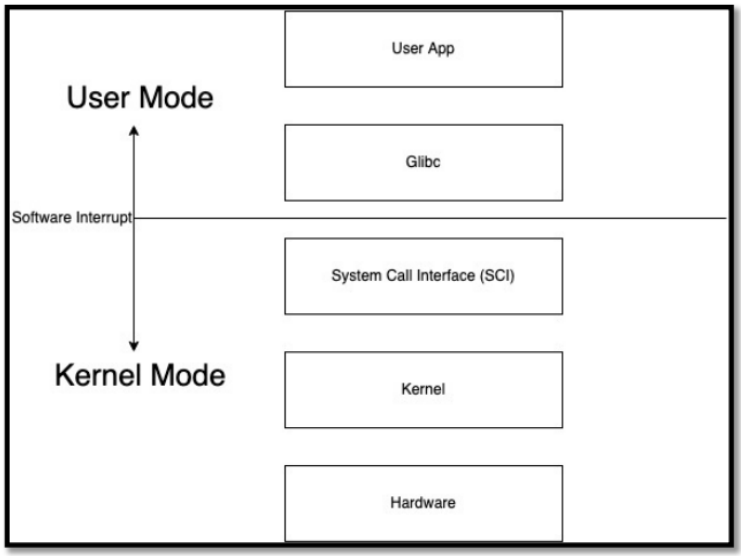
- Mkdir indirectly calls kernel and asked the file mgmt. module to create a new directory.
- Mkdir is just a wrapper of actual system calls.
- Mkdir interacts with kernel using system calls.

E.g.: Creating Process

- User executes a process. (User space)
- Gets system call. (US)
- Exec system call to create a process. (KS)
- Return to US.

How do apps interact with Kernel? -> using system calls.

System Calls are the only way through which a process can go into **kernel mode from user mode.**



Types of System Call:

1. **Process Control**
   a. end, abort ;
   b. load, execute;
   c. create process, terminate process;
   d. get process attributes, set process attributes;

e. wait for time;
f. wait event, signal event;
g. Allocate and free memory.

2. **File Management**
   a. create file, delete file;
   b. open, close;
   c. read, write, reposition;
   d. Get file attributes, set file attributes.

3. **Device Management**
   a. request device, release device;
   b. read, write, reposition;
   c. get device attributes, set device attributes;
   d. Logically attach or detach devices.

4. **Information Maintenance**
   a. get time or date, set time or date;
   b. get system data, set system data;
   c. get process, file, or device attributes;
   d. Set process, file, or device attributes.

5. **Communication Management**
   a. create, delete communication connection;
   b. send, receive messages;
   c. transfer status information;
   d. Attach or detach remote devices.

Examples of Windows and UNIX System Call:

Category	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle() SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	open () read () write () close () chmod() umask(chown()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Management	GetCurrentProcessID() SetTimer() Sleep()	getpid () alarm () sleep ()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe () shmget () mmap()

- **fork():** create child process (in computer no new processes are created, only child processes are created).
- **pipe():** create a tunnel between the two processes for communication.

What happen when you turn on your computer?

1. PC On
2. CPU initializes itself and looks for a firmware program (BIOS) stored in BIOS Chip (Basic input-output system chip is a ROM chip found on mother board that allows to access & setup computer system at most basic level.)
    a. In modern PCs, CPU loads UEFI (Unified extensible firmware interface)

3. CPU runs the BIOS which tests and initializes system hardware. Bios loads configuration settings. If something is not appropriate (like missing RAM) error is thrown and boot process is stopped. This is called POST (Power on self-test) process.
    a. (UEFI can do a lot more than just initialize hardware; it's really a tiny operating system. For example, Intel CPUs have the Intel Management Engine. This provides a variety of features, including powering Intel's Active Management Technology, which allows for remote management of business PCs.)

4. BIOS will handoff responsibility for booting your PC to your OS's bootloader.
    a. BIOS looked at the MBR (master boot record), a special boot sector at the beginning of a disk. The MBR contains code that loads the rest of the operating system, known as a "bootloader." The BIOS executes the bootloader, which takes it from there and begins booting the actual operating system—Windows or Linux.

    For example.
    In other words, the BIOS or UEFI examines a storage device on your system to look for a small program, either in the MBR or on an EFI system partition, and runs it.

5. The bootloader is a small program that has the large task of booting the rest of the operating system (Boots Kernel then, User Space). Windows uses a bootloader named Windows Boot Manager (Bootmgr.exe), most Linux systems use GRUB, and Macs use something called boot.ef.

PART 03: 32-bit OS, 64-bit OS, Storage, Architecture of Process, PCB, Process State, GHz etc.

32-bit Operating System:

A **32-bit operating system** is a computer operating system that is designed to work on processors that are 32 bits wide. The "32-bit" refers to the way a computer's processor handles information. In a 32-bit system, the processor can handle data and memory addresses that are 32 bits long.

Here are the key characteristics of 32-bit operating systems:

- **Memory Addressing:**
    - **Maximum Addressable Memory:** A 32-bit system can theoretically address up to 4 GB (2^32 bytes) of memory. However, due to hardware and system requirements, a typical 32-bit operating system may effectively use around 3 GB to 3.5 GB of RAM.

    - **Addressing Limitation:** The 32-bit architecture imposes a limitation on the total memory that can be used by the system. If a computer has more than 4 GB of RAM, a 32-bit operating system will not be able to utilize the additional memory.

- **Data Handling:**

- **Data Representation:** The CPU processes data in chunks of 32 bits at a time. This means it can perform operations on 32-bit integers, memory addresses, and data units in a single clock cycle.

- **Software Compatibility:**
  - **32-bit Applications:** 32-bit operating systems can run both 32-bit and 16-bit applications. Most software developed for personal computers during the late 20th century and early 2000s was designed for 32-bit systems.

- **Performance:**
  - **Limited Performance:** 32-bit systems have limitations regarding the amount of memory they can use, which can affect the performance of memory-intensive applications. For tasks that require a large amount of memory, 64-bit systems are more suitable.

- **Hardware Compatibility:**
  - **Peripheral Devices:** 32-bit operating systems are compatible with a wide range of peripheral devices, making them suitable for older hardware components.

- **Legacy Support:**
  - **Legacy Software:** Many older systems and software applications are designed for 32-bit architectures. 32-bit operating systems are necessary for running legacy software that has not been updated to support 64-bit systems.

It's important to note that with the advancement of technology, many modern computers are now equipped with 64-bit processors, allowing for the use of 64-bit operating systems. 64-bit systems can handle larger amounts of memory and offer better performance for memory-intensive tasks. However, 32-bit operating systems are still encountered in specific scenarios where legacy hardware or software constraints exist.

64-bit Operating System:

A **64-bit operating system** is an operating system that is designed to work on processors that are 64 bits wide. The "**64-bit**" refers to the way a computer's processor handles information. In a 64-bit system, the processor can handle data and memory addresses that are 64 bits long.

Here are the key characteristics of 64-bit operating systems:

- **Memory Addressing:**
  - **Maximum Addressable Memory:** A 64-bit system can theoretically address up to 18.4 million TB (2^64 bytes) of memory. This vast address space allows 64-bit operating systems to handle much larger amounts of RAM compared to 32-bit systems. In practical terms, modern 64-bit systems can support several terabytes of RAM.

  - **Breaking the 4 GB Barrier:** One of the significant advantages of 64-bit systems is their ability to utilize more than 4 GB of RAM, which is the addressing limitation of 32-bit systems. This enables smoother multitasking and the handling of memory-intensive applications.

- **Data Handling:**
  - **Data Representation:** The CPU processes data in chunks of 64 bits at a time. This larger data width allows for more efficient processing of large data sets and enhances performance for memory-intensive tasks.

- **Software Compatibility:**
  - **32-bit and 64-bit Applications:** 64-bit operating systems can run both 32-bit and 64-bit applications. Most modern software and applications are developed to be compatible with 64-bit

systems. However, 32-bit software can still run on 64-bit systems with the help of compatibility layers.

- **Performance:**
  - **Efficiency:** 64-bit systems can handle larger chunks of data, leading to improved computational efficiency and faster processing of complex tasks, especially those involving large datasets.

- **Security and Robustness:**
  - **Enhanced Security Features:** 64-bit architectures often come with enhanced security features such as Data Execution Prevention (DEP) and Kernel Patch Protection (Patch Guard), which help protect the system against various types of attacks.

- **Multitasking and Multithreading:**
  - **Efficient Multitasking:** The increased memory capacity of 64-bit systems allows for more efficient multitasking, enabling users to run multiple memory-intensive applications simultaneously without experiencing performance bottlenecks.

  - **Multithreading:** 64-bit systems can efficiently handle multithreaded applications, allowing for parallel execution of tasks and improved overall system responsiveness.

- **Graphics and Multimedia:**
  - **Graphics Processing:** 64-bit systems are beneficial for graphics-intensive applications, including video editing, computer-aided design (CAD), and gaming, as they can efficiently process large graphical datasets.

Modern computers, especially those designed for power users, professionals, and server environments, commonly use 64-bit operating systems due to their ability to handle large amounts of memory and process data more efficiently.

Difference between 32-bit Operating System and 64-bit Operating System:

Feature	32-bit OS	64-bit OS
Memory	Maximum of 4 GB RAM	Maximum of several terabytes of RAM
Processor	Can run on both 32-bit and 64-bit processors	Requires a 64-bit processor
Performance	Limited by the maximum amount of RAM it can access	Can take advantage of more memory, enabling faster performance
Compatibility	Can run 32-bit and 16-bit applications	Can run 32-bit and 64-bit applications
Address Space	Uses 32-bit address space	Uses 64-bit address space
Hardware	May not support newer hardware	Supports newer hardware with 64-bit

support		drivers
Security	Limited security features	More advanced security features, such as hardware-level protection
Application support	Limited support for new software	Supports newer software designed for 64-bit architecture
Price	Less expensive than 64-bit OS	More expensive than 32-bit OS
Multitasking	Can handle multiple tasks but with limited efficiency	Can handle multiple tasks more efficiently
Gaming	Can run high graphical games, but may not be as efficient as with 64-bit OS	Can run high graphical games and handle complex software more efficiently
Virtualization	Limited support for virtualization	Better support for virtualization

Notes:

1. A 32-bit OS has 32-bit registers, and it can access 2^32 unique memory addresses. i.e., 4GB of physical memory.
2. A 64-bit OS has 64-bit registers, and it can access 2^64 unique memory addresses. i.e., 17,179,869,184 GB of physical memory.
3. 32-bit CPU architecture can process 32 bits of data & information.
4. 64-bit CPU architecture can process 64 bits of data & information.
5. **Advantages of 64-bit over the 32-bit operating system:**
    a. **Addressable Memory**: 32-bit CPU -> 2^32 memory addresses, 64-bit CPU -> 2^64 memory addresses.

    b. **Resource usage**: Installing more RAM on a system with a 32-bit OS doesn't impact performance. However, upgrade that system with excess RAM to the 64-bit version of Windows, and you'll notice a difference.

    c. **Performance**: All calculations take place in the registers. When you're performing math in your code, operands are loaded from memory into registers. So, having larger registers allow you to perform larger calculations at the same time. 32-bit processor can execute 4 bytes of data in 1 instruction cycle while 64-bit means that processor can execute 8 bytes of data in 1 instruction cycle. (In 1 sec, there could be thousands to billons of instruction cycles depending upon a processor design)

    d. **Compatibility**: 64-bit CPU can run both 32-bit and 64-bit OS. While 32-bit CPU can only run 32-bit OS.

    e. **Better Graphics performance**: 8-bytes graphics calculations make graphics-intensive apps run faster.

Storage Devices:

1. **Register:**
   - **Type:** Internal storage within the CPU.
   - **Importance:** Registers are the smallest and fastest type of memory in a computer. They store data in CPU that is currently processing. Registers facilitate quick access to data, allowing the CPU to perform arithmetic and logic operations efficiently. Registers are essential for the CPU to execute instructions and perform computations.

2. **Cache Memory:**
   - **Type:** Located between the main memory and the CPU.
   - **Importance:** Cache memory stores frequently accessed data and instructions. It acts as a high-speed buffer between the main memory and the CPU, reducing the time the CPU spends waiting for data. Faster access to frequently used data significantly improves the overall speed and performance of the computer.

3. **Main Memory (RAM - Random Access Memory):**
   - **Type:** Volatile memory (loses data when power is off).
   - **Importance:** RAM is the primary memory used by the computer to temporarily store data that the CPU and running programs need to access quickly. It provides fast read and write access and is crucial for running applications. The more RAM a system has, the more data it can store temporarily, allowing for smoother multitasking and efficient program execution.

4. **Electronic Disk (Solid State Drive - SSD):**
   - **Type:** Non-volatile memory (retains data even when power is off).
   - **Importance:** SSDs use NAND flash memory to store data electronically. They provide significantly faster read and write speeds compared to traditional hard disk drives (HDDs) because they lack moving parts. SSDs enhance overall system responsiveness, reduce boot times, and improve the loading times of applications and files.

5. **Magnetic Disk (Hard Disk Drive - HDD):**
   - **Type:** Non-volatile memory (retains data even when power is off).
   - **Importance:** HDDs store data using magnetic storage. They offer large storage capacities at a lower cost per gigabyte compared to SSDs. HDDs are commonly used for mass storage of files, documents, videos, and applications in computers. While they are slower than SSDs due to mechanical read/write heads and spinning disks, they remain popular for bulk storage due to their cost-effectiveness.

6. **Optical Disk (CDs, DVDs, Blu-ray Discs):**
   - **Type:** Non-volatile memory (retains data even when power is off).
   - **Importance:** Optical disks are used for long-term storage and distribution of large amounts of data. They are commonly used for software distribution, music, movies, and archival purposes. Different types of optical discs, such as DVDs and Blu-ray discs, offer varying levels of storage capacity, making them suitable for different applications.

7. **Magnetic Tape:**
   - **Type:** Non-volatile memory (retains data even when power is off).
   - **Importance:** Magnetic tape is a sequential access storage medium commonly used for large-scale backups, archival storage, and data recovery. It offers high capacity and is often used in enterprise environments for cost-effective, long-term data storage.

Each storage device plays a specific role in the computer's memory hierarchy, providing different trade-offs between speed, capacity, and cost. Computer systems often utilize a combination of these storage devices to balance performance and storage needs based on specific use cases.

Comparison:

- **Cost:**
  - Primary storages are costly.
  - Registers are most expensive due to expensive semiconductors & labour.
  - Secondary storages are cheaper than primary.

- **Access Speed:**
  - Primary has higher access speed than secondary memory.
  - Registers has highest access speed, then comes cache, then main memory.

- **Storage size:**
  - Secondary has more space.

- **Volatility:**
  - Primary memory is volatile.
  - Secondary is non-volatile.

How OS create Process:

Creating a process in an operating system involves several steps to set up the necessary data structures and resources that allow the program to run. **Here is a general overview of how an operating system creates a process:**

1. **Process Creation System Call:**
   - **Trigger:** The process creation usually starts with a system call (like `fork()` in Unix-based systems) initiated by a running process or by the operating system itself.
   - **Request Processing:** The operating system's kernel intercepts the system call request and begins the process creation procedure.
2. **Allocate Process Control Block (PCB):**
   - **Definition:** A Process Control Block (PCB) is a data structure that contains information about the process, such as its program counter, stack pointer, memory allocation, and state.
   - **Allocation:** The operating system allocates a new PCB to store information about the new process. This includes allocating memory for the process's stack, heap, and other required data structures.
3. **Memory Allocation:**
   - **Virtual Address Space:** The operating system allocates a virtual address space for the process. This space includes the code segment, data segment, heap (for dynamically allocated memory), and stack.
   - **Mapping Physical Memory:** The virtual addresses are mapped to physical memory addresses. If demand paging is used, actual physical memory pages might not be allocated until they are accessed.
4. **Loading the Program:**

- **Load Program into Memory:** The executable file associated with the process is loaded into the allocated memory space. This involves reading the program file from storage (like a hard disk) into the allocated memory.

- **Setting Program Counter:** The program counter (PC) is set to the entry point of the program, indicating the first instruction to be executed.

5. **Process Initialization:**
   - **Initialization:** Initial values for program variables and data structures are set. This step ensures that the process starts in a known state.

   - **File Descriptors:** If the process needs access to files, standard input, output, and error file descriptors are initialized.

6. **Parent-Child Relationship (Optional, for fork-like Systems Calls):**
   - **Duplicate Process (for fork()):** In systems with `fork()`-like system calls, the new process is a duplicate of the calling process (the parent). The child process receives a copy of the parent's memory and resources but operates independently thereafter.

7. **Process Scheduling:**
   - **Ready State:** The process is marked as ready to run and is added to the ready queue in the scheduling algorithm of the operating system.

   - **Scheduler:** The operating system scheduler determines when and which process will run next based on scheduling policies (like round-robin, priority scheduling, etc.).

8. **Process Execution:**
   - **Execution:** The newly created process is now ready to execute. When the scheduler selects this process for execution, it starts running the program instructions from the designated entry point.

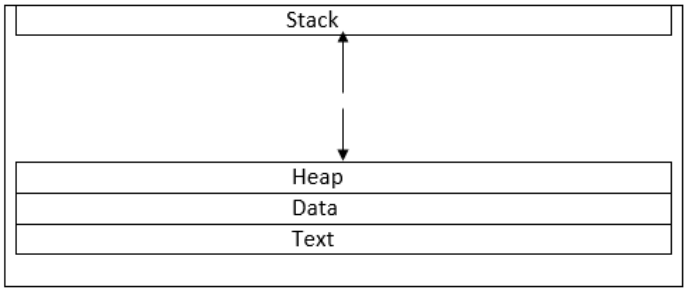
The specifics of process creation can vary slightly between different operating systems and programming languages. However, the fundamental steps mentioned above are common to most modern operating systems. The operating system's kernel manages these steps to create and execute processes effectively.

Architecture of Process:

A process is actively running software or a computer code. Any procedure must be carried out in a precise order. An entity that helps in describing the fundamental work unit that must be implemented in any system is referred to as a process.

In other words, we create computer programs as text files that, when executed, create processes that carry out all of the tasks listed in the program.

When a program is loaded into memory, it may be divided into the four components stack, heap, text, and data to form a process. The simplified depiction of a process in the main memory is shown in the diagram below.



Process Diagram

- **Stack:**
  The process stack stores temporary information such as method or function arguments, the return address, and local variables.

- **Heap:**
  This is the memory where a process is dynamically allotted while it is running.
- **Text:**
  This consists of the information stored in the processor's registers as well as the most recent activity indicated by the program counter's value.

- **Data:**
  In this section, both global and static variables are discussed.

Attribute of Process:

a. Feature that allows identifying a process uniquely.
b. Process table.
  i. All processes are being tracked by OS using a table like data structure.
  ii. Each entry in that table is process control block (PCB).
c. PCB: Stores info/attributes of a process.
  i. Data Structure used for each process that stores information of a process such as process id, program counter, process state, priority etc.

PCB (Process Control Block) of Process:

An Operating System helps in process creation, scheduling, and termination with the help of Process Control Block. The Process Control Block (PCB), which is part of the Operating System, aids in managing how processes operate. Every OS process has a Process Control Block related to it. By keeping data on different things including their state, I/O status, and CPU Scheduling, a PCB maintains track of processes.
Now, let us understand the Process Control Block with the help of the components present in the Process Control Block.

A Process Control Block (PCB) is a data structure that contains information about the process, such as its program counter, stack pointer, memory allocation, and state.

A Process Control Block consists of:

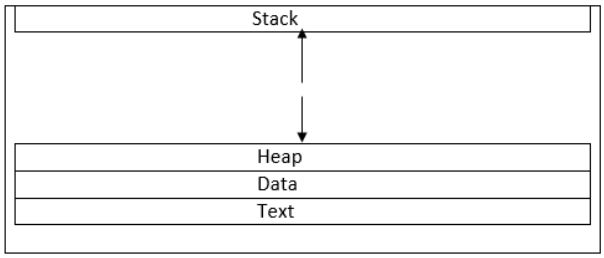
1. Process ID
2. Process State
3. Program Counter
4. CPU Registers

5. CPU Scheduling Information
6. Accounting and Business Information
7. Memory Management Information
8. Input Output Status Information

Process ID:

It is an Identification mark which is present for the Process. This is very useful for finding the process. It is also very useful for identifying the process also.

Process State:



Process Diagram

Now, let us know about each and every process states in detail. Let me explain about each and every state

i. **New State:**
A Program which is going to be taken up by the Operating System directly into the Main Memory is known as a New Process State.

ii. **Ready State:**
The ready state, when a process waits for the CPU to be assigned, is the first state it enters after being formed. The operating system pulls new processes from secondary memory and places them all in main memory.

The term "ready state processes" refers to processes that are in the main memory and are prepared for execution. Numerous processes could be active at the moment.

iii. **Running State:**
The Operating System will select one of the processes from the ready state based on the scheduling mechanism. As a result, if our system only has one CPU, there will only ever be one process operating at any given moment. We can execute n processes concurrently in the system if there are n processors.

iv. **Waiting or Blocking State:**
Depending on the scheduling mechanism or the inherent behaviour of the process, a process can go from the Running state to the Block or Wait states.

The OS switches a process to the block or wait state and allots the CPU to the other processes while it waits for a specific resource to be allocated or for user input.

v. **Terminated State:**
A process enters the termination state once it has completed its execution. The operating system will end the process and erase the whole context of the process (Process Control Block).

Program Counter:
The address of the following instruction to be executed from memory is stored in a CPU register called a program counter (PC) in the computer processor. It is a digital counter required for both task execution speed and for monitoring the present stage of execution.

An instruction counter, instruction pointer, instruction addresses register, or sequence control register are other names for a program counter.

CPU Register:
When the process is in a running state, here is where the contents of the processor registers are kept. Accumulators, index and general-purpose registers, instruction registers, and condition code registers are the many categories of CPU registers.

CPU Scheduling Information:
It is necessary to arrange a procedure for execution. This schedule determines when it transitions from ready to running. Process priority, scheduling queue pointers (to indicate the order of execution), and several other scheduling parameters are all included in CPU scheduling information.

Accounting and Business Information:
The State of Business Addressing and Information includes information such as CPU use, the amount of time a process uses in real time, the number of jobs or processes, etc.

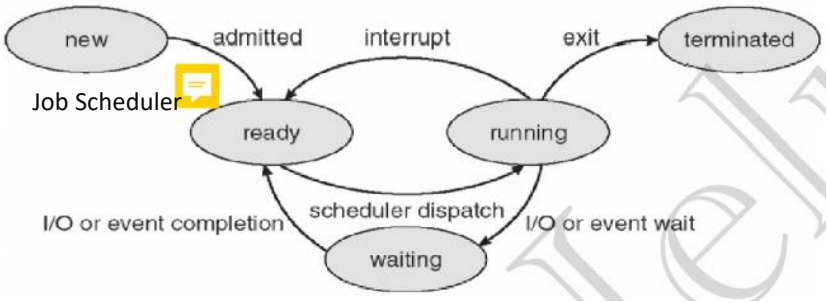
Memory Management Information:
The Memory Management Information section contains information on the page, segment tables, and the value of the base and limit registers. It relies on the operating system's memory system.

Input Output Status Information:
This Input Output Status Information section consists of Input and Output related information which includes about the process statuses, etc.

Registers in the PCB, it is a data structure. When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.

Process State | Process Queue:

1. **Process State:** As process executes, it changes state. Each process may be in one of the following states.
    a. **New:** OS is about to pick the program & convert it into process. OR the process is being created.
    b. **Run:** Instructions are being executed; CPU is allocated.
    c. **Waiting:** Waiting for IO.
    d. **Ready:** The process is in memory, waiting to be assigned to a processor.
    e. **Terminated:** The process has finished execution. PCB entry removed from process table.



2. **Process Queue:**
    a. **Job Queue:**
        i. Processes in new state.
        ii. Present in secondary memory.
        iii. Job Scheduler (Long-term scheduler (LTS)) picks process from the pool and loads them into memory for execution.

    b. **Ready Queue:**
        i. Processes in Ready state.
        ii. Present in main memory.
        iii. CPU Scheduler (Short-term scheduler) picks process from ready queue and dispatch it to CPU. (CPU Scheduler called **Short Term Scheduler** because, it works in high frequency, there is no delay to handle the processes)

    c. **Waiting Queue:**
        i. Process in Wait State

3. **Degree of multi-programming:**
    - In one at a time, how many processes are present in **"ready"** state or **"ready queue".**
    - **It is handle by LTS (Long Term Scheduler)**

4. **Dispatcher:** The module of OS that gives control of CPU to a process selected by **STS**.

Gigahertz (GHz):

GHz stands for gigahertz, which is a unit of frequency representing one billion cycles per second. In the context of CPUs (Central Processing Units), GHz refers to the clock speed or the number of cycles a CPU can execute in one second. A CPU with a higher clock speed (measured in GHz) can perform more instructions per second and is generally faster than a CPU with a lower clock speed.

For example:

- **1 GHz CPU:** This means the CPU can execute 1 billion cycles per second.
- **4 GHz CPU:** This means the CPU can execute 4 billion cycles per second.

A higher clock speed generally indicates a faster CPU, but it's essential to note that the clock speed is not the only factor determining a processor's performance. Modern CPUs are designed with multiple cores, which can handle parallel tasks, and they have various optimizations, cache sizes, and instruction sets that influence their overall performance.

When comparing CPUs, it's crucial to consider factors such as the number of cores, the architecture (like Intel Core, AMD Ryzen, etc.), and the specific tasks the CPU will be used for. Different applications and tasks may benefit more from certain CPU features than others. Therefore, while clock speed (GHz) provides a basic indication of a CPU's speed, it's not the sole factor to consider when evaluating a processor's performance.

PART 04: All about Process Scheduling Algorithms.

Swapping:

Swapping is a memory management scheme in which **any process can be temporarily swapped from main memory to secondary memory** so that the main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

The purpose of the swapping in operating system is to access the data present in the hard disk and bring it to RAM so that the application programs can use it. The thing to remember is that swapping is used only when data is not present in RAM.

Although the process of swapping affects the performance of the system, it helps to run larger and more than one process. This is the reason why swapping is also referred to as memory compaction.

The concept of swapping has divided into two more concepts: Swap-in and Swap-out.

- Swap-out is a method of removing a process from RAM and adding it to the hard disk.
- Swap-in is a method of removing a program from a hard disk and putting it back into the main memory or RAM.

Example: Suppose the user process's size is 2048KB and is a standard hard disk where swapping has a data transfer rate of 1Mbps. Now we will calculate how long it will take to transfer from main memory to secondary memory.

User process size is 2048Kb

Data transfer rate is 1Mbps = 1024 kbps

Time = process size / transfer rate

= 2048 / 1024

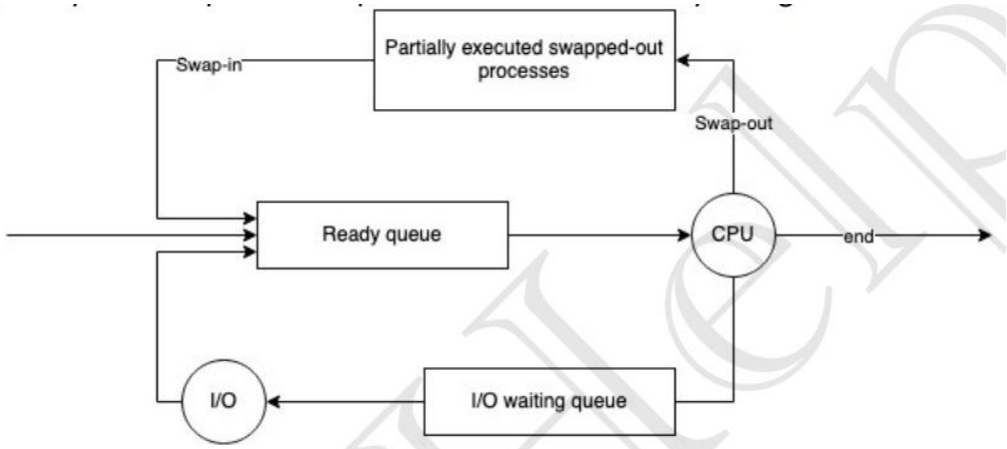
= 2 seconds

= 2000 milliseconds

Now taking swap-in and swap-out time, the process will take 4000 milliseconds.

a. Time-sharing system may have medium term scheduler (MTS).
b. Remove processes from memory to reduce degree of multi-programming.
c. These removed processes can be reintroduced into memory, and its execution can be continued where it left off. This is called Swapping.
d. Swap-out and swap-in is done by MTS.
e. Swapping is necessary to improve process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

f. Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.



Advantages of Swapping:

- It helps the CPU to manage multiple processes within a single main memory.
- It helps to create and use virtual memory.
- Swapping allows the CPU to perform multiple tasks simultaneously. Therefore, processes do not have to wait very long before they are executed.
- It improves the main memory utilization.

Disadvantages of Swapping:

- If the computer system loses power, the user may lose all information related to the program in case of substantial swapping activity.
- If the swapping algorithm is not good, the composite method can increase the number of Page Fault and decrease the overall processing performance.

Context Switching:

The Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system. When switching perform in the system, it stores the old running process's status in the form of registers and assigns the CPU to a new process to execute its tasks. While a new process is running in the system, the previous process must wait in a ready queue. The execution of the old process starts at that point where another process stopped it. It defines the characteristics of a multitasking operating system in which multiple processes shared the same CPU to perform multiple tasks without the need for additional processors in the system.

a. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.

b. When this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

c. It is pure overhead, because the system does no useful work while switching.

d. Speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

The need of Context Switching:

A context switching helps to share a single CPU across all processes to complete its execution and store the system's tasks status. When the process reloads in the system, the execution of the process starts at the same point where there is conflicting.

Following are the reasons that describe the need for context switching in the Operating system.

1. The switching of one process to another process is not directly in the system. A context switching helps the operating system that switches between the multiple processes to use the CPU's resource to accomplish its tasks and store its context. We can resume the service of the process at the same point later. If we do not store the currently running process's data or context, the stored data may be lost while switching between processes.

2. If a high priority process falls into the ready queue, the currently running process will be shut down or stopped by a high priority process to complete its tasks in the system.

3. If any running process requires I/O resources in the system, the current process will be switched by another process to use the CPUs. And when the I/O requirement is met, the old process goes into a ready state to wait for its execution in the CPU. Context switching stores the state of the process to resume its tasks in an operating system. Otherwise, the process needs to restart its execution from the initials level.

4. If any interrupts occur while running a process in the operating system, the process status is saved as registers using context switching. After resolving the interrupts, the process switches from a wait state to a ready state to resume its execution at the same point later, where the operating system interrupted occurs.

5. A context switching allows a single CPU to handle multiple process requests simultaneously without the need for any additional processors.

Example of Context Switching:
Suppose that multiple processes are stored in a Process Control Block (PCB). One process is running state to execute its task with the use of CPUs. As the process is running, another process arrives in the ready queue, which has a high priority of completing its task using CPU. Here we used context switching that switches the current process with the new process requiring the CPU to finish its tasks. While switching the process, a context switch saves the status of the old process in registers. When the process reloads into the CPU, it starts the execution of the process when the new process stops the old process. If we do not save the state of the process, we have to start its execution at the initial level. In this way, context switching helps the operating system to switch between the processes, store or reload the process when it requires executing its tasks.

Context Switching triggers:
Following are the three types of context switching triggers as follows.

1. Interrupts
2. Multitasking
3. Kernel/User switch

Interrupts: A CPU requests for the data to read from a disk, and if there are any interrupts, the context switching automatic switches a part of the hardware that requires less time to handle the interrupts.

Multitasking: A context switching is the characteristic of multitasking that allows the process to be switched from the CPU so that another process can be run. When switching the process, the old state is saved to resume the process's execution at the same point in the system.

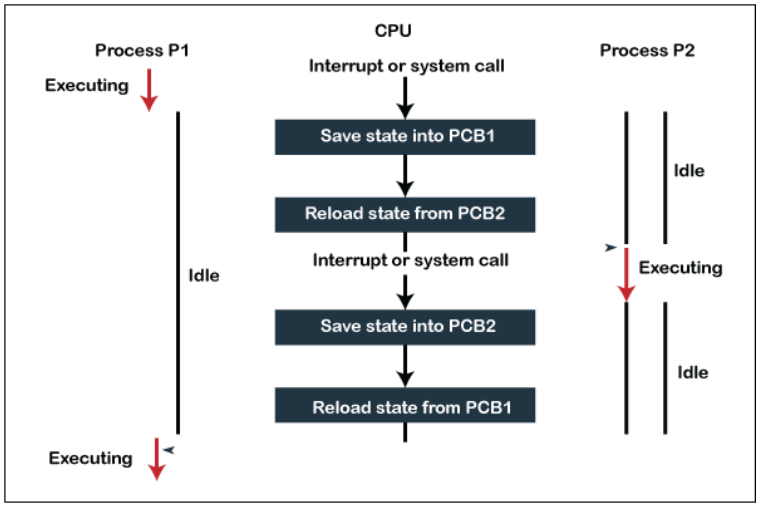
Kernel/User Switch: It is used in the operating systems when switching between the user mode, and the kernel/user mode is performed.

What is PCB?

A PCB (Process Control Block) is a data structure used in the operating system to store all data related information to the process. For example, when a process is created in the operating system, updated information of the process, switching information of the process, terminated process in the PCB.

Step of Context Switching:

There are several steps involves in context switching of the processes. The following diagram represents the context switching of two processes, P1 to P2, when an interrupt, I/O needs, or priority-based process occurs in the ready queue of PCB.



As we can see in the diagram, initially, the P1 process is running on the CPU to execute its task, and at the same time, another process, P2, is in the ready state. If an error or interruption has occurred or the process requires input/output, the P1 process switches its state from running to the waiting state. Before changing the state of the process P1, context switching saves the context of the process P1 in the form of registers and the program counter to the **PCB1**. After that, it loads the state of the P2 process from the ready state of the **PCB2** to the running state.

The following steps are taken when switching Process P1 to Process 2:

1. First, this context switching needs to save the state of process P1 in the form of the program counter and the registers to the PCB (Program Counter Block), which is in the running state.

2. Now update PCB1 to process P1 and moves the process to the appropriate queue, such as the ready queue, I/O queue and waiting queue.

3. After that, another process gets into the running state, or we can select a new process from the ready state, which is to be executed, or the process has a high priority to execute its task.

4. Now, we have to update the PCB (Process Control Block) for the selected process P2. It includes switching the process state from ready to running state or from another state like blocked, exit, or suspend.

5. If the CPU already executes process P2, we need to get the status of process P2 to resume its execution at the same time point where the system interrupt occurs.

Similarly, process P2 is switched off from the CPU so that the process P1 can resume execution. P1 process is reloaded from PCB1 to the running state to resume its task at the same point. Otherwise, the information is lost, and when the process is executed again, it starts execution at the initial level.

Orphan Process:

In the context of operating systems, an **orphan process** is a child process that continues to run after its parent process has terminated or finished executing. When a parent process spawns a child process, it may wait for the child to complete its execution. However, if the parent process terminates before the child process, the child process becomes an orphan.

Orphan processes are typically adopted by the operating system to ensure they do not remain running indefinitely and do not consume system resources unnecessarily. This adoption process is known as **process adoption** or **reprinting**. When the parent process terminates, the orphan process is reassigned a new parent, usually the init process (or its equivalent on different operating systems). The init process is the first process started by the operating system during boot and has a process ID of 1.

By adopting orphan processes, the init process ensures that all processes in the system eventually have a parent, preventing them from becoming zombie processes (a state where a process has terminated but its exit status has not yet been collected by its parent).

In summary, an orphan process is a child process that continues to run after its parent process has terminated. The operating system assigns a new parent (typically the init process) to orphan processes to prevent them from becoming zombies and to properly manage their termination and resource clean-up.

a. The process whose parent process has been terminated and it is still running.
b. Orphan processes are adopted by init process.
c. Init is the first process of OS.

Zombie Process or Defunct Process:

In the context of operating systems, a **zombie process**, also known as a **defunct process**, is a process that has completed execution but still has an entry in the process table. This entry is needed to allow the parent process to retrieve the child process's exit status after its termination. A process enters the zombie state after it has finished its execution, but its parent process has not yet collected its exit status.

Here is how the lifecycle of a zombie process occurs:

1. A parent process creates a child process and continues its execution.
2. The child process executes and completes its task.
3. The child process terminates, but its exit status is still needed by the parent process.
4. The child process becomes a zombie (defunct) process, indicating that it has finished execution but still has an entry in the process table.
5. The parent process should collect the exit status of the terminated child process using system calls like `wait()` or `waitpid()`. This operation reaps the zombie process, freeing up the resources associated with it.
6. Once the exit status is collected by the parent process, the zombie process is removed from the process table, and its resources are completely freed.

If a parent process fails to collect the exit status of its terminated child processes (perhaps due to improper coding or design), these processes can remain in the zombie state, wasting system resources. Properly designed programs

and scripts should ensure that they handle the termination status of their child processes to prevent the accumulation of zombie processes.
In Unix-like operating systems, you can view zombie processes using commands like `ps aux` or `ps -ef`. Zombie processes are displayed with a status of 'Z' in the output of these commands.

a. A zombie process is a process whose execution is completed but it still has an entry in the process table.

b. Zombie processes usually occur for child processes, as the parent process still needs to read it's child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as **reaping** the zombie process.

c. It is because parent process maycall wait () on child process for a longer time duration and child process got terminated much earlier.

d. As entry in the process table can only be removed, after the parent process reads the exit status of child process. Hence, the child process remains a zombie till it is removed from the process table.

Process Scheduling:

Process scheduling in operating systems is the act of determining which process should run next on a CPU (Central Processing Unit). The CPU is a shared resource, and there are usually more processes in a system than there are CPUs to run them. Process scheduling ensures that multiple processes can share the CPU efficiently and fairly, allowing for multitasking and concurrent execution of programs. The primary goals of process scheduling are to maximize CPU utilization, provide fairness, minimize response time, and optimize system throughput.

a. Basis of Multi-programming OS.
b. By switching the CPU among processes, the OS can make the computer more productive.
c. Many processes are kept in memory at a time, when a process must wait or time quantum expires, the OS takes the CPU away from that process & gives the CPU to another process & this pattern continues.

Here are the key concepts related to process scheduling:

1. **Scheduling Queues:**
   - **Ready Queue:** Processes that are ready to execute but are waiting for CPU time are placed in the ready queue. The process scheduler selects processes from this queue for execution.
   - **Waiting Queue:** Processes that are waiting for I/O operations to complete are moved to the waiting queue. They are not considered for execution until the I/O operation is finished.
2. **Scheduling Algorithms:**
   - **First-Come, First-Served (FCFS):** The process that arrives first is the first to be executed.
   - **Shortest Job Next (SJN) / Shortest Job First (SJF):** The process with the smallest total burst time is selected next.
   - **Round Robin (RR):** Each process is assigned a fixed time slice or quantum. When a process's time slice expires, it is moved to the end of the queue.
   - **Priority Scheduling:** Each process is assigned a priority. The process with the highest priority is selected for execution. This can be either pre-emptive or non-pre-emptive.
   - **Multilevel Queue Scheduling:** Processes are divided into different priority levels, and each queue has its scheduling algorithm.

- **Multilevel Feedback Queue Scheduling:** Similar to multilevel queue scheduling, but processes can move between queues based on their behaviour and CPU bursts.

3. **Pre-emption:**
   - **Pre-emptive Scheduling:** The operating system can interrupt a running process and move it to the ready queue, allowing another process to run. Pre-emptive scheduling ensures fairness and responsiveness.
   - **Non-Pre-emptive Scheduling:** Once a process starts running, it continues until it finishes or voluntarily gives up the CPU.
4. **Context Switching:**
   - **Context Switch:** When the operating system switches from executing one process to another, it performs a context switch. This involves saving the current process's state (registers, program counter, etc.) and loading the state of the next process to be executed.
   - **Cost:** Context switches incur a small overhead, and scheduling algorithms aim to minimize the number of context switches to improve overall system performance.
5. **Scheduling Criteria:**
   - **CPU Burst Time:** Predicting the time a process will need the CPU before it requires I/O operations.
   - **I/O Burst Time:** The time a process requires for I/O operations before it can continue with CPU execution.
   - **Priority:** Some processes are more critical than others and need to be executed with higher priority.

Effective process scheduling is crucial for the efficient utilization of system resources and ensures that users experience responsive and smooth multitasking environments. Different operating systems and applications might require different scheduling algorithms based on their specific needs and use cases.

CPU Scheduler:

A **CPU scheduler** is a component of the operating system that selects and assigns processes in the ready queue to the CPU for execution. Its primary goal is to maximize CPU utilization and throughput while ensuring fairness and responsiveness. The CPU scheduler operates in the kernel space and is an essential part of the operating system's process management system.

a. Whenever the CPU become ideal, OS must select one process from the ready queue to be executed.
b. Done by STS.

Here are its key functions and responsibilities:

1. **Process Selection:**
   - The CPU scheduler selects the next process to run from the pool of processes that are ready to execute. These processes are typically in the ready queue, waiting for CPU time.
2. **Scheduling Algorithms:**
   - The CPU scheduler uses various scheduling algorithms (such as First-Come, First-Served, Shortest Job Next, Round Robin, Priority Scheduling, etc.) to determine the order in which processes are

executed. Each algorithm has different properties and trade-offs, catering to specific system requirements.

3. **Context Switching:**
   - When the CPU scheduler decides to switch from one process to another, it performs a context switch. This involves saving the state of the currently running process (registers, program counter, etc.) and loading the state of the new process to be executed. Context switches are essential for multitasking but come at a performance cost.

4. **Priority Management:**
   - Many scheduling algorithms involve assigning priorities to processes. The CPU scheduler manages these priorities, ensuring that higher-priority processes get more CPU time. Priority management is crucial for real-time systems and applications with varying levels of importance.

5. **Pre-emption:**
   - Pre-emptive scheduling allows the CPU scheduler to interrupt a running process and allocate the CPU to a higher-priority process that has become available. Pre-emption ensures responsiveness, especially in multitasking environments.

6. **Fairness and Balance:**
   - The CPU scheduler aims to be fair, ensuring that all processes get a fair share of the CPU time. It prevents any single process from monopolizing the CPU, ensuring equitable resource allocation.

7. **Performance Optimization:**
   - The CPU scheduler plays a role in optimizing system performance by minimizing the overhead of context switches and maximizing CPU throughput. This optimization involves choosing appropriate scheduling algorithms based on the system's workload and requirements.

In summary, the CPU scheduler is a fundamental part of the operating system responsible for managing the execution of processes on the CPU. It ensures that processes are executed in an efficient, fair, and responsive manner, allowing the computer system to function smoothly in multitasking scenarios. Different operating systems implement various CPU scheduling algorithms, selecting the most suitable one based on the system's specific needs and use cases.

Non-Preemptive Scheduling:

Non-preemptive scheduling, also known as **cooperative scheduling**, is a type of scheduling where once a process starts its execution, it continues to run until it voluntarily releases the CPU. In other words, the operating system does not forcibly interrupt a running process to give CPU time to another process. Instead, processes have to explicitly yield the CPU, typically by entering an I/O operation or terminating.

a. Once CPU has been allocated to a process, the process keeps the CPU until it releases CPU either by terminating or by switching to wait-state.
b. Starvation, as a process with long burst time may starve less burst time process.
c. Low CPU utilization.

In the context of I/O operations, non-preemptive scheduling works as follows:

- **Process Initiation:** When a process is initiated, it starts executing and continues to run until it explicitly requests an I/O operation, such as reading from a file, receiving user input, or sending data over a network.

- **I/O Request:** When a process requests an I/O operation, it enters a waiting state. The operating system places the process in a waiting queue associated with the specific I/O device.

- **Blocked State:** While waiting for the I/O operation to complete, the process is in a blocked state. It does not consume CPU time during this period.

- **Completion of I/O Operation:** Once the I/O operation is completed (e.g., data is read from a file), the process is moved back to the ready queue, indicating it is ready to execute again.

- **Process Resumption:** The process resumes its execution only when it is selected from the ready queue by the CPU scheduler. The operating system does not interrupt a running process to execute a waiting process, unlike in pre-emptive scheduling.

Non-preemptive scheduling has the advantage of simplicity and low overhead. Since there are no forced context switches, the system doesn't incur the overhead associated with preemptive scheduling, making it more efficient in some cases. However, it also has drawbacks, primarily concerning responsiveness and fairness. If a process does not voluntarily yield the CPU (e.g., due to an infinite loop or a programming error), other processes might be delayed or starved of CPU time.
A classic example of non-preemptive scheduling is the scheduling used by early versions of cooperative multitasking systems, where processes had to yield explicitly to allow other processes to run. However, in modern operating systems, preemptive scheduling is more common because it offers better responsiveness and fairness, especially in environments where processes can't be trusted to yield the CPU voluntarily.

Preemptive Scheduling:
Preemptive scheduling, also known as **preemptive multitasking**, is a scheduling technique in which the operating system can interrupt a currently executing process and allocate the CPU to another process. This interruption is called a **pre-emption**, and it occurs without the cooperation of the running process. Preemptive scheduling is designed to ensure fairness, responsiveness, and efficient use of system resources, especially in multitasking environments where multiple processes compete for CPU time.

a. CPU is taken away from a process after time quantum expires along with terminating or switching to wait-state.
b. Less Starvation.
c. High CPU utilization.

In preemptive scheduling:

1. **Time Slicing:** Each process is assigned a fixed time slice or quantum. When a process's time slice expires, it is moved to the back of the ready queue, and the CPU scheduler selects the next process to run. This technique is commonly known as **round-robin scheduling**.

2. **Priority-Based Pre-emption:** Processes are assigned priorities, and the CPU is given to the process with the highest priority that is ready to execute. Lower-priority processes can be pre-empted to allow higher-priority processes to run. This ensures that critical tasks are executed promptly.

3. **Interrupt-Driven Pre-emption:** Pre-emption can occur due to hardware interrupts (such as I/O operations completing) or software interrupts (such as timer interrupts). When an interrupt occurs, the CPU switches to the appropriate interrupt service routine, and the currently running process might be pre-empted to handle the interrupt.

4. **Real-Time Pre-emption:** In real-time operating systems, tasks have strict deadlines. Pre-emptive scheduling ensures that high-priority real-time tasks are executed within their deadlines, even if lower-priority tasks are currently running.

Benefits of preemptive scheduling include:

- **Responsiveness:** Pre-emptive scheduling ensures that no single process can monopolize the CPU for an extended period. It allows for a more responsive system, even if some processes are resource-intensive.

- **Fairness:** Pre-emptive scheduling provides fairness by giving all processes an opportunity to run, preventing any single process from starving others of CPU time.

- **Priority Handling:** It allows the operating system to manage processes based on their priorities, ensuring that critical tasks are executed with higher precedence.

However, preemptive scheduling also introduces challenges, such as the overhead of context switches (switching between processes) and the need for synchronization mechanisms to ensure data consistency when multiple processes access shared resources.
Despite these challenges, preemptive scheduling is widely used in modern operating systems to provide efficient multitasking, real-time processing, and a responsive user experience. Different algorithms, like shortest job next, priority scheduling, and multilevel feedback queues, are employed in preemptive scheduling based on specific system requirements and use cases.

Goal of Scheduling:

The goals of CPU scheduling in operating systems are designed to optimize the performance, responsiveness, and fairness of the system. These goals ensure efficient utilization of CPU resources and contribute to a smooth user experience. Here are the primary objectives of CPU scheduling:

1. **Maximize CPU Utilization:**
   - **Goal:** Ensure that the CPU is utilized to its fullest capacity.
   - **Rationale:** Keeping the CPU busy with processes at all times maximizes the overall system throughput and efficiency.

2. **Fairness:**
   - **Goal:** Provide fair allocation of CPU time to all processes.
   - **Rationale:** Prevent any process from monopolizing the CPU, ensuring that every process gets a fair share of CPU time. This leads to equitable distribution of resources among users and applications.

3. **Minimize Turnaround Time:**
   - **Goal:** Minimize the total time taken by a process to complete its execution, from the submission to the termination.
   - **Rationale:** Faster turnaround time improves the system's ability to execute tasks promptly, enhancing user satisfaction and the overall system responsiveness.

4. **Minimize Waiting Time:**
   - **Goal:** Minimize the total time processes spend waiting in the ready queue before they get executed.
   - **Rationale:** Reducing waiting time ensures that processes start execution quickly after becoming ready, which leads to efficient utilization of resources.

5. **Minimize Response Time:**
   - **Goal:** Minimize the time it takes for a system to respond to user input.
   - **Rationale:** Quick response time provides users with a more interactive and responsive computing experience. This is especially critical in interactive systems, such as desktop computers and web servers.

6. **Maximize Throughput:**

- **Goal:** Maximize the number of processes completed within a given time period.
- **Rationale:** Maximizing throughput ensures that the system can handle a large number of tasks efficiently. This is particularly important for server environments and batch processing systems.

7. **Ensure Predictability and Stability:**
   - **Goal:** Provide consistent and predictable behaviour in CPU scheduling.
   - **Rationale:** Predictable behaviour helps in system planning and ensures that users and applications can rely on a consistent level of performance, leading to a stable computing environment.

8. **Support Real-Time Requirements:**
   - **Goal:** Provide guarantees for the timely execution of critical real-time tasks.
   - **Rationale:** Real-time systems have stringent requirements for task completion within specific deadlines. Meeting these requirements ensures the reliable functioning of systems used in critical applications like aerospace, industrial automation, and healthcare.

Operating systems use various scheduling algorithms and techniques to achieve these goals. Different systems might prioritize these goals differently based on the specific requirements and use cases of the computing environment.

Important Points:

1. **Throughput:** No. of processes completed per unit time.
2. **Arrival time (AT):** Time when process is arrived at the ready queue.
3. **Burst time (BT):** The time required by the process for its execution.
4. **Turnaround time (TAT):** Time taken from first time process enters ready state till it terminates. (CT - AT)
5. **Wait time (WT):** Time process spends waiting for CPU. (WT = TAT – BT) 11. Response time: Time duration between process getting into ready queue and process getting CPU for the first time.
6. **Completion Time (CT):** Time taken till process gets terminated.

CPU Scheduling Algorithms:

Modes of CPU Scheduling Algorithm:

There are two modes in CPU Scheduling Algorithms. They are:

1. Pre-emptive Approach
2. Non Pre-emptive Approach

In **Pre Emptive-Approach** the process once starts its execution then the CPU is allotted to the same process until the completion of process. There would be no shift of Processes by the Central Processing Unit. The complete CPU is allocated to the Process and there would be no change of CPU allocation until the process is complete.

In **Non-Pre Emptive-Approach** the process once stars its execution, then the CPU is not allotted to the same process until the completion of process. There is a shift of Processes by the Central Processing Unit. The complete CPU is allocated to the Process when only certain required conditions are achieved and there will be change of CPU allocation if there is break or false occurrence in the required conditions.

Types of CPU Scheduling Algorithm:

- **First Come First Serve**
- **Shortest Job First**

- **Priority Scheduling**
- **Round Robin Scheduling**

First Come First Scheduling Algorithm (FCFS):

This is the first type of CPU Scheduling Algorithms. Here, in this CPU Scheduling Algorithm we are going to learn how CPU is going to allot resources to the certain process.

Here, in the First Come First Serve CPU Scheduling Algorithm, the CPU allots the resources to the process in a certain order. The order is serial way. The CPU is allotted to the process in which it has occurred.

We can also say that First Come First Serve CPU Scheduling Algorithm follows First In First Out in Ready Queue.
First Come First Serve can be called as FCFS in short form.

Characteristics of FCFS:

- First Come First Serve can follow or can be executed in Pre-emptive Approach or Non-Pre-emptive Approach.
- The Process which enters the Ready Queue is executed first. So, we say that FCFS follows First in First out Approach.
- First Come First Come First Serve is only executed when the Arrival Time (AT) is greater than or equal to the Time which is at present.

Advantages:

- Very easy to perform by the CPU
- Follows FIFO Queue Approach

Disadvantages:

- First Come First Serve is not very efficient.
- First Come First Serve suffers because of Convoy Effect.

Convoy Effect:

The **convoy effect**, also known as **convoy phenomenon**, refers to a situation in CPU scheduling where short processes are held up by longer processes waiting in the ready queue, leading to inefficient use of CPU time and poor system performance.

This effect occurs when shorter processes, which require relatively less CPU time to execute, are stuck waiting in the ready queue behind longer processes that have higher burst times. As a result, even though the short processes could be executed quickly, they end up waiting for a long time behind the longer processes, creating a "convoy" of processes in the queue.

The convoy effect can significantly impact the system's overall throughput, response time, and efficiency. Short processes might have to wait unnecessarily, leading to decreased system performance and utilization of resources. This phenomenon is particularly problematic in scheduling algorithms like First-Come, First-Served (FCFS) or Shortest Job Next (SJN) when the shortest processes are not given priority. Preemptive scheduling algorithms, such as Round Robin or Priority Scheduling, help mitigate the convoy effect by allowing shorter processes to be executed even if longer processes are in the system. These algorithms allow the scheduler to interrupt a running process and allocate the CPU to a shorter process if it becomes available, thereby reducing the waiting time for short processes and preventing the convoy effect.

Shortest Job First CPU Scheduling Algorithm (SJF):

This is another type of CPU Scheduling Algorithms. Here, in this CPU Scheduling Algorithm we are going to learn how CPU is going to allot resources to the certain process.

The Shortest Job is heavily dependent on the Burst Times. Every CPU Scheduling Algorithm is basically dependent on the Arrival Times. Here, in this Shortest Job First CPU Scheduling Algorithm, the CPU allots its resources to the process which is in ready queue and the process which has least Burst Time.

If we face a condition where two processes are present in the Ready Queue and their Burst Times are same, we can choose any process for execution. In actual Operating Systems, if we face the same problem then sequential allocation of resources takes place.

Shortest Job First can be called as SJF in short form.

Characteristics of SJF:

- SJF (Shortest Job First) has the least average waiting time. This is because all the heavy processes are executed at the last. So, because of this reason all the very small, small processes are executed first and prevent starvation of small processes.
- It is used as a measure of time to do each activity.
- If shorter processes continue to be produced, hunger might result. The idea of aging can be used to overcome this issue.
- Shortest Job can be executed in Pre-emptive and also non pre-emptive way also.

Advantages of SJF:

- SJF is used because it has the least average waiting time than the other CPU Scheduling Algorithms
- SJF can be termed or can be called as long term CPU scheduling algorithm.

Disadvantages of SJF:

- Starvation is one of the negative traits Shortest Job First CPU Scheduling Algorithm exhibits.
- Often, it becomes difficult to forecast how long the next CPU request will take.

Shortest Job First (SJF) [Non-preemptive]

a. Process with least BT will be dispatched to CPU first.
b. Must do estimation for BT for each process in ready queue beforehand, Correct estimation of BT is an impossible task (ideally.)
c. Run lowest time process for all time then, choose job having lowest BT at that instance.
d. This will suffer from convoy effect as if the very first process which came is Ready state is having a large BT.
e. Process starvation might happen.
f. Criteria for SJF algos, AT + BT

SJF [Preemptive]

a. Less starvation.
b. No convoy effect.
c. Gives average WT less for a given set of processes as scheduling short job before a long one decreases the WT of short job more than it increases the WT of the long process.

Priority CPU Scheduling Algorithm:

This is another type of CPU Scheduling Algorithms. Here, in this CPU Scheduling Algorithm we are going to learn how CPU is going to allot resources to the certain process.

The Priority CPU Scheduling is different from the remaining CPU Scheduling Algorithms. Here, each and every process has a certain Priority number.

There are two types of Priority Values.

- Highest Number is considered as Highest Priority Value
- Lowest Number is considered as Lowest Priority Value

Priority for Prevention The priority of a process determines how the CPU Scheduling Algorithm operates, which is a preemptive strategy. Since the editor has assigned equal importance to each function in this method, the most crucial steps must come first. The most significant CPU planning algorithm relies on the FCFS (First Come First Serve) approach when there is a conflict, that is, when there are many processors with equal value.

Characteristics

- Priority CPU scheduling organizes tasks according to importance.
- When a task with a lower priority is being performed while a task with a higher priority arrives, the task with the lower priority is replaced by the task with the higher priority, and the latter is stopped until the execution is finished.
- A process's priority level rises as the allocated number decreases.

Advantages:

- The typical or average waiting time for Priority CPU Scheduling is shorter than First Come First Serve (FCFS).
- It is easier to handle Priority CPU scheduling.
- It is less complex.

Disadvantages:

- The Starvation Problem is one of the Pre-emptive Priority CPU Scheduling Algorithm's most prevalent flaws. Because of this issue, a process must wait a longer period of time to be scheduled into the CPU. The hunger problem or the starvation problem is the name given to this issue.

Priority Scheduling [Non-preemptive]

a. Priority is assigned to a process when it is created.
b. SJF is a special case of general priority scheduling with priority inversely proportional to BT.

Priority Scheduling [Preemptive]

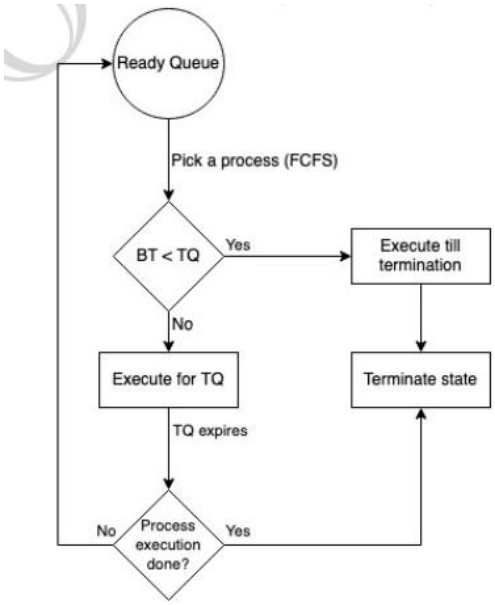
a. Current RUN state job will be pre-empted if next job has higher priority.
b. May cause indefinite waiting (Starvation) for lower priority jobs. (Possibility is they won't get executed ever). (True for both pre-emptive and non-pre-emptive version).
    i. Solution: Ageing is the solution.
    ii. Gradually increase priority of process that wait so long. E.g., increase priority by 1 every 15 minutes.

Round Robin Algorithm:

Round Robin is a CPU scheduling mechanism those cycles around assigning each task a specific time slot. It is the First come, first served CPU Scheduling technique with preemptive mode. The Round Robin CPU algorithm frequently emphasizes the Time-Sharing method.

a. Most popular.
b. Like FCFS but pre-emptive.
c. Designed for time sharing systems.
d. Criteria: AT + time quantum (TQ), doesn't depend on BT.

e. No process is going to wait forever, hence very low starvation. [No convoy effect].
f. Easy to implement.
g. If TQ is small, more will be the context switch (more overhead).



Round Robin CPU Scheduling Algorithm characteristics include:

- Because all processes receive a balanced CPU allocation, it is straightforward, simple to use, and starvation-free.
- One of the most used techniques for CPU core scheduling. Because the processes are only allowed access to the CPU for a brief period of time, it is seen as preemptive.

The benefits of round robin CPU Scheduling:

- Every process receives an equal amount of CPU time, therefore round robin appears to be equitable.
- To the end of the ready queue is added the newly formed process.

Advantages:

1. **Fairness:** RR ensures fairness by giving each process an equal share of CPU time. No process can monopolize the CPU for an extended period.

2. **Easy to Implement:** RR is easy to implement, making it a popular choice in many operating systems. It uses a simple circular queue data structure.

3. **Low Latency for Short Jobs:** Short jobs or processes with lower burst times are given quick access to the CPU since they are served in a timely manner due to the time quantum.

4. **Responsive:** It provides relatively fast response times for interactive tasks since no process has to wait for an extended period before getting CPU time.

5. **Predictable Performance: R**R provides predictable performance in terms of response time. Every process gets an equal opportunity, ensuring consistent behaviour.

6. **Prevents Starvation:** No process is indefinitely postponed. Every process eventually gets CPU time, preventing starvation (where a process is unable to proceed due to lack of resources).

Disadvantages:

1. **Poor Performance for Long-Running Jobs:** Long-running jobs suffer in RR because they are repeatedly placed at the back of the queue after their time quantum expires. This can lead to inefficient CPU utilization.

2. **Inefficient Use of CPU Time:** RR can lead to inefficient use of CPU time due to context-switching overhead. Context switches occur every time a process's time quantum expires, consuming CPU cycles.

3. **High Waiting Times for CPU-Bound Processes:** CPU-bound processes may experience high waiting times, especially if the time quantum is set too low. They would be repeatedly interrupted and placed back in the queue.

4. **Dependency on Time Quantum:** The performance of RR is significantly influenced by the choice of the time quantum. Setting the time quantum too short can lead to frequent context switches, while setting it too long can diminish the responsiveness of the system.

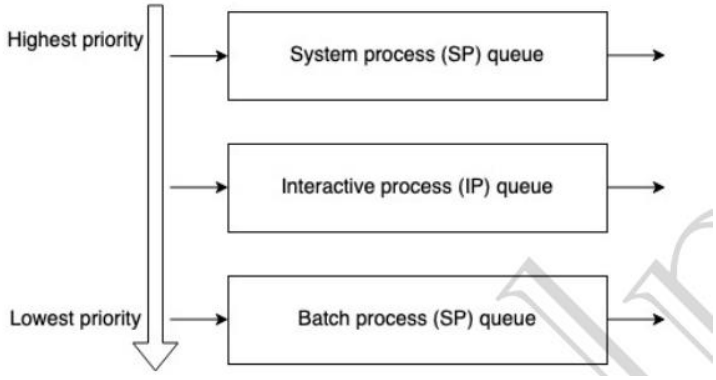
5. **Doesn't Prioritize Important Processes:** RR does not prioritize important or critical processes. A high-priority task has the same chance of getting CPU time as a low-priority one.

6. **Doesn't Consider Process Priority:** RR treats all processes equally, disregarding their importance or priority. This can be a disadvantage in systems where certain tasks require immediate attention.

Multi-Level Queue Scheduling (MLQ):

Multi-level queue scheduling (MLQ) is a scheduling algorithm that partitions the ready queue into several separate queues, each with its own scheduling algorithm. Each queue has a different priority level, and processes are assigned to these queues based on their properties, such as the type of task, priority, or other criteria. Each queue can have its own scheduling algorithm, making it suitable for a wide range of application scenarios.

a. Ready queue is divided into multiple queues depending upon priority.
b. A process is permanently assigned to one of the queues (inflexible) based on some property of process, memory, size, process priority or process type.
c. Each queue has its own scheduling algorithm. E.g., SP -> RR, IP -> RR & BP -> FCFS



d. System process: Created by OS (Highest priority) Interactive process (Foreground process): Needs user input (I/O). Batch process (Background process): Runs silently, no user input required.
e. Scheduling among different sub-queues is implemented as fixed priority pre-emptive scheduling. E.g., foreground queue has absolute priority over background queue.
f. If an interactive process comes & batch process is currently executing. Then, batch process will be pre-empted.
g. Problem: Only after completion of all the processes from the top-level ready queue, the further level ready queues will be scheduled. This came starvation for lower priority process.
h. Convoy effect is present.

In multi-level queue scheduling, the ready queue is divided into multiple priority levels, typically based on the priority of processes.

Here are the key features of multi-level queue scheduling:

1. **Multiple Queues:** The ready queue is divided into several queues, each representing a different priority level. Processes are placed in these queues based on their priority, with higher-priority processes in higher-priority queues.

2. **Different Scheduling Algorithms:** Each queue can have its own scheduling algorithm. For example, high-priority queues might use a pre-emptive scheduling algorithm like Round Robin to ensure responsiveness, while lower-priority queues might use non-pre-emptive algorithms to prioritize longer execution times for CPU-bound tasks.

3. **Priority-Based Assignment:** Processes are assigned to different queues based on their priority levels. Priority can be determined by various factors, such as process type, user, or importance of the task.

4. **Pre-emption and Aging:** Pre-emption can occur within the queues based on the scheduling algorithm used. Additionally, aging mechanisms can be implemented to prevent starvation, ensuring that lower-priority processes eventually get CPU time.

5. **Example Scenario:**
   - **Queue 1:** High-priority queue with short time quantum (e.g., Round Robin) for interactive tasks.
   - **Queue 2:** Medium-priority queue with longer time quantum for general tasks.
   - **Queue 3:** Low-priority queue with non-pre-emptive scheduling for background tasks.

Advantages of Multi-level Queue Scheduling:

- **Flexibility:** Different types of processes can be managed effectively based on their priority or characteristics.
- **Fairness:** Higher-priority tasks are given preference, ensuring fairness and responsiveness for important tasks.
- **Efficiency:** Tailoring scheduling algorithms to specific queues can optimize system performance for different types of applications.

Disadvantages of Multi-level Queue Scheduling:

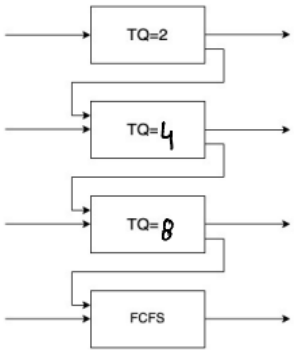
- **Complexity:** Managing multiple queues and algorithms adds complexity to the scheduling mechanism.
- **Tuning Challenges:** Setting appropriate parameters and thresholds for each queue and algorithm can be challenging.
- **Overhead:** Context switching between multiple queues and algorithms can introduce additional overhead.

Multi-Level Feedback Queue Scheduling:

Multi-level Feedback Queue Scheduling (MLFQ) is a sophisticated scheduling algorithm that extends the concept of multi-level queue scheduling. In MLFQ, processes are assigned to various priority queues similar to multi-level

queue scheduling, but the key difference is that MLFQ dynamically adjusts the priority of processes based on their behavior. This dynamic adjustment allows the scheduler to adapt to the characteristics of different processes and optimize the scheduling decisions.

a. Multiple sub-queues are present.
b. Allows the process to move between queues. The idea is to separate processes according to the characteristics of their BT. If a process uses too much CPU time, it will be moved to lower priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queue. In addition, a process that waits too much in a lower-priority queue may be moved to a higher priority queue. This form of ageing prevents starvation.
c. Less starvation then MLQ.
d. It is flexible.
e. Can be configured to match a specific system design requirement.



Here are the main features of Multi-level Feedback Queue Scheduling:

1. **Multiple Queues with Different Priorities:**
   - MLFQ maintains several priority queues, each with a different priority level. Typically, there is a fixed number of queues, each representing a different priority.

2. **Dynamic Priority Adjustment:**
   - MLFQ dynamically adjusts the priority of processes based on their behaviour. For example:
   - **Aging:** Processes waiting in a higher-priority queue for a long time are moved to a lower-priority queue. This prevents starvation, ensuring that processes eventually get CPU time.
   - **Boosting:** Periodically, all processes are boosted to a higher-priority queue, allowing processes that have not received CPU time for a while to move up in priority.

3. **Pre-emption and Queue Selection:**
   - MLFQ can use different scheduling algorithms for each queue. Commonly, round-robin scheduling is used. When a process's time quantum expires, it is moved to a lower-priority queue. If a process uses too much CPU time in a lower-priority queue, it is demoted to a higher-priority queue.

4. **Example Scenario:**
   - **Queue 1:** Highest priority queue with a short time quantum for interactive tasks.
   - **Queue 2:** Medium priority queue with a slightly longer time quantum.
   - **Queue 3:** Lower priority queue with a longer time quantum for CPU-bound tasks.
   - **Aging and Boosting:** Aging ensures that processes waiting for a long time move to higher queues. Periodic boosting moves all processes to higher queues to provide fairness.

Advantages of Multi-level Feedback Queue Scheduling:

1. **Adaptability:** MLFQ adapts to different types of processes, dynamically adjusting priorities based on their behaviour.

2. **Fairness:** Aging and boosting mechanisms prevent starvation, ensuring that all processes eventually receive CPU time.

3. **Optimized Performance:** MLFQ optimizes system performance by giving priority to interactive tasks and adjusting priorities based on process behaviour.

Disadvantages of Multi-level Feedback Queue Scheduling:

1. **Complexity:** MLFQ is more complex to implement and manage compared to simpler scheduling algorithms, requiring careful tuning of parameters.

2. **Tuning Challenges:** Setting appropriate parameters for aging, boosting, and time quantum values can be challenging to achieve desired system performance.

What is the limit of Process Table entry?

The maximum number of process table entries, also known as the **process table size**, is determined by the operating system's design and configuration. There is no fixed standard for this value, as it depends on various factors, such as the operating system type, the architecture it runs on, and the available system resources.

For example:

- **32-bit Operating Systems:** In older 32-bit systems, the address space limitations often imposed a maximum number of process table entries, typically in the range of several thousand processes.

- **64-bit Operating Systems:** With the advent of 64-bit systems, the address space limitations were significantly expanded. Modern 64-bit operating systems can support a much larger number of process table entries, potentially in the order of millions.

- **System Resources:** The amount of available system resources, such as RAM (Random Access Memory) and the capacity of the storage device where the process table is stored, can also influence the maximum number of process table entries.

- **Configuration Settings:** Some operating systems allow system administrators to configure the process table size based on their specific requirements. This configuration can be done through system parameters and settings.

It's important to note that the practical limit of process table entries is also influenced by the system's overall workload and the system's ability to efficiently manage and switch between processes. Too many processes can lead to increased context switching overhead and decreased system performance.

When designing systems or applications, it's essential to consider the limitations of the operating system and hardware being used to ensure optimal performance and resource utilization. The specific limits for process table entries can be determined by consulting the documentation of the operating system in use.

PART 05: Concurrency, Thread, Thread Scheduling, Thread Context Switching, Multi-Threading.

Concurrency:

Concurrency is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel.

It refers to the execution of multiple instruction sequences at the same time. It occurs in an operating system when multiple process threads are executing concurrently. These threads can interact with one another via shared memory or message passing. Concurrency results in resource sharing, which causes issues like deadlocks and resource scarcity. It aids with techniques such as process coordination, memory allocation, and execution schedule to maximize throughput.

Problems in Concurrency:

- **Locating the programming errors:** It's difficult to spot a programming error because reports are usually repeatable due to the varying states of shared components each time the code is executed.

- **Sharing Global Resources:** Sharing global resources is difficult. If two processes utilize a global variable and both alter the variable's value, the order in which the many changes are executed is critical.

- **Locking the channel:** It could be inefficient for the OS to lock the resource and prevent other processes from using it.

- **Optimal Allocation of Resources:** It is challenging for the OS to handle resource allocation properly.

How process mapped with memory:

In modern operating systems, a **process** is mapped to memory through a combination of hardware and software mechanisms. This mapping allows processes to access memory addresses for data storage and execution.

Here's how this mapping generally occurs:

1. **Virtual Memory:**
   - Most operating systems use a concept called **virtual memory**. Each process has its own virtual address space, which is a range of memory addresses that it can use. This virtual address space is much larger than the physical memory (RAM) available in the system.

   - Processes interact with virtual addresses, thinking they are accessing the physical memory directly, but these addresses are translated to physical addresses by the operating system and hardware.

2. **Address Translation:**
   - When a process accesses a memory address, the CPU's memory management unit (MMU) translates the virtual address to a physical address. This translation is done using a data structure called a **page table** or a **translation lookaside buffer (TLB)**.

   - The page table maintains the mapping between virtual addresses and physical addresses. If the translation is not already in the TLB, it involves a lookup in the page table. If the mapping is found, the physical address is determined.

3. **Page Faults:**
   - If the mapping is not found in the page table (a situation called a **page fault**), the operating system is involved. The OS then loads the required data from the secondary storage (usually a hard drive or SSD) into physical memory, updates the page table, and the process can continue execution.

4. **Memory Segmentation:**
   - Some operating systems use **memory segmentation** in addition to paging. Segmentation divides the virtual address space into segments such as code, data, and stack. Each segment can grow or shrink dynamically.

   - The operating system maintains a segment table, which is used for address translation, similar to how a page table works for paging.

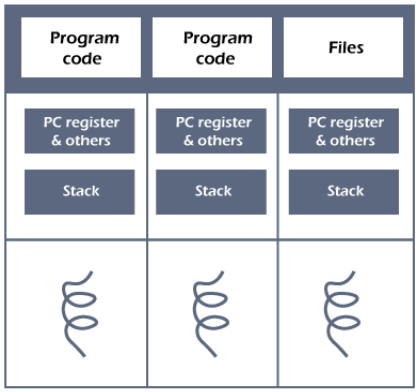
5. **Shared Memory and Memory-Mapped Files:**
   - Operating systems allow processes to share memory regions or map files directly into memory. This shared memory or memory-mapped files mechanism allows multiple processes to communicate and share data efficiently.

6. **Memory Protection:**
   - Memory protection mechanisms ensure that processes do not interfere with each other's memory. Unauthorized access attempts or accessing unallocated memory regions result in segmentation faults or access violation errors, protecting the integrity and security of the system.

In summary, the operating system, working with the hardware, manages the mapping between virtual addresses used by processes and physical addresses in the system's RAM. This mapping is crucial for enabling efficient use of memory, protecting processes from one another, and facilitating features like dynamic memory allocation, process communication, and file handling.

Thread in OS:

A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control. There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process. Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks. Thread is often referred to as a lightweight process.



The process can be split down into so many threads. **For example**, in a browser, many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.

- Single sequence stream within a process.
- An independent path of execution in a process.
- Light-weight process.
- Used to achieve parallelism by dividing a process's tasks which are independent path of execution.

- E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell checking, formatting of text and saving the text are done concurrently by multiple threads.)

Need of Thread:

- It takes far less time to create a new thread in an existing process than to create a new process.
- Threads can share the common data, they do not need to use Inter- Process communication.
- Context switching is faster when working with threads.
- It takes less time to terminate a thread than a process.

Types of Thread:

1. Kernel level thread.
2. User-level thread.

User Level Thread:

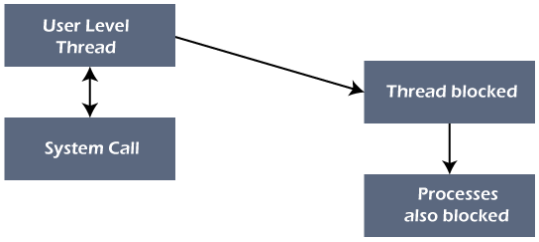
The operating system does not recognize the user-level thread. User threads can be easily implemented and it is implemented by the user. If a user performs a user-level thread blocking operation, the whole process is blocked. The kernel level thread does not know nothing about the user level thread. The kernel-level thread manages user-level threads as if they are single-threaded processes? Examples: Java thread, POSIX threads, etc.

Advantages of User Level Thread:

- The user threads can be easily implemented than the kernel thread.
- User-level threads can be applied to such types of operating systems that do not support threads at the kernel-level.
- It is faster and efficient.
- Context switch time is shorter than the kernel-level threads.
- It does not require modifications of the operating system.
- User-level threads representation is very simple. The register, PC, stack, and mini thread control blocks are stored in the address space of the user-level process.
- It is simple to create, switch, and synchronize threads without the intervention of the process.

Disadvantages of User Level Thread:

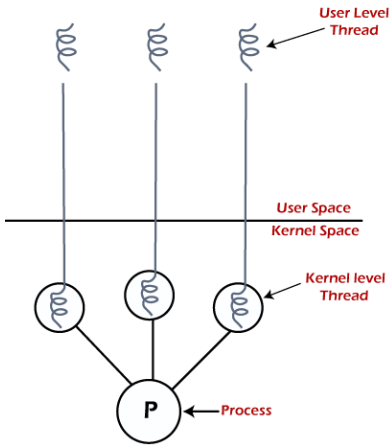
- User-level threads lack coordination between the thread and the kernel.
- If a thread causes a page fault, the entire process is blocked.



Kernel Level Thread:

The kernel thread recognizes the operating system. There is a thread control block and process control block in the system for each thread and process in the kernel-level thread. The kernel-level thread is implemented by the operating system. The kernel knows about all the threads and manages them. The kernel-level thread offers a system call to create and manage the threads from user-space. The implementation of kernel threads is more

difficult than the user thread. Context switch time is longer in the kernel thread. If a kernel thread performs a blocking operation, the Banky thread execution can continue. Example: Window Solaris.



Advantages of Kernel Level Thread:

- The kernel-level thread is fully aware of all threads.
- The scheduler may decide to spend more CPU time in the process of threads being large numerical.
- The kernel-level thread is good for those applications that block the frequency.

Disadvantages of Kernel Level Thread:

- The kernel thread manages and schedules all threads.
- The implementation of kernel threads is difficult than the user thread.
- The kernel-level thread is slower than user-level threads.

Components of Threads:

1. **Program Counter (PC):** The program counter keeps track of the address of the next instruction to be executed in the thread. When the thread is interrupted, the program counter helps in saving the current execution state so that the thread can be resumed later from the same point.

2. **Register Set:** Threads have their own set of registers, including general-purpose registers, stack pointer, and often a base pointer. These registers store intermediate data and addresses during the execution of instructions. When a thread is context-switched, the contents of these registers need to be saved to the thread's context data and restored when the thread is resumed.

3. **Stack Space:** Each thread has its own stack space in memory. The stack is used to store function call information, local variables, and the state of the function calls. When a function is called, the current state of the calling function is saved on the stack. This stack space is essential for managing function calls and local variables and plays a crucial role in the execution of the thread.

These components are fundamental for the execution and management of threads in a multithreaded program. The program counter helps in tracking the next instruction, the register set stores temporary data, and the stack space manages function calls and local variables, ensuring the thread's proper execution and state management.

How threads are napped with memory:

Threads, like processes, are entities in a computer program that can be scheduled for execution. Threads are lighter than processes and share the same memory space within a process. When threads are created within a process, they share the process's memory and resources, allowing them to communicate and synchronize efficiently.

Here's how threads are mapped to memory:

1. **Shared Memory Space:**
   - Threads within a process share the same memory space. They have access to the process's memory, including global variables, heap, and code segment. This shared memory space enables threads to communicate and share data without the need for inter-process communication mechanisms.

2. **Thread-Specific Data (TSD):**
   - Threads can also have thread-specific data. This data is unique to each thread and is stored in a data structure known as **Thread-Specific Data (TSD)** or **Thread-Local Storage (TLS)**. TSD allows threads to have their own private data within the shared memory space.

3. **Thread Control Block (TCB):**
   - Each thread has its own **Thread Control Block (TCB)**, which contains information about the thread, including its program counter, stack pointer, register values, and state. The TCB is managed by the operating system and is crucial for the context switching process, where one thread is paused, and another is resumed.

4. **Stack Space:**
   - Each thread has its own **stack space**. The stack is used for storing function call information, local variables, and maintaining the execution context of the thread. Unlike the heap and global variables, the stack space is private to each thread and is not shared.

5. **Thread Safety and Synchronization:**
   - Threads within a process need to coordinate their activities to avoid race conditions and ensure data consistency. Synchronization mechanisms such as **mutexes**, **semaphores**, and **condition variables** are used to control access to shared resources and maintain thread safety.

6. **Memory Protection:**
   - Modern operating systems and processors provide **memory protection mechanisms** that prevent one thread from accessing the memory of another thread without proper synchronization. Access violations and segmentation faults are raised if a thread attempts to access unauthorized memory regions.

In summary, threads within a process share the same memory space, allowing them to efficiently communicate and share data. However, to ensure data consistency and avoid conflicts, synchronization mechanisms are used. Each thread has its own stack space for function calls and local variables, and access to shared resources is controlled using synchronization primitives. The operating system manages thread-specific information through the Thread Control Block and provides memory protection to prevent unauthorized access to memory regions.

Thread Scheduling:

Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.

Thread scheduling in an operating system refers to the mechanism by which the system determines the order in which threads from different processes or within the same process are executed by the CPU. Efficient thread scheduling is crucial for maximizing CPU utilization, ensuring fairness among threads, and providing responsive performance for applications. Thread scheduling involves various algorithms and policies to manage the execution of threads.

Here are some key aspects of thread scheduling in an OS:

1. **Scheduling Algorithms:**
   - **Pre-emptive Scheduling:** In pre-emptive scheduling, the operating system can interrupt a running thread and switch to another thread based on predefined criteria such as time slices, thread priorities, or events. Common pre-emptive scheduling algorithms include Round Robin, Priority Scheduling, and Multilevel Queue Scheduling.

   - **Non-Pre-emptive Scheduling:** Non-pre-emptive scheduling allows threads to run until they voluntarily yield the CPU, block on I/O operations, or terminate. Non-pre-emptive algorithms include First-Come, First-Served (FCFS) and Shortest Job Next (SJN).

2. **Thread Priorities:**
   - Threads are often assigned priorities indicating their importance. Higher-priority threads are given preference in execution. Priority levels can be static or dynamic, changing based on the thread's behaviour and system requirements.

3. **Time Slicing:**
   - Time slicing involves dividing the CPU time into small intervals or time slices. Each thread is allowed to execute for a fixed time slice before being pre-empted. Time slicing ensures fairness and prevents any single thread from monopolizing the CPU for too long.

4. **Affinity and Load Balancing:**
   - Thread affinity policies specify which CPU cores are eligible to run a particular thread. Load balancing algorithms distribute threads across CPU cores to optimize resource utilization and prevent overload on specific cores.

5. **Real-Time Thread Scheduling:**
   - Real-time operating systems (RTOS) implement thread scheduling for tasks with strict timing requirements. Threads in real-time systems have specific deadlines, and the scheduler ensures that these deadlines are met, making real-time systems suitable for critical applications like robotics, avionics, and industrial control.

6. **Thread States:**
   - Threads can be in various states, including running, ready, blocked, and terminated. The scheduler manages the transitions between these states, ensuring that threads are moved efficiently based on their activities and system events.

7. **Context Switching:**
   - Context switching is the process of saving the current state of a running thread and restoring the state of another thread. Efficient context switching is crucial for quick thread switching and optimal system performance.

Thread scheduling mechanisms are designed to balance competing objectives, such as maximizing throughput, minimizing response time, ensuring fairness, and meeting real-time constraints, depending on the specific requirements of the system and its applications. Different scheduling algorithms and policies are employed based on the system's workload and the desired trade-offs between these objectives.

Thread Context Switching:

Thread context switching refers to the process of saving the current state of a running thread and restoring the state of another thread for execution. Context switching is a fundamental operation in multitasking environments where multiple threads or processes share a single CPU. The operating system performs context switching to enable multitasking, allowing the CPU to execute different threads or processes in an interleaved manner.

- OS saves current state of thread & switches to another thread of same process.
- Doesn't includes switching of memory address space. (But Program counter, registers & stack are included.)
- Fast switching as compared to process switching.
- CPU's cache state is preserved.

Here's how thread context switching works:

1. **Saving the Current Thread's Context:**
   - When a running thread's time slice expires (in pre-emptive scheduling) or when the thread voluntarily yields the CPU (in cooperative scheduling), the operating system saves the thread's context. This context includes the values of CPU registers, program counter, stack pointer, and other necessary information that represents the thread's state at that moment.

2. **Selecting the Next Thread:**
   - The operating system's scheduler selects the next thread to run. This can be a thread from the same process or a different process, depending on the scheduling algorithm and policies in place.

3. **Restoring the Next Thread's Context:**
   - The saved context of the selected thread (which was saved during its previous context switch) is loaded back into the CPU registers and other relevant hardware components. This effectively restores the thread's state to the point where it left off the last time it was scheduled.

4. **Resuming Execution:**
   - The CPU starts executing instructions from the restored thread's context. The thread continues its execution from the point where it was interrupted, unaware that it was temporarily suspended.

Context switching is a complex and resource-intensive operation because it involves saving and restoring a considerable amount of data, especially when dealing with threads that have complex data structures or large stack sizes. Efficient context switching is essential for maintaining system responsiveness and ensuring that the CPU is utilized optimally across multiple threads and processes.
While context switching is a critical part of multitasking systems, excessive context switches can lead to overhead and reduced performance due to the time spent on saving and restoring thread contexts. Therefore, operating systems aim to strike a balance by optimizing context switch operations to minimize their impact on system performance.

How each thread get access to the CPU:

- Each thread has its own program counter.
- Depending upon the thread scheduling algorithm, OS schedule these threads.
- OS will fetch instructions corresponding to PC of that thread and execute instruction.

I/O or TQ, based context switching is done here as well:

- We have TCB (Thread control block) like PCB for state storage management while performing context switching.

Will single CPU system would gain by multi-threading technique:

- Never.
- As two threads have to context switch for that single CPU.
- This won't give any gain.

Multi-Threading:

Multi-threading is a programming and execution technique that allows multiple threads (smaller units of a process) to run concurrently within the same program. Threads are the smallest units of execution within a CPU, and multi-threading enables a program to perform multiple tasks or processes simultaneously, improving overall performance and responsiveness.

In a single-threaded application, tasks are executed sequentially, one after another. In a multi-threaded application, different threads can execute independently, allowing the program to perform tasks concurrently. Multi-threading can be used to achieve various goals, such as improving responsiveness, optimizing resource utilization, and enhancing the user experience.

Multi-threading is commonly used in applications that require concurrent processing, such as web servers, multimedia applications, video games, and scientific simulations. However, it also introduces challenges such as race conditions and deadlocks, which need to be carefully managed through proper synchronization techniques.

Advantages of Multi-Threading:

1. **Improved Responsiveness:**
   - Multi-threading allows applications to remain responsive to user interactions even when performing time-consuming tasks in the background. This is crucial for providing a smooth user experience, especially in graphical user interfaces and interactive applications.

2. **Increased Concurrency:**
   - Multi-threading enables concurrent execution of multiple tasks within a single program. Different threads can work on different parts of a task simultaneously, leading to more efficient use of CPU resources and faster task completion.

3. **Resource Sharing:**
   - Threads within the same process share the same memory space. This facilitates efficient data sharing and communication among threads without the need for complex inter-process communication mechanisms. Shared memory simplifies collaboration and data exchange between threads.

4. **Simplified Code Design:**
   - Multi-threading can simplify the design of complex applications. Tasks that can be naturally divided into smaller sub-tasks can be implemented as separate threads, making the code more modular and easier to manage.

5. **Parallelism on Multi-Core Processors:**
   - Modern computers often have multi-core processors. Multi-threading allows applications to take advantage of these multiple cores by executing threads in parallel. This leads to significant performance improvements for multi-threaded applications.

6. **Efficient Task Execution:**
   - Certain tasks can be parallelized, allowing multiple threads to work on them simultaneously. This is particularly beneficial for computationally intensive tasks, such as rendering, simulations, and data processing, where parallel execution leads to faster results.

7. **Faster I/O Operations:**
   - Multi-threading is beneficial for I/O-bound tasks, such as file operations and network communications. While one thread is waiting for I/O operations to complete, other threads can continue executing, making the most of the CPU's processing capabilities.

8. **Improved Scalability:**
   - Multi-threading enables applications to scale better with increasing workloads. As the number of threads increases, applications can handle more concurrent tasks efficiently, accommodating a larger number of users or processing more data.

9. **Enhanced User Experience:**
   - Multi-threading can lead to a more responsive user interface, allowing users to interact with applications seamlessly while background tasks are being executed concurrently. This responsiveness contributes to a better user experience.

10. **Optimized Resource Utilization:**
    - Multi-threading helps in optimizing the use of system resources, ensuring that the CPU is utilized to its full potential. Efficient use of resources leads to better system performance and responsiveness.

However, it's important to note that multi-threading also comes with challenges, such as race conditions, deadlocks, and increased complexity in code management. Proper synchronization mechanisms and careful design are essential to harness the advantages of multi-threading while mitigating potential issues.

Note:

- **Skips the Parts:**
  - **Critical Section Problem and How to address it.**
  - **Conditional Variable and Semaphores for Thread synchronization.**
  - **The dining philosopher's problem.**

PART 06: Deed Lock, Resource Allocation Graph, Memory Management Workflow, Virtual Memory, Thrashing

PART 07: Linux Operating System:

PART 07: Windows Operating System: