UNIT – 1

PART – 1: INTRODUCTION TO OS AND PROCESS MANAGEMENT

Introduction to Operating Systems:

An Operating System (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs.

When you start using a Computer System then it's the Operating System (OS) which acts as an interface between you and the computer hardware.

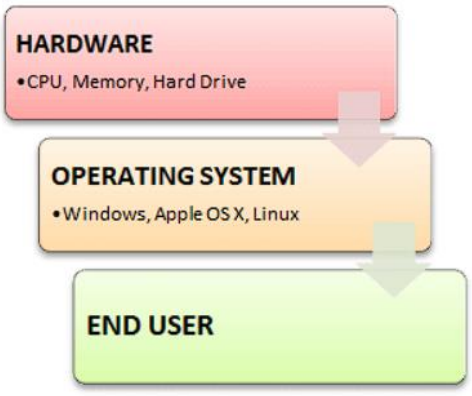
The operating system is really a low-level **Software** which is categorized as a **System Software** and supports a computer's basic functions, such as memory management, tasks scheduling and controlling peripherals etc.

Some well-known operating systems are Microsoft Windows, macOS, Linux, Unix, Android, and iOS.

Definition: ***An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.***

Every computer system must have at least one operating system to run other programs. Applications like Browsers, MS Office, Notepad Games, etc., need some environment to run and perform its tasks.

If we consider a Computer Hardware is body of the Computer System, then we can say an Operating System is its soul which brings it alive i.e., operational. We can never use a Computer System if it does not have an Operating System installed on it.



Introduction to Operating System

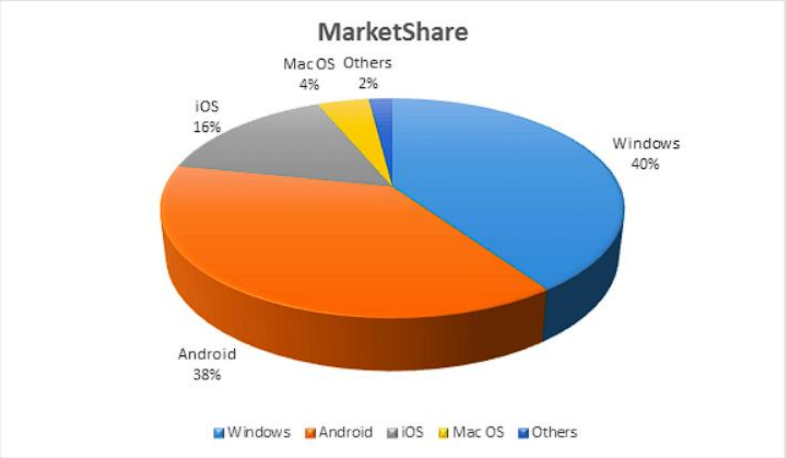
History Of OS:

- Initially, the computers made did not have an Operating system and to run each program a different code was used. This had made the processing of data more complex and time taking
- Operating systems were first developed in the late 1950s to manage tape storage
- The General Motors Research Lab implemented the first OS in the early 1950s for their IBM 701
- In the mid-1960s, operating systems started to use disks
- In the late 1960s, the first version of the Unix OS was developed
- The first OS built by Microsoft was DOS. It was built in 1981 by purchasing the 86-DOS software from a Seattle company
- The present-day popular OS Windows first came to existence in 1985 when a GUI was created and paired with MS-DOS.

Operating System Generations:

Generation	Year	Electronic device used	Types of OS Devices
First	1945-55	Vacuum Tubes	Plug Boards
Second	1955-65	Transistors	Batch Systems
Third	1965-80	Integrated Circuits (IC)	Multiprogramming
Fourth	Since 1980	Large Scale Integration	PC

Examples of Operating System with Market Share:



Market Share of Operating Systems

Types of Operating System (OS):

Given below are the different types of Operating System along with brief information about each of them:

1. Batch Operating System:

- There is no direct communication between the computer and the OS
- There is an intermediate, the Operator, which needs to distribute the work into batches and sort similar jobs
- Multiple users can use it
- Can easily manager a large amount of work

2. Real-Time Operating System:

- It has a data processing system
- The processing time is very small between the user's command and the output
- Used in fields where the response needs to be quick and rapid

3. Time-Sharing Operating System:

- Multiple people at various terminals can use a program at the same time
- The main motive is to minimize the response time

4. Distributed Operating System:

- When two or more systems are connected to each other and one can open files which are not present in their system but in other devices connected in the network
- Its usage has now increased over the years
- They use multiple central processors to serve real-time applications
- Failure of one system does not affect the other systems connected in the network

5. Embedded Operating System:

- These special Operating systems are built into larger systems
- They generally are limited to single specific functions like an ATM

6. Network Operating System:

- They have one main server which is connected to other client servers
- All the management of files, processing of data, access to sharing files, etc. are performed over this small network
- It is also a secure operating system for working with multiple users

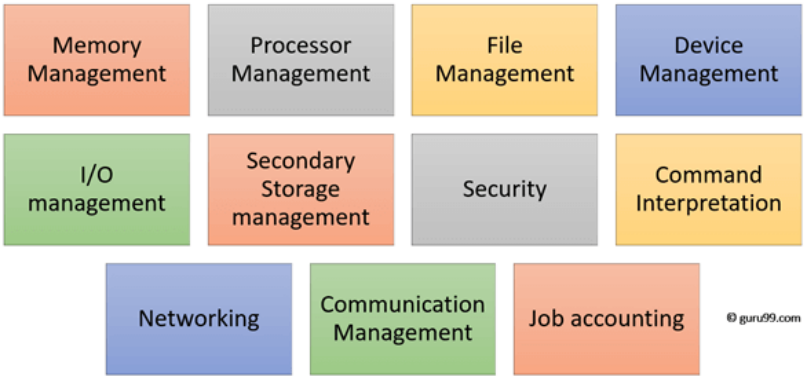
7. Mobile Operating System:

- With the advancement in the field of technology, smartphones now are released with an Operating system.
- They are designed in a manner that they can help a small device work efficiently
- Some most famous mobile operating systems are Android and iOS, but others include BlackBerry, Web, and watchOS

Functions/Characteristics of Operating System:

Some typical operating system functions may include managing memory, files, processes, I/O system & devices, security, etc.

Below are the main functions of Operating System:



1. **Process management**: Process management helps OS to create and delete processes. It also provides mechanisms for synchronization and communication among processes.
2. **Memory management:** Memory management module performs the task of allocation and de-allocation of memory space to programs in need of this resources.
3. **File management**: It manages all the file-related activities such as organization storage, retrieval, naming, sharing, and protection of files.
4. **Device Management**: Device management keeps tracks of all devices. This module also responsible for this task is known as the I/O controller. It also performs the task of allocation and de-allocation of the devices.
5. **I/O System Management:** One of the main objects of any OS is to hide the peculiarities of that hardware devices from the user.
6. **Secondary-Storage Management**: Systems have several levels of storage which includes primary storage, secondary storage, and cache storage. Instructions and data must be stored in primary storage or cache so that a running program can reference it.

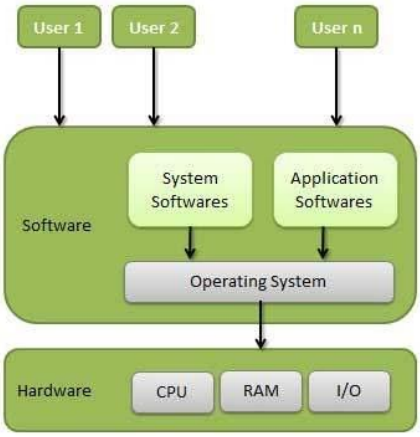
7. **Security**: Security module protects the data and information of a computer system against malware threat and authorized access.
8. **Command interpretation**: This module is interpreting commands given by the and acting system resources to process those commands.
9. **Networking:** A distributed system is a group of processors which do not share memory, hardware devices, or a clock. The processors communicate with one another through the network.
10. **Job accounting**: Keeping track of time & resource used by various job and users.
11. **Communication management**: Coordination and assignment of compilers, interpreters, and another software resource of the various users of the computer systems.

Features of Operating System (OS): Here is a list important features of OS -

- Protected and supervisor mode
- Allows disk access and file systems Device drivers Networking Security
- Program Execution
- Memory management Virtual Memory Multitasking
- Handling I/O operations
- Manipulation of the file system
- Error Detection and handling
- Resource allocation
- Information and Resource Protection

Architecture:

We can draw a generic architecture diagram of an Operating System which is as follows:



Purposes of an Operating System:

- It controls the allocation and use of the computing System's resources among the various user and tasks.
- It provides an interface between the computer hardware and the programmer that simplifies and makes it feasible for coding and debugging of application programs.

Tasks of an Operating System:

1. Provides the facilities to create and modify programs and data files using an editor.
2. Access to the compiler for translating the user program from high-level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

Advantage of Operating System:

- Allows you to hide details of hardware by creating an abstraction
- Easy to use with a GUI
- Offers an environment in which a user may execute programs/applications
- The operating system must make sure that the computer system convenient to use
- Operating System acts as an intermediary among applications and the hardware components
- It provides the computer system resources with easy-to-use format
- Acts as an intermediator between all hardware's and software's of the system

Disadvantages of Operating System:

- If any issue occurs in OS, you may lose all the contents which have been stored in your system
- Operating system's software is quite expensive for small size organization which adds burden on them. Example Windows
- It is never entirely secure as a threat can occur at any time

Components of an Operating Systems:

There are two basic components of an Operating System.

1. Shell
2. Kernel

1. **Shell:** Shell is the outermost layer of the Operating System and it handles the interaction with the user. The main task of the Shell is the management of interaction between the User and OS. Shell provides better communication with the user and the Operating System Shell does it by giving proper input to the user it also interprets input for the OS and handles the output from the OS. It works as a way of communication between the User and the OS.

2. **Kernel:** The kernel is one of the components of the Operating System which works as a core component. The rest of the components depends on Kernel for the supply of the important services that are provided by the Operating System. The kernel is the primary interface between the Operating system and Hardware.

Functions of Kernel:

The following functions are to be performed by the Kernel.

- It helps in controlling the System Calls.
- It helps in I/O Management.
- It helps in the management of applications, memory, etc.

Types of Kernel:

There are four types of Kernel that are mentioned below.

- Monolithic Kernel
- Microkernel
- Hybrid Kernel
- Exokernel

There are many types of kernels that exists, but among them, the two most popular kernels are:

1. Monolithic: A monolithic kernel is a single code or block of the program. It provides all the required services offered by the operating system. It is a simplistic design which creates a distinct communication layer between the hardware and software.

2. Microkernels: Microkernel manages all system resources. In this type of kernel, services are implemented in different address space. The user services are stored in user address space, and kernel services are stored under kernel address space. So, it helps to reduce the size of both the kernel and operating system.

Difference Between Firmware and Operating System:

Below are the Key Differences between Firmware and Operating System:

Firmware	Operating System
Define Firmware: Firmware is one kind of programming that is embedded on a chip in the device which controls that specific device.	***Define Operating System:*** OS provides functionality over and above that which is provided by the firmware.
Firmware is programs that been encoded by the manufacture of the IC or something and cannot be changed.	OS is a program that can be installed by the user and can be changed.
It is stored on non-volatile memory.	OS is stored on the hard drive.

Operating System Structure:

The operating system can be implemented with the help of various structures. The structure of the OS depends mainly on how the various standard components of the operating system are interconnected and melded into the kernel.

A design known as an operating system enables user application programs to communicate with the machine's hardware. Given its complex design and need to be easy to use and modify, the operating system should be constructed with the utmost care. A straightforward way to do this is to supernaturally develop the operating system. These parts must each have unique inputs, outputs, and functionalities.

Depending on this, we have the following structures in the operating system -

1. Simple/Monolithic Structure
2. Micro-Kernel Structure
4. Hybrid-Kernel Structure
4. Exo-Kernel Structure
6. Layered Structure
6. Modular Structure
7. Virtual Machines

What is a System Structure for an Operating System?

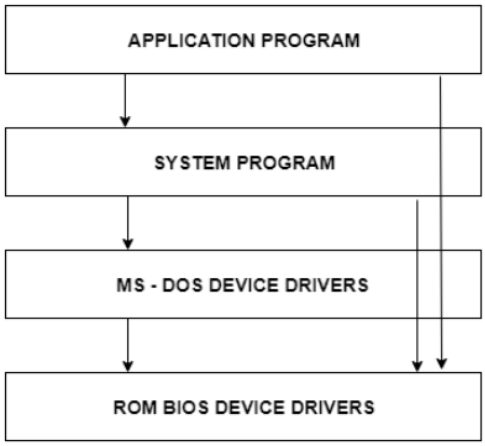
Because operating systems have complex structures, we want a structure that is easy to understand so that we can adapt an operating system to meet our specific needs. Similar to how we break down larger problems into smaller, more manageable subproblems, building an operating system in pieces is simpler. The operating system is a component of every segment. The strategy for integrating different operating system components within the kernel can be thought of as an operating system structure.

As will be discussed below, various types of structures are used to implement operating systems.

1. Simple/Monolithic Structure:

Such operating systems do not have well-defined structures and are small, simple, and limited. The interfaces and levels of functionality are not well separated. MS-DOS is an example of such an operating system. In MS-DOS, application programs are able to access the basic I/O routines. These types of operating systems cause the entire system to crash if one of the user programs fails.

A diagram of the structure of MS-DOS is shown below –



MS-DOS STRUCTURE

Advantages of Simple/Monolithic structure

- It delivers better application performance because of the few interfaces between the application program and the hardware.
- It is easy for kernel developers to develop such an operating system.

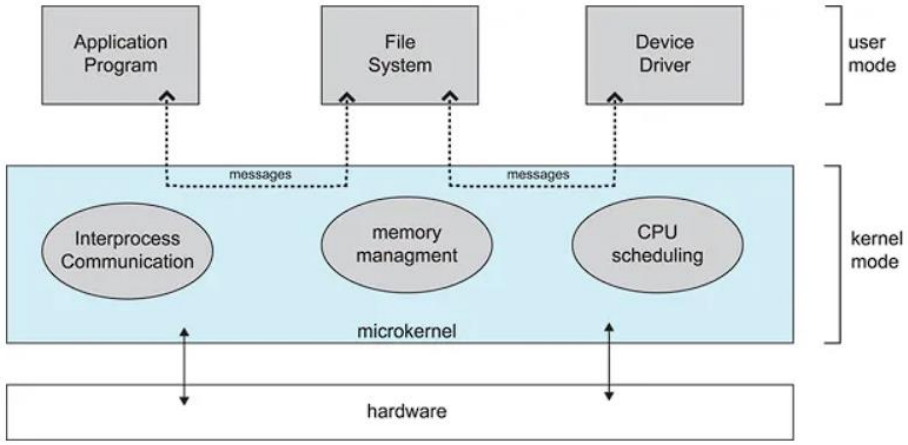
Disadvantages of Simple/Monolithic structure

- The structure is very complicated, as no clear boundaries exist between modules.
- It does not enforce data hiding in the operating system.

2. Micro-kernel Structure:

This structure designs the operating system by removing all non-essential components from the kernel and implementing them as system and user programs. This results in a smaller kernel called the micro-kernel. Advantages of this structure are that all new services need to be added to user space and does not require the kernel to be modified. Thus, it is more secure and reliable as if a service fails, then rest of the operating system remains untouched.

Mac OS is an example of this type of OS.



Architecture of a typical microkernel

Advantages of Micro-kernel structure:

- It makes the operating system portable to various platforms.
- As microkernels are small so these can be tested effectively.

Disadvantages of Micro-kernel structure:

- Increased level of inter module communication degrades system performance.

3. Hybrid-Kernel Structure:

Hybrid-kernel structure is nothing but just a combination of both monolithic-kernel structure and micro-kernel structure. Basically, it combines properties of both monolithic and micro-kernel and make a more advance and helpful approach. It implements speed and design of monolithic and modularity and stability of micro-kernel structure. Most OSes today do not strictly adhere to one architecture, but are hybrids of several.

Advantages of Hybrid-Kernel Structure:

- It offers good performance as it implements the advantages of both structure in it.
- It supports a wide range of hardware and applications.
- It provides better isolation and security by implementing micro-kernel approach.
- It enhances overall system reliability by separating critical functions into micro-kernel for debugging and maintenance.

Disadvantages of Hybrid-Kernel Structure:

- It increases overall complexity of system by implementing both structure (monolithic and micro) and making the system difficult to understand.
- The layer of communication between micro-kernel and other component increases time complexity and decreases performance compared to monolithic kernel.

4. Exo-Kernel Structure:

Exokernel is an operating system developed at MIT to provide application-level management of hardware resources. By separating resource management from protection, the exokernel architecture aims to enable application-specific customization. Due to its limited operability, exokernel size typically tends to be minimal.

The OS will always have an impact on the functionality, performance, and scope of the apps that are developed on it because it sits in between the software and the hardware. The exokernel operating system makes an attempt to address this problem by rejecting the notion that an operating system must provide abstractions upon which to base applications. The objective is to limit developers use of abstractions as little as possible while still giving them freedom.

Advantages of Exo-kernel:

- Support for improved application control.
- Separates management from security.
- It improves the performance of the application.
- A more efficient use of hardware resources is made possible by accurate resource allocation and revocation.
- It is simpler to test and create new operating systems.
- Each user-space program is allowed to use a custom memory management system.

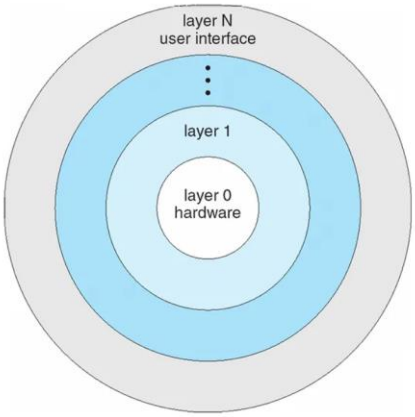
Disadvantages of Exo-kernel:

- A decline in consistency
- Exokernel interfaces have a complex architecture.

5. Layered Structure:

An OS can be broken into pieces and retain much more control over the system. In this structure, the OS is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware, and the topmost layer (layer N) is the user interface. These layers are so designed that each layer uses the functions of the lower-level layers. This simplifies the debugging process, if lower-level layers are debugged and an error occurs during debugging, then the error must be on that layer only, as the lower-level layers have already been debugged.

The main disadvantage of this structure is that at each layer, the data needs to be modified and passed on which adds overhead to the system. Moreover, careful planning of the layers is necessary, as a layer can use only lower-level layers. UNIX is an example of this structure.



Layered Operating System

Advantages of Layered structure:

- Layering makes it easier to enhance the operating system, as the implementation of a layer can be changed easily without affecting the other layers.
- It is very easy to perform debugging and system verification.

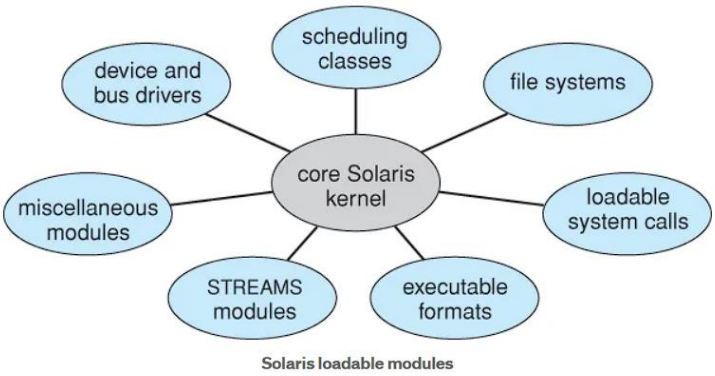
Disadvantages of Layered structure:

- In this structure, the application's performance is degraded as compared to simple structure.
- It requires careful planning for designing the layers, as the higher layers use the functionalities of only the lower layers.

6. Modular Structure or Approach:

It is considered as the best approach for an OS. It involves designing of a modular kernel. The kernel has only a set of core components and other services are added as dynamically loadable modules to the kernel either during runtime or boot time. It resembles layered structure due to the fact that each kernel has defined and protected interfaces, but it is more flexible than a layered structure as a module can call any other module.

For example, Solaris OS is organized as shown in the figure -



Solaris loadable modules

7. VMs (Virtual Machines):

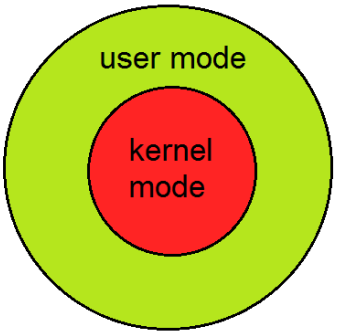
Based on our needs, a virtual machine abstracts the hardware of our personal computer, including the CPU, disc drives, RAM, and NIC (Network Interface Card), into a variety of different execution contexts, giving us the impression that each execution environment is a different computer. An illustration of it is a virtual box.

An operating system enables us to run multiple processes concurrently while making it appear as though each one is using a different processor and virtual memory by using CPU scheduling and virtual memory techniques.

The fundamental issue with the virtual machine technique is disc systems. Let's say the physical machine only has three-disc drives, but it needs to host seven virtual machines. The program that creates virtual machines would need a significant amount of disc space in order to provide virtual memory and spooling, so it should be clear that it is impossible to assign a disc drive to every virtual machine. The answer is to make virtual discs available.

Introduction to System Calls:

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.



Modes supported by the operating system:

Kernel Mode:

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

User Mode:

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

System Call:

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via **Application Program Interface (API).**

It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

Services Provided by System Calls:

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

How are system calls made?

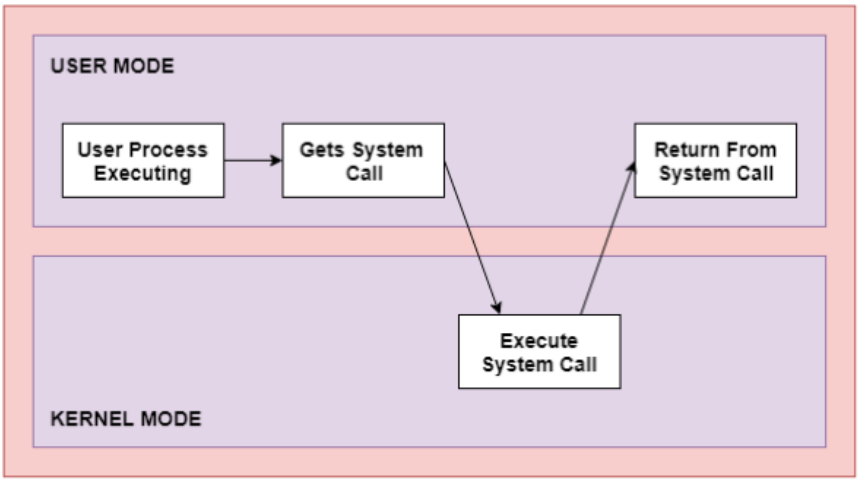
When a computer software needs to access the operating system's kernel, it makes a system call. The system call uses an API to expose the operating system's services to user programs. It is the only method to access the kernel system. All programs or processes that require resources for execution must use system calls, as they serve as an interface between the operating system and user programs.

Below are some examples of how a system call varies from a user function -

1. A system call function may create and use kernel processes to execute the asynchronous processing.
2. A system call has greater authority than a standard subroutine. A system call with kernel-mode privilege executes in the kernel protection domain.
3. System calls are not permitted to use shared libraries or any symbols that are not present in the kernel protection domain.
4. The code and data for system calls are stored in global kernel memory.

How System Call Works?

Here are the steps for System Call in OS:



As you can see in the above-given System Call example diagram.

Step 1) The processes executed in the user mode till the time a system call interrupts it.

Step 2) After that, the system call is executed in the kernel-mode on a priority basis.

Step 3) Once system call execution is over, control returns to the user mode.,

Step 4) The execution of user processes resumed in Kernel mode.

Types of System Calls: There are 5 different categories of system calls –

1. Process control**:** end, abort, create, terminate, allocate and free memory.
2. File management**:** create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

1. **Process Control:** It handles the system calls for process creation, deletion, etc. Examples of process control system calls are: Load, Execute, Abort, and Wait for Signal events for process.
2. **File Management:** File manipulation events like Creating, Deleting, Reading Writing etc are being classified under file management system calls.
3. **Device Management:** Device Management system calls are being used to request the device, release the device, and logically attach and detach the device.
4. **Information Maintenance:** This type of system call is used to maintain the information about the system like time and date.
5. **Communications:** In order to have interprocess communications like send or receive the message, create or delete the communication connections, to transfer status information etc. communication system calls are used.

Examples of Windows and Unix System Calls –

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Rules for Passing Parameters in System Call:

1. The floating-point parameters can’t be passed as a parameter in the system call.
2. Only a limited number of arguments can be passed in the system call.
3. If there are more arguments, then they should be stored in the block of memory and the address of that memory block is stored in the register.
4. Parameters can be pushed and popped from the stack only by the operating system.

Important System Calls Used in OS:

1. **wait():** There are scenarios where the current process needs to wait for another process to complete the task. So to wait in this scenario, wait() system call is used.
2. **fork():** This system call creates the copy of the process that has called it. The one which has called fork is called the parent process and the newly created copy is called the child process.
3. **exec():** This system call is called when the running process wants to execute another executable file. The process id remains the same while the other resources used by the process are replaced by the newly created process.
4. **kill():** Sometimes while working, we require to terminate a certain process. So kill system call is called which sends the termination signal to the process.
5. **exit():** When we are actually required to terminate the program, an exit system call is used. The resources that are occupied by the process are released after invoking the exit system call.

Process Concept:

What is a Process in an Operating System?

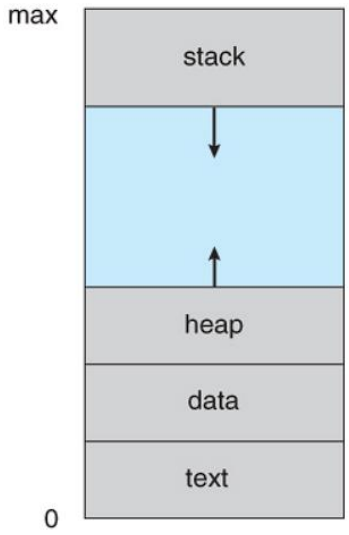
A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

The process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to the program which is considered to be a 'passive' entity. Attributes held by the process include hardware state, memory, CPU, etc.

A program can be segregated into four pieces when put into memory to become a process: **stack, heap, text, and data**. The diagram below depicts a simplified representation of a process in the main memory.

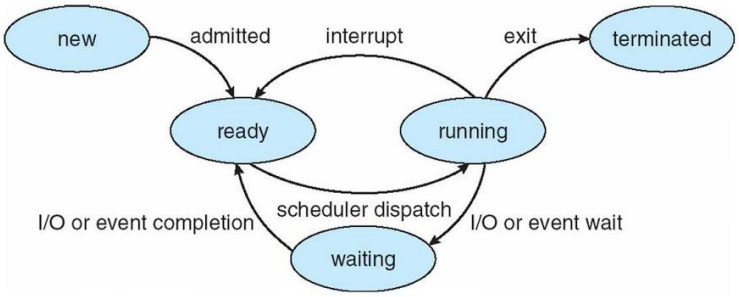
Components of a Process: It is divided into the following four sections:



1. **Stack:** Temporary data like method or function parameters, return address, and local variables are stored in the process stack.
2. **Heap:** This is the memory that is dynamically allocated to a process during its execution.
3. **Text:** This comprises the contents present in the processor's registers as well as the current activity reflected by the value of the program counter.
4. **Data:** The global as well as static variables are included in this section.

Process Life Cycle/States of Process:

When a process runs, it goes through many states. Distinct operating systems have different stages, and the names of these states are not standardized. In general, a process can be in one of the five states listed below at any given time.



Processes may be in one of 5 states, as shown in Figure above -

1. **New** - The process is in the stage of being created.
2. **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
3. **Running** - The CPU is working on this process's instructions.

4. **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example, the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
5. **Terminated -** The process has completed.

- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.

Process Control Block (PCB):

Every process has a process control block, which is a data structure managed by the operating system. An integer process ID (or PID) is used to identify the PCB. As shown below, PCB stores all of the information required to maintain track of a process.

1. Process State: The process's present state, such as whether it's ready, waiting, running, or whatever.

2. Process Privileges: This is required in order to grant or deny access to system resources.

3. Process ID: Each process in the OS has its own unique identifier.

4. Pointer: It refers to a pointer that points to the parent process.

5. Program Counter: The program counter refers to a pointer that points to the address of the process's next instruction.

6. CPU Registers: Processes must be stored in various CPU registers for execution in the running state.

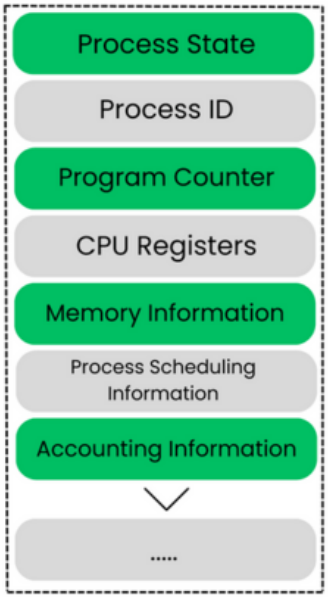
7. CPU Scheduling Information: Process priority and additional scheduling information are required for the process to be scheduled.

8. Memory Management Information: This includes information from the page table, memory limitations, and segment table, all of which are dependent on the amount of memory used by the OS.

9. Accounting Information: This comprises CPU use for process execution, time constraints, and execution ID, among other things.

10. IO Status Information: This section includes a list of the process's I/O devices.

The PCB architecture is fully dependent on the operating system, and different operating systems may include different information. A simplified diagram of a PCB is shown below –



The PCB is kept for the duration of a procedure and then removed once the process is finished.

Process vs Program:

Let us take a look at the differences between Process and Program:

Process	Program
The process is basically an instance of the computer program that is being executed.	A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer.
A process has a **shorter lifetime**.	A Program has a **longer lifetime**.
A Process requires resources such as memory, CPU, Input-Output devices.	A Program is stored by hard-disk and does not require any resources.
A process has a dynamic instance of code and data	A Program has static code and static data.
Basically, a process is the **running instance** of the code.	On the other hand, the program is the **executable code**.

Type of Process:

A process may be

- Either OS process (e.g., system call) or User process
- Either I/O bound process or CPU bound process
  - I/O bound processes: Issue lots of I/O requests and little computation
  - CPU bound processes: Use CPU frequently and I/O infrequently
- Either Independent process or Cooperating process
  - Independent processes:
    - Does not affect the execution of other processes
    - Is not affected by other processes
    - Does not share any data with other processes
  - Cooperative processes (e.g., producer/consumer example):
    - Can affect the execution of other processes
    - Can be affected by other processes
    - Share data with other processes
- Either Foreground process or Background process
  - Foreground processes:
    - Hold the terminal.
    - Can receive input and return output from/to the user
  - Background processes:
    - Detached from the terminal it was started
    - Runs without user interaction.
    - The "&" sign will allow you to execute a process as a background process in UNIX.
    - Run "netscape" and "netscape &" and observe the difference.

Operations on Processes:

There are many operations that can be performed on processes. Some of these are process creation, process preemption, process blocking, and process termination. These are given in detail as follows −

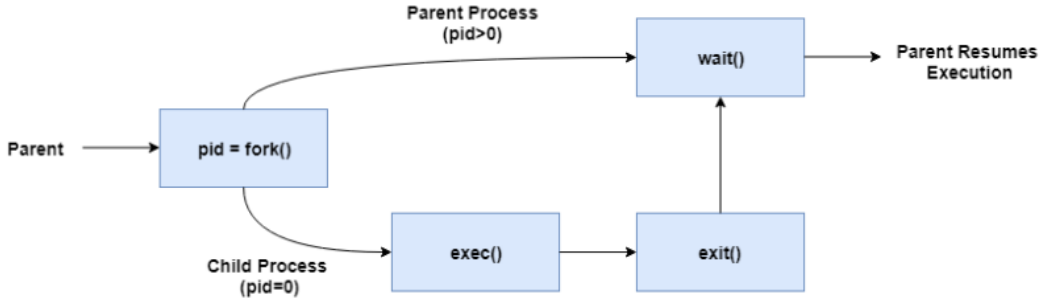
Process Creation:

Processes need to be created in the system for different operations. This can be done by the following events –

- User request for process creation
- System initialization
- Execution of a process creation system call by a running process
- Batch job initialization

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows −



Process Creation Using fork()

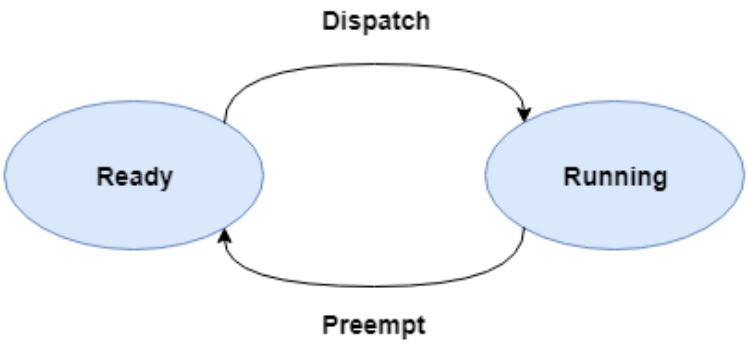
Scheduling/Dispatching:

The event or activity in which the state of the process is changed from ready to run. It means the operating system puts the process from the ready state into the running state. Dispatching is done by the operating system when the resources are free or the process has higher priority than the ongoing process. There are various other cases in which the process in the running state is preempted and the process in the ready state is dispatched by the operating system.

Process Preemption:

An interrupt mechanism is used in preemption that suspends the process executing currently and the next process to execute is determined by the short-term scheduler. Preemption makes sure that all processes get some CPU time for execution.

A diagram that demonstrates process preemption is as follows −

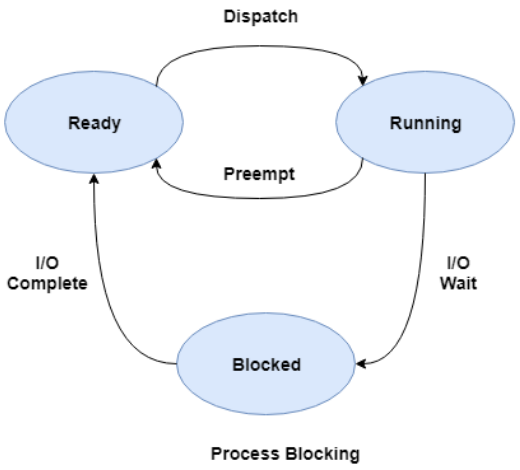


Process Preemption

Process Blocking:

The process is blocked if it is waiting for some event to occur. This event may be I/O as the I/O events are executed in the main memory and don't require the processor. After the event is complete, the process again goes to the ready state.

A diagram that demonstrates process blocking is as follows −



Process Blocking

Process Termination:

Process termination is the activity of ending the process. In other words, process termination is the relaxation of computer resources taken by the process for the execution. Like creation, in termination also there may be several events that may lead to the process of termination. Some of them are:

1. The process completes its execution fully and it indicates to the OS that it has finished.
2. The operating system itself terminates the process due to service errors.
3. There may be a problem in hardware that terminates the process.
4. One process can be terminated by another process.

Advantages of Process in Operating System:

Here are the advantages of process in operating system:

- It has efficient for resource allocation.
- They have enhanced system stability and security through process isolation.
- It is independent process execution for fault tolerance.
- These have inter-process communication for collaboration.
- It is easy to debug individual processes.

Disadvantages of Process in Operating System:

Here are the disadvantages of process in operating system:

- They have limited system resources can cause performance issues.
- It has synchronization issues can arise when processes access shared resources.
- The process of switching can consume time.
- It has increased memory usage due to process duplication.
- These have security risks from malicious processes accessing sensitive data.

Cooperating Processes:

In an operating system, everything is around the process. How the process goes through several different states. So, in this topic, we are going to discuss one type of process called as Cooperating Process. In the operating system, there are two types of processes:

- **Independent Process:** Independent Processes are those processes whose task is not dependent on any other processes.
- **Cooperating Process:** Cooperating Processes are those processes that depend on other processes or processes. They work together to achieve a common task in an operating system. These processes interact with each other by sharing the resources such as CPU, memory, and I/O devices to complete the task.

Need of Cooperating Processes:

There may be many reasons for the requirement of cooperating processes. Some of these are given as follows −

1. Modularity: Modularity involves dividing complicated tasks into smaller subtasks. These subtasks can be completed by different cooperating processes. This leads to faster and more efficient completion of the required tasks.

2. Information Sharing: Sharing of information between multiple processes can be accomplished using cooperating processes. This may include access to the same files. A mechanism is required so that the processes can access the files in parallel to each other.

3. Convenience: There are many tasks that a user needs to do such as compiling, printing, editing etc. It is convenient if these tasks can be managed by cooperating processes.

4. Computation Speedup: Subtasks of a single task can be performed parallelly using cooperating processes. This increases the computation speedup as the task can be executed faster. However, this is only possible if the system has multiple processing elements.

Methods of Cooperating Process:

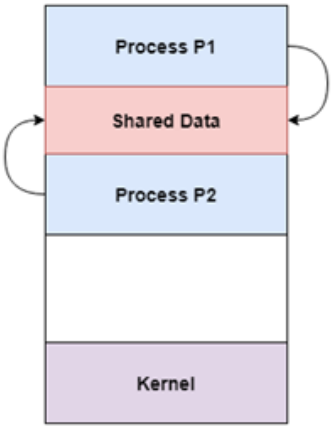
Cooperating processes in OS requires a communication method that will allow the processes to exchange data and information.

There are two methods by which cooperating process in OS can communicate:

1. Cooperation by Sharing:

The cooperating processes can cooperate with each other using shared data such as memory, variables, files, databases etc. Critical section is used to provide data integrity and writing is mutually exclusive to prevent inconsistent data.

A diagram that demonstrates cooperation by sharing is given as follows −

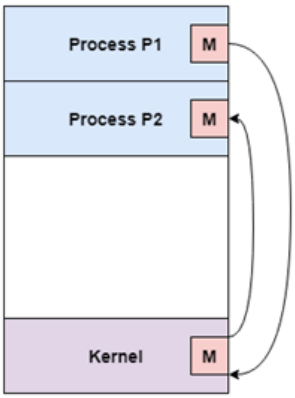


In the above diagram, Process P1 and P2 can cooperate with each other using shared data such as memory, variables, files, databases etc.

2. Cooperation by Communication:

The cooperating processes can cooperate with each other using messages. This may lead to deadlock if each process is waiting for a message from the other to perform a operation. Starvation is also possible if a process never receives a message.

A diagram that demonstrates cooperation by communication is given as follows −



In the above diagram, Process P1 and P2 can cooperate with each other using messages to communicate.

Advantages of Cooperating Process:

Let's discuss several advantages of cooperating processes in the operating system:

- With help of data and information sharing, the processes can be executed with much faster speed and efficiency as processes can access the same files concurrently.
- Modularity gives the advantage of breaking a complex task into several modules which are later put together to achieve the goal of faster execution of processes.
- Cooperating processes provide convenience as different processes running at the same time can cooperate without any interruption among them.
- The computation speed of the processes increases by dividing processes into different subprocesses and executing them parallelly at the same time.

Disadvantages of Cooperating Process:

Let's discuss the disadvantages of cooperating processes in the operating system:

- During the communication method of the cooperation processes, there may be a problem of **deadlock** if a consumer process waits for the message from the production process and the message does not receive at the consuming end.
- There may be a condition of process starvation where the next process will have to wait until the message is received by the previous consuming process.
- The cooperating processes in the operating system can damage the data which may occur due to **modularity**.
- During information sharing, it may also share any sensitive data of the user with the other process that the user might not want to share.

Example of Cooperating Process:

Let's take the example of the **producer-consumer** problem which is also known as a **bounded buffer problem** to understand cooperating processes in more detail:

Producer: The process which generates the message that a consumer may consume is known as the **producer**.

Consumer: The process which consumes the information generated by the produces.

A producer produces a piece of information and stores it in a **buffer** (**critical section**) and the consumer consumes that information.

For **Example**, A web server produces web pages that are consumed by the client. A compiler produces an assembly code that is consumed by the assembler.

Buffer memory can be of two types:

- **Unbounded buffer:** It is a kind of buffer that has no practical limit on the size buffer. The producer can produce new information but the consumer might have to wait for them.
- **Bounded buffer:** It is a kind of buffer that assumes a fixed size. Here, the consumer has to wait if the buffer is empty, while the producer has to wait if the buffer is full. But here in the **process-consumer** problem, we have used a **bounded buffer**.

Producer and consumer both processes execute simultaneously. The problem arises when a consumer wants to consume information when the buffer is **empty** or there is nothing to be consumed and a producer produces a piece of information when the buffer is **full** or the memory of the consumer is already full.

Producer Process:

```
while(true)
    {
        produce an item &
        while(counter = = buffer-size);
        buffer[int] = next produced;
        in = (in+1) % buffer- size;
        counter ++;
    }
```

Consumer Process:

```
while(true)
    {
        while (counter = = 0);
        next consumed = buffer[out];
        out= (out+1) % buffer size;
        counter--;
    }
```

In the above **producer code** and **consumer code**, we have the following variables:

- **counter**: counter is used to identify the size of the buffer which is used by the producer as well as consumer processes.
- **in:** in a variable is used by the producer to detect the next empty slot in the buffer region.
- **out:** The consumer uses the out variable to detect where the items are stored.

Shared Resources:

In the **process-consumer** problem we have used two types of shared resources:

1. Buffer
2. Counter

When both the producer process and consumer process do not execute on time then it may cause inconsistency in the process. The value of the counter variable will be **incorrect** if both the producer and consumer processes will be executed **concurrently**.

Producer and consumer processes both shares the following variables:

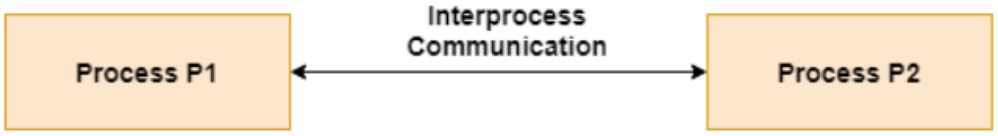
```
var n;
    type item = .....;
    var Buffer : array [0,n-1] of item;
    in, out:0..n-1;
```

The shared buffer region holds two logical pointers i.e, **in** and **out**, and are implemented in the form circular array. By default, the values of both the variables (**in** and **out**) are initialized to 0. As we discussed earlier, the **out** variable points to the first filled location in the buffer while the **in** variable points to the first free location in the buffer. The buffer will be empty if in=out. The buffer will be filled if in+1 mod n=out.

Inter-Process Communication:

Inter-process communication is the mechanism provided by the **operating system** that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows –



Why Inter Process Communication (IPC) is Required?

IPC helps achieve these things:

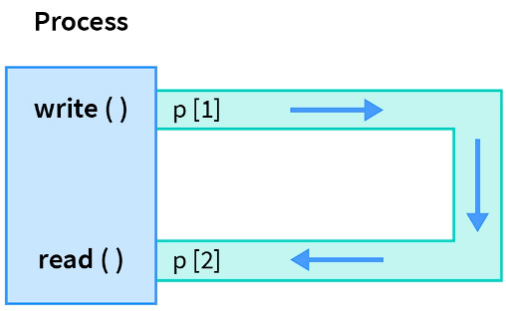
- Computational Speedup
- Modularity
- Information and data sharing
- Privilege separation
- Processes can communicate with each other and synchronize their action.

Approaches to Inter-process Communication:

The different approaches to implement inter-process communication are given as follows –

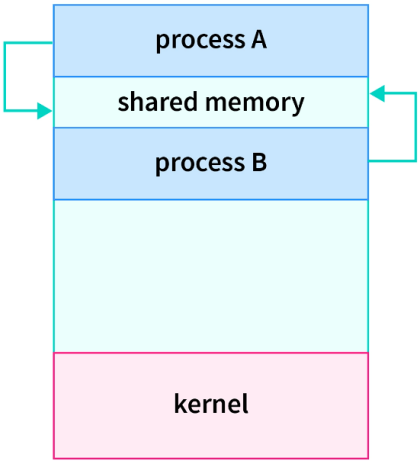
Pipes:

- It is a half-duplex method (or one-way communication) used for IPC between two related processes.
- It is like a scenario like filling the water with a tap into a bucket. The filling process is writing into the pipe and the reading process is retrieved from the pipe.



Shared Memory:

Multiple processes can access a common shared memory. Multiple processes communicate by shared memory, where one process makes changes at a time and then others view the change. Shared memory does not use kernel.

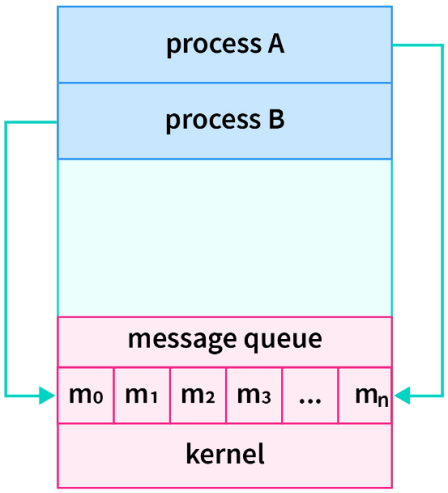


Message Passing:

- In IPC, this is used as a process for communication and synchronization.
- Processes can communicate without any shared variables, therefore it can be used in a distributed environment on a network.
- It is slower than the shared memory technique.
- It has two actions sending (fixed size message) and receiving messages.

Message Queues:

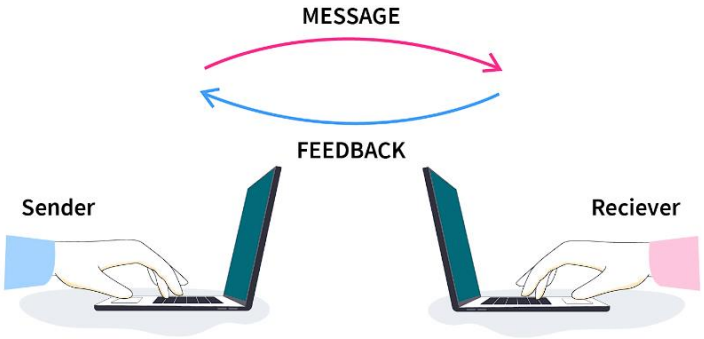
We have a linked list to store messages in a kernel of OS and a message queue is identified using "message queue identifier".



Direct Communication:

In this, the process that wants to communicate must name the sender or receiver.

- A pair of communicating processes must have one link between them.
- A link (generally bi-directional) is established between every pair of communicating processes.



Indirect Communication:

- Pairs of communicating processes have shared mailboxes.
- Link (uni-directional or bi-directional) is established between pairs of processes.
- The sender process puts the message in the port or mailbox of a receiver process and the receiver process takes out (or deletes) the data from the mailbox.

FIFO:

- Used to communicate between two processes that are not related.
- Full-duplex method - Process P1 is able to communicate with Process P2, and vice versa.

What is Like FIFOS and Unlike FIFOS?

Like FIFOS	Unlike FIFOS
FIFO or first in first out method is followed	There are methods to remove urgent messages before these messages arrive at the front.
Both receiving and sending processes are not required for FIFO to occur.	It is always ready, therefore opening or closing is not required.
Data transfer is allowed to occur among processes that are unrelated.	There are no synchronization problems between opening and closing.

Advantages of Inter-Process Communication (IPC):

- **Data Sharing:** IPC allows processes to share data with each other. This can be useful in situations where one process needs to access data that is held by another process.
- **Resource Sharing:** IPC allows processes to share resources such as memory, files, and devices. This can help reduce the amount of memory or disk space that is required by a system.
- **Synchronization:** IPC allows processes to synchronize their activities. For example, one process may need to wait for another process to complete its task before it can continue.
- **Modularity:** IPC allows processes to be designed in a modular way, with each process performing a specific task. This can make it easier to develop and maintain complex systems.
- **Scalability:** IPC allows processes to be distributed across multiple systems, which can help improve performance and scalability.

Disadvantages of Inter-Process Communication (IPC):

- **Complexity:** IPC can add complexity to the design and implementation of software systems, as it requires careful coordination and synchronization between processes. This can lead to increased development time and maintenance costs.
- **Overhead:** IPC can introduce additional overhead, such as the need to serialize and deserialize data, and the need to synchronize access to shared resources. This can impact the performance of the system.
- **Scalability:** IPC can also limit the scalability of a system, as it may be difficult to manage and coordinate large numbers of processes communicating with each other.
- **Security:** IPC can introduce security vulnerabilities, as it creates additional attack surfaces for malicious actors to exploit. For example, a malicious process could attempt to gain unauthorized access to shared resources or data.
- **Compatibility:** IPC can also create compatibility issues between different systems, as different operating systems and programming languages may have different IPC mechanisms and APIs. This can make it difficult to develop cross-platform applications that work seamlessly across different environments.

Examples of IPC Systems:

1. **POSIX:** uses shared memory method.
2. **Mach:** uses message passing
3. **Windows XP:** uses message passing using local procedural calls

Role of Synchronization in Inter Process Communication:

It is one of the essential parts of inter process communication. Typically, this is provided by inter-process communication control mechanisms, but sometimes it can also be controlled by communication processes.

These are the following methods that used to provide the synchronization:

1. Mutual Exclusion: It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.

2. Semaphore: Semaphore is a type of variable that usually controls the access to the shared resources by several processes. Semaphore is further divided into two types which are as follows:

1) **Binary Semaphore**: A synchronization primitive with two states (typically 0 and 1), used to protect access to a single shared resource, ensuring that only one process can access the resource at a time.
2) **Counting Semaphore**: A synchronization primitive with a non-negative integer value, employed to control access to multiple instances of a resource or limit the number of concurrent processes or threads that can access a specific resource.

3. Barrier: A barrier typically not allows an individual process to proceed unless all the processes don't reach it. It is used by many parallel languages, and collective routines impose barriers.

4. Spinlock: Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not. It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

Mutual Exclusion:

Mutual exclusion also known as Mutex is a unit of code that avert contemporaneous access to shared resources. Mutual exclusion is concurrency control's property that is installed for the objective of averting race conditions.

In simple words, it's a condition in which a thread of execution does not ever get involved in a critical section at the same time as a concurrent thread of execution so far using the critical section. This critical section can be a period for which the thread of execution uses the shared resource which can be defined as a data object, that different concurrent threads maybe attempt to alter (where the number of concurrent read operations allowed is two but on the other hand two write or one read and write is not allowed, as it may guide it to **data instability**).

Mutual exclusion in OS is designed so that when a write operation is in the process then another thread is not granted to use the very object before the first one has done writing on the critical section after that releases the object because the rest of the processes have to read and write it.

Why is Mutual Exclusion Required?

An easy example of the importance of Mutual Exclusion can be envisioned by implementing a linked list of multiple items, considering the fourth and fifth need removal. The deletion of the node which sits between the other two nodes is done by modifying the previous node's next reference directing the succeeding node.

In a simple explanation, whenever node "i" want to be removed, at that moment node "ith - 1" 's next reference is modified, directing towards the node "ith + 1". Whenever a shared linked list is in the middle of many threads, two separate nodes can be removed by two threads at the same time meaning the first thread modifies node "ith - 1" next reference, directing towards the node "ith + 1", at the same time second thread modifies node "ith" next reference, directing towards the node "ith + 2". Despite the removal of both achieved, linked lists required state is not yet attained because node "i + 1" still exists in the list, due to node "ith - 1" next reference still directing towards the node "i + 1".

Now, this situation is called a race condition. **Race conditions** can be prevented by mutual exclusion so that updates at the same time cannot happen to the very bit about the list.

Necessary Conditions for Mutual Exclusion:

There are four conditions applied to mutual exclusion, which are mentioned below:

- Mutual exclusion should be ensured in the middle of different processes when accessing shared resources. There must not be two processes within their critical sections at any time.
- Assumptions should not be made as to the respective speed of the unstable processes.
- The process that is outside the critical section must not interfere with another for access to the critical section.
- When multiple processes access its critical section, they must be allowed access in a finite time, i.e., they should never be kept waiting in a loop that has no limits.

Approaches To Implementing Mutual Exclusion:

1. Software method: Leave the responsibility to the processes themselves. These methods are usually highly error-prone and carry high overheads.

2. Hardware method: Special-purpose machine instructions are used for accessing shared resources. This method is faster but cannot provide a complete solution. Hardware solutions cannot give guarantee the absence of deadlock and starvation.

3. Programming language method: Provide support through the operating system or through the programming language.

Example of Mutual Exclusion:

There are many types of mutual exclusion, some of them are mentioned below:

- **Locks:** It is a mechanism that applies restrictions on access to a resource when multiple threads of execution exist.
- **Recursive lock:** It is a certain type of mutual exclusion (mutex) device that is locked several times by the very same process/thread, without making a deadlock. While trying to perform the "lock" operation on any mutex may fail or block when the mutex is already locked, while on a recursive mutex the operation will be a success only if the locking thread is the one that already holds the lock.
- **Semaphore:** It is an abstract data type designed to control the way into a shared resource by multiple threads and prevents critical section problems in a concurrent system such as a multitasking operating system. They are a kind of **synchronization primitive**.
- **Readers writer (RW) lock:** It is a synchronization primitive that works out reader-writer problems. It grants concurrent access to the read-only processes, and writing processes require exclusive access. This conveys that multiple threads can read the data in parallel however exclusive lock is required for writing or making changes in data. It can be used to manipulate access to a data structure inside the memory.

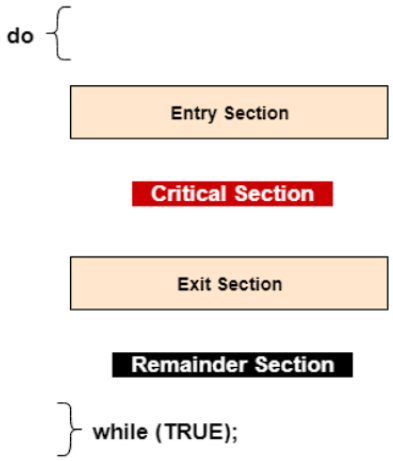
What is a Race Condition?

A race condition occurs when multiple processes or threads access shared data concurrently, and the final outcome of the program depends on the relative timing of their execution. This can lead to unexpected and undesired results, such as data corruption, incorrect calculations, or program crashes.

Critical Section Problem:

What is the Critical Section?

- Critical Section refers to the segment of code or the program that tries to access or modify the value of the variables in a shared resource.
- The section above the critical section is called the **Entry Section**. The process that is entering the critical section must pass the entry section.
- The section below the critical section is called the **Exit Section**.
- The section below the exit section is called the **Reminder Section** and this section has the remaining code that is left after execution.



What is the Critical Section Problem?

When there is more than one process accessing or modifying a shared resource at the same time, then the value of that resource will be determined by the last process. This is called the **race condition**.

Consider an **example** of two processes, p1 and p2. Let value=3 be a variable present in the shared resource.

Let us consider the following actions are done by the two processes,

```
value+3  // process p1
value=6
value-3  // process p2
value=3
```

The original value of, value should be 6, but due to the interruption of the process p2, the value is changed back to 3. This is the problem of synchronization.

The **critical section problem** is to make sure that only one process should be in a critical section at a time. When a process is in the critical section, no other processes are allowed to enter the critical section. This solves the race condition.

Example of Critical Section Problem:

Let us consider a classic bank example, this example is very similar to the example we have seen above.

- Let us consider a scenario where money is withdrawn from the bank by both the cashier (through cheque) and the ATM at the same time.
- Consider an account having a balance of ₹10,000. Let us consider that when a cashier withdraws the money, it takes 2 seconds for the balance to be updated in the account.
- It is possible to withdraw ₹7000 from the cashier and within the balance update time of 2 seconds, also withdraw an amount of ₹6000 from the ATM.
- Thus, the total money withdrawn becomes greater than the balance of the bank account.

This happened because of two withdrawals occurring at the same time. In the case of the critical section, only one withdrawal should be possible and it can solve this problem.

Solutions to the Critical Section Problem:

A solution developed for the critical section should have the following properties,

- If a process enters the critical section, then no other process should be allowed to enter the critical section. This is called **mutual exclusion**.
- If a process is in the critical section and another process arrives, then the new process must wait until the first process exits the critical section. In such cases, the process that is waiting to enter the critical section should not wait for an unlimited period. This is called **progress**.
- If a process wants to enter into the critical section, then there should be a specified time that the process can be made to wait. This property is called **bounded waiting**.
- The solution should be independent of the system's architecture. This is called **neutrality**

Some of the software-based solutions for critical section problems are **Peterson's solution**, **semaphores**, **monitors**. Some of the hardware-based solutions for the critical section problem involve atomic instructions such as **TestAndSet**, **compare and swap**, **Unlock and Lock**.

Software-Based Solutions:

Semaphores: A semaphore is a data structure that is used to control access to shared resources. It is typically implemented as a counter that is incremented or decremented when a process enters or exits the critical section. When the counter reaches zero, no other process is allowed to enter the critical section.

Monitors: A monitor is a software construct that provides a way for processes to synchronize access to shared resources. It is essentially a collection of procedures, each of which defines a critical section, and a mechanism for controlling access to those procedures. Monitors are typically implemented using semaphores.

Peterson's Algorithm: Peterson's Algorithm is a solution to the Critical Section Problem in OS for two processes. It uses shared memory and atomic instructions (such as test-and-set) to ensure that only one process can enter the critical section at a time.

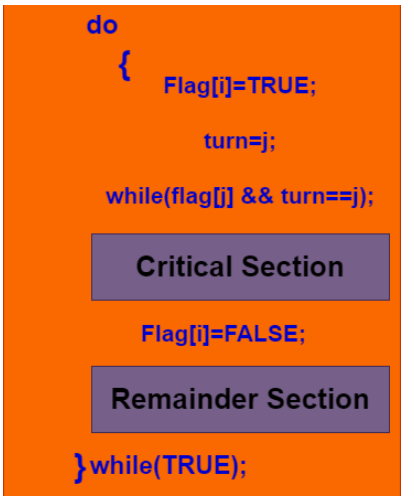
Peterson's Solution:

This is widely used and software-based solution to critical section problems. Peterson's solution was developed by a computer scientist Peterson that's why it is named so.

With the help of this solution whenever a process is executing in any critical state, then the other process only executes the rest of the code, and vice-versa can happen. This method also helps to make sure of the thing that only a single process can run in the critical section at a specific time.

This solution preserves all three conditions:

- Mutual Exclusion is comforted as at any time only one process can access the critical section.
- Progress is also comforted, as a process that is outside the critical section is unable to block other processes from entering into the critical section.
- Bounded Waiting is assured as every process gets a fair chance to enter the Critical section.



The above shows the structure of process **Pi in Peterson's solution.**

- Suppose there are **N processes (P1, P2, ... PN)** and as at some point of time every process requires to enter in the **Critical Section**
- A **FLAG[]** array of size N is maintained here which is by default false. Whenever a process requires to enter in the critical section, it has to set its flag as true. Example: If Pi wants to enter it will set **FLAG[i]=TRUE.**
- Another variable is called **TURN** and is used to indicate the process number that is currently waiting to enter into the critical section.
- The process that enters into the critical section while exiting would change the **TURN** to another number from the list of processes that are ready.
- Example: If the turn is 3 then P3 enters the Critical section and while exiting turn=4 and therefore P4 breaks out of the wait loop.

Lamport's Bakery Algorithm: Lamport's Bakery Algorithm is a solution to the Critical Section Problem in OS for multiple processes. It uses shared memory and atomic instructions (such as test-and-set) to ensure that only one process can enter the critical section at a time.

Software and Hardware Techniques: Software and Hardware techniques such as spin locks, lock-free data structures, and atomic operations can be used to solve the Critical Section Problem in OS.

Hardware-Based Solutions:

test_and_set: Uses a shared boolean variable lock and the test_and_set instruction that executes atomically and sets the passed parameter value to true.

compare_and_swap: Same as test_and_set, except that the passed parameter value is set to true if it is equal to expected in the parameter of the compare_and_swap instruction.

Mutex locks: Software method that provides the acquire() and release() functions that execute atomically.

Condition variables: Utilizes a queue of processes that are waiting to enter the critical section.

Pseudocode of Critical Section Problem:

The concept of a critical section problem can be illustrated with the following pseudocode -

```
while(true){
// entry section
    process_entered=1;
//loop until process_eneterd in 0
    while(process_entered);
        // critical section
// exit section
    process_entered=0
// remainder section

    }
```

Consider a process is in the critical region and a new process wants to enter the critical section. In this case, the new process will be made to wait in the entry section by the while loop. The while loop will continue to stall the process until the variable process_entered is made 0 by the process exiting from the critical section in the exit section.

Thus, this mechanism prevents two processes from entering the critical section at the same time.

Synchronization Hardware:

Synchronization hardware is a hardware-based solution to resolve the critical section problem. In our earlier content of the critical section, we have discussed how the multiple processes sharing common resources must be synchronized to avoid inconsistent results.

Well, we can synchronize the processes sharing a common variable in two ways. First is the **software-based solution** which includes Peterson's solution and the second is the **hardware-based solution** which is also referred to as synchronization hardware or hardware synchronization.

Synchronization hardware i.e. hardware-based solution for the critical section problem which introduces the **hardware instructions** that can be used to resolve the critical section problem effectively. Hardware solutions are often easier and also improves the efficiency of the system.

Consider, a system with a single processor where just by preventing the occurrence interrupts while a shared variable is being modified can assure that only the current sequence of the instructions will be executed without any preemption.

This assures you that no unfair modifications will be made to the shared variable. This method is often used by the systems with non-preemptive kernels.

Now, consider the system with multiple processors, where disabling interrupts would require a message to be sent to all the processors which would cause unnecessary delay to the processes to enter into their critical section. This also decreases the efficiency of the system.

The hardware-based solution to critical section problem is based on a simple tool i.e. **lock.** The solution implies that before entering into the critical section the process must acquire a lock and must release the lock when it exits its critical section. Using of lock also prevent the ***race condition***.

The hardware synchronization provides two kinds of hardware instructions that are **TestAndSet** and **Swap.** We will discuss each of the instruction briefly. But before jumping on to the instructions let us discuss what conditions they must verify to resolve the critical section problem.

1. **Mutual Exclusion:** The hardware instruction must verify that at a point in time only one process can be in its critical section.
2. **Bounded Waiting:** The processes interested to execute their critical section must not wait for long to enter their critical section.
3. **Progress:** The process not interested in entering its critical section must not block other processes from entering into their critical section.

There are three hardware approaches to solve process synchronization problems:

1. Test and Set: ***Test and set algorithm*** uses a boolean variable '**lock**' which is initially initialized to false. This lock variable determines the entry of the process inside the critical section of the code. Let's first see the algorithm and then try to understand what the algorithm is doing.

```
boolean lock = false;

boolean TestAndSet(boolean &target){
    boolean returnValue = target;
    target = true;
    return returnValue;
}

while(1){
while(TestAndSet(lock));

CRITICAL SECTION CODE;
lock = false;
REMAINDER SECTION CODE;

}
```

In the above algorithm the **TestAndSet()** function takes a boolean value and returns the same value. **TestAndSet()** function sets the lock variable to true.

When lock varibale is initially false the **TestAndSet(lock)** condition checks for TestAndSet(false). As TestAndSet function returns the same value as its argument, **TestAndSet(false)** returns false. Now, while loop **while(TestAndSet(lock))** breaks and the process enters the critical section.

As one process is inside the critical section and lock value is now '**true**', if any other process tries to enter the critical section then the new process checks for **while(TestAndSet(true))** which will return **true** inside while loop and as a result the other process keeps executing the while loop.

```
while(true); // this keeps executing until lock becomes false.
```

As no queue is maintained for the processes stuck in the while loop, bounded waiting is not ensured. **If a process waits for a set amount of time before entering the critical section, it is said to be a bounded waiting condition.**

In test and set algorithm the incoming process trying to enter the critical section does not wait in a queue so any process may get the chance to enter the critical section as soon as the process finds the lock variable to be false. It may be possible that a particular process never gets the chance to enter the critical section and that process waits indefinitely.

2. Swap: ***Swap function*** uses two boolean variables lock and key. Both lock and key variables are initially initialized to false. **Swap algorithm is the same as lock and set algorithm. The Swap algorithm uses a temporary variable to set the lock to true when a process enters the critical section of the program.**

Let's see the swap algorithm pseudo-code:

```
boolean lock = false;
individual key = false;

void swap(boolean &a, boolean &b){
    boolean temp = a;
    a = b;
    b = temp;
}

while(1){
    key=true;
    while(key){
        swap(lock,key);
    }

    CRITICAL SECTION CODE
    lock = false;
    REMAINDER SECTION CODE
}
```

In the code above when a process P1 enters the critical section of the program it first executes the while loop

```
while(key){
        swap(lock,key);
    }
```

As key value is set to true just before the for loop so swap(lock, key) swaps the value of lock and key. Lock becomes true and the key becomes false. In the next iteration of the while loop breaks and the process, P1 enters the critical section.

The value of lock and key when P1 enters the critical section is lock = true and key = false.

Let's say another process, P2, tries to enter the critical section while P1 is in the critical section. Let's take a look at what happens if P2 tries to enter the critical section.

key is set to true again after the first while loop is executed i.e. while(1). Now, the second while loop in the program i.e., while(key) is checked. As key is true the process enters the second while loop. swap(lock, key) is executed again. as both key and lock are true so after swapping also both will be true. So, the while keeps executing and the process P2 keeps running the while loop until Process P1 comes out of the critical section and makes **lock false**.

When Process P1 comes out of the critical section the value of lock is again set to false so that other processes can now enter the critical section.

When a process is inside the critical section than all other incoming process trying to enter the critical section is not maintained in any order or queue. So any process out of all the waiting process can get the chance to enter the critical section as the lock becomes false. So, there may be a process that may wait indefinitely. So, **bounded waiting is not ensured in Swap algorithm also**.

3. Unlock and lock:

Unlock and lock algorithm uses the TestAndSet method to control the value of lock. Unlock and lock algorithm uses a variable waiting[i] for each process i. Here i is a positive integer i.e 1,2,3,... which corresponds to processes P1, P2, P3... and so on. waiting[i] checks if the process i is waiting or not to enter into the critical section.

All the processes are maintained in a ready queue before entering into the critical section. The processes are added to the queue with respect to their process number. The queue is the circular queue.

Let's see the Unlock and lock algorithm pseudo-code first:

```
boolean lock = false;
Individual key = false;
Individual waiting[i];

boolean TestAndSet(boolean &target){
    boolean returnValue = target;
    target = true;
    return returnValue;
}

while(1){
    waiting[i] = true;
    key = true;
    while(waiting[i] && key){
        key = TestAndSet(lock);
    }
    CRITICAL SECTION CODE
    j = (i+1) % n;
    while(j != i && !waiting[j])
         j = (j+1) % n;
    if(j == i)
         lock = false;
    else
         waiting[j] = false;
    REMAINDER SECTION CODE
}
```

In Unlock and lock algorithm the lock is not set to false as one process comes out of the critical section. In other algorithms like swap and Test and set the lock was being set to false as the process comes out of the critical section so that any other process can enter the critical section.

But in Unlock and lock, once the ith process comes out of the critical section the algorithm checks the waiting queue for the next process waiting to enter the critical section i.e. jth process. If there is a jth process waiting in the ready queue to enter the critical section, the waiting[j] of the jth process is set to false so that the while loop while(waiting[i] && key) becomes false and the jth process enters the critical section.

If no process is waiting in the ready queue to enter the critical section the algorithm, then sets the lock to false so that any other process comes and enters the critical section easily.

Since a ready queue is always maintained for the waiting processes, the Unlock and lock algorithm ensures bounded waiting.

Advantages of Hardware Instruction:

- Hardware instructions are easy to implement and improves the efficiency of the system.
- Supports any number of processes may it be on the single or multiple processor system.
- With hardware instructions, you can implement multiple critical sections each defined with a unique variable.

Disadvantages of Hardware Instruction:

- Processes waiting for entering their critical section consumes a lot of processors time which increases busy waiting.
- As the selection of processes to enter their critical section is arbitrary. It may happen that some processes are waiting for the indefinite time which leads to process starvation.
- Deadlock is also possible.

This is how we can use a hardware solution to overcome the critical section problem. But overviewing the drawback of hardware solution there we have to opt for some other mechanism i.e., semaphore.

Semaphores:

What is Semaphore?

- **Semaphore** is simply a variable that is non-negative and shared between threads.

- The least value for a Semaphore is zero (0). The Maximum value of a Semaphore can be anything.

- A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.

- It uses two atomic operations, 1) Wait, and 2) Signal for the process synchronization.

- A semaphore either allows or disallows access to the resource, which depends on how it is set up.

Characteristic of Semaphore:

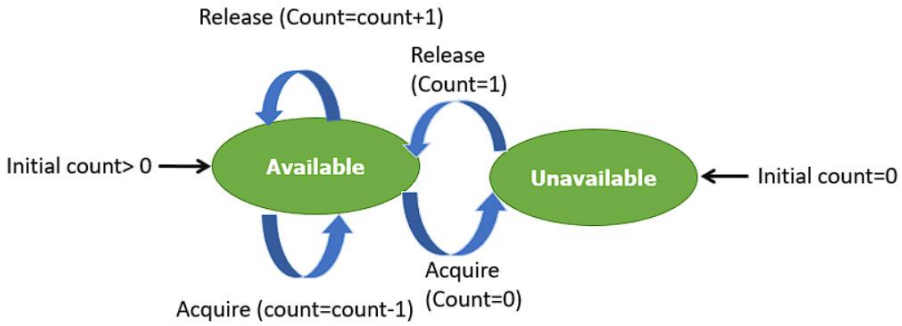
Here, are characteristic of a semaphore:

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- Semaphore will always hold a non-negative integer value.
- Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

Types of Semaphores: The two common kinds of semaphores are -

1. Counting Semaphores:

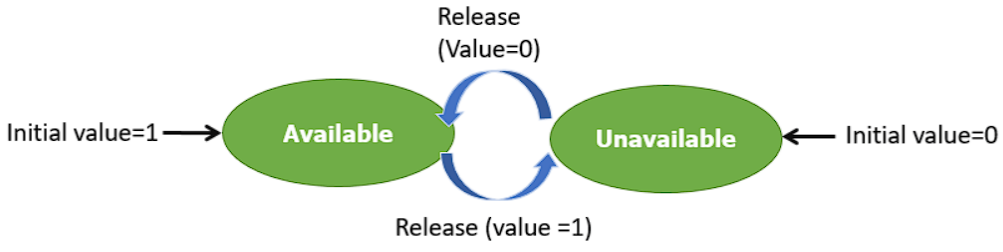
This type of Semaphore uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state.



However, If the count is > 0, the semaphore is created in the available state, and the number of tokens it has equals to its count.

2. Binary Semaphores:

The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0. It is easy to implement than counting semaphores.



Example of Semaphore:

The below-given program is a step-by-step implementation, which involves usage and declaration of semaphore.

```
Shared var mutex: semaphore = 1;
Process i
    begin
    .
    .
    P(mutex);
    execute CS;
    V(mutex);
    .
    .
    End;
```

Wait and Signal Operations/Procedures in Semaphores:

Both of these operations are used to implement process synchronization. The goal of this semaphore operation is to get mutual exclusion.

Wait for Operation:

This type of semaphore operation helps you to control the entry of a task into the critical section. However, If the value of wait is positive, then the value of the wait argument X is decremented. In the case of negative or zero value, no operation is executed. It is also called P(S) operation.

After the semaphore value is decreased, which becomes negative, the command is held up until the required conditions are satisfied.

```
Copy CodeP(S)
{
  while (S<=0);
  S--;
}
```

The Wait for Operation is also known as:

1. Sleep Operation
2. Down Operation
3. Decrease Operation
4. P Function (most important alias name for wait operation)

Signal Operation:

This type of Semaphore operation is used to control the exit of a task from a critical section. It helps to increase the value of the argument by 1, which is denoted as V(S).

```
Copy CodeP(S)
{
  while (S>=0);
  S++;
}
```

The Signal Operation is also known as:

1. Wake up Operation 2. Up Operation

3. Increase Operation 4. V Function (most important alias name for signal operation)

Counting Semaphore vs. Binary Semaphore:

Criteria	Binary Semaphore	Counting Semaphore
Definition	A Binary Semaphore is a semaphore whose integer value range over 0 and 1.	A counting semaphore is a semaphore that has multiple values of the counter. The value can range over an unrestricted domain.

Structure Implementation	typedef struct {
	int semaphore_variable;
	}binary_semaphore;
		typedef struct {
		int semaphore_variable;
		Queue list; //A queue to store the list of task
		}counting_semaphore;

Representation	0 means that a process or a thread is accessing the critical section, other process should wait for it to exit the critical section. 1 represents the critical section is free.	The value can range from 0 to N, where N is the number of process or thread that has to enter the critical section.
Mutual Exclusion	Yes, it guarantees mutual exclusion, since just one process or thread can enter the critical section at a time.	No, it doesn't guarantees mutual exclusion, since more than one process or thread can enter the critical section at a time.
Bounded wait	No, it doesn't guarantees bounded wait, as only one process can enter the critical section, and there is no limit on how long the process can exist in the critical section, making another process to starve.	Yes, it guarantees bounded wait, since it maintains a list of all the process or threads, using a queue, and each process or thread get a chance to enter the critical section once. So no question of starvation.

Starvation	No waiting queue is present then FCFS (first come first serve) is not followed so, starvation is possible and busy wait present	Waiting queue is present then FCFS (first come first serve) is followed so, no starvation hence no busy wait.
Number of instance	Used only for a single instance of resource type R.it can be usedonly for 2 processes.	Used for any number of instance of resource of type R.it can be used for any number of processes.

Difference Between Semaphore vs. Mutex:

Parameters	**Semaphore**	**Mutex**
Mechanism	It is a type of signaling mechanism.	It is a locking mechanism.
Data Type	Semaphore is an integer variable.	Mutex is just an object.
Modification	The wait and signal operations can modify a semaphore.	It is modified only by the process that may request or release a resource.
Resource management	If no resource is free, then the process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0.	If it is locked, the process has to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked.
Thread	You can have multiple program threads.	You can have multiple program threads in mutex but not simultaneously.
Ownership	Value can be changed by any process releasing or obtaining the resource.	Object lock is released only by the process, which has obtained the lock on it.
Types	Types of Semaphore are counting semaphore and binary semaphore and	Mutex has no subtypes.
Operation	Semaphore value is modified using wait () and signal () operation.	Mutex object is locked or unlocked.
Resources Occupancy	It is occupied if all resources are being used and the process requesting for resource performs wait () operation and blocks itself until semaphore count becomes >1.	In case if the object is already locked, the process requesting resources waits and is queued by the system before lock is released.

Advantages of Semaphores:

Here, are pros/benefits of using Semaphore:

- It allows more than one thread to access the critical section
- Semaphores are machine-independent.
- Semaphores are implemented in the machine-independent code of the microkernel.
- They do not allow multiple processes to enter the critical section.
- As there is busy waiting in semaphore, there is never a wastage of process time and resources.
- They are machine-independent, which should be run in the machine-independent code of the microkernel.
- They allow flexible management of resources.

Disadvantage of semaphores:

Here, are cons/drawback of semaphore

- One of the biggest limitations of a semaphore is priority inversion.
- The operating system has to keep track of all calls to wait and signal semaphore.
- Their use is never enforced, but it is by convention only.
- In order to avoid deadlocks in semaphore, the Wait and Signal operations require to be executed in the correct order.
- Semaphore programming is a complicated, so there are chances of not achieving mutual exclusion.
- It is also not a practical method for large scale use as their use leads to loss of modularity.
- Semaphore is more prone to programmer error.
- It may cause deadlock or violation of mutual exclusion due to programmer error.

Classic Problems of Synchronization:

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

Below are some of the classical problems depicting flaws of process synchronization in systems where cooperating processes are present.

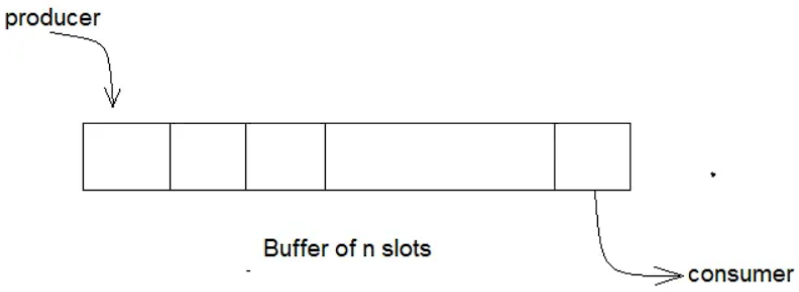
1. Bounded Buffer (Producer-Consumer) Problem
2. Printer Spooler problem
3. Dining Philosophers Problem
4. The Readers Writers Problem

1. Bounded Buffer Problem:

The bounded buffer problem, which is also called the **producer-consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution to this problem is to use semaphores. The semaphores which will be used here are:

- **m**, a **binary semaphore** that is used to acquire and release the lock.
- **empty**, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and fully represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
    // wait until empty > 0 and then decrement 'empty'
    wait(empty);
    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
    signal(mutex);
    // increment 'full'
    signal(full);
}
while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is at least one empty slot.
- Then it decrements the **empty** semaphore because there will now be one less empty slot since the producer is going to insert data in one of those slots.
- Then, it acquires a lock on the buffer, so that the consumer cannot access the buffer until the producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

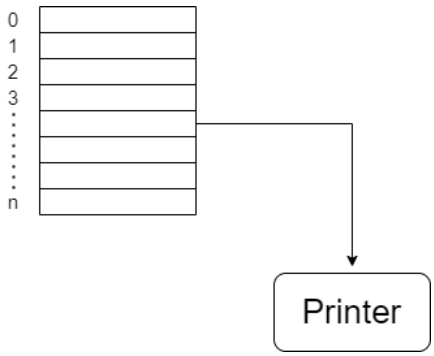
The pseudocode for the consumer function looks like this:

```
do
{
    // wait until full > 0 and then decrement 'full'
    wait(full);
    // acquire the lock
    wait(mutex);

    /* perform the remove operation in a slot */

    // release the lock
    signal(mutex);
    // increment 'empty'
    signal(empty);
}
while(TRUE);
```

- The consumer waits until there is at least one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one after the consumer completes its operation.
- After that, the consumer acquires a lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots are removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

2. Printer Spooler Problem:

- As we know the printer is a peripheral device, so it is slower in comparison to CPU and memory.
- So, if multiple users send some file to the printer to print then the spooler comes into play.
- The spooler is a program in a printer that stores all the files coming to print and when the printer is free it gives it to the printer in a sequential manner.



This four-line code is executed by each process in order to store its file in the spooler directory to print.

```
Load Ri, m[in]
store SD[Ri], "F-N"
INCR Ri
store m[in], Ri
```

`in: Shared variable` `m: Memory location` `Ri: Register` `F-N: File name` `SD: Spooler directory`

1. Line 1: Inline one we are loading free memory location m[in], in register Ri
2. Line 2: Inline two we are storing file name *(F-N)* in the spooler directory *(SD)* at position Ri, which is for instance 0
3. Line 3: In line three we are incrementing the count of Ri from 0 to 1, so the next file can be stored in at index 1
4. Line 4: In line four the new file will be stored at incremented memory location m[in]

The problem Statement:

Say there are two processes P1 and P2 with file names F1.txt and F2.txt that arrive at a concurrent time, let's say that the first three instructions of process P1 with file name F1 get executed, and then preemption occurs. Due to this m[in] does not gets incremented and still points to the location of f4.txt in the spooler directory.

now the next process P2 starts with filename F5 and the first instructions load the m[in] (which points to the f1.txt) to the register and then writes the file f2.txt on it.

Due to this F1 is overwritten by F2 and data is lost. After P2 is completed, the last instruction of P1 gets executed (store m[in], Ri) which then points to the next location in the spooler directory.

This leads to loss of data and needs to be solved using process synchronization.

Solution

We can initialize a semaphore with value 1 for the first time the spooler directory gets accessed. Then in the entry section of the code, we check if the semaphore is 1 or 0. The process gets executed if and only if the semaphore is 1.And then we assign the semaphore with value 0 while the process is being executed and then again assign it to 1 in the end block after the process gets completely executed. This way if any process gets preempted at any instruction and a new process comes, the semaphore is going to remain 0 since P1 has not been completed which forces the other process to wait till the P1 gets executed and the semaphore becomes 1 again.

Pseudocode

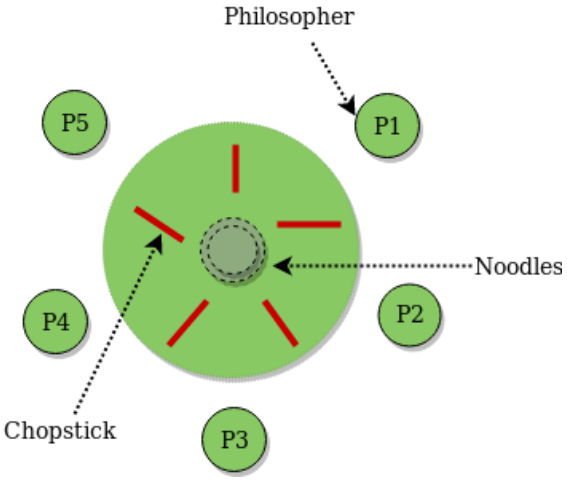
```
//semaphore=1(for the first time spooler directory is accesed)

{in entry block}
//if (semaphore==1) [enter critical section else go to block state]

{in critical section}
//semaphore=0 (which blocks any other process till
current process gets completely executed)

Load Ri, m[in]
store SD[Ri], “F-N”
INCR Ri
store m[in], Ri


{in end block}
semaphore=1 (new process can enter)
```

3. Dining Philosophers Problem:

The dining philosophers problem is another classic synchronization problem that is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks — one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point in time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5]for each of the five chopsticks.

The code for each philosopher looks like this:

```
while(TRUE)
{
    wait(stick[i]);
    /*
        mod is used because if i=5, next
        chopstick is 1 (dining table is circular)
    */
    wait(stick[(i+1) % 5]);

    /* eat */
    signal(stick[i]);

    signal(stick[(i+1) % 5]);
    /* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them picks up one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

4. Readers Writer Problem:

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource that should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **reader** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** m and a **semaphore** w. An integer variable read_count is used to maintain the number of readers currently accessing the resource. The variable read_count is initialized to 0. A value of 1 is given initially to m and w.

Instead of having the process to acquire a lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

The code for the **writer** process looks like this:

```
while(TRUE)
{
    wait(w);

   /* perform the write operation */

   signal(w);
}
```

And, the code for the **reader** process looks like this:

```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
    signal(m);

    /* perform the reading operation */

    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.

- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource, and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader who exits the critical section.
- The reason for this is, when the first readers enter the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Possibilities-

Case	Process 1	Process 2	Allowed / Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed
Case 4	Reading	Reading	Allowed

Critical Regions:

In an operating system, a critical region refers to a section of code or a data structure that must be accessed exclusively by one method or thread at a time. Critical regions are utilized to prevent concurrent entry to shared sources, along with variables, information structures, or devices, that allow you to maintain information integrity and keep away from race conditions.

The concept of important regions is carefully tied to the want for synchronization and mutual exclusion in multi-threaded or multi-manner environments. Without proper synchronization mechanisms, concurrent admission to shared resources can lead to information inconsistencies, unpredictable conduct, and mistakes.

To implement mutual exclusion and shield important areas, operating structures provide synchronization mechanisms, inclusive of locks, semaphores, or monitors. These mechanisms ensure that the handiest one procedure or thread can get the right of entry to the vital location at any given time, even as other procedures or threads are averted from entering till the cutting-edge occupant releases the lock.

Critical Region Characteristics and Requirements:

Following are the characteristics and requirements for critical regions in an operating system.

1. Mutual Exclusion: Only one procedure or thread can access the important region at a time. This ensures that concurrent entry does not bring about facts corruption or inconsistent states.

2. Atomicity: The execution of code within an essential region is dealt with as an indivisible unit of execution. It way that after a system or thread enters a vital place, it completes its execution without interruption.

3. Synchronization: Processes or threads waiting to go into a essential vicinity are synchronized to prevent simultaneous access. They commonly appoint synchronization primitives, inclusive of locks or semaphores, to govern access and put in force mutual exclusion.

4. Minimal Time Spent in Critical Regions: It is perfect to reduce the time spent inside crucial regions to reduce the capacity for contention and improve gadget overall performance. Lengthy execution within essential regions can increase the waiting time for different strategies or threads.

Monitors:

What is a monitor in OS?

A feature of programming languages called monitors helps control access to shared data. The Monitor is a collection of shared actions, data structures, and synchronization between parallel procedure calls. A monitor is therefore also referred to as a synchronization tool. Some of the languages that support the usage of monitors include Java, C#, Visual Basic, Ada, and concurrent Euclid. Although they can call the monitor's procedures, processes running outside of the monitor are unable to access its internal variables.

For example, the Java programming language provides synchronization mechanisms like the wait() and notify() constructs.

Syntax of Monitor:

Monitor in OS has a simple syntax similar to how we define a class, it is as follows:

```
Monitor monitorName{
    variables_declaration;
    condition_variables;

    procedure p1{ ... };
    procedure p2{ ... };
    ...
    procedure pn{ ... };

    {
        initializing_code;
    }

}
```

In an operating system, a monitor is only a class that includes variable_declarations, condition_variables, different procedures (functions), and an initializing_code block for synchronizing processes.

Characteristics of Monitors:

Monitors in operating systems possess several key characteristics that make them valuable tools for managing concurrent access to shared resources. Here are the main characteristics of monitors:

- **Mutual Exclusion:** Monitors ensure mutual exclusion, which means only one process or thread can be inside the monitor at any given time. This property prevents concurrent processes from accessing shared resources simultaneously and eliminates the risk of data corruption or inconsistent results due to race conditions.
- **Encapsulation:** Monitors encapsulate both the shared resource and the procedures that operate on it. By bundling the resource and the relevant procedures together, monitors provide a clean and organized approach to managing concurrent access. This encapsulation simplifies the design and maintenance of concurrent programs, as the necessary synchronization logic is localized within the monitor.
- **Synchronization Primitives:** Monitors often support synchronization primitives, such as condition variables. Condition variables enable threads within the monitor to wait for specific conditions to become true or to signal other threads when certain conditions are met. These primitives allow for efficient coordination among threads and help avoid busy-waiting, which can waste CPU cycles.

- **Blocking Mechanism:** When a process or thread attempts to enter a monitor that is already in use, it is blocked and put in a queue (entry queue) until the monitor becomes available. This blocking mechanism avoids busy-waiting and allows other processes to proceed while waiting for their turn to access the monitor.
- **Local Data:** Each thread that enters a monitor has its own local data or stack, which means the variables declared within a monitor procedure are unique to each thread's execution. This feature prevents interference between threads and ensures that data accessed within the monitor remains consistent for each thread.
- **Priority Inheritance:** In some advanced implementations of monitors, a priority inheritance mechanism can be used to prevent priority inversion. When a higher-priority thread is waiting for a lower-priority thread to release a resource inside the monitor, the lower-priority thread's priority may be temporarily elevated to avoid unnecessary delays caused by priority inversion scenarios.
- **High-Level Abstraction:** Monitors provide a higher-level abstraction for concurrency management compared to low-level synchronization mechanisms like semaphores or spinlocks. This abstraction reduces the complexity of concurrent programming and makes it easier to write correct and maintainable code.

Components of Monitor:

In an operating system, a monitor is a synchronization construct that helps manage concurrent access to shared resources by multiple processes or threads.

A monitor typically consists of the following main components:

- **Shared Resource:** The shared resource is the data or resource that multiple processes or threads need to access in a mutually exclusive manner. Examples of shared resources can include critical sections of code, global variables, or any data structure that needs to be accessed atomically.
- **Entry Queue:** The entry queue is a data structure that holds the processes or threads that are waiting to enter the monitor and access the shared resource. When a process or thread tries to enter the monitor while it is already being used by another process, it is placed in this queue, and its execution is temporarily suspended until the monitor becomes available.
- **Entry Procedures (or Monitor Procedures):** Entry procedures are special procedures that provide access to the shared resource and enforce mutual exclusion. When a process or thread wants to access the shared resource, it must call one of these entry procedures. The monitor's implementation ensures that only one process or thread can execute an entry procedure at a time, thus achieving mutual exclusion.

- **Local Data (or Local Variables):** Each process or thread that enters the monitor has its own set of local data or local variables. These variables are unique to each thread's execution and are not shared between threads. Local data allows each thread to work independently within the monitor without interfering with other threads' data.
- **Condition Variables:** Condition variables enable communication and synchronization between processes or threads within the monitor. They allow threads to wait until a specific condition is satisfied or to signal other threads when certain conditions become true. Condition variables are crucial for avoiding busy-waiting, which can be inefficient and wasteful of system resources.

  The condition variables of the monitor can be subjected to two different types of operations:

  1. Wait 2. Signal

Consider a condition variable (y) is declared in the monitor:

y.wait(): The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.

y.signal(): If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

Advantages of Monitor:

- Compared to semaphore-based solutions, monitors have the advantage of making concurrent or parallel programming simpler and less error-prone.
- It helps in operating system process synchronization.
- Monitors have mutual exclusion built in.
- Semaphores are more difficult to set up than monitors.
- Semaphores can lead to timing errors, which monitors may be able to fix.

Disadvantages of Monitor:

- Monitors must be implemented with the programming language.
- Monitor puts more work on the compiler.
- The monitor needs to be aware of the features offered by the operating system for managing critical steps in the parallel processes.

Process Scheduling and Algorithms:

What is Process Scheduling in Operating Systems?

The process manager's activity is process scheduling, which involves removing the running process from the CPU and selecting another process based on a specific strategy.

Multiprogramming OS's process scheduling is critical. Multiple processes could be loaded into the executable memory at the same time in such operating systems, and the loaded processes share the CPU utilising temporal multiplexing.

Process Scheduling Queues:

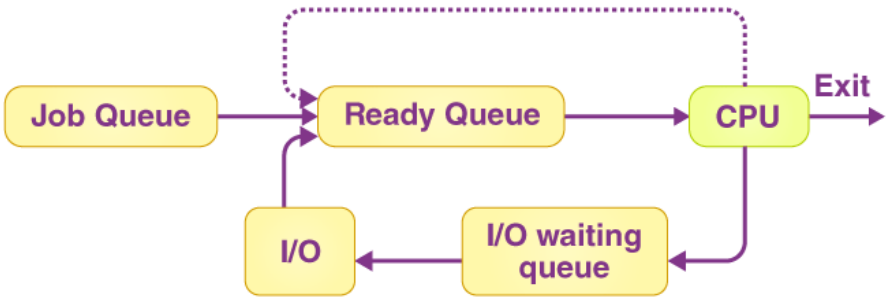
All PCBs (Process Scheduling Blocks) are kept in Process Scheduling Queues by the OS. Each processing state has its own queue in the OS, and PCBs from all processes in the same execution state are put in the very same queue. A process's PCB is unlinked from its present queue and then moved to its next state queue when its status changes.

The following major process scheduling queues are maintained by the Operating System:

Job queue – It contains all of the system's processes.

Ready queue – This queue maintains a list of all processes in the main memory that are ready to run. This queue is always filled with new processes.

Device queue – This queue is made up of processes that are stalled owing to the lack of an I/O device.



Each queue can be managed by the OS using distinct policies (FIFO, Priority, Round Robin, etc.). The OS scheduler governs how tasks are moved between the ready and run queues, each of which can only have one item per processor core on a system; it has been integrated with the CPU in the preceding figure.

Two-State Process Model:

The running and non-running states of a two-state process paradigm are detailed below.

Running

Whenever a new process is formed, it is immediately put into the operating state of the system.

Not currently running

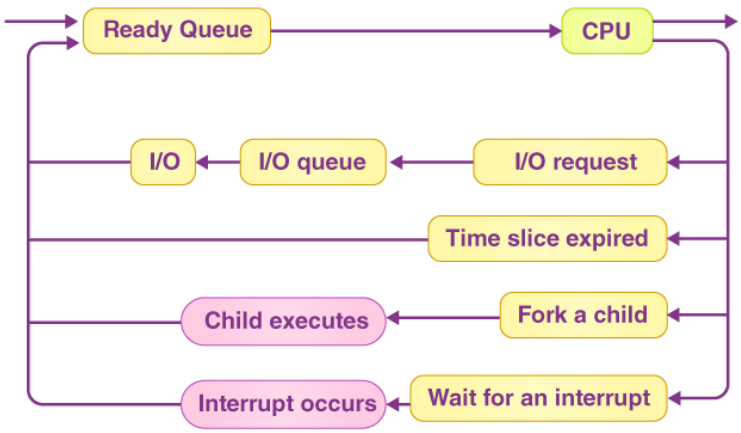
Processes that are not now executed are queued and will be executed when their turn comes. Each queue entry is a pointer to a certain process. Linked lists are used to implement queues. The following is how to use a dispatcher. When a process is halted, it is automatically shifted to the waiting list. The procedure is discarded once it has been completed or failed. In either scenario, the dispatcher chooses a process to run from the queue.

What are Scheduling Queues?

- The Job Queue stores all processes that are entered into the system.
- The Ready Queue holds processes in the ready state.
- Device Queues hold processes that are waiting for any device to become available. For each I/O device, there are separate device queues.

The ready queue is where a new process is initially placed. It sits in the ready queue, waiting to be chosen for execution or dispatched. One of the following occurrences can happen once the process has been assigned to the CPU and is running:

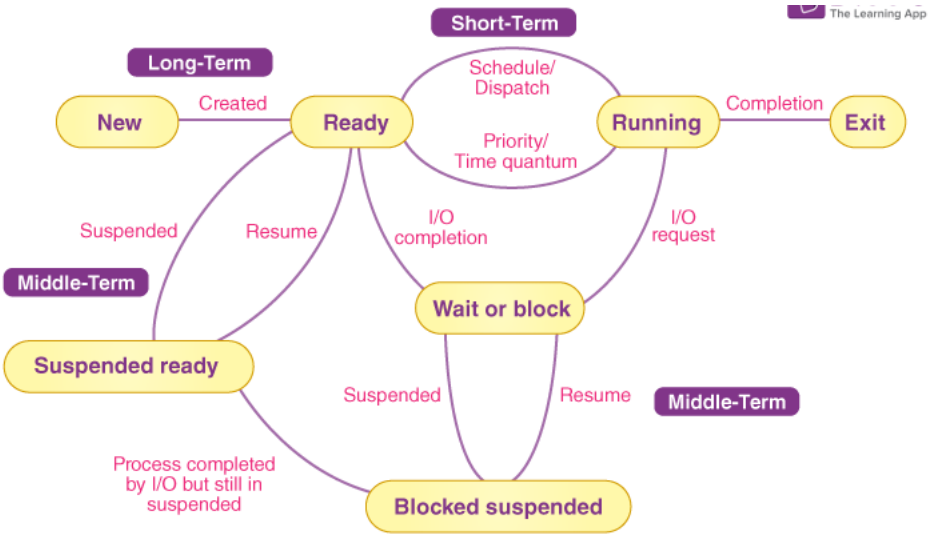
- The process could send out an I/O request before being placed in the I/O queue.
- The procedure could start a new one and then wait for it to finish.
- As a result of an interrupt, the process could be forcibly removed from the CPU and returned to the ready queue.



The process finally moves from the waiting to ready state in the first two circumstances and then returns to the ready queue. This cycle is repeated until a process is terminated, at which point it is withdrawn from all queues, and its PCB and resources are reallocated.

What is a Process Scheduler in an Operating System?

The process manager's activity is process scheduling, which involves removing the running process from the CPU and selecting another process based on a specific strategy. The scheduler's purpose is to implement the virtual machine so that each process appears to be running on its own computer to the user.



Multiprogramming OS's process scheduling is critical. Multiple processes could be loaded into executable memory at the same time in such an OS, and the loaded processes share the CPU utilising temporal multiplexing.

Types of Process Schedulers:

Process schedulers are divided into three categories.

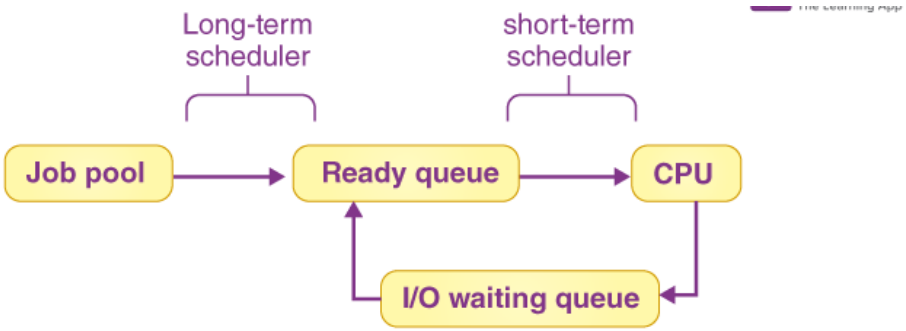
1. Long-Term Scheduler or Job Scheduler

The job scheduler is another name for Long-Term scheduler. It selects processes from the pool (or the secondary memory) and then maintains them in the primary memory's ready queue.

The Multiprogramming degree is mostly controlled by the Long-Term Scheduler. The goal of the Long-Term scheduler is to select the best mix of IO and CPU bound processes from the pool of jobs.

If the job scheduler selects more IO bound processes, all of the jobs may become stuck, the CPU will be idle for the majority of the time, and multiprogramming will be reduced as a result. Hence, the Long-Term scheduler's job is crucial and could have a Long-Term impact on the system.

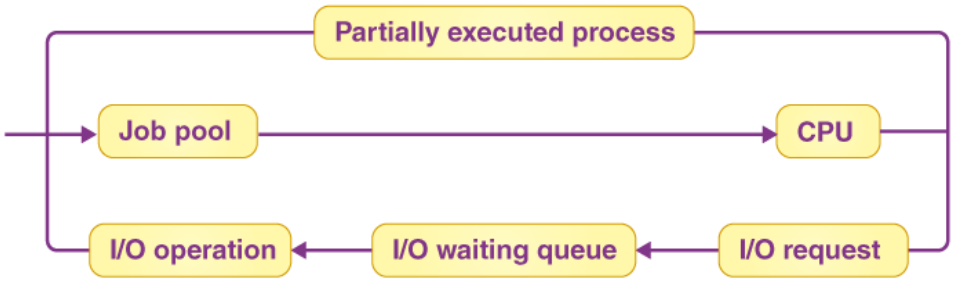
2. Short-Term Scheduler or CPU Scheduler



CPU scheduler is another name for Short-Term scheduler. It chooses one job from the ready queue and then sends it to the CPU for processing.

To determine which work will be dispatched for execution, a scheduling method is utilised. The Short-Term scheduler's task can be essential in the sense that if it chooses a job with a long CPU burst time, all subsequent jobs will have to wait in a ready queue for a long period. This is known as hunger, and it can occur if the Short-Term scheduler makes a mistake when selecting the work.

3. Medium-Term Scheduler



The switched-out processes are handled by the Medium-Term scheduler. If the running state processes require some IO time to complete, the state must be changed from running to waiting.

This is accomplished using a Medium-Term scheduler. It stops the process from executing in order to make space for other processes. Swapped out processes are examples of this, and the operation is known as swapping. The Medium-Term scheduler here is in charge of stopping and starting processes.

The degree of multiprogramming is reduced. To have a great blend of operations in the ready queue, swapping is required.

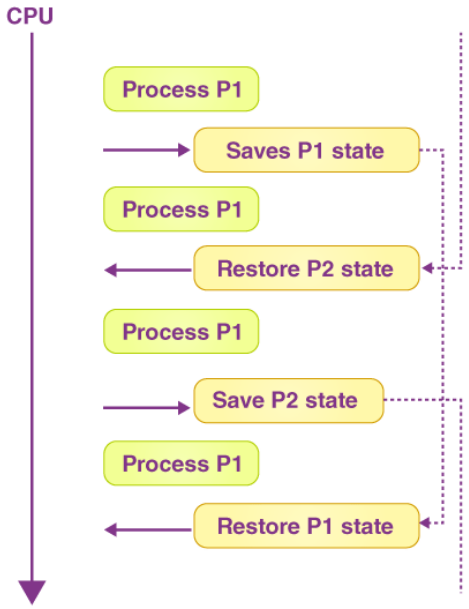
Comparison among Schedulers:

Parameters	Long-Term	Short-Term	Medium-Term
Type of Scheduler	It is a type of job scheduler.	It is a type of CPU scheduler.	It is a type of process swapping scheduler.
Speed	Its speed is comparatively less than that of the Short-Term scheduler.	It is the fastest among the other two.	Its speed is in between both Long and Short-Term schedulers.
Purpose	A Long-Term Scheduler helps in controlling the overall degree of multiprogramming.	The Short-Term Scheduler provides much less control over the degree of multiprogramming.	Medium-Term reduces the overall degree of multiprogramming.
Minimal time-sharing system	Almost absent	Minimal	Present
Function	Selects processes from the pool and then loads them into the memory for execution.	Selects all those processes that are ready to be executed.	Can re-introduce the given process into memory. The execution can then be continued.

Context Switch:

A context switch is a mechanism in the Process Control block that stores and restores the state or context of a CPU so that a process execution can be resumed from the very same point at a later time. A context switcher makes use of this technique to allow numerous programmes to share a single CPU. Thus, context switching is one of the most important elements of a multitasking operating system.

The state of the currently running process is recorded in the PCB when the scheduler changes the CPU from one process to another. The state for the following operation is then loaded from its PCB and used to set the registers, PC, and so on. The second process can then begin its execution.



Because register and memory states must be preserved as well as recovered, context switches can be computationally demanding. Some hardware systems use multiple sets of processor registers in order to save context switching time. The following information is saved for further use when the process is switched.

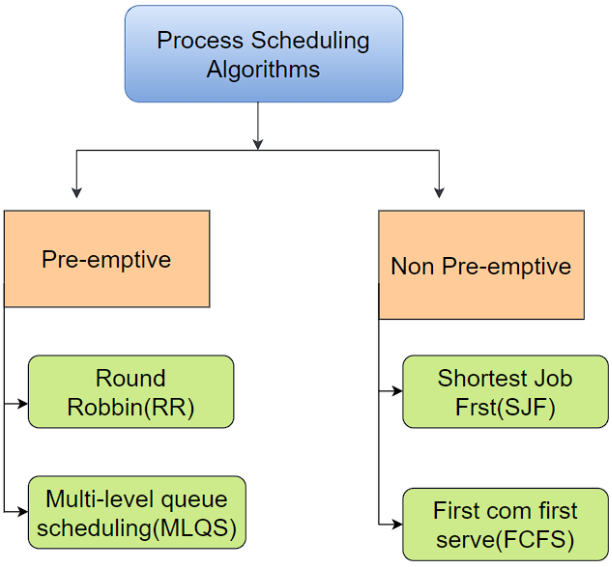
- Program counter
- Base and limit register value
- Scheduling information
- Currently used register
- I/O State information
- Changed state
- Accounting information

Process Scheduling Algorithms:

Process scheduling algorithms determine which processes are selected from the ready queue and granted CPU time for execution. They ensure that each process receives an appropriate amount of CPU time they prevent **starvation.** They play a critical role in managing the transitions between these process states, ensuring efficient resource utilization, fair allocation, and optimal system performance.

Process scheduling algorithms are divided into two categories:

1. Preemptive scheduling.
2. Non-preemptive scheduling.



Process Scheduling algorithms

Preemptive Scheduling Algorithms:

In **preemptive scheduling**, a running process can be forcefully interrupted and moved out of the CPU before its completion if a higher-priority process becomes ready or if its time slice (quantum) expires. Preemptive scheduling algorithms are:

1. Round robin (RR).
2. Multi-level queue scheduling (MLQS).

Non-preemptive Scheduling Algorithms:

Non-preemptive scheduling algorithms are process scheduling techniques where a running process cannot be interrupted until it voluntarily releases the CPU.

Here are some examples of non-preemptive scheduling algorithms:

1. First Come, First Served (FCFS)
2. Shortest Job First (SJF)

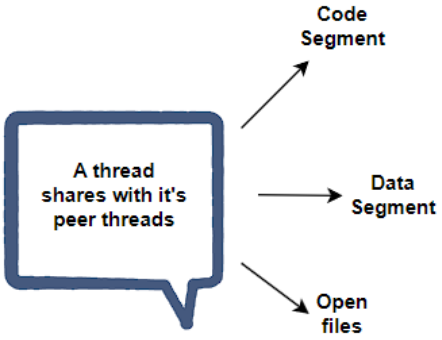
Threads:

What is Thread in Operating System?

Thread is a sequential flow of tasks within a process. Threads in an operating system can be of the same or different types. Threads are used to increase the performance of the applications.

Each thread has its own program counter, stack, and set of registers. However, the threads of a single process might share the same code and data/file. **Threads are also termed lightweight processes as they share common resources**.

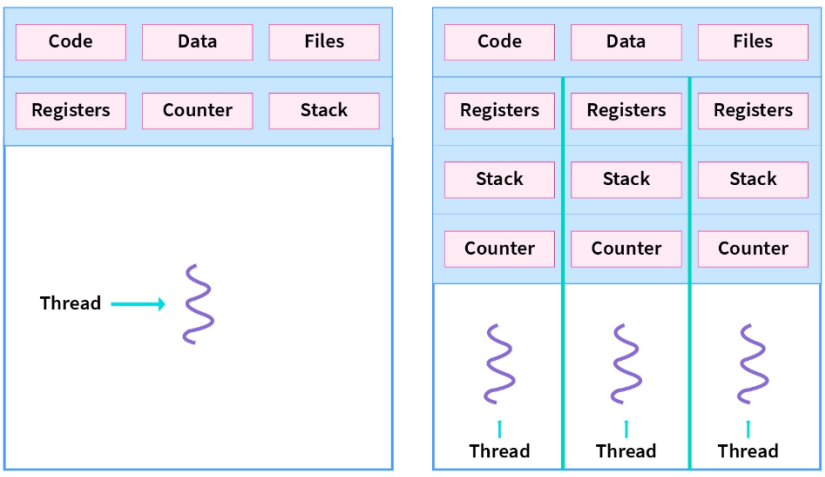
Eg: While playing a movie on a device the audio and video are controlled by different threads in the background.



Types of threads per process:

There are two sorts of process based on the number of threads:

1. **Single thread process**: Only one thread in an entire process
2. **Multi-thread process**: Multiple threads within an entire process.



Single-threaded process Multithreaded process

Components of Thread:

Threads are a fundamental component of modern operating systems and programming languages, and they consist of several key components that enable them to perform concurrent and parallel processing. The main components of a thread include:

- Thread ID
- Program counter
- Stack Space
- Register set
- Thread priority
- Thread state
- Thread-safety
- Synchronization

These components work together to enable threads to execute concurrently and perform complex tasks in a parallel and efficient manner.

Needs of Threads:

We need threads for several reasons, including:

1. **Improved performance:** Threads can help improve the performance of an application by allowing it to execute multiple tasks concurrently, thereby reducing the overall processing time.
2. **Responsiveness:** Threads can help improve the responsiveness of an application by allowing it to respond to user input while performing time-consuming tasks in the background.
3. **Resource sharing:** Threads can share resources such as memory, files, and network connections.
4. **Modularity:** Threads can help improve the modularity of an application by allowing it to break complex tasks into smaller, more manageable units of work that can be executed concurrently.
5. **Asynchronous processing:** Threads can be used to perform asynchronous processing, such as handling input/output operations.
6. **Parallel processing:** Threads can enable an application to perform parallel processing.

Types of Thread:

1. User Level Thread:

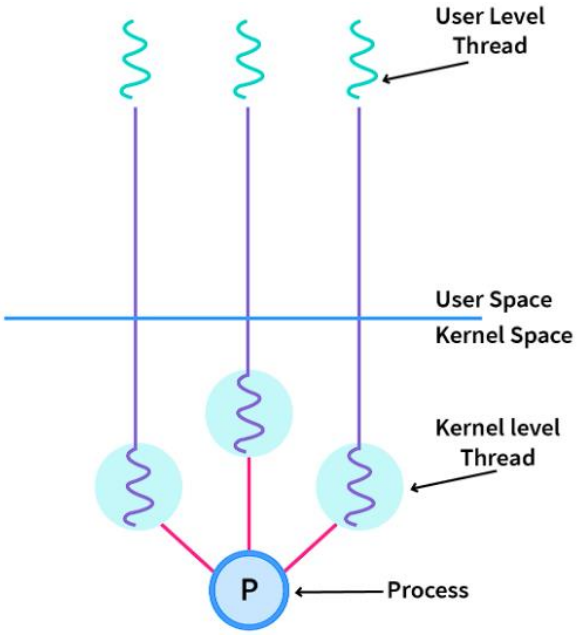
User-level threads are implemented and managed by the user and the kernel is not aware of it.

- User-level threads are **implemented using user-level libraries and the OS does not recognize these threads**.
- User-level thread is **faster to create and manage compared to kernel-level thread**.
- **Context switching in user-level threads is faster**.
- If one user-level thread performs a blocking operation then the entire process gets blocked. Eg: POSIX threads, Java threads, etc.

2. Kernel level Thread:

Kernel level threads are implemented and managed by the OS.

- Kernel level threads are **implemented using system calls and Kernel level threads are recognized by the OS**.
- Kernel-level threads are **slower to create and manage compared to user-level threads**.
- **Context switching in a kernel-level thread is slower**.
- Even if one kernel-level thread performs a blocking operation, it does not affect other threads. Eg: **Window Solaris**.



Advantages of Threading:

- Threads improve the overall performance of a program.
- Threads increases the responsiveness of the program
- Context Switching time in threads is faster.
- Threads share the same memory and resources within a process.
- Communication is faster in threads.
- Threads provide concurrency within a process.
- Enhanced throughput of the system.
- Since different threads can run parallelly, threading enables the utilization of the multiprocessor architecture to a greater extent and increases efficiency.

Issues with Threading:

There are a number of issues that arise with threading. Some of them are mentioned below:

- **The semantics of fork() and exec() system calls: The fork() call is used to create a duplicate child process**. During a fork() call the issue that arises is whether the whole process should be duplicated or just the thread which made the fork() call should be duplicated. **The exec() call replaces the whole process that called it including all the threads in the process with a new program**.
- **Thread cancellation:** The termination of a thread before its completion is called thread cancellation and the terminated thread is termed as target thread. Thread cancellation is of two types:
    1. **Asynchronous Cancellation:** In asynchronous cancellation, one thread immediately terminates the target thread.
    2. **Deferred Cancellation:** In deferred cancellation, the target thread periodically checks if it should be terminated.
- **Signal handling:** In UNIX systems, a signal is used to notify a process that a particular event has happened. Based on the source of the signal, signal handling can be categorized as:
    1. **Asynchronous Signal:** The signal which is generated outside the process which receives it.
    2. **Synchronous Signal:** The signal which is generated and delivered in the same process.

Process vs Thread:

The given table summarizes the differences between the threads and Processes in the Operating system.

Parameters	Process	Thread
Definition	An independent program with its own memory space and resources	A lightweight unit of execution within a process, sharing the same memory space
Creation	Created by the operating system when a program is launched	Created by the program itself
Memory	Each process has its own memory space, including its own stack, heap, and code segment	Threads within a process share the same memory space, including stack and heap
Communication	Inter-process communication is necessary for processes to communicate	Threads within a process can communicate directly through shared memory
Resource allocation	Processes are allocated system resources, including memory and I/O resources	Threads share the same resources as the process they belong to
Control	Each process runs independently and can be controlled separately	Threads within a process share the same control flow and can communicate directly
Overhead	Processes have a higher overhead due to the need for inter-process communication and separate memory space	Threads have lower overhead due to shared memory space and direct communication capabilities
Parallelism	Processes can run in parallel on different processors	Threads within a process can also run in parallel on different processors

Multithreading:

What is Multithreading?

The term multithreading refers to an operating system's capacity to support execution among fellow threads within a single process. The advantages of implementation of threads are resource ownership (Address space and utilized file I/O) and execution (TCB and scheduling). All threads inside a process will have to share compute resources such as code, data, files, and memory space with its peer thread, but stacks and registers will not be shared, and each new thread will have its own stacks and registers.

What is important to note here is that requests from one thread do not block requests from other separate threads, which improves application responsiveness. The term 'multithreading' also reduces the number of computing resources used and makes them more efficient. The concept of multithreading is the event of a system executing many threads, with the execution of these threads being of two types: Concurrent and Parallel multithread executions.

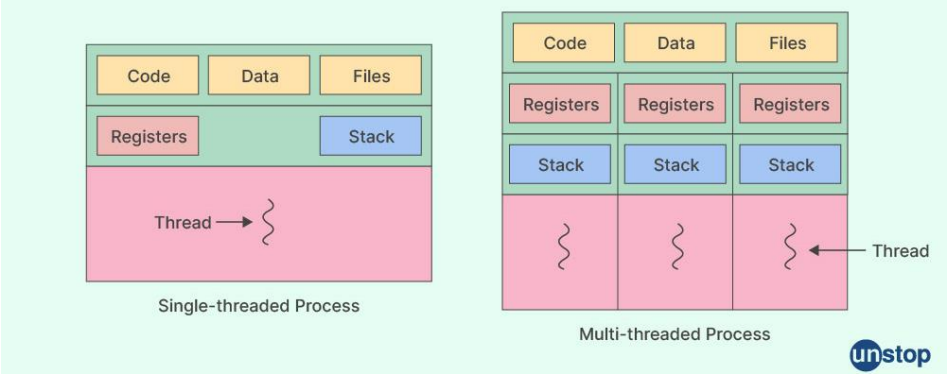
The concurrent process of threads is defined as the ability of a processor to move execution resources between threads in a multithreaded process on a single processor. When each thread in a multithreaded program executes on a separate processor at the same time, it is referred to as parallel execution.

The threads of the same process share the following items during multithreading:

- Address space
- Global variables
- Accounting information
- Opened files, used memory and I/O
- Child processes
- Pending alarms
- Signal and signal handlers

The following things are private (not shared) to the individual thread of a process in multithreading.

1. **Stack** (parameters, temporary, variables return address, etc)
2. **TCB (Thread Control Block):** It contains 'thread ids', 'CPU state information' (user-visible, control and status registers, stack pointers), and 'scheduling information' (state of thread priority, etc.)



Advantages of Multithreading:

- Improves execution speed (by combining CPU machine and I/O wait times).
- Multithreading can be used to achieve concurrency.
- Reduces the amount of time it takes for context switching.
- Improves responsiveness.
- Make synchronous processing possible (separates the execution of independent threads).
- Increases throughput.
- Performs foreground and background work in parallel.

Disadvantages of Multithreading:

- Because the threads share the same address space and may access resources such as open files, difficulties might arise in managing threads if they utilize incompatible data structures.
- If we don't optimize the number of threads, instead of gaining performance, we may lose performance.
- If a parent process requires several threads to function properly, the child processes should be multithreaded as well, as they may all be required.

> Process Image vs. Multi Thread Process Image: PCB, Stack, Data, and Code segments are all shown in the process image. PCB for the parent process is included in multithread process images. Each individual thread has a TCB and a stack and address space for everything (data and code)

Multithreading Models:

There are three sorts of models in multithreading.

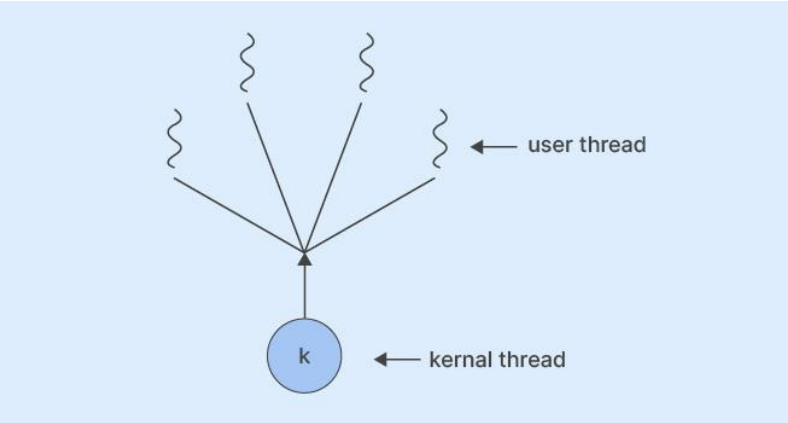
- Many-to-many models
- Many-to-one model
- One-to-one model

1. Many-to-One:

There will be a many-to-one relationship model between threads, as the name implies. Multiple user threads are linked or mapped to a single kernel thread in this case. Management of threads is done on a user-by-user basis, which makes it more efficient. It converts a large number of user-level threads into a single Kernel-level thread.

The following issues arise in many to one model:

- A block statement on a user-level thread stops all other threads from running.
- Use of multi-core architecture is inefficient.
- There is no actual concurrency.

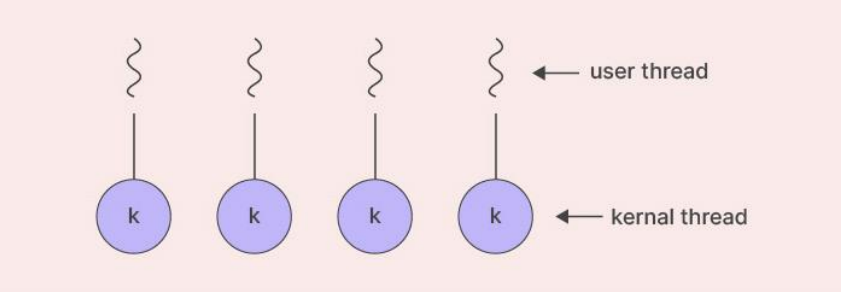


2. One-to-One:

We can deduce from the name that one user thread is mapped to one separate kernel thread. The user-level thread and the kernel-level thread have a one-to-one relationship. The many-to-one model for thread provides less concurrency than this architecture. When a thread performs a blocking system call, it also allows another thread to run.

It provides the following benefits over the many-to-one model:

- A block statement on one thread does not cause any other threads to be blocked.
- Concurrency in the true sense.
- Use of a multi-core system that is efficient.

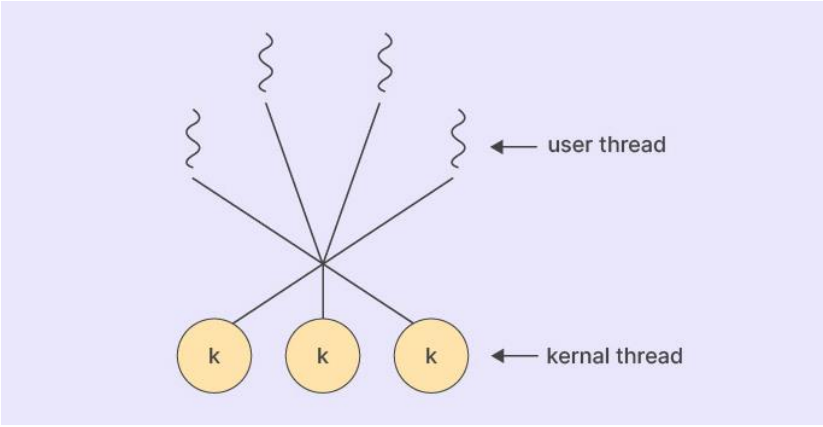


3. Many-to-Many:

From the name itself, we may assume that there are numerous user threads mapped to a lesser or equal number of kernel threads. This model has a version called a two-level model, which incorporates both many-to-many and one-to-one relationships. When a thread makes a blocked system call, the kernel can schedule another thread for execution.

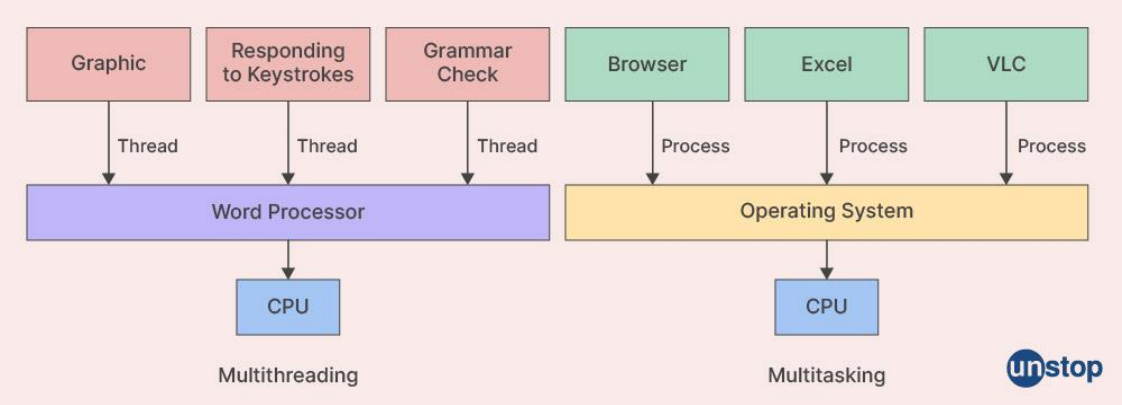
The particular application or machine determines the number of kernel threads. The many-to-many model multiplexes any number of user threads onto the same number of kernel threads or fewer kernel threads. On the utilization of multiprocessor architectures, developers may build as many user threads as they need, and the associated kernel threads can execute in parallel.

So, in a multithreading system, this is the optimal paradigm for establishing the relationship between user thread and the kernel thread.

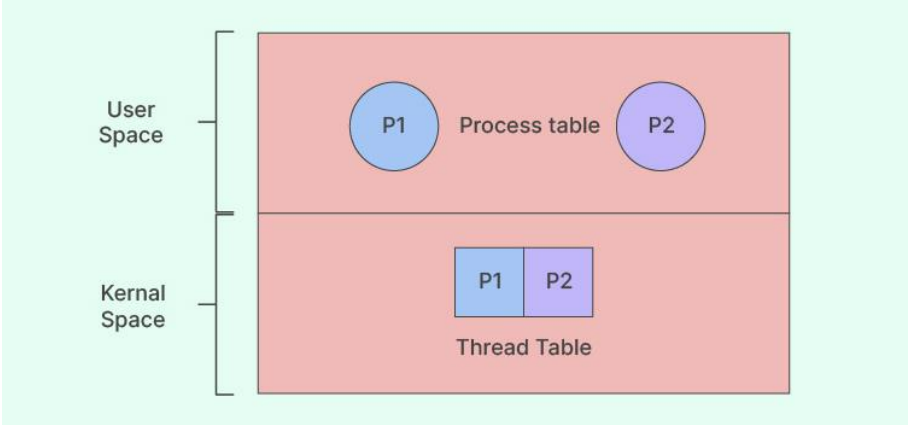


Multithreading Vs. Multitasking:

- Terminating a thread in multithreading takes lesser time than terminating a process in multitasking.
- As opposed to switching between processes, switching between threads takes less time.
- Multithreading allows threads in the same process to share memory, I/O, and file resources, allowing threads to communicate at the user level.
- When compared to a process, threads are light.
- Web-client browsers, network servers, bank servers, word processors, and spreadsheets are examples of multithreading programs.
- When compared to multitasking, multithreading is faster.
- Creating a thread in multithreading requires less time than creating a process in multitasking.
- Multiprocessing relies on pickling objects in memory to send to other processes, but Multithreading avoids it.



Difference between Process, Kernel Thread and User Thread:



Process	Kernel Thread	User Thread
There is no collaboration between processes.	Share address space	Share address space
When one process is stopped, it has no effect on the others.	A process's other threads are unaffected by blocking one thread.	When one thread is blocked, the entire thread's process is blocked.
In a multiprocessor system, each process can operate on a distinct processor to facilitate parallelism.	In a multiprocessor system, each thread might execute on a distinct processor to allow parallelism, and Kernel manages all threads in a process.	Only one processor should be used by all threads, and only one thread should be active at any given moment. Because a process's threads are all interdependent and governed by it.
The process has a lot of overhead.	Kernel threads have a moderate overhead.	The overhead of the user thread is minimal.
The procedure is a lengthy procedure.	Kernel threading is a lightweight method that does not require the use of a process.	A user thread is a small process that belongs to a larger process.
The operating system schedules processes using a process table.	The OS uses a thread table to schedule threads.	Thread library uses a thread table to schedule threads.
It is possible to suspend a process without affecting other processes.	Suspending a process causes all of its threads to stop executing, which isn't conceivable.	Thread suspension is not possible since suspending a process causes the current thread to stop operating.
OS support is required for efficient communication between processes.	OS support is required for thread communication.	The threads communicate with each other at the user level (does not require OS support).

PART – 2 : CPU SCHEDULING

Introduction:

What is CPU Scheduling?

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

Types of CPU Scheduling:

Here are two kinds of Scheduling methods:

1. Preemptive Scheduling:

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

2. Non-Preemptive Scheduling:

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

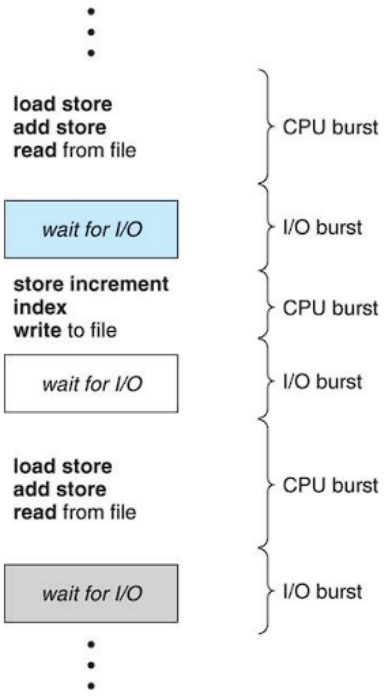
1. A process switches from the running to the waiting state.
2. Specific process switches from the running state to the ready state.
3. Specific process switches from the waiting state to the ready state.
4. Process finished its execution and terminated.

Only conditions 1 and 4 apply, the scheduling is called non- preemptive.

All other scheduling are preemptive.

Important CPU Scheduling Terminologies:

- **Burst Time/Execution Time:** It is a time required by the process to complete execution. It is also called running time.
- **Arrival Time:** when a process enters in a ready state
- **Finish Time:** when process complete and exit from a system
- **Multiprogramming:** A number of programs which can be present in memory at the same time.
- **Jobs:** It is a type of program without any kind of user interaction.
- **User:** It is a kind of program having user interaction.
- **Process:** It is the reference that is used for both job and user.
- **CPU/IO burst cycle:** Characterizes process execution, which alternates between CPU and I/O activity. CPU times are usually shorter than the time of I/O.
    - Process execution consists of a cycle of CPU execution and I/O wait.
    - Processes alternate between these two states.
    - Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst ... etc
    - The last CPU burst will end with a system request to terminate execution rather than with another I/O burst.
    - The duration of these CPU burst have been measured.
    - An I/O-bound program would typically have many short CPU bursts, A CPU-bound program might have a few very long CPU bursts.
    - This can help to select an appropriate CPU-scheduling algorithm.



What is the Need for CPU Scheduling Algorithm?

CPU (Central Processing Unit) scheduling algorithm is needed to efficiently allocate the available processing time of a CPU among multiple processes that are competing for the CPU's resources.

The need for CPU scheduling arises because modern operating systems allow multiple processes to execute concurrently on a single CPU. When multiple processes are running, they contend for the CPU's resources, and the CPU must choose which process to execute at any given moment.

The CPU scheduling algorithm determines the order in which processes are executed and how much CPU time each process is allocated. A good CPU scheduling algorithm should ensure that each process gets a fair share of the CPU time, while also maximizing overall system throughput and minimizing response time.

Preemptive vs Non-Preemptive Scheduling:

Preemptive Scheduling	Non-Preemptive Scheduling
In Preemptive Scheduling, the CPU is assigned for some time	In Non-Preemptive Scheduling, CPU is assigned to the process until the process completes the execution.
Interrupt may occur between the execution of the process.	Interrupt cannot occur until the process completes its execution.
Preemptive Scheduling is flexible.	Non-Preemptive Scheduling is rigid.
In preemptive Scheduling there may be a chance of overhead the scheduling process.	In Non-Preemptive Scheduling there is no chance of overhead the scheduling the process.
Preemptive scheduling is cost- related.	Non-Preemptive Scheduling does not cost-related.
In preemptive scheduling, when the high-priority process reaches into the ready queue, then due to of high-priority process the low-priority process can starve.	In Non-Preemptive Scheduling, when in a CPU, the process which has more burst time is running, then due to this, the process which has less burst time can starve.

Various Times related to the Process in OS:

A process is a program in execution, and there are several different times associated with a process that is important to understand when working with operating systems. These times include:

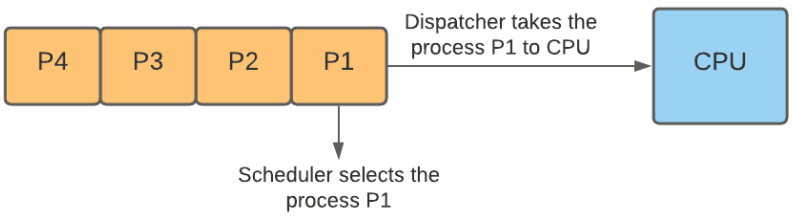
1. **Arrival Time:** This is the time at which a process arrives in the system and is ready to be executed.
2. **Burst Time:** This is the amount of time that a process needs to execute its next instruction or complete its next task.
3. **Waiting Time:** This is the amount of time that a process spends waiting in the ready queue for the CPU to become available.
4. **Turnaround Time:** This is the total time that elapses from the time a process arrives in the system to the time it completes execution. It is the sum of the burst time and the waiting time.
5. **Response Time:** This is the amount of time that elapses from the time a process submits a request to the time it receives a response.
6. **Execution Time:** This is the amount of time that a process spends executing on the CPU.

Dispatcher in OS:

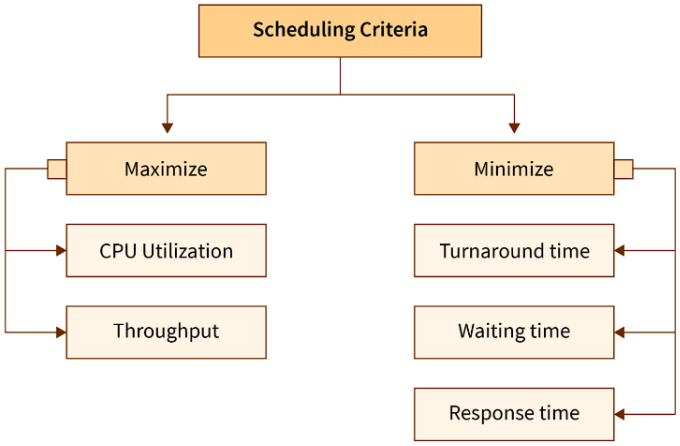
The dispatcher is another component involved in CPU scheduling. **The duty of the dispatcher is to give control of the CPU to the process selected by the CPU scheduler for execution**. The dispatcher should work very fast since it is invoked every time the CPU is allocated to a process. The time taken by the dispatcher to stop one process and start another process is known as the **dispatch latency**. Thus, the functions of the dispatcher are -

- Switching context (switching from one process to another process)
- Switching to user mode (In dual-mode, switching the system from kernel mode to user mode)
- In the newly loaded program, navigate to the correct location.

In the figure below, the process P1 selected by the CPU scheduler is taken to the CPU by the dispatcher.



Scheduling Criteria:



Maximize

Throughput: - Throughput is defined as the total number of processes that complete its execution per unit time. Depending on the specific processes, this can vary from 10/second to 1/hour.

CPU Utilization: - CPU utilization is an essential job in the operating system. For effective CPU utilization, the operating system must ensure that the CPU stays as active as possible most of the time**.** It can be between 0 to 100 percent, but in the Real-Time operating system, the range is 40 percent, for low-level, and for the high-level system, it can be 90 percent. for better CPU utilization, CPU must busy at all the time.

CPU utilization can be defined as the percentage of time CPU was handling process execution to total time

Formulae – CPU Utilization = (Total time – Total idle time) / (Total Time)

A computer with 75% CPU utilization is better than 50%, since the CPU time was better utilized to handle process execution in 75% and lesser time was wasted in idle time.

Minimize:

Load Average: - load average is the average number of processes which is existing in the ready queue and waiting for a CPU.

Turnaround Time: - Turnaround Time is defined as the total amount of time process consumed from its arrival to its completion. In other words, it is the total amount of time to execute a specific process.

Turn-around time (TAT) = Completion time (CT) – Arrival time (AT) or,

TAT = Burst time (BT) + Waiting time (WT)

Waiting Time: - Waiting time is the cumulative amount of time for which the process has waited for the allocation of the CPU.

Waiting time (WT) = Turn-around time (TAT) – Burst time (BT)

Response Time: - Response Time is defined as the difference between the time of arrival and the time in which the process gets the CPU first.

Interval Timer:

Timer interruption is a method that is closely related to preemption. When a certain process gets the CPU allocation, a timer may be set to a specified interval. Both timer interruption and preemption force a process to return the CPU before its CPU burst is complete.

Most of the multi-programmed operating system uses some form of a timer to prevent a process from tying up the system forever.

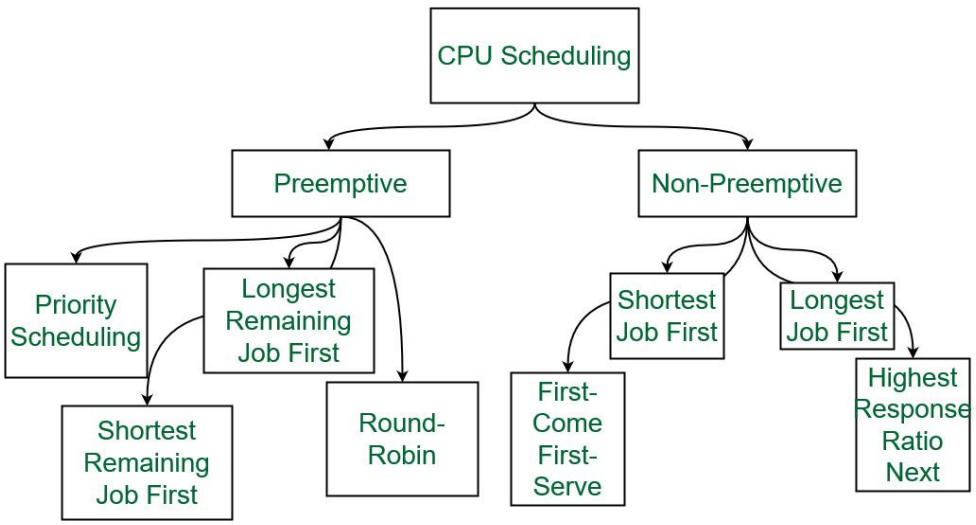
Scheduling Algorithms/Techniques:

Objective of Scheduling Algorithms:

There are the following objectives of Scheduling Algorithms:

1. Fare allocation of CPU
2. Increased throughput
3. Improved CPU Utilization
4. Minimum Turnaround time
5. Minimum response time
6. Minimum waiting time

Various Scheduling Algorithms:



1. First Come, First Served (FCFS):

FCFS scheduling is a **non-preemptive algorithm** where processes are executed in the order they arrive. Once a process starts executing, it continues until it completes or enters a waiting state voluntarily. FCFS scheduling algorithm acts similarly to a **queue**.

This strategy can be easily implemented by using FIFO queue, FIFO means First In First Out. When CPU becomes free, a process from the first position in a queue is selected to run.

In case of a tie, if two processes request CPU simultaneously, the process with a **smaller process ID** gets the CPU allocation first. This is the simplest and easiest-to-implement CPU scheduling algorithm.

How First Come First Serve CPU Scheduling Algorithm Work?

- The waiting time for the first process is 0 as it is executed first.
- The waiting time for the upcoming process can be calculated by:

  wt[i] = (at[i – 1] + bt[i – 1] + wt[i – 1]) – at[i]

  where,

  - ***wt[i]*** *= waiting time of current process*
  - ***at[i-1]*** *= arrival time of previous process*
  - ***bt[i-1]*** *= burst time of previous process*
  - ***wt[i-1]*** *= waiting time of previous process*
  - ***at[i]*** *= arrival time of current process*

- The Average waiting time can be calculated by:

  Average Waiting Time = (sum of all waiting time)/(Number of processes)

> **Note:** Use **FCFS scheduling** when the order of process arrival is important or needs to be preserved.

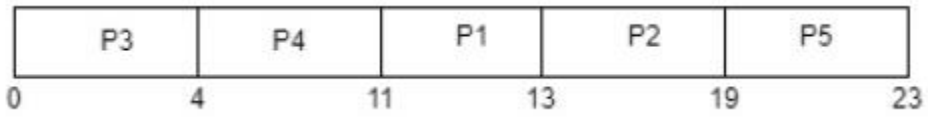
Gantt Chart: Gantt chart is a visualization which helps to scheduling and managing particular tasks in a project. It is used while solving scheduling problems, for a concept of how the processes are being allocated in different algorithms.

Problem 1: Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	5	6
P3	0	4
P4	0	7
P5	7	4

Solution:

Gantt chart -



For this problem CT, TAT, WT, RT is shown in the given table –

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	13	13-2= 11	11-2= 9	9
P2	5	6	19	19-5= 14	14-6= 8	8
P3	0	4	4	4-0= 4	4-4= 0	0
P4	0	7	11	11-0= 11	11-7= 4	4
P5	7	4	23	23-7= 16	16-4= 12	12

Average Waiting time = (9+8+0+4+12)/5 = 33/5 = 6.6 time unit (time unit can be considered as milliseconds)

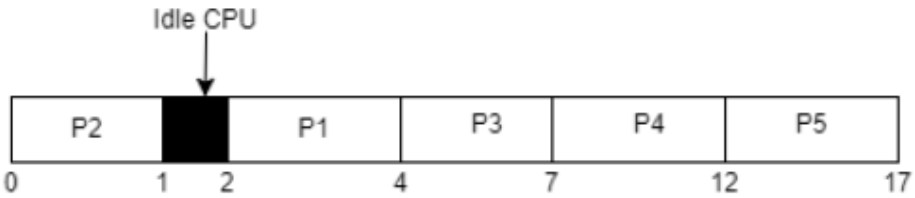
Average Turn-around time = (11+14+4+11+16)/5 = 56/5 = 11.2 time unit (time unit can be considered as milliseconds)

Problem 2: Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	0	1
P3	2	3
P4	3	5
P5	4	5

Solution:

Gantt chart –



For this problem CT, TAT, WT, RT is shown in the given table −

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	4	4-2= 2	2-2= 0	0
P2	0	1	1	1-0= 1	1-1= 0	0
P3	2	3	7	7-2= 5	5-3= 2	2
P4	3	5	12	12-3= 9	9-5= 4	4
P5	4	5	17	17-4= 13	13-5= 8	8

Average Waiting time = (0+0+2+4+8)/5 = 14/5 = 2.8 time unit (time unit can be considered as milliseconds)

Average Turn-around time = (2+1+5+9+13)/5 = 30/5 = 6 time unit (time unit can be considered as milliseconds)

*In idle (not-active) CPU period, no process is scheduled to be terminated so in this time it remains void for a little time.

Advantages:

- This algorithm is simple to implement and easy to understand.
- It uses a very simple data structure queue for maintaining processes.
- FCFS does not lead the process to **starvation**.

Disadvantages:

- As it is a non-preemptive algorithm, it doesn't consider any process's priority or burst time.
- This algorithm mostly suffers from the **convoy effect**.
- The average waiting time for processes is not optimal.
- Parallel utilization of resources for processes is not possible.

Starvation: Starvation is the problem that occurs when low priority processes get jammed for an unspecified time as the high priority processes keep executing.

Convoy Effect: In convoy effect, consider processes with higher burst time arrived before the processes with smaller burst time. Then, smaller processes have to wait for a long time for longer processes to release the CPU.

2. Shortest-Job-Next (SJN):

The **Shortest Job Next (SJN)** scheduling algorithm is also known as the **Shortest Job First (SJF)** scheduling.

In this scheduling, the algorithm selects the process in the waiting queue with the shortest **burst time** to execute next.

If two processes have the same burst time, we use the **FCFS scheduling** to break the tie, and the rest of the processes get scheduled using SJN again.

To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.

Characteristics of SJF Scheduling:

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- This is used in Batch Systems.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)

- It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
- SJF can be used in specialized environments where accurate estimates of running time are available.

Algorithm:

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

How to compute below times in SJF using a program?

- **Completion Time(CT):** Time at which process completes its execution.
- **Turn Around Time(TAT):** Time Difference between completion time and arrival time.
  Turn Around Time = Completion Time – Arrival Time
- **Waiting Time(WT):** Time Difference between turn around time and burst time.
  Waiting Time = Turn Around Time – Burst Time

Types: There are two types of SJN scheduling:

- Preemptive SJN
- Non-Preemptive SJN

Preemptive SJF scheduling:

Preemptive SJF scheduling is sometime called **Shortest Remaining Time First Scheduling.**

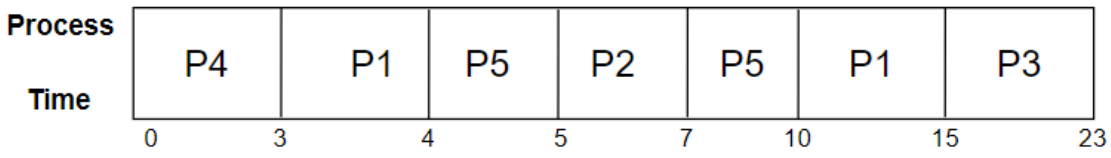
In the **preemptive SJN scheduling**, all the processes are put into the ready queue as they request to use the CPU. The process with the shortest burst time starts execution. If another process arrives with a shorter burst time, the current process gets preempted from execution, and a shorter process is allocated to use the CPU.

Example:

Suppose we have a set of five processes whose burst time and arrival time are given, schedule the processes using preemptive SJN scheduling, and calculate the average waiting time.

Process Queue	Burst Time	Arrival Time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Gantt chart:



Waiting time:

$P4 = 0 - 0 = 0$
$P1 = (3 - 2) + 6 = 7$
$P2 = 5 - 5 = 0$
$P5 = 4 - 4 + 2 = 2$
$P3 = 15 - 1 = 14$

Average waiting time:

$$\text{Average waiting time} = \frac{0 + 7 + 0 + 2 + 14}{5} = \frac{23}{5} = 4.6$$

Non-Preemptive SJN:

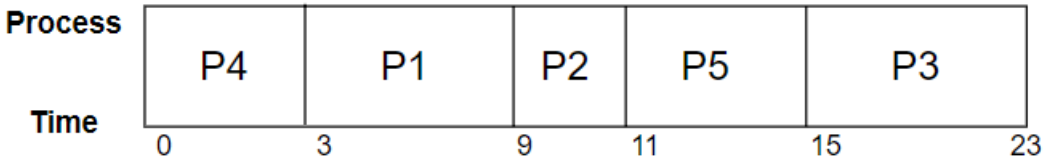
In the **non-preemptive SJN scheduling**, if the CPU cycle is allocated to any process, the process holds the CPU until it reaches a waiting state or is terminated.

Example

Suppose we have a set of five processes whose burst time and arrival time are given, schedule the processes using non-preemptive SJN scheduling, and calculate the average waiting time.

Process Queue	Burst Time	Arrival Time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Gantt chart:



Waiting time:

$P4=0-0=0$

$P1=3-2=1$

$P2=9-5=4$

$P5=11-4=7$

$P3=15-1=14$

Average waiting time:

$$\text{Average waiting time} = \frac{0+1+4+7+14}{5} = \frac{26}{5} = 5.2$$

Advantages of SJF:

- SJF is better than the First come first serve (FCFS) algorithm as it reduces the average waiting time.
- SJF is generally used for long term scheduling.
- It is suitable for the jobs running in batches, where run times are already known.
- SJF is probably optimal in terms of average turnaround time.
- This is the best scheduling to reduce waiting time.

Disadvantages of SJF:

- SJF may cause very long turn-around times or starvation.
- In SJF job completion time must be known earlier, but sometimes it is hard to predict.
- Sometimes, it is complicated to predict the length of the upcoming CPU request.
- It leads to the starvation that does not reduce average turnaround time.

Prediction of CPU Burst Time for a process in SJF:

The SJF algorithm is one of the best scheduling algorithms since it provides the maximum throughput and minimal waiting time but the problem with the algorithm is, the CPU burst time can't be known in advance.

We can approximate the CPU burst time for a process. There are various techniques which can be used to assume the CPU Burst time for a process. Our Assumption needs to be accurate in order to utilize the algorithm optimally.

There are the following techniques used for the assumption of CPU burst time for a process.

1. Static Techniques:

Process Size:

We can predict the Burst Time of the process from its size. If we have two processes **T_OLD** and **T_New** and the actual burst time of the old process is known as **20 secs** and the size of the process is **20 KB**. We know that the size of **P_NEW is 21 KB**. Then the probability of **P_New** having the similar burst time as **20 secs** is maximum.

If, P_OLD → 20 KB

P_New → 21 KB

BT(P_OLD) → 20 Secs

Then,

BT(P_New) → 20 secs

Hence, in this technique, we actually predict the burst time of a new process according to the burst time of an old process of similar size as of new process.

Process Type:

We can also predict the burst time of the process according to its type. A Process can be of various types defined as follows.

- **OS Process**

A Process can be an Operating system process like schedulers, compilers, program managers and many more system processes. Their burst time is generally lower for example, 3 to 5 units of time.

- **User Process**

The Processes initiated by the users are called user processes. There can be three types of processes as follows.

- **Interactive Process**

The Interactive processes are the one which interact with the user time to time or Execution of which totally depends upon the User inputs for example various games are such processes. There burst time needs to be lower since they don't need CPU for a large amount of time, they mainly depend upon the user's interactivity with the process hence they are mainly IO bound processes.

- **Foreground process**

Foreground processes are the processes which are used by the user to perform their needs such as MS office, Editors, utility software etc. These types of processes have a bit higher burst time since they are a perfect mix of CPU and IO bound processes.

- **Background process**

Background processes supports the execution of other processes. They work in hidden mode. For example, key logger is the process which records the keys pressed by the user and activities of the user on the system. They are mainly CPU bound processes and needs CPU for a higher amount of time.

2. Dynamic Techniques:

Simple Averaging:

In simple averaging, there are given list of n processes P(i).....P(n). Let T(i) denotes the burst time of the process P(i). Let τ(n) denotes the predicted burst time of Pth process. Then according to the simple averaging, the predicted burst time of process n+1 will be calculated as,

$$\tau(n+1) = (1/n) \sum T(i)$$

Where, 0<=i<=n and $\sum T(i)$ is the summation of actual burst time of all the processes available till now.

Exponential Averaging or Aging:

Let, Tn be the actual burst time of nth process. τ(n) be the predicted burst time for nth process then the CPU burst time for the next process (n+1) will be calculated as,

$$\tau(n+1) = \alpha . Tn + (1-\alpha) . \tau(n)$$

Where, α is the smoothing. Its value lies between 0 and 1.

3. Priority Based Scheduling:

Priority scheduling is scheduling a set of processes each one with a specific priority relative to other processes.

There are 2 types of Priority scheduling: Fixed Priority scheduling and Dynamic Priority Scheduling.

In Fixed Priority scheduling, process is assigned a fixed priority at the start.

In Dynamic Priority Scheduling, during execution, the priority is calculated and assigned to process.

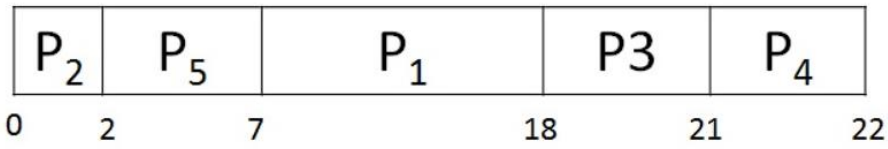
- Each process is assigned a priority. Here the CPU is allocated to the process which has the highest priority.
- If two processes have an equal priority, then those processes are scheduled in FCFS order.
- In general, the priority is indicated by number in the range from 0 to 7 or 0 to 4095. Here, we assume that 0 represents high priority.
- The priority can be calculated based on time limits, memory requirement, number of open files, the ratio from average I/O burst to average CPU burst.

- It can be either preemptive or nonpreemptive.
- In case of **Preemptive priority scheduling** algorithm, the CPU will be preempted only if the priority of newly arrived process is higher than the currently running process.
- In case of **nonpreemptive priority scheduling** algorithm, the newly arrived process will be placed at the head of the ready queue.
- The main drawback of this scheduling algorithm is indefinite blocking or starvation (this algorithm can leave the low priority processes waiting indefinitely).

For Example, consider the set of processes that arrive at time 0 and CPU-burst time given in millisecond:

Process	CPU-Burst Time	Priority
P_1	11	3
P_2	2	1
P_3	3	4
P_4	1	5
P_5	5	2

Gantt Chart:



Process	Arrival Time	CPU-Burst	Priority	Waiting Time	Turnaround Time
P_1	0	11	3	7	18
P_2	0	2	1	0	2
P_3	0	3	4	18	21
P_4	0	1	5	21	22
P_5	0	5	2	2	7

Average Waiting Time = (7+0+18+21+2) / 5 = 9.6 ms

Average Turnaround Time=(18+2+21+22+7) / 5 = 14 ms

4. Round Robin Scheduling:

The Round robin scheduling algorithm is one of the CPU scheduling algorithms in which every process gets a fixed amount of **time quantum** to execute the process.

In this algorithm, every process gets executed cyclically. This means that processes that have their **burst time remaining** after the expiration of the time quantum are sent back to the ready state and wait for their next turn to complete the execution until it **terminates**. This processing is done in **FIFO** order which suggests that processes are executed on a first-come, first-serve basis.

The Round-robin scheduling algorithm is a kind of **preemptive** First come, First Serve CPU Scheduling algorithm where each process in the ready state gets the CPU for a fixed time in a cyclic way (turn by turn). It is the oldest scheduling algorithm, which is mainly used for **multitasking**.

Characteristics of Round-Robin Scheduling:

Here are the important characteristics of Round-Robin Scheduling:

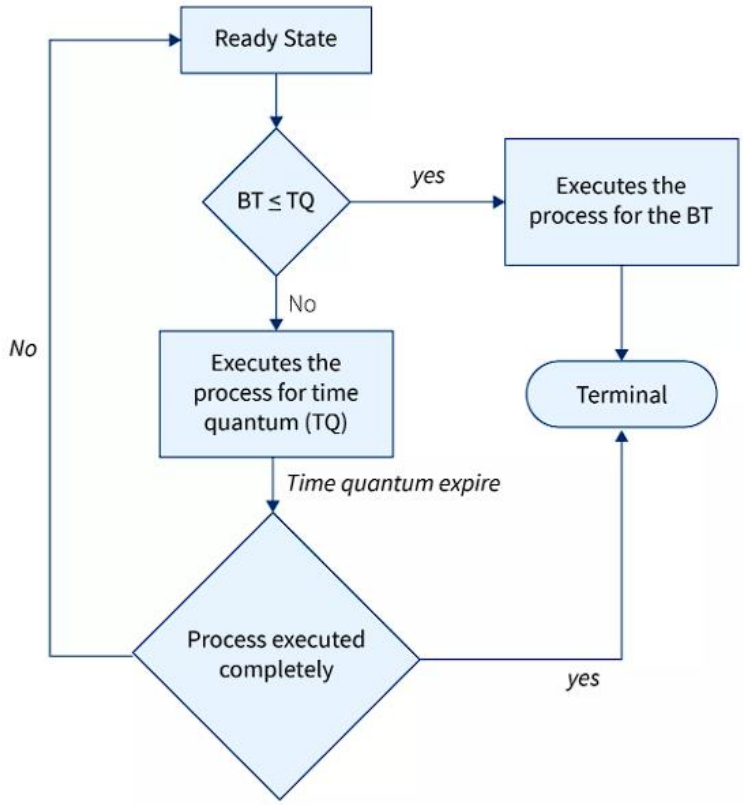
- Round robin is **a pre-emptive** algorithm.
- The CPU is shifted to the next process after fixed interval time, which is **called time quantum/time slice.**
- The process that is preempted is added to the **end of the queue.**
- Round robin is a **hybrid model** which is **clock-driven.**
- Time slice should be **minimum**, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.
- It is a **real time algorithm** which responds to the event within a specific time limit.
- Round robin is one of the **oldest, fairest**, and **easiest** algorithm.
- **Widely used** scheduling method in traditional OS.

How does the Round Robin Algorithm Work?

1. All the processes are added to the ready queue.
2. At first, The burst time of every process is compared to the time quantum of the CPU.
3. If the burst time of the process is **less than or equal** to the time quantum in the round-robin scheduling algorithm, the process is executed to its burst time.

4. If the burst time of the process is **greater than** the time quantum, the process is executed up to the time quantum (TQ).
5. When the time quantum expires, it checks if the process is executed completely or not.
6. On completion, the process terminates. Otherwise, it goes back again to the *ready state*.

Consider the below flow diagram for a better understanding of Round Robin scheduling algorithm:



Example of Round Robin Scheduling Algorithm:

Consider the following 6 processes: **P1, P2, P3, P4, P5, and P6** with their arrival time and burst time as given below:

Q. What are the average waiting and turnaround times for the round-robin scheduling algorithm (RR) with a time quantum of **4 units**?

Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	6
P3	2	3
P4	3	1
P5	4	5
P6	6	4

Ready Queue:

At first, In the ready queue, process P1 will be executed for a time slice of 4 units. Since there are no processes initially, Process P1, with a burst time of 5 units, will be the only process in the ready queue.

P1
5

Ready Queue:

Along with the execution of P1, four more processes, P2, P3, P4, and P5, arrive in the ready queue. P1 will be added to the ready queue due to the remaining 1 unit.

P2	P3	P4	P5	P1
6	3	1	5	1

Ready Queue:

During the execution of P2, P6 arrived in the ready queue. Since P2 has not been completed, P2 will be added to the ready queue.

P3	P4	P5	P1	P6	P2
3	1	5	1	4	2

Ready Queue:

Similarly, P3 and P4 have been completed, but P5 has a remaining burst time of 1 unit. Hence it will be added back to the queue.

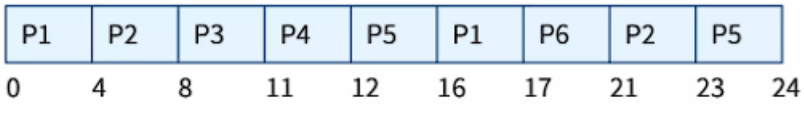
P1	P6	P2	P5
1	4	2	1

Ready Queue:

The next processes, P6 and P2, will be executed. Only P5 will be left with 1 unit of burst time.

P5
1

GANTT Chart:



As we know,

Turn Around Time = *Completion Time - Arrival Time*

Waiting Time = *Turn Around Time - Burst Time*

Processes	Arrival Time(AT)	Burst Time(BT)	Turn Around Time(TAT)	Waiting Time(WT)
P1	0	5	17	12
P2	1	6	22	16
P3	2	3	9	6
P4	3	1	9	8
P5	4	5	20	15
P6	6	4	15	11

Avg Waiting Time = (11+5+8+6+16+12)/6 = 68/6 = **11.33 units**.

Advantages of Round Robin in OS

1. This round-robin algorithm offers starvation-free execution of processes.
2. Each process gets equal priority and fair allocation of CPU.
3. Round Robin scheduling algorithm enables the Context switching method to save the states of preempted processes.
4. It is easily implementable on the system because round-robin scheduling in os doesn’t depend upon burst time.

Disadvantages of Round Robin in OS

1. The waiting and response times are higher due to the short time slot.
2. Lower time quantum results in higher context switching.
3. We cannot set any special priority for the processes.

5. Multilevel Queue Scheduling:

Operating systems use a particular kind of scheduling algorithm called multilevel queue scheduling to control how resources are distributed across distinct tasks. It is an adaptation of the conventional queue-based scheduling method, in which processes are grouped according to their priority, process type, or other factors.

The system can allocate system resources based on the priority and needs of the processes by assigning a separate scheduling algorithm to each queue. For instance, the background queue may employ first-come-first-serve scheduling to maximize the usage of system resources for longer-running activities, while the foreground queue might use Round Robin scheduling to prioritize interactive processes and speed up reaction time.

The goal of multilevel queue scheduling is to strike a compromise between fairness and performance. The system can increase overall performance while making sure that all processes are treated equitably by giving some processes more priority than others and distributing resources accordingly.

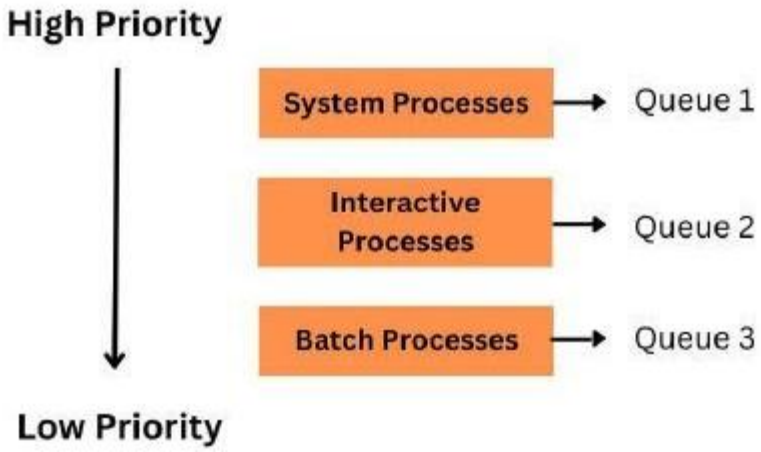
Features of Multilevel Queue (MLQ) CPU Scheduling:

- **Multiple queues:** In MLQ scheduling, processes are divided into multiple queues based on their priority, with each queue having a different priority level. Higher-priority processes are placed in queues with higher priority levels, while lower-priority processes are placed in queues with lower priority levels.
- **Priorities assigned:** Priorities are assigned to processes based on their type, characteristics, and importance. For example, interactive processes like user input/output may have a higher priority than batch processes like file backups.
- **Preemption:** Preemption is allowed in MLQ scheduling, which means a higher priority process can preempt a lower priority process, and the CPU is allocated to the higher priority process. This helps ensure that high-priority processes are executed in a timely manner.
- **Scheduling algorithm:** Different scheduling algorithms can be used for each queue, depending on the requirements of the processes in that queue. For example, Round Robin scheduling may be used for interactive processes, while First Come First Serve scheduling may be used for batch processes.
- **Feedback mechanism:** A feedback mechanism can be implemented to adjust the priority of a process based on its behavior over time. For example, if an interactive process has been waiting in a lower-priority queue for a long time, its priority may be increased to ensure it is executed in a timely manner.

- **Efficient allocation of CPU time:** MLQ scheduling ensures that processes with higher priority levels are executed in a timely manner, while still allowing lower priority processes to execute when the CPU is idle.
- **Fairness:** MLQ scheduling provides a fair allocation of CPU time to different types of processes, based on their priority and requirements.
- **Customizable:** MLQ scheduling can be customized to meet the specific requirements of different types of processes.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of processes System processes, Interactive processes, and Batch Processes. All three processes have their own queue.

Now, look at the below figure -



The Description of the processes in the above diagram is as follows:

- **System Processes:** The CPU itself has its own process to run which is generally termed a System Process. The system process is always given the highest priority.
- **Interactive Processes:** An Interactive Process is a type of process in which there should be the same type of interaction.
- **Batch Processes:** Batch processing is generally a technique in the Operating system that collects the programs and data together in the form of a batch before the processing starts.

Different Levels of Queues:

Processes are divided into many tiers of queues in multilevel queue scheduling according to their features, priority, and time limitations. The following various layers of queues can be utilized in multilevel queue scheduling:

- **Foreground Queue** − Interactive processes that demand quick system responses often use this queue. Usually, these procedures receive first attention
- **Background Queue** − This queue is used for non-interactive processes that take longer to complete. These tasks are prioritized less than foreground tasks.
- **Interactive queue** − The interactive queue is used for processes that demand a system response in a reasonable amount of time
- **Batch Queue** − This queue is used for batch processing, or processing many jobs at once. These tasks typically call for lengthy execution times and are submitted in advance.
- **System Queue** − This queue is used for system processes, such as device drivers and interrupt handlers, that need special permissions to complete their tasks. Usually, these procedures receive first attention.
- **Real-time Queue** − Real-time processes, which demand a prompt response from the system, are queued in this way. Usually, these procedures receive first attention.

Scheduling among the queues:

What will happen if all the queues have some processes? Which process should get the CPU? To determine this Scheduling among the queues is necessary. There are two ways to do so –

1. **Fixed priority preemptive scheduling method –** Each queue has absolute priority over the lower priority queue. Let us consider the following priority order **queue 1 > queue 2 > queue 3**. According to this algorithm, no process in the batch queue(queue 3) can run unless queues 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process(queue 2) entered the ready queue the batch process is preempted.
2. **Time slicing** – In this method, each queue gets a certain portion of CPU time and can use it to schedule its own processes. For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

Example Problem:

Consider the below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Below is the **Gantt chart** of the problem:

Working:

- At starting, both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round-robin fashion and completes after 7 units
- Then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 seconds and
- After its completion P3 takes the CPU and completes its execution.

Advantages of Multilevel Queue Scheduling

In comparison to other scheduling algorithms, multilevel queue scheduling has several advantages. Some of these advantages are

1. **Efficient Resource Utilization:** Multilevel queue scheduling allows the system to allocate resources more efficiently by grouping processes with similar resource requirements into separate queues.
2. **Improved Response Time:** By assigning higher priority to interactive processes that require a fast response time, multilevel queue scheduling can reduce response time and improve system performance.
3. **Better Throughput:** Multilevel queue scheduling can improve the overall throughput of the system by executing processes more efficiently. The system can execute multiple processes concurrently from different queues, improving the overall efficiency of the system.
4. **Flexibility:** Multilevel queue scheduling can be customized to suit different types of applications or workloads by adjusting the priority levels of the queues. This allows the system to adapt to changing workload demands, ensuring that resources are allocated efficiently.
5. **Fairness:** Multilevel queue scheduling can provide a fair allocation of CPU time to all processes by ensuring that each queue is executed in turn. This can prevent any single process from monopolizing the CPU and starving other processes of CPU time.

Disadvantages of Multilevel Queue Scheduling:

While multilevel queue scheduling has several advantages, there are also some drawbacks to using this algorithm. Here are some of them:

1. **Complexity:** Multilevel queue scheduling is a relatively complex scheduling algorithm that can be difficult to implement and manage. The system needs to maintain multiple separate queues with different priority levels, which can be challenging to maintain.
2. **Overhead:** Dividing the ready queue into multiple queues can increase the overhead associated with the scheduling algorithm, which can negatively impact system performance.
3. **Inflexibility:** The fixed priority levels of multilevel queue scheduling may not be suitable for all types of applications or workloads. Some applications may require more or less CPU time than the priority level assigned to them, which can result in inefficient resource allocation.
4. **Starvation:** If a queue has a large number of processes with a higher priority than those in other queues, the processes in the lower-priority queues may be starved of CPU time.

6. Multilevel Feedback Queue Scheduling:

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling is like Multilevel Queue (MLQ) Scheduling but in this process can move between the queues. And thus, much more efficient than multilevel queue scheduling.

Characteristics of Multilevel Feedback Queue Scheduling:

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system, and processes are allowed to move between queues.
- As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead,

Features of Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling:

Multiple queues: Similar to MLQ scheduling, MLFQ scheduling divides processes into multiple queues based on their priority levels. However, unlike MLQ scheduling, processes can move between queues based on their behavior and needs.

Priorities adjusted dynamically: The priority of a process can be adjusted dynamically based on its behavior, such as how much CPU time it has used or how often it has been blocked. Higher-priority processes are given more CPU time and lower-priority processes are given less.

Time-slicing: Each queue is assigned a time quantum or time slice, which determines how much CPU time a process in that queue is allowed to use before it is preempted and moved to a lower priority queue.

Feedback mechanism: MLFQ scheduling uses a feedback mechanism to adjust the priority of a process based on its behavior over time. For example, if a process in a lower-priority queue uses up its time slice, it may be moved to a higher-priority queue to ensure it gets more CPU time.

Preemption: Preemption is allowed in MLFQ scheduling, meaning that a higher-priority process can preempt a lower-priority process to ensure it gets the CPU time it needs.

Advantages of Multilevel Feedback Queue Scheduling:

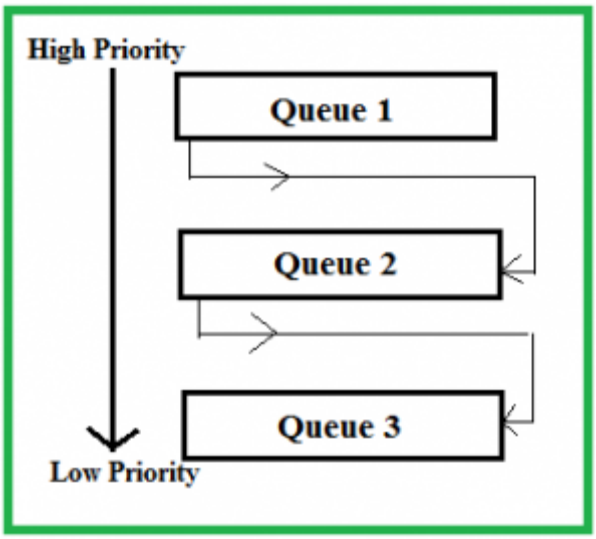
- It is more flexible.
- It allows different processes to move between different queues.
- It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.

Disadvantages of Multilevel Feedback Queue Scheduling:

- The selection of the best scheduler, it requires some other means to select the values.
- It produces more CPU overheads.
- It is the most complex algorithm.

Multilevel feedback queue scheduling, however, allows a process to move between queues. Multilevel Feedback Queue Scheduling **(MLFQ)** keeps analyzing the behavior (time of execution) of processes and according to which it changes its priority.

Now, look at the diagram and explanation below to understand it properly.



Now let us suppose that queues 1 and 2 follow round robin with time quantum 4 and 8 respectively and queue 3 follow FCFS.

Implementation of MFQS is given below –

- When a process starts executing the operating system can insert it into any of the above three queues depending upon its **priority**. For example, if it is some background process, then the operating system would not like it to be given to higher priority queues such as queues 1 and 2. It will directly assign it to a lower priority queue i.e. queue 3. Let's say our current process for consideration is of significant priority so it will be given **queue 1**.
- In queue 1 process executes for 4 units and if it completes in these 4 units or it gives CPU for I/O operation in these 4 units then the priority of this process does not change and if it again comes in the ready queue then it again starts its execution in Queue 1.
- If a process in queue 1 does not complete in 4 units then its priority gets reduced and it is shifted to queue 2.
- Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 units. In a general case if a process does not complete in a time quantum then it is shifted to the lower priority queue.
- In the last queue, processes are scheduled in an FCFS manner.
- A process in a lower priority queue can only execute only when higher priority queues are empty.
- A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

Also, the above implementation may differ for the example in which the last queue will follow **Round-robin Scheduling.**

In the above Implementation, there is a problem and that is; Any process that is in the lower priority queue has to suffer starvation due to some short processes that are taking all the CPU time.

And the solution to this problem is : There is a solution that is to boost the priority of all the process after regular intervals then place all the processes in the highest priority queue.

What is the need for such complex Scheduling?

- Firstly, it is more flexible than multilevel queue scheduling.
- To optimize turnaround time algorithms like SJF are needed which require the running time of processes to schedule them. But the running time of the process is not known in advance. MFQS runs a process for a time quantum and then it can change its priority(if it is a long process). Thus it learns from past behavior of the process and then predicts its future behavior. This way it tries to run a shorter process first thus optimizing turnaround time.
- MFQS also reduces the response time.

Example: Consider a system that has a CPU-bound process, which requires a burst time of 40 seconds. The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum '2' seconds and in each level it is incremented by '5' seconds. Then how many times the process will be interrupted and in which queue the process will terminate the execution?

Solution:

- Process P needs 40 Seconds for total execution.
- At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.
- At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
- At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
- At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
- At Queue 5 it executes for 2 seconds and then it completes.
- Hence the process is interrupted 4 times and completed on queue 5.

ROM and Its Role in CPU Scheduling:

ROM is a non-volatile memory that stores essential data even when the computer is turned off. Its influence in CPU scheduling in OS is akin to a conductor's sheet music, guiding the orchestration of tasks.

Defining ROM: Much like a reference book, ROM is a memory chip that stores fixed data. This data includes essential instructions and algorithms crucial for various system functions.

Role in CPU Scheduling: In CPU scheduling, ROM serves as a repository of scheduling algorithms, rules, and criteria. When the scheduler decides which task should run next, it consults this stored knowledge in ROM.

Example: Think of ROM as a rulebook in a game. Just as players follow the rules to keep the game fair, CPU scheduling algorithms in OS adhere to the guidelines stored in ROM to ensure balanced task execution.

ROM ensures consistency in CPU scheduling, providing a standardized approach to task management. It's like a navigator for the operating system, guiding it through the complex terrain of task prioritization. While it might not be directly visible, ROM's influence is woven into a smooth and efficient system performance fabric.

Multiple Processor Scheduling:

A multi-processor is a system that has more than one processor but shares the same memory, bus, and input/output devices. The bus connects the processor to the RAM, to the I/O devices, and to all the other components of the computer.

The system is a tightly coupled system. This type of system works even if a processor goes down. The rest of the system keeps working. **In multi-processor scheduling, more than one processors (CPUs) share the load to handle the execution of processes smoothly**.

The scheduling process of a multi-processor is more complex than that of a single processor system because of the following reasons -

- Load balancing is a problem since more than one processors are present.
- Processes executing simultaneously may require access to shared data.
- Cache affinity should be considered in scheduling.

Approaches to Multiple Processor Scheduling:

- **Symmetric Multiprocessing**: In symmetric multi-processor scheduling, the processors are self-scheduling. The scheduler for each processor checks the ready queue and selects a process to execute. Each of the processors works on the same copy of the operating system and communicates with each other. If one of the processors goes down, the rest of the system keeps working.
    - **Symmetrical Scheduling with global queues**: If the processes to be executed are in a common queue or a global queue, the scheduler for each processor checks this global-ready queue and selects a process to execute.

    - **Symmetrical Scheduling with per queues**: If the processors in the system have their own private ready queues, the scheduler for each processor checks their own private queue to select a process.

- **Asymmetric Multiprocessing**: In asymmetric multi-processor scheduling, there is a master server, and the rest of them are slave servers. The master server handles all the scheduling processes and I/O processes, and the slave servers handle the users' processes. If the master server goes down, the whole system comes to a halt. However, if one of the slave servers goes down, the rest of the system keeps working.

Processor Affinity:

A process has an affinity for a processor on which it runs. This is called **processor affinity**.

Let's try to understand why this happens.

- When a process runs on a processor, the data accessed by the process most recently is populated in the cache memory of this processor. The following data access calls by the process are often satisfied by the cache memory.
- However, if this process is migrated to another processor for some reason, the content of the cache memory of the first processor is invalidated, and the second processor's cache memory has to be repopulated.
- To avoid the cost of invalidating and repopulating the cache memory, the Migration of processes from one processor to another is avoided.

There are two types of processor affinity.

- **Soft Affinity**: The system has a rule of trying to keep running a process on the same processor but does not guarantee it. This is called soft affinity.
- **Hard Affinity**: The system allows the process to specify the subset of processors on which it may run, i.e., each process can run only some of the processors. Systems such as Linux implement soft affinity, but they also provide system calls such as sched_setaffinity() to support hard affinity.

Load Balancing:

In a multi-processor system, all processors may not have the same workload. Some may have a long ready queue, while others may be sitting idle. To solve this problem, load balancing comes into the picture. **Load Balancing** is the phenomenon of distributing workload so that the processors have an even workload in a symmetric multi-processor system.

In symmetric multiprocessing systems which have a global queue, load balancing is not required. In such a system, a processor examines the global ready queue and selects a process as soon as it becomes ideal.

However, in asymmetric multi-processor with private queues, some processors may end up idle while others have a high workload. There are two ways to solve this.

- **Push Migration**: In push migration, a task routinely checks the load on each processor. Some processors may have long queues while some are idle. If the workload is unevenly distributed, it will extract the load from the overloaded processor and assign the load to an idle or a less busy processor.
- **Pull Migration**: In pull migration, an idle processor will extract the load from an overloaded processor itself.

Multi-Core Processors:

A **multi-core processor** is a single computing component comprised of two or more CPUs called cores. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. A processor register can hold an instruction, address, etc. Since each core has a register set, the system behaves as a multi-processor with each core as a processor.

Symmetric multiprocessing systems which use multi-core processors allow higher performance at low energy.

Symmetric multiprocessor

In **Symmetric multi-processors**, the memory has only one operating system, which can be run by any central processing unit. When a system call is made, the CPU on which the system call was made traps the kernel and processed that system call. The model works to balance processes and memory dynamically. As the name suggests, it uses symmetric multiprocessing to schedule processes, and every processor is self-scheduling. Each processor checks the global or private ready queue and selects a process to execute it.

Note: The kernel is the central component of the operating system. It connects the system hardware to the application software.

There are three ways of conflict that may arise in a symmetric multi-processor system. These are as follows.

- **Locking system**: The resources in a multi-processor are shared among the processors. To the access safe of these resources to the processors, a locking system is required. This is done to serialize the access of the resources by the processors.

- **Shared data**: Since multiple processors are accessing the same data at any given time, the data may not be consistent across all of these processors. To avoid this, we must use some kind of strategy or locking scheme.
- **Cache Coherence**: When the resource data is stored in multiple local caches and shared by many clients, it may be rendered invalid if one of the clients changes the memory block. This can be resolved by maintaining a consistent view of the data.

Master-Slave Multiprocessor:

In a **master-slave multi-processor**, one CPU works as a master while all others work as slave processors. This means the master processor handles all the scheduling processes and the I/O processes while the slave processors handle the user's processes. **The memory and input-output devices are shared among all the processors, and all the processors are connected to a common bus. It uses asymmetric multiprocessing to schedule processes.**

Virtualization and Threading:

Virtualization is the process of running multiple operating systems on a computer system. So, a single CPU can also act as a multi-processor. This can be achieved by having a host operating system and other guest operating systems.

- Different applications run on different operating systems without interfering with one another.
- A virtual machine is a virtual environment that functions as a virtual computer with its CPU, memory, network interface, and storage, created on a physical hardware system.
- In a time-sharing OS, 100ms(millisecond) is allocated to each time slice to give users a reasonable response time. But it takes more than 100ms, maybe 1 second or more. This results in a poor response time for users logged into the virtual machine.
- Since the virtual operating systems receive a fraction of the available CPU cycles, the clocks in virtual machines may be incorrect. This is because their timers do not take any longer to trigger than they do on dedicated CPUs.

Use Cases of Multiple Processors Scheduling in Operating System:

Now, we will discuss a few of the use cases of Multiple Processor Scheduling in Operating Systems−

1. **High-Performance Computing (HPC):** Enables parallel execution for faster computation in scientific simulations and data analysis.
2. **Server Virtualization:** Ensures fair resource allocation among virtual machines on a single physical server with multiple processors.
3. **Real-Time Systems:** Meets strict timing requirements in aerospace, defense, and industrial automation through algorithms like Earliest Deadline First (EDF).
4. **Multimedia Processing:** Enables parallel execution for faster video rendering and audio processing, ensuring smooth real-time performance.
5. **Distributed Computing:** Improves load balancing, fault tolerance, and resource utilization in collaborative processing across multiple nodes.
6. **Cloud Computing:** Optimizes virtual machine and container allocation for fairness, scalability, and efficient resource utilization in cloud environments.
7. **Big Data Processing:** Accelerates data processing tasks in big data analytics through parallel execution, enabling real-time insights.
8. **Scientific Simulations:** Facilitates parallel execution of numerical simulations, reducing time for obtaining results in scientific research.
9. **Gaming:** Maximizes utilization of multiple processors for complex graphics rendering, physics simulations, and AI computations in gaming systems.
10. **Embedded Systems:** Ensures timely response and coordination of tasks in automotive systems, IoT devices, and robotics with multiple processors.

Types of Multiprocessor Scheduling Algorithms:

Operating systems utilize a range of multiprocessor scheduling algorithms. Among the most typical types are −

Round-Robin Scheduling − The round-robin scheduling algorithm allocates a time quantum to each CPU and configures processes to run in a round-robin fashion on each processor. Since it ensures that each process gets an equivalent amount of CPU time, this strategy might be useful in systems wherein all programs have the same priority.

Priority Scheduling − Processes are given levels of priority in this method, and those with greater priorities are scheduled to run first. This technique might be helpful in systems where some jobs, like real-time tasks, call for a higher priority.

Scheduling with the shortest job first (SJF) − This algorithm schedules tasks according to how long they should take to complete. It is planned for the shortest work to run first, then the next smallest job, and so on. This technique can be helpful in systems with lots of quick processes since it can shorten the typical response time.

Fair-share scheduling − In this technique, the number of processors and the priority of each process determine how much time is allotted to each. As it ensures that each process receives a fair share of processing time, this technique might be helpful in systems with a mix of long and short processes.

Earliest deadline first (EDF) scheduling − Each process in this algorithm is given a deadline, and the process with the earliest deadline is the one that will execute first. In systems with real-time activities that have stringent deadlines, this approach can be helpful.

Scheduling using a multilevel feedback queue (MLFQ) − Using a multilayer feedback queue (MLFQ), processes are given a range of priority levels and are able to move up or down the priority levels based on their behavior. This strategy might be useful in systems with a mix of short and long processes.

Advantages of Multiple-Processor Scheduling:

- **Efficient Multitasking**: Improved multitasking capabilities with the ability to handle multiple tasks simultaneously.
- **Resource Utilization**: Maximizes the use of system resources, optimizing efficiency.
- **Fault Tolerance**: Redundant processors can provide system reliability in case of failures.
- **Scalability**: Easy scalability by adding more processors to meet growing demands.

Disadvantages of Multiple-Processor Scheduling:

- **Increased Complexity:** Managing multiple processors adds complexity to scheduling and resource allocation.
- **Synchronization Overhead:** Coordinating tasks among processors can introduce overhead.
- **Software Compatibility:** Some software may not be optimized for multiprocessor systems.
- **Cost:** Multiprocessor systems can be more expensive to build and maintain.

Real Time Scheduling:

What are Real-time operating systems?

Real-time operating systems (RTOS) play an important role in ensuring that embedded systems meet strict timing constraints. These systems are employed in a wide range of applications, which include automotive control systems, medical devices, aerospace, and industrial automation, etc. Here, the timely and deterministic execution of tasks is paramount. One of the key features that make RTOS suitable for these applications is their multi-thread scheduling mechanism.

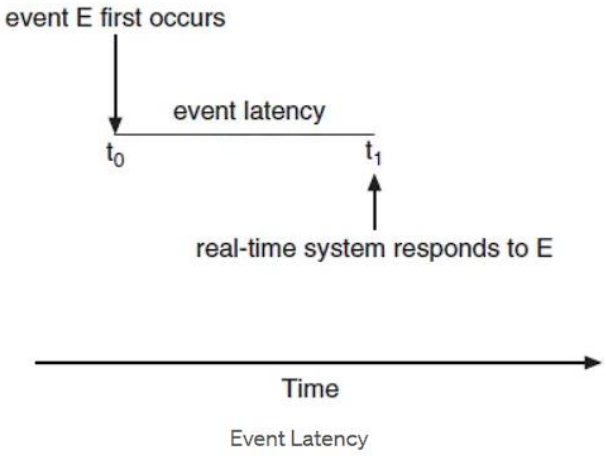
CPU scheduling for real-time operating systems involves special issues. There are 2 major types of systems:

1. **Soft real time systems** — They provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes.
2. **Hard real time systems** — They have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

There are certain issues associated with these types of scheduling. One such is the task of minimising latency.

Minimising Latency:

Event latency refers to the amount of time that elapsed between the event occurring and when it is serviced. Different events may have different latency requirements.

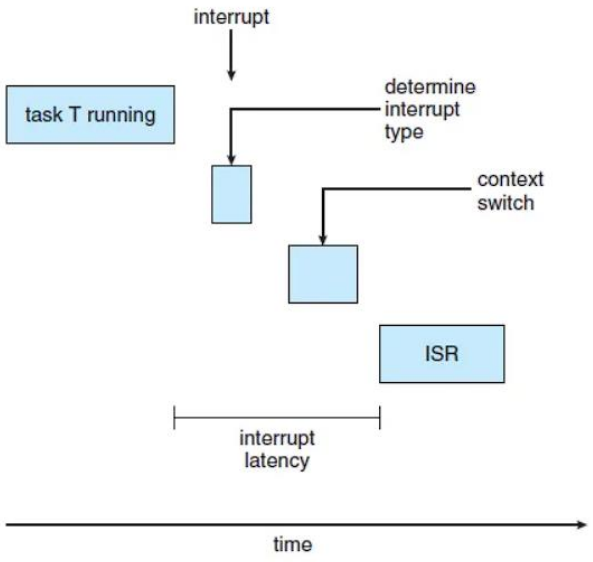


Event Latency

There are 2 types of latency which affect real time systems. They are -

1. *Interrupt Latency*
2. *Dispatch Latency*

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the **interrupt latency**.

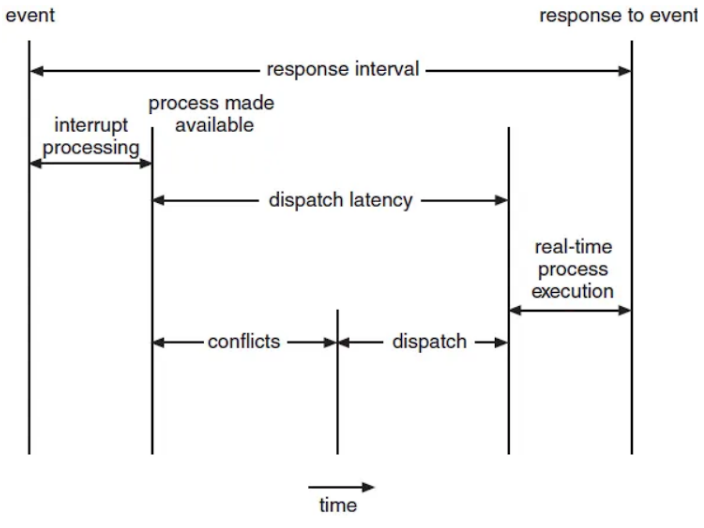


Interrupt Latency

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.

Dispatch latency is the amount of time required by the scheduling dispatcher to stop one process and start another. The most effective technique for keeping dispatch latency low is to provide preemptive kernels.

The below figure illustrates the dispatch latency:



Dispatch Latency

The conflict phase in dispatch latency has 2 parts:

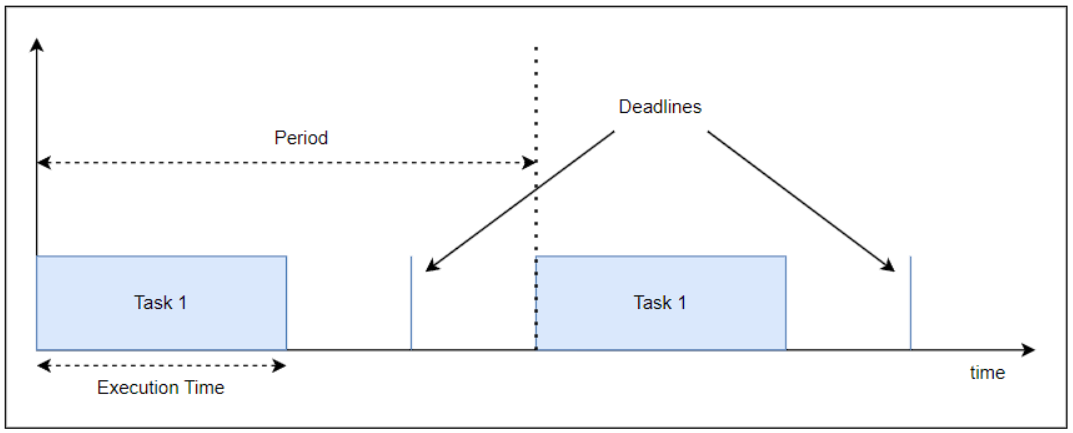
1. Preemption of any process running in the kernel.
2. Release by low-priority processes of resources needed by a high-priority process

Real-time Scheduling:

Real-time scheduling in RTOS involves the management and execution of multiple threads (also known as tasks) with the objective of ensuring that tasks meet their deadlines consistently. Each task in the system has associated properties, including priority, execution time, and periodicity, which the scheduler uses to determine the order in which tasks are executed. The primary goal of real-time scheduling is to minimize task latencies and guarantee the completion of critical tasks within their specified time constraints.

Before exploring the different scheduling algorithms, it is important to understand the fundamental characteristics of a single task, namely:

- **Period:** The period of the task is the time between two subsequent occurrences of the task. It quantifies how often we can expect the task to occur.
- **Deadlines:** The deadline is the time within which each task that must be executed. If a task is not completed within the deadline, it is considered a failure.
- **Execution time:** The execution time is a measure of how long does it take to fully process as task.

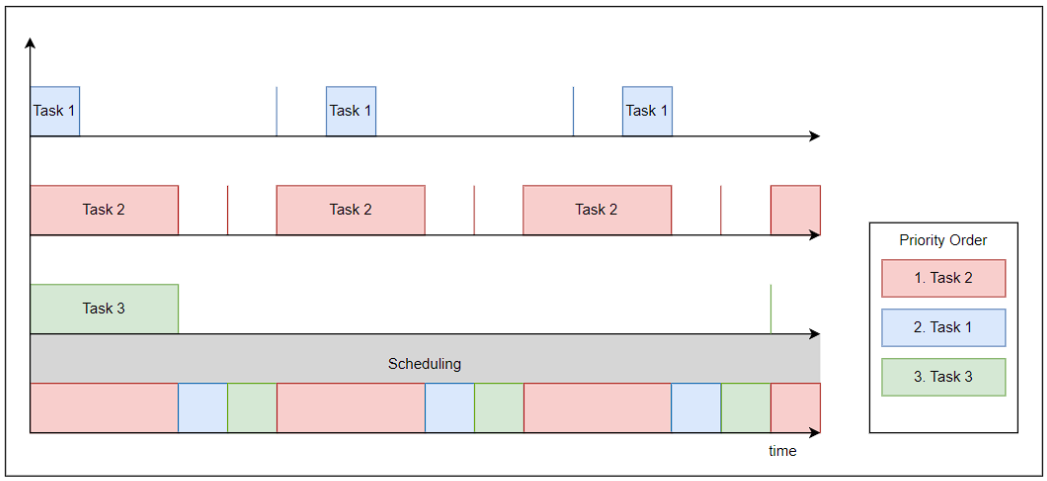


Task characteristics

RTOS systems implement various scheduling algorithms to manage the execution of tasks. Let’s look at some of the most commonly used scheduling algorithms -

Fixed-priority preemptive scheduling:

In this approach, each task is assigned a fixed priority level, and the scheduler selects the highest priority task ready to run. Tasks with higher priority preempt tasks with lower priority, ensuring that higher priority tasks are executed without delay. Fixed-priority scheduling is simple and suitable for systems where task priorities remain constant.

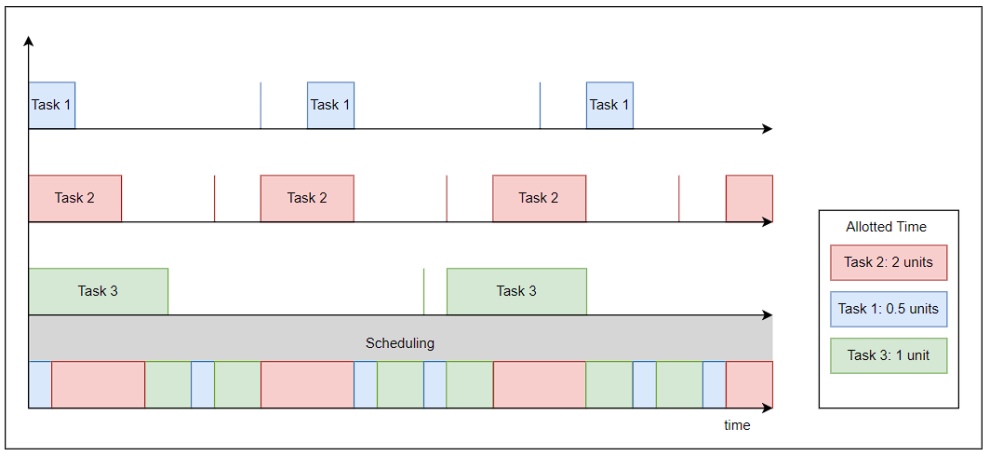


Fixed-priority preemptive scheduling

In the illustration above, we can see that as Task 2 has the highest priority, the scheduler abandons any other task it is processing to first cater to it. Only after Task 2 has been executed does the scheduler move on to the other tasks, according to the priority list.

Round-robin scheduling:

Round-robin is a time-sharing scheduling algorithm where tasks are assigned a fixed time range. Each task runs for its allotted time, and if it isn't completed within the range, it is moved to the back of the queue, allowing other tasks to execute. Round-robin is useful but may not be ideal for real-time systems with strict deadlines.

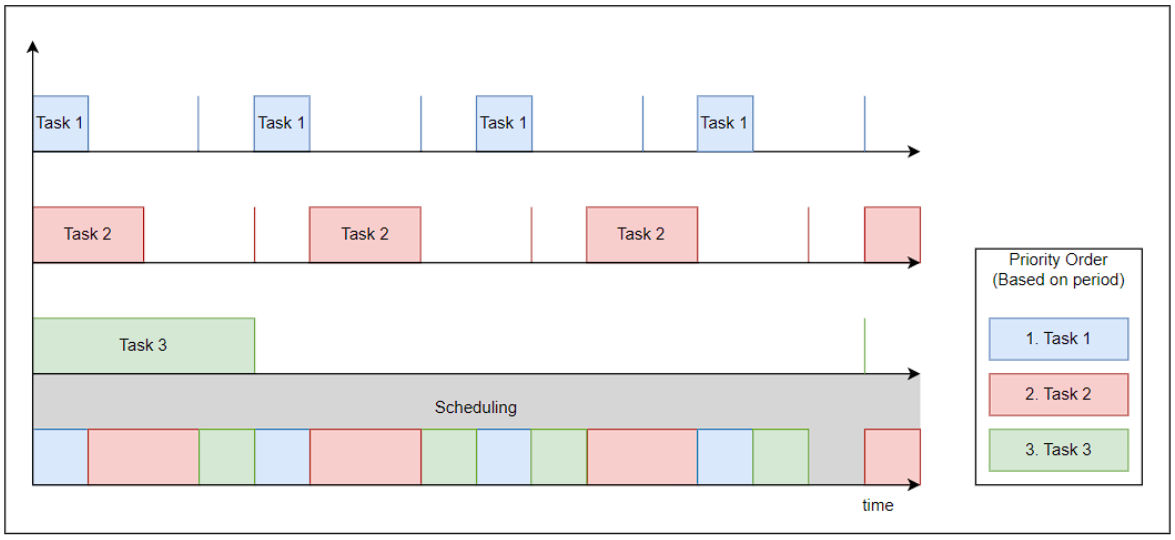


Round-robin scheduling

In the illustration above, we can see that each task is only allowed to run its specific allotted time in one execution. Task 1 executes for 0.5 units and is abandoned for task 2, which is also succeeded by Task 3 after completing its allotted time. As the allotted time of Task 2 is just as much as its execution time, it is always processed in one go. On the contrary, the allotted time for Task 3 is less than its execution time, so it is processed in multiple attempts.

Rate-monotonic scheduling:

Rate-monotonic scheduling algorithm assigns priorities based on a task's period, with shorter periods receiving higher priorities. Tasks with shorter deadlines are scheduled more frequently, making this algorithm suitable for periodic tasks in hard real-time systems.

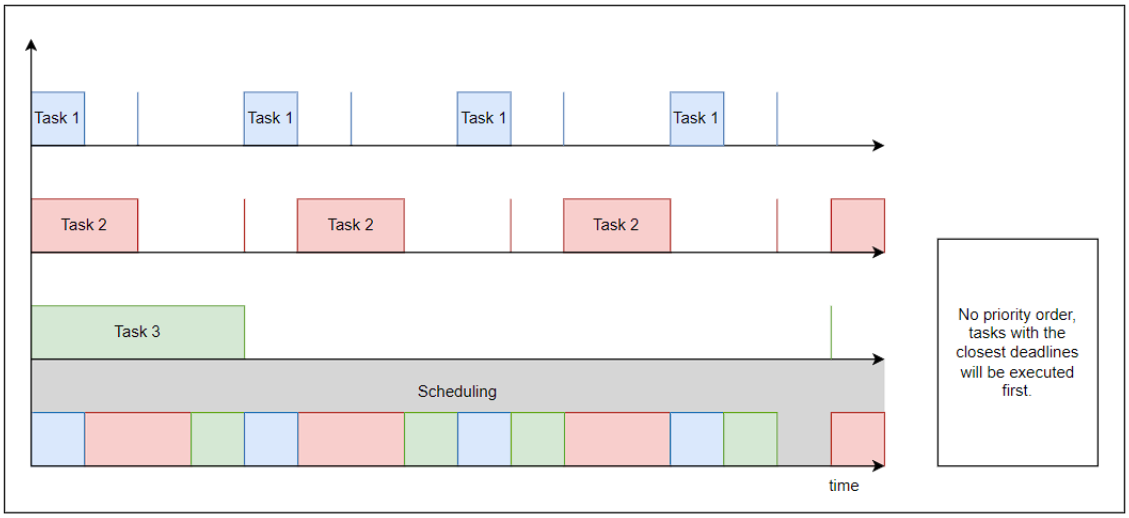


Rate-monotonic scheduling

In the illustration above, we can see that Task 1 occurs most frequently, thus, it is given the highest priority. Task 2 comes next on the priority list, followed by Task 3. The scheduler always prioritizes Task 1, abandoning other tasks if Task 1 occurs. The other tasks are only processed after Task 1 is completed.

Earliest deadline first scheduling:

Commonly known as **EDF**, this algorithm assigns priorities dynamically based on each task's next deadline. The task with the earliest deadline is given the highest priority. EDF ensures that tasks with impending deadlines are executed first, but requires more complex scheduling algorithms.



EDF scheduling

In the illustration above, we can see that the task that has the closest deadlines is prioritized over other tasks. When Task 3 is being executed, Task 1 occurs, and because its deadline is closer, the scheduler abandons Task 3 to process Task 1 first. Thus, tasks with the shortest deadlines are mostly processed in one go, while tasks with very long deadlines are mostly processed in small chunks.

Selecting a scheduling algorithm:

Choosing the right scheduling algorithm for an RTOS system depends on various factors:

- **Task characteristics:** Consider the nature of the tasks, whether they are periodic, aperiodic, or sporadic. Some algorithms, like rate-monotonic, are well-suited for periodic tasks, while others, like EDF, handle aperiodic tasks more effectively.
- **System requirements:** Analyze the system's real-time requirements, including deadlines, response times, and throughput. Ensure that the selected algorithm can meet these requirements.
- **Resource constraints:** Assess the hardware resources available, such as CPU cores, memory, and I/O devices. Some algorithms may be more efficient in utilizing these resources than others.
- **Predictability:** Different algorithms offer varying levels of predictability. Fixed-priority algorithms are generally more predictable, while dynamic algorithms like EDF offer better responsiveness but can be harder to analyze.
- **Complexity:** Consider the complexity of the scheduling algorithm in relation to the available computing resources. Simpler algorithms are easier to implement and debug.

In addition to the scheduling algorithms introduced above, there are several other algorithms, such as cooperative scheduling, fixed-priority non-preemptive scheduling, critical section preemptive scheduling, and static time scheduling, etc. Each algorithm has its strengths and weaknesses, and deciding which one to use requires careful analysis of our system and the required outputs.

Disadvantages:

- In real-time tasks, testing is a tedious process where each task has to be verified for its completion within the deadlines, and validating the same if any delay happens takes more time.
- Resources needed for all the tasks assigned to meet its requirements may not be available in the system when there are several tasks to execute at the same time and thus resulting in starvation of resources.
- Complexity is higher as real-time tasks must be analyzed in terms of requirements, deadlines, and resources.

FINALLY, THE END 😴