OPERATING SYSTEMS

MODULE-2

Processes: Process Concept, Process scheduling, Operations on processes, Inter-process communication.
Threads and Concurrency: Multithreading models, Thread libraries, Threading issues. **CPU Scheduling:** Basic concepts, Scheduling criteria, Scheduling algorithms, Multiple processor scheduling.

THE PROCESS

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs. These needs resulted in the notion of a **Process,** which is a program in execution.

A process is the unit of work in a modern computing system. Even if a computer can execute only one program at a time the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **Processes.** The status of the current activity of a process is represented by the value of the **program counter** and the **contents of the processor's registers.** The memory layout of a process is typically divided into multiple sections.

These sections include:

- Text section: the executable code
- Data section: global variables
- Heap section: memory that is dynamically allocated during program run time.
- Stack section: temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

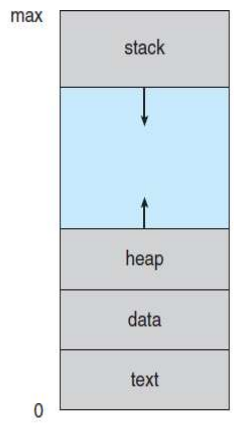


Figure Layout of a process in memory.

The sizes of the **Text** and **Data** sections are fixed, as their sizes do not change during program runtime. However, the **Stack** and **Heap** sections can shrink and grow dynamically during program execution. Each time a function is called, an **activation record** containing function parameters, local variables**,** and the return address is pushed onto the **Stack**. When control is returned from the function, the activation record is popped from the stack.

Similarly, the **heap** will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow toward one another, the operating system must ensure they do not overlap one another**.** A process is an **active entity,** with a program counter specifying the next instruction to execute and a set of associated resources.

Process scheduling

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Categories of Scheduling

There are two categories of scheduling:

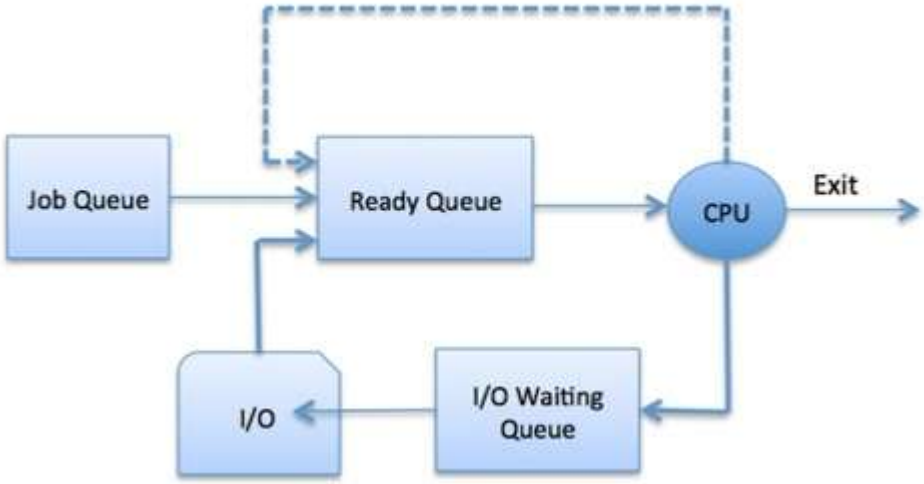
1. **Non-preemptive:** Here the resource can't be taken from a process until the process completes execution. The switching of resources occurs when the running process terminates and moves to a waiting state.
2. **Preemptive:** Here the OS allocates the resources to a process for a fixed amount of time. During resource allocation, the process switches from running state to ready state or from waiting state to ready state. This switching occurs as the CPU may give priority to other processes and replace the process with higher priority with the running process.

Process Scheduling Queues

The OS maintains all Process Control Blocks (PCBs) in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running and non-running states which are described below −

S.N.	State & Description
1	**Running** When a new process is created, it enters into the system as in the running state.
2	**Not Running** Processes that are not running are kept in queue, waiting for their turn to

execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler

It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

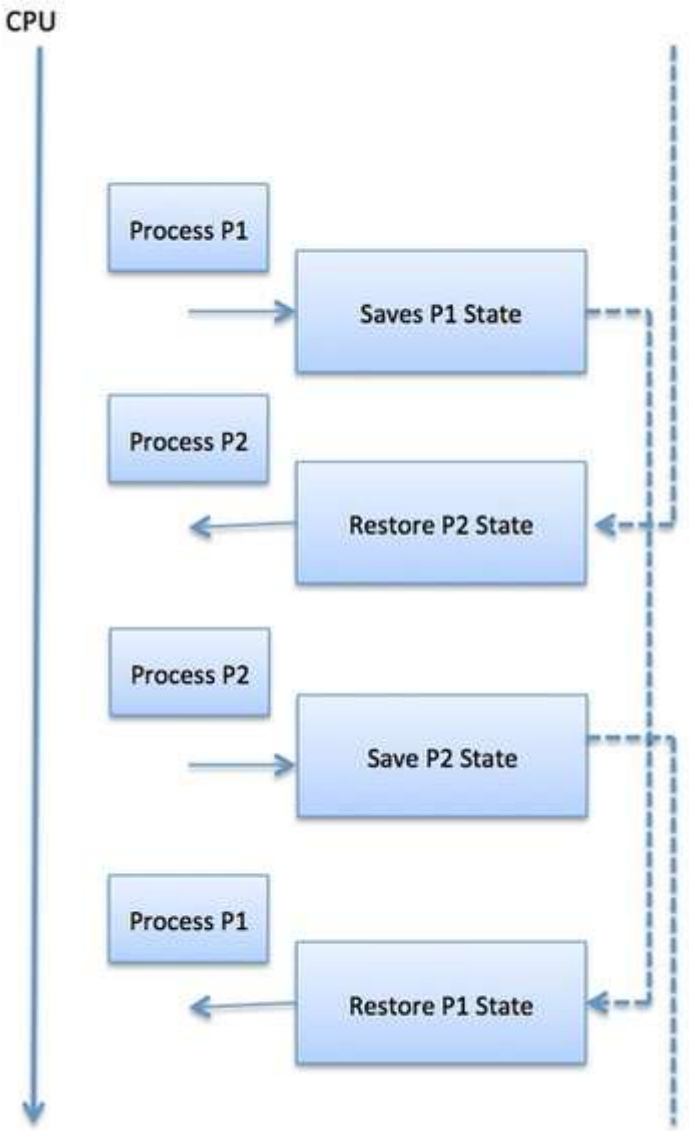
Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switching

A context switching is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

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

- Accounting information

Operations on Process

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 −

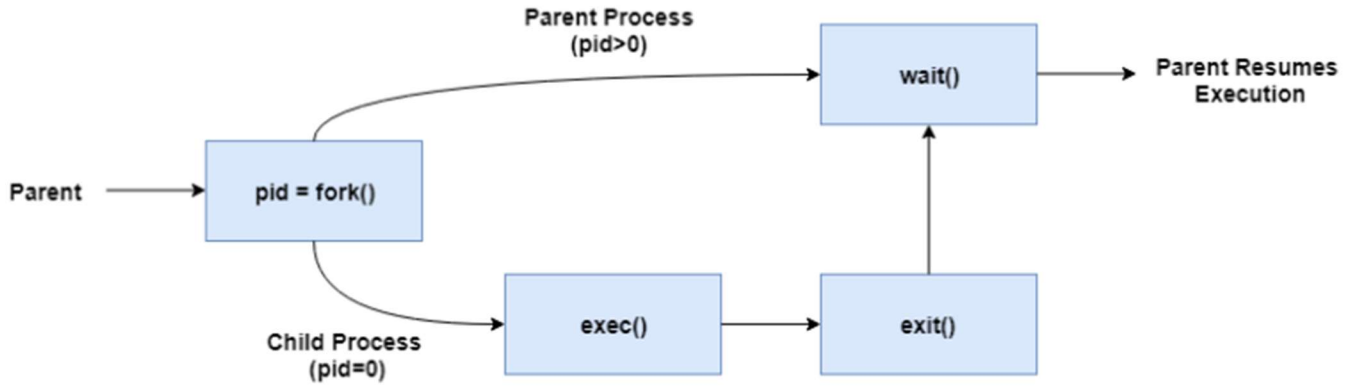
Process Creation

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 −

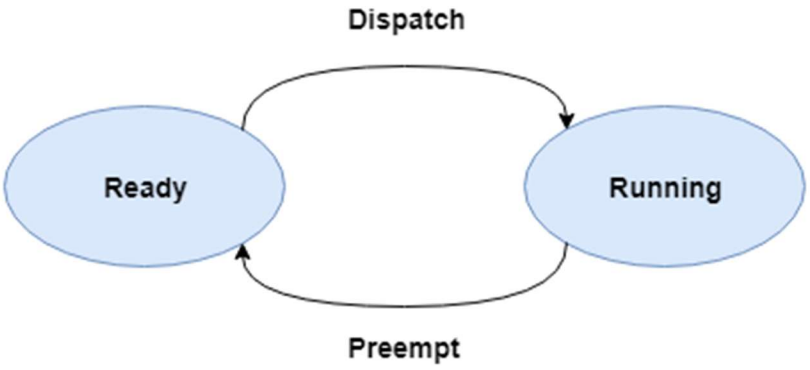


Process Creation Using fork()

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 −

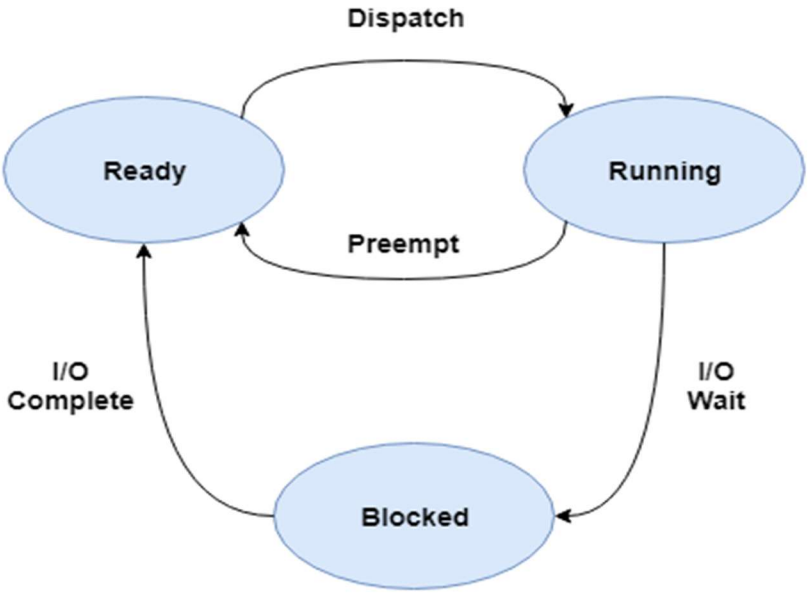


Process Preemption

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

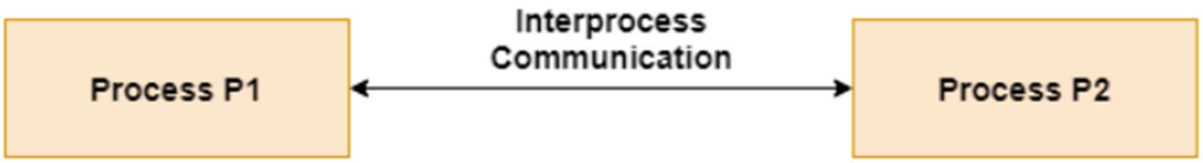
After the process has completed the execution of its last instruction, it is terminated. The resources held by a process are released after it is terminated.

A child process can be terminated by its parent process if its task is no longer relevant. The child process sends its status information to the parent process before it terminates. Also, when a parent process is terminated, its child processes are terminated as well as the child processes cannot run if the parent processes are terminated.

Inter-process communication

Interprocess 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 −



Synchronization in Interprocess Communication

Synchronization is a necessary part of interprocess communication. It is either provided by the interprocess control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows −

- **Semaphore**
  A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.
- **Mutual Exclusion**
  Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.
- **Barrier**
  A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.

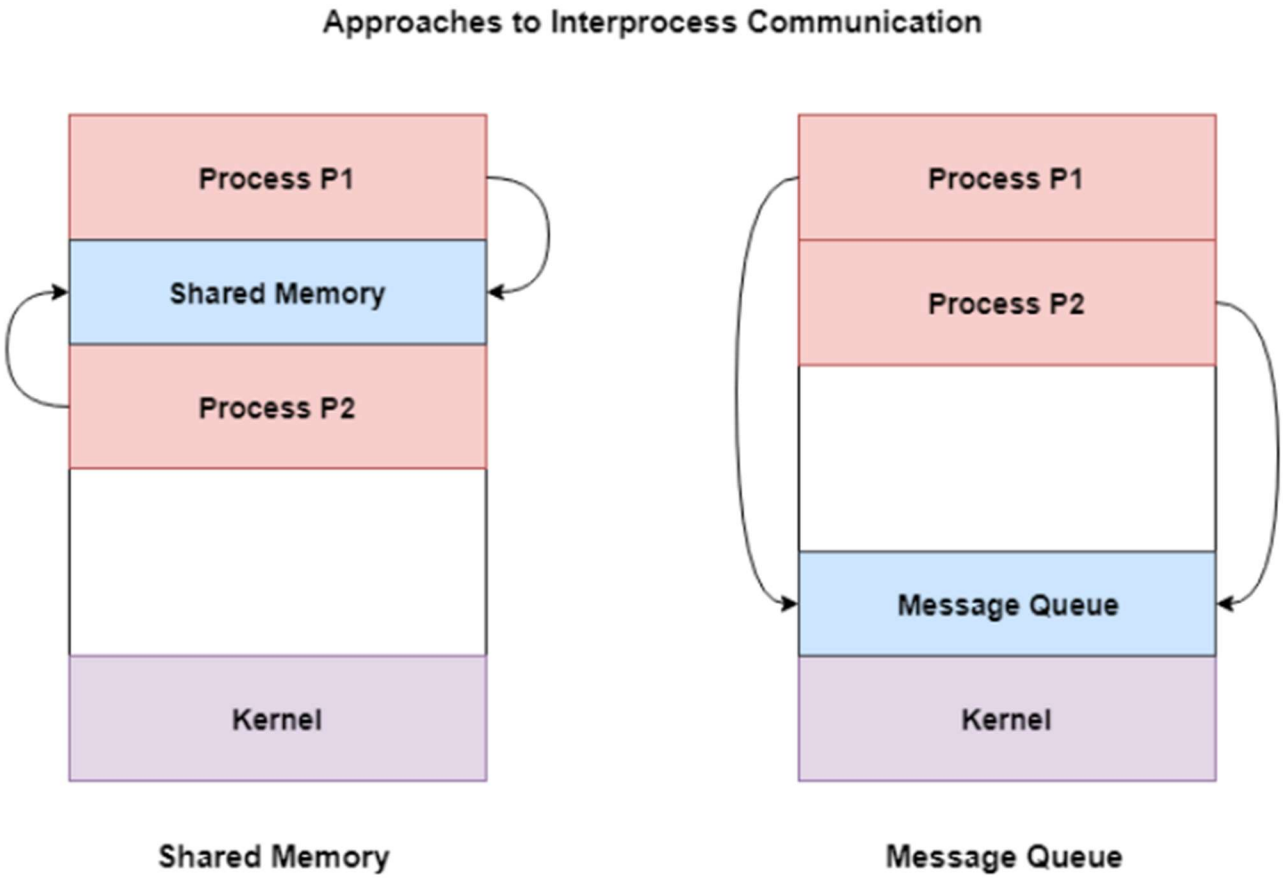
- **Spinlock**
  This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Approaches to Inter-process Communication

The different approaches to implement interprocess communication are given as follows –

- **Pipe**
  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.
- **Socket**
  The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.
- **File**
  A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.
- **Signal**
  Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.
- **Shared Memory**
  Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.
- **Message Queue**
  Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message queue and shared memory methods of interprocess communication is as follows −



Multithreading models

Multithreading allows the execution of multiple parts of a program at the same time. These parts are known as threads and are lightweight processes available within the process. Therefore, multithreading leads to maximum utilization of the CPU by multitasking.

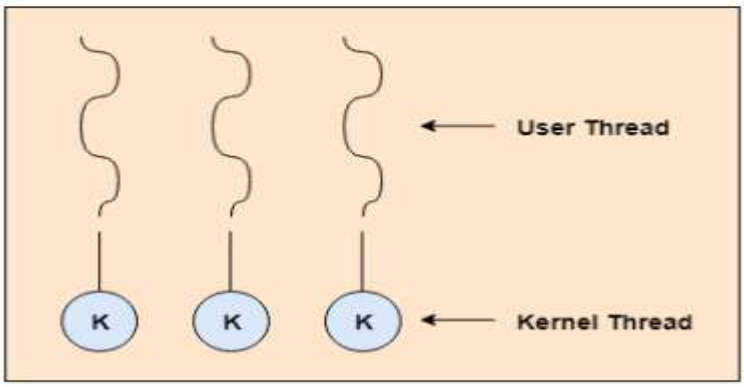
The main models for multithreading are one to one model, many to one model and many to many model. Details about these are given as follows −

One to One Model

The one to one model maps each of the user threads to a kernel thread. This means that many threads can run in parallel on multiprocessors and other threads can run when one thread makes a blocking system call.

A disadvantage of the one to one model is that the creation of a user thread requires a corresponding kernel thread. Since a lot of kernel threads burden the system, there is restriction on the number of threads in the system.

A diagram that demonstrates the one to one model is given as follows −



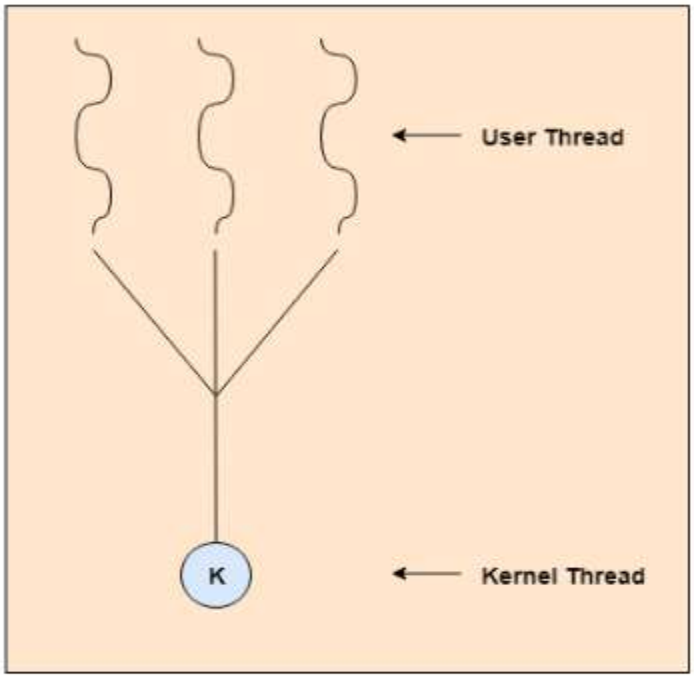
One to One Model

Many to One Model

The many to one model maps many of the user threads to a single kernel thread. This model is quite efficient as the user space manages the thread management.

A disadvantage of the many to one model is that a thread blocking system call blocks the entire process. Also, multiple threads cannot run in parallel as only one thread can access the kernel at a time.

A diagram that demonstrates the many to one model is given as follows −



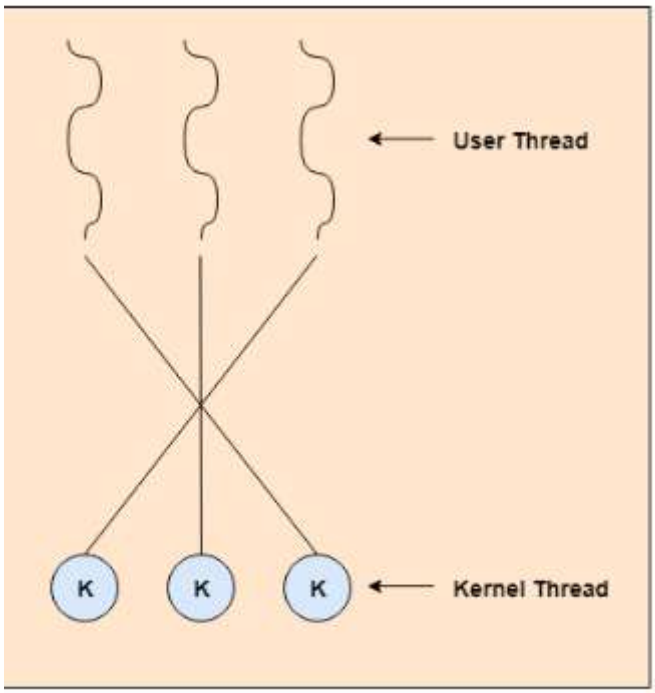
Many to One Model

Many to Many Model

The many to many model maps many of the user threads to a equal number or lesser kernel threads. The number of kernel threads depends on the application or machine.

The many to many does not have the disadvantages of the one to one model or the many to one model. There can be as many user threads as required and their corresponding kernel threads can run in parallel on a multiprocessor.

A diagram that demonstrates the many to many model is given as follows −

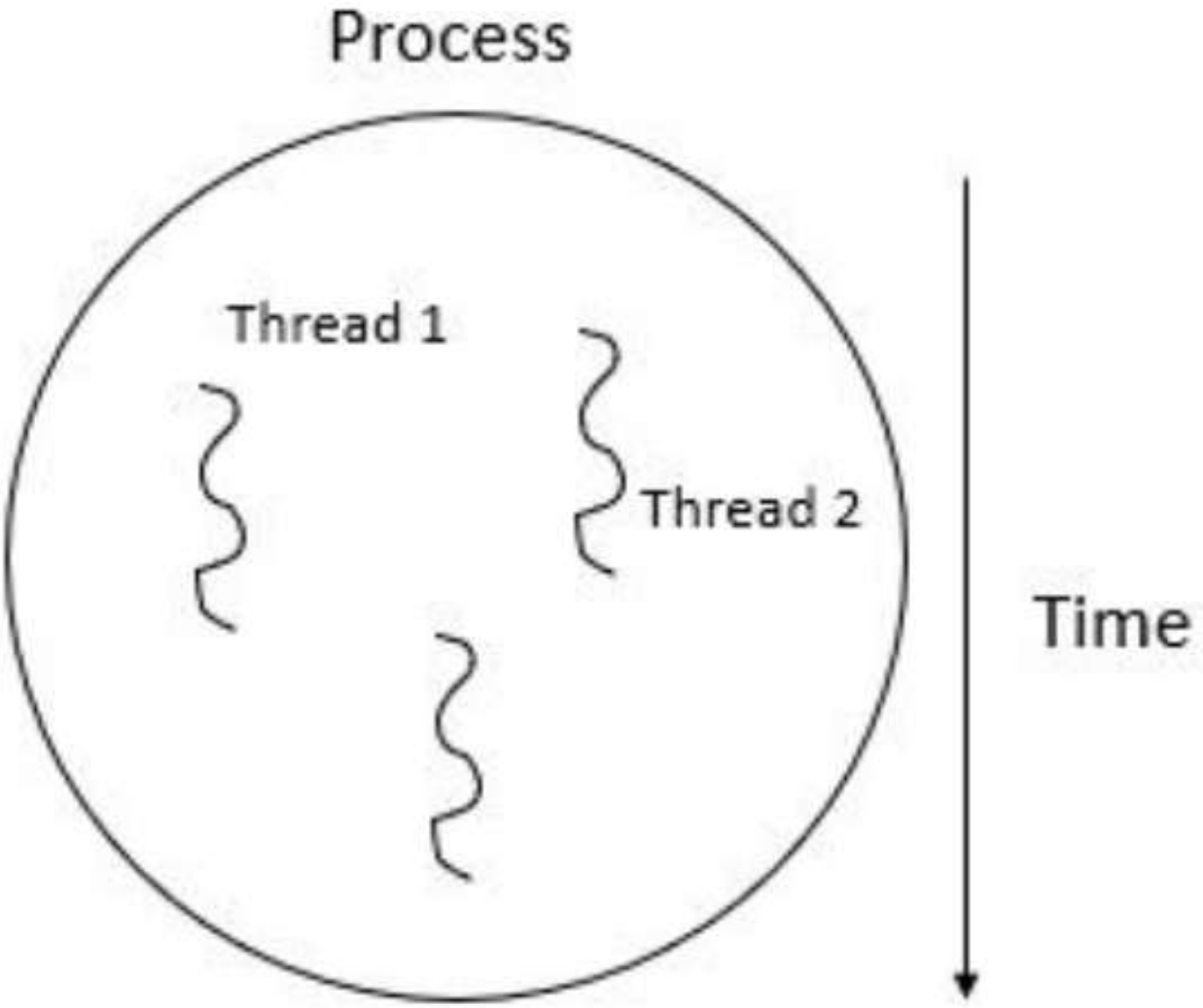


Many to Many Model

Thread libraries

A thread is a lightweight of process and is a basic unit of CPU utilization which consists of a program counter, a stack, and a set of registers.

Given below is the structure of thread in a process −



A process has a single thread of control where one program can counter and one sequence of instructions is carried out at any given time. Dividing an application or a program into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

Thread has the ability to share an address space and all of its data among themselves. This ability is essential for some specific applications.

Threads are lighter weight than processes, but they are faster to create and destroy than processes.

Thread Library

A thread library provides the programmer with an Application program interface for creating and managing thread.

Ways of implementing thread library

There are two primary ways of implementing thread library, which are as follows –

- The first approach is to provide a library entirely in user space with kernel support. All code and data structures for the library exist in a local function call in user space and not in a system call.
- The second approach is to implement a kernel level library supported directly by the operating system. In this case the code and data structures for the library exist in kernel space.

Invoking a function in the application program interface for the library typically results in a system call to the kernel.

The main thread libraries which are used are given below –

- **POSIX threads** − Pthreads, the threads extension of the POSIX standard, may be provided as either a user level or a kernel level library.
- **WIN 32 thread** − The windows thread library is a kernel level library available on windows systems.
- **JAVA thread** − The JAVA thread API allows threads to be created and managed directly as JAVA programs.

Threading Issues

We can discuss some of the issues to consider in designing multithreaded programs. These issued are as follows −

The fork() and exec() system calls

The fork() is used to create a duplicate process. The meaning of the fork() and exec() system calls change in a multithreaded program.

If one thread in a program which calls fork(), does the new process duplicate all threads, or is the new process single-threaded? If we take, some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

If a thread calls the exec() system call, the program specified in the parameter to exec() will replace the entire process which includes all threads.

Signal Handling

Generally, signal is used in UNIX systems to notify a process that a particular event has occurred. A signal received either synchronously or asynchronously, based on the source of and the reason for the event being signalled.

All signals, whether synchronous or asynchronous, follow the same pattern as given below –

- A signal is generated by the occurrence of a particular event.
- The signal is delivered to a process.
- Once delivered, the signal must be handled.

Cancellation

Thread cancellation is the task of terminating a thread before it has completed.

For example − If multiple database threads are concurrently searching through a database and one thread returns the result the remaining threads might be cancelled.

A target thread is a thread that is to be cancelled, cancellation of target thread may occur in two different scenarios −

- **Asynchronous cancellation** − One thread immediately terminates the target thread.
- **Deferred cancellation** − The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an ordinary fashion.

Thread polls

Multithreading in a web server, whenever the server receives a request it creates a separate thread to service the request.

Some of the problems that arise in creating a thread are as follows −

- The amount of time required to create the thread prior to serving the request together with the fact that this thread will be discarded once it has completed its work.
- If all concurrent requests are allowed to be serviced in a new thread, there is no bound on the number of threads concurrently active in the system.
- Unlimited thread could exhaust system resources like CPU time or memory.

A thread pool is to create a number of threads at process start-up and place them into a pool, where they sit and wait for work.

CPU Scheduling Basic concepts

CPU scheduling is a fundamental concept in operating systems that determines which processes run on the CPU at any given time. Efficient CPU scheduling is crucial for optimizing system performance, responsiveness, and resource utilization. Below are the basic concepts and criteria for CPU scheduling:

1. **Process**: A program in execution. Processes can be in different states like ready, running, or waiting.

2. **CPU Burst and I/O Burst:**

CPU Burst: The amount of time a process spends executing on the CPU before needing I/O.

I/O Burst: The time a process waits for I/O operations to complete.

3. **Scheduler**: The part of the operating system that selects processes for execution on the CPU. There are three types:

Long-term scheduler: Decides which processes should be brought into the ready queue.

Short-term scheduler (CPU scheduler): Decides which process in the ready queue will execute next.

Medium-term scheduler: Handles swapping of processes in and out of memory.

4. **Dispatcher**: The dispatcher is responsible for giving control of the CPU to the process selected by the short-term scheduler. This involves:

Switching context (saving and restoring process states)

Switching to user mode

Jumping to the correct location in the user program to resume execution

5. **Preemptive vs. Non-preemptive Scheduling**:

Preemptive Scheduling: The OS can take the CPU away from a running process before it completes its CPU burst (e.g., Round Robin, Priority Scheduling).

Non-preemptive Scheduling: Once the CPU is assigned to a process, it cannot be taken away until the process completes its CPU burst (e.g., First Come First Served (FCFS), Shortest Job First (SJF)).

SCHEDULING CRITERIA

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU- scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU Utilization**
- **Throughput**
- **Turnaround Time**
- **Waiting Time**
- **Response Time**

CPU UTILIZATION

We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). CPU utilization can be obtained by using the **TOP** command on Linux, macOS, and UNIX systems.

THROUGHPUT

If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **Throughput.** For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

TURNAROUND TIME

From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the **turnaround time.** Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.

WAITING TIME

The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

RESPONSE TIME

In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to **maximize CPU utilization and throughput** and to **minimize turnaround time, waiting time, and response time**. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average.

SCHEDULING ALGORITHMS

A **Scheduling Algorithm** is the algorithm which tells us how much CPUtime we can allocate to the processes. These scheduling algorithms are either **Preemptive or Non-preemptive**

Preemptive Scheduling Algorithms are those which are based on the priority of the processes. By preference, when a high priority process enters, it preempts a low priority process in between and executes the high priority process first.

Non-preemptive Scheduling Algorithms are those who can't be preempted in between, i.e. we can not take control of CPU in between until the current process completes its execution.

To understand scheduling, first, we discuss some terms that we need in each scheduling process.

- **Arrival time** is the time at which the process arrives in the ready queue for execution, and it is given in our table when we need to calculate the average waiting time.
- **Completion time** is the time at which the process finishes its execution.
- **Turnaround time** is the difference between completion time and arrival time.

Turnaround Time = Completion Time- Arrival Time

- **Burst time** is the time required by the process for execution of the process by CPU.
- **Waiting time** (W.T) is the difference between turnaround time and burst time.

Waiting Time= Turnaround Time – Burst Time

FIRST COME FIRST SERVE SCHEDULING ALGORITHM

- The simplest CPU-scheduling algorithm is the **First-Come First-Serve (FCFS)** scheduling algorithm.
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.

IMPLEMENTATION

- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

EXAMPLES

1. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

PROCESS	BURST TIME
P1	24
P2	3
P3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart. **Gantt Chart** is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

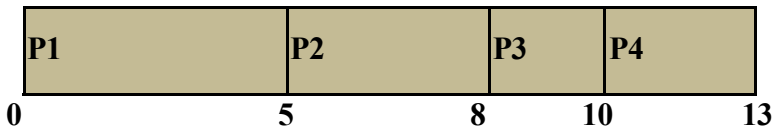
PROCESS	BURST TIME	ARRIVAL TIME	COMPLETION TIME	TURNAROUND TIME	WAITING TIME
P1	24	0	24	24	0
P2	3	0	27	27	24
P3	3	0	30	30	27

The waiting time is 0 milliseconds for process P1, 24 milliseconds for processP2, and 27 milliseconds for process P3. Thus, the **Average Waiting Time is (0 + 24 + 27)/3 = 17 milliseconds**

2. **Suppose there are four processes with process ID's P1, P2, P3, and P4 and they enter into the CPU as follows:**

PROCESS	BURST TIME	ARRIVAL TIME
P1	5	0
P2	3	2
P3	2	6
P4	3	7

GANTT CHART



PROCESS	BURST TIME	ARRIVAL TIME	COMPLETION TIME	TURNAROUND TIME	WAITING TIME
P1	5	0	5	5	0
P2	3	2	8	6	3
P3	2	6	10	4	2
P4	3	7	13	6	3

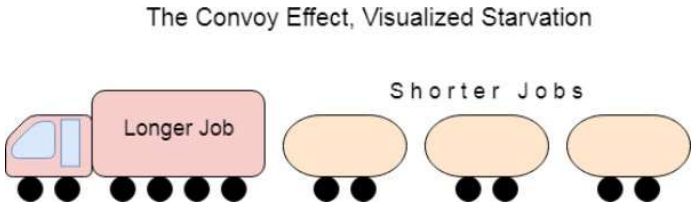
Total Turnaround Time = 5 + 6 + 4 + 6
= 21 milliseconds

Average Turnaround Time = Total Turn Around Time / Total No. of Processes
= 21 / 4
= 5.25 milliseconds

Total Waiting Time = 0 + 3 + 2 + 3
= 8 milliseconds

Average Waiting Time = Total Waiting Time / Total No. of Processes
= 8 / 4
= 2 milliseconds

Consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. There is a Convoy Effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.



The FCFS scheduling algorithm is nonpreemptive so, once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

SHORTEST JOB FIRST SCHEDULING ALGORITHM

Shortest Job First scheduling works on the process with the shortest **Burst Time** or duration first. **Shortest Job First (SJF)** is an algorithm in which the process having the **smallest execution time** is chosen for the next execution. It significantly **reduces the average waiting time** for other processes awaiting execution.

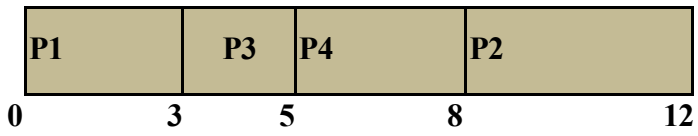
In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a **waiting state** or **terminated.** This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)

EXAMPLE

Let us take an example to understand it better. Suppose there are four processes with process ID's **P1**, **P2**, **P3**, and **P4** and they enter into the CPU as follows:

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	4	1
P3	2	2
P4	3	5

GANTT CHART



PROCESS	BURST TIME	ARRIVAL TIME	COMPLETION TIME	TURNAROUND TIME	WAITING TIME
P1	3	0	3	3	0
P2	4	1	12	11	7
P3	2	2	5	3	1
P4	3	5	8	3	0

Total Turn Around Time = 3 + 11 + 3 + 3
= 20 milliseconds

Average Turnaround Time = Total Turn Around Time / Total No. of Processes
= 20 / 4
= 5 milliseconds

Total Waiting Time = 0 + 7 + 1 + 0
= 8 milliseconds

Average Waiting Time = Total Waiting Time / Total No. of Processes
= 8 / 4
= 2 milliseconds

ROUND ROBIN CPU SCHEDULING

The **Round-Robin (RR) Scheduling Algorithm** is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum or time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

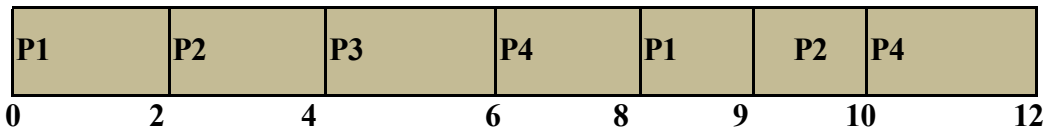
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue. The average waiting time under the RR policy is often long.

EXAMPLE:

Suppose we have 4 processes: P1, P2, P3, and P4, with the following properties and suppose that the time quantum decided is 2 units.

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	3	1
P3	2	3
P4	4	5

GANTT CHART



PROCESS	BURST TIME	ARRIVAL TIME	COMPLETION TIME	TURNAROUND TIME	WAITING TIME
P1	3	0	9	9	6
P2	3	1	10	9	6
P3	2	3	6	3	1
P4	4	5	12	7	3

Total Turn-Around Time = 9 + 9 + 3 + 7
= 28 milliseconds

Average Turn-Around Time= Total Turn-Around Time / Total No. of Processes
= 28 / 4
= 7 milliseconds

Total Waiting Time = 6 + 6 + 1 + 3
= 16 milliseconds

Average Waiting Time = Total Waiting Time / Total No. of Processes
= 16 / 4
= 4 milliseconds

PRIORITY CPU SCHEDULING

Priority scheduling in OS is the scheduling algorithm which schedules processes according to the priority assigned to each of the processes. Higher priority processes are executed before lower priority processes.

Priority of processes depends on some factors such as:

- Time limit
- Memory requirements of the process
- Ratio of average I/O to average CPU burst time

These priorities of processes are represented as simple integers in a fixed range such as 0 to 7. There are two types of priority scheduling algorithm in OS:

- **Non- Preemptive Scheduling**
- **Preemptive Scheduling**

Non- Preemptive Scheduling

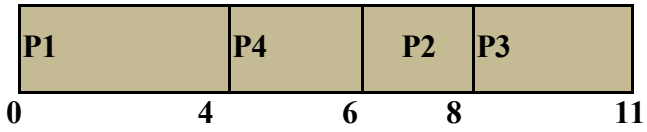
If during the execution of a process, another process with a higher priority arrives for execution, even then the currently executing process will not be disturbed. The newly arrived high priority process will be put in next for execution since it has higher priority than the processes that are in waiting state for execution. All the other processes will remain in the waiting queue. Once the execution of the current process is done, the high priority process will be given the CPU for execution.

For example, suppose we have 4 processes: **P1**, **P2**, **P3** and **P4** and they enter the CPU as follows:

Note: Here, lower the priority number, higher is the priority.

PROCESS	BURST TIME	ARRIVAL TIME	PRIORITY
P1	4	0	3
P2	2	1	2
P3	3	2	4
P4	2	4	1

GANTT CHART



PROCESS	BURST TIME	ARRIVAL TIME	PRIORITY	COMPLETION TIME	TURNAROUND TIME	WAITING TIME
P1	4	0	3	4	4	0
P2	2	1	2	8	7	5
P3	3	2	4	11	9	6
P4	2	4	1	6	2	0

Total Turn Around Time = 4 + 7 + 9 + 2
= 22 milliseconds

Average Turn Around Time= Total Turn Around Time / Total No. of Processes

= 22 / 4

= 5.5 milliseconds

Total Waiting Time = 0 + 5 + 6 + 0

= 11 milliseconds

Average Waiting Time = Total Waiting Time / Total No. of Processes

= 11 / 4

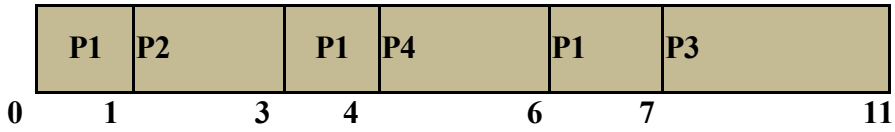
= 2.75 milliseconds

Preemptive Scheduling

Preemptive Sheduling will preempt the currently running process if a higher priority process enters the waiting state for execution and will execute the higher priority process first and then resume executing the previous process.

PROCESS	BURST TIME	ARRIVAL TIME	PRIORITY
P1	4	0	3
P2	2	1	2
P3	3	2	4
P4	2	4	1

GANTT CHART



PROCESS	BURST TIME	ARRIVAL TIME	PRIORITY	COMPLETION TIME	TURNAROUND TIME	WAITING TIME
P1	4	0	3	8	8	4
P2	2	1	2	3	2	0
P3	3	2	4	11	9	6
P4	2	4	1	6	2	0

Total Turn-Around Time = 8 + 2 + 9 + 2
= 21 milliseconds

Average Turn-Around Time= Total Turn-Around Time / Total No. of Processes
= 21 / 4
= 5.25 milliseconds

Total Waiting Time = 4 + 0 + 6 + 0
= 10 milliseconds

Average Waiting Time = Total Waiting Time / Total No. of Processes
= 10 / 4
= 2.5 milliseconds

Multiple processor scheduling

The goal of multiple processor scheduling, also known as multiprocessor scheduling, is to create a system's scheduling function that utilizes several processors. In multiprocessor scheduling, multiple CPUs split the workload (load sharing) to enable concurrent execution of multiple processes. In comparison to single-processor scheduling, multiprocessor scheduling is generally more complicated. There are many identical processors in the multiprocessor scheduling system, allowing us to perform any process at any moment.

The system's numerous CPUs communicate often and share a common bus, memory, and other peripherals. As a result, the system is said to be strongly connected. These systems are employed whenever large amounts of data need to be processed, and they are mostly used in satellite, weather forecasting, etc.

In some instances of multiple-processor scheduling, the functioning of the processors is homogeneous, or identical. Any process in the queue can run on any available processor.

Multiprocessor systems can be homogeneous (the same CPU) or heterogeneous (various types of CPUs). Special scheduling restrictions, such as devices coupled to a single CPU through a private bus, may apply.

The ideal scheduling method for a system with a single processor cannot be determined by any rule or policy. There is also no ideal scheduling strategy for a system with several CPUs.

Approaches to Multiple Processor Scheduling

There are two different architectures utilized in multiprocessor systems: −

Symmetric Multiprocessing

In an SMP system, each processor is comparable and has the same access to memory and I/O resources. The CPUs are not connected in a master-slave fashion, and they all use the same memory and I/O subsystems. This suggests that every memory location and I/O device are accessible to every processor without restriction. An operating system manages the task distribution among the processors in an SMP system, allowing every operation to be completed by any processor.

Asymmetric Multiprocessing

In the AMP asymmetric architecture, one processor, known as the master processor, has complete access to all of the system's resources, particularly memory and I/O devices. The master processor is in charge of allocating tasks to the other processors, also known as slave processors. Every slave processor is responsible for doing a certain set of tasks that the master processing has assigned to it. The master processor receives tasks from the operating system, which the master processor then distributes to the subordinate processors.

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−

- **High-Performance Computing** − Multiple processor scheduling is crucial in high-performance computing (HPC) environments where large-scale scientific simulations, data analysis, or complex computations are performed. Efficient scheduling of processes across multiple processors enables parallel execution, leading to faster computation times and increased overall system performance.
- **Server Virtualization** − In virtualized environments, where multiple virtual machines (VMs) run on a single physical server with multiple processors, effective scheduling ensures fair allocation of resources to VMs. It enables optimal utilization of processing power while maintaining performance isolation and ensuring that each VM receives its allocated share of CPU time.
- **Real-Time Systems** − Real-time systems, such as those used in aerospace, defense, and industrial automation, have strict timing requirements. Multiple processor scheduling algorithms like Earliest Deadline First (EDF) ensure that critical tasks with imminent deadlines are executed promptly, guaranteeing timely response and meeting stringent timing constraints.
- **Multimedia Processing** − Multimedia applications, such as video rendering or audio processing, often require significant computational power. Scheduling processes across multiple processors allows for parallel execution of multimedia tasks, enabling faster processing and smooth real-time performance.

- **Distributed Computing** − In distributed computing systems, tasks are distributed across multiple processors or nodes for collaborative processing. Efficient scheduling algorithms ensure load balancing, fault tolerance, and effective resource utilization across the distributed infrastructure, improving overall system efficiency and scalability.
- **Cloud Computing** − Cloud service providers employ multiple processors to serve numerous client requests simultaneously. Scheduling algorithms optimize the allocation of virtual machines and containers across the available processors, ensuring fairness, scalability, and efficient resource utilization in cloud computing environments.
- **Big Data Processing** − Big data analytics involves processing and analyzing massive volumes of data. Multiple processor scheduling enables parallel execution of data processing tasks, such as data ingestion, transformation, and analysis, significantly reducing the time required for data processing and enabling real-time or near-real-time insights.
- **Scientific Simulations and Modeling** − Numerical simulations and scientific modeling often require extensive computational resources. Multiple processor scheduling allows for the parallel execution of simulation tasks, accelerating the time it takes to obtain results and enabling researchers to explore complex phenomena and perform more accurate simulations.
- **Gaming** − In modern gaming systems, multiple processors are utilized to handle complex graphics rendering, physics simulations, and AI computations. Effective scheduling ensures smooth gameplay, minimizes lag, and maximizes the utilization of available processing power to deliver an immersive gaming experience.
- **Embedded Systems** − Embedded systems with multiple processors, such as automotive systems, IoT devices, or robotics, require efficient scheduling to ensure a timely response, real-time control, and coordination of various tasks running on different processors. Scheduling algorithms prioritize critical tasks and manage resource allocation to meet system requirements.

Conclusion

Multiple processor scheduling is a key concept in operating systems and is essential for managing how actions are executed in systems with multiple processors. Due to the growing need for computing power, multiprocessor systems are becoming more widespread, and efficient scheduling techniques are needed to manage the allocation of system resources.

END OF MODULE-2