MODULE-2

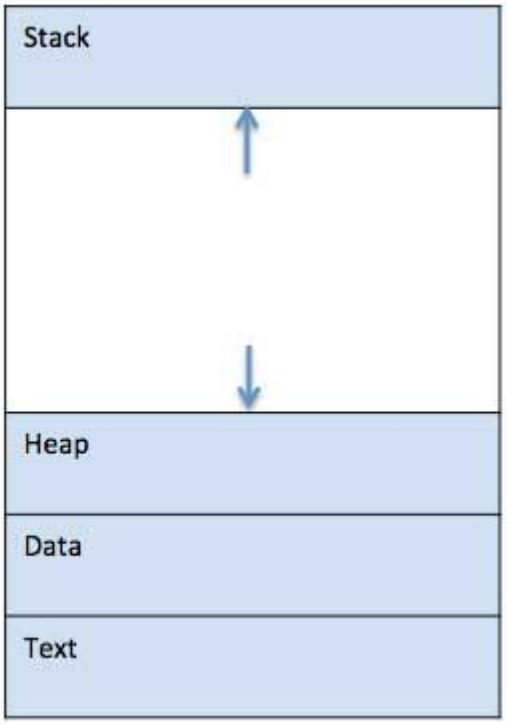
Process

A process is basically a program in execution. 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.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory



Different states of a Process-

A process is a program in execution and it is more than a program code called as text section and this concept works under all the operating system because all the task perform by the operating system needs a process to perform the task

The process executes when it changes the state. The state of a process is defined by the current activity of the process.

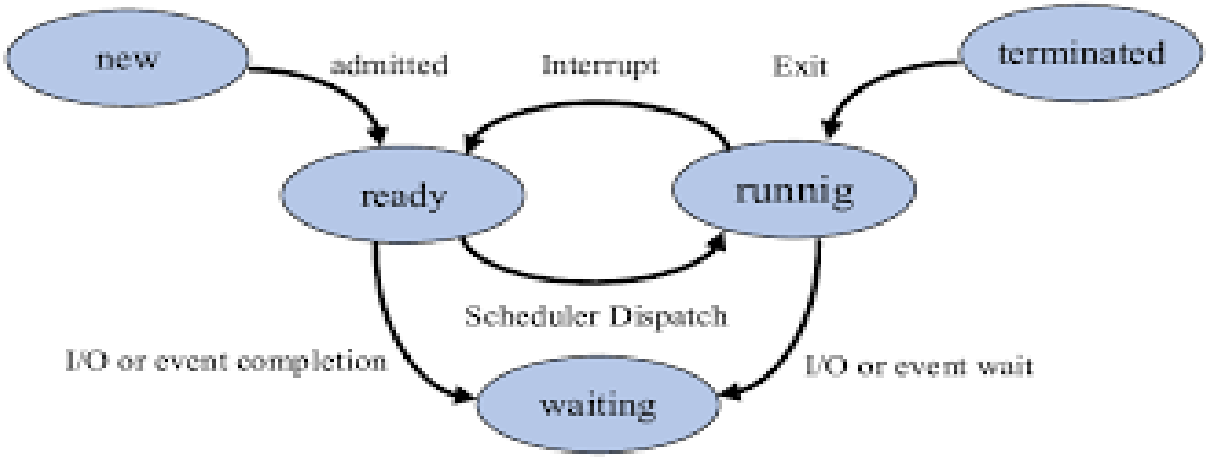
Each process may be in any one of the following states −

- **New** − The process is being created.

- **Running** − In this state the instructions are being executed.
- **Waiting** − The process is in waiting state until an event occurs like I/O operation completion or receiving a signal.
- **Ready** − The process is waiting to be assigned to a processor.
- **Terminated** − the process has finished execution.

It is important to know that only one process can be running on any processor at any instant. Many processes may be ready and waiting.

Now let us see the state diagram of these process states −



Explanation

Step 1 − Whenever a new process is created, it is admitted into ready state.

Step 2 − If no other process is present at running state, it is dispatched to running based on scheduler dispatcher.

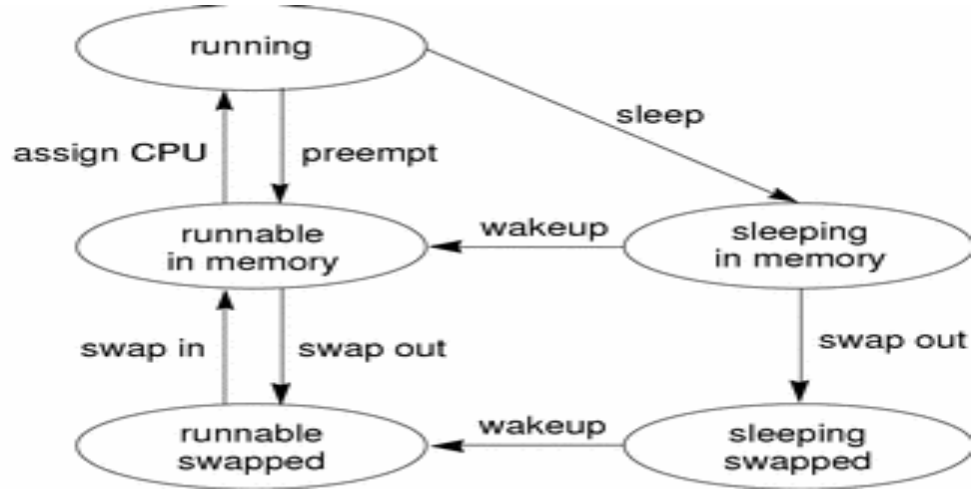
Step 3 − If any higher priority process is ready, the uncompleted process will be sent to the waiting state from the running state.

Step 4 − Whenever I/O or event is completed the process will send back to ready state based on the interrupt signal given by the running state.

Step 5 − Whenever the execution of a process is completed in running state, it will exit to terminate state, which is the completion of process.

Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. A simplified overview of UNIX process states and the transitions between states is shown in the following figure.



An active process is normally in one of the five states in the diagram. The arrows show how the process changes states.

- A process is running if the process is assigned to a CPU. A process is removed from the running state by the scheduler if a process with a higher priority becomes runnable. A process is also pre-empted if a process of equal priority is runnable when the original process consumes its entire time slice.
- A process is runnable in memory if the process is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if the process is in primary memory but is waiting for a specific event before continuing execution. For example, a process sleeps while waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, a wakeup call is sent to the process. If the reason for its sleep is gone, the process becomes runnable.
- When a process' address space has been written to secondary memory, and that process is not waiting for a specific event, the process is runnable and swapped.
- If a process is waiting for a specific event and has had its whole address space written to secondary memory, the process is sleeping and swapped.

  If a machine does not have enough primary memory to hold all its active processes, that machine must page or swap some address space to secondary memory.

- When the system is short of primary memory, the system writes individual pages of some processes to secondary memory but leaves those processes runnable. When a running process, accesses those pages, the process sleeps while the pages are read back into primary memory.

- When the system encounters a more serious shortage of primary memory, the system writes all the pages of some processes to secondary memory. The system marks the pages that have been written to secondary memory as swapped. Such processes can only be scheduled when the system scheduler daemon selects these processes to be read back into memory.

Both paging and swapping cause delay when a process is ready to run again. For processes that have strict timing requirements, this delay can be unacceptable.

To avoid swapping delays, real-time processes are never swapped, though parts of such processes can be paged. A program can prevent paging and swapping by locking its text and data into primary memory. For more information, see the memcntl(2) man page. How much memory can be locked is limited by how much memory is configured. Also, locking too much can cause intolerable delays to processes that do not have their text and data locked into memory.

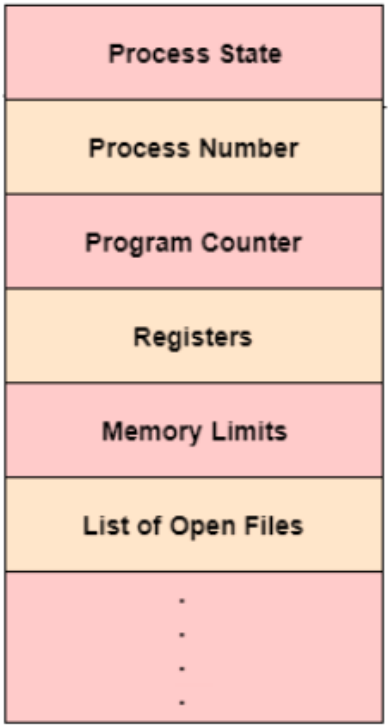
Process Control Block (PCB)

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

Structure of the Process Control Block

The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given diagram −



Process Control Block (PCB)

The following are the data items −

Process State

This specifies the process state i.e. new, ready, running, waiting or terminated.

Process Number

This shows the number of the particular process.

Program Counter

This contains the address of the next instruction that needs to be executed in the process.

Registers

This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

List of Open Files

These are the different files that are associated with the process

CPU Scheduling Information

The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

Memory Management Information

The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

I/O Status Information

This information includes the list of I/O devices used by the process, the list of files etc.

Accounting information

The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

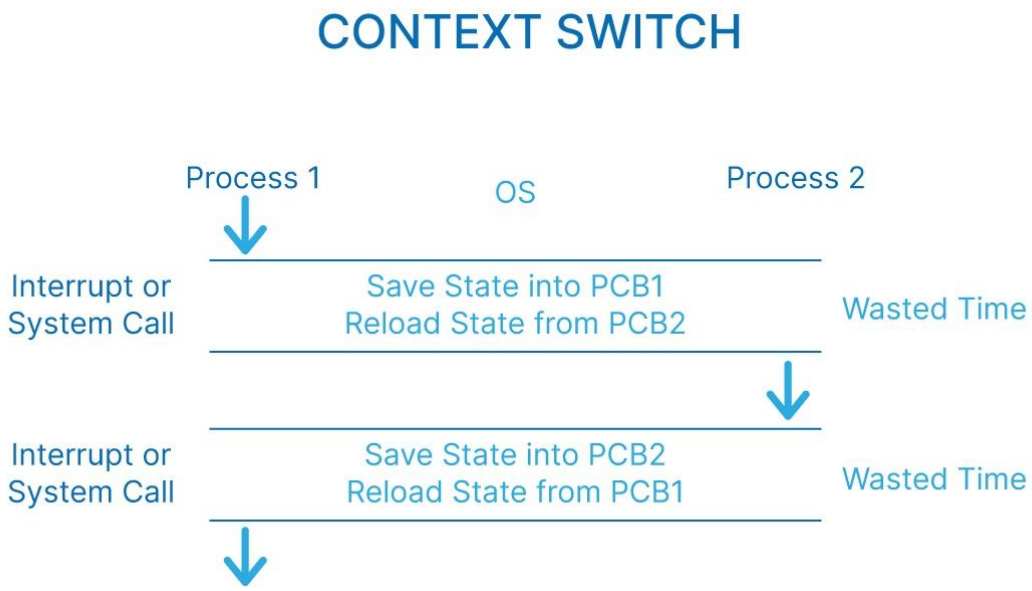
Location of the Process Control Block

The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

Context switching:-

Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

A diagram that demonstrates context switching is as follows −



In the above diagram, initially Process 1 is running. Process 1 is switched out and Process 2 is switched in because of an interrupt or a system call. Context switching involves saving the state of Process 1 into PCB1 and loading the state of process 2 from PCB2. After some time again a context switch occurs and Process 2 is switched out and Process 1 is switched in again. This involves saving the state of Process 2 into PCB2 and loading the state of process 1 from PCB1.

Context switching triggers

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

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

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

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

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

Context Switching Steps

The steps involved in context switching are as follows −

- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the process control block and registers.

Context Switching Cost

Context Switching leads to an overhead cost because of TLB flushes, sharing the cache between multiple tasks, running the task scheduler etc. Context switching between two threads of the same process is faster than between two different processes as threads have the same virtual memory maps. Because of this TLB flushing is not required.

The need for Context switching

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

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

1. The switching of one process to another process is not directly in the system. A context switching helps the operating system that switches between the multiple processes to use the CPU's resource to accomplish its tasks and store its context. We can resume the service of the process at the same point later. If we do not store the currently running process's data or context, the stored data may be lost while switching between processes.
2. If a high priority process falls into the ready queue, the currently running process will be shut down or stopped by a high priority process to complete its tasks in the system.
3. If any running process requires I/O resources in the system, the current process will be switched by another process to use the CPUs. And when the I/O requirement is met, the old process goes into a ready state to wait for its execution in the CPU. Context switching stores the state of the process to resume its tasks in an operating system. Otherwise, the process needs to restart its execution from the initials level.
4. If any interrupts occur while running a process in the operating system, the process status is saved as registers using context switching. After resolving the interrupts, the process switches from a wait state to a ready state to resume its execution at the same point later, where the operating system interrupted occurs.
5. A context switching allows a single CPU to handle multiple process requests simultaneously without the need for any additional processors.

Example of Context Switching

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

Thread

A process is divided into several light-weight processes, each light-weight process is said to be a **thread**. The thread has a program counter that keeps the tracks of which instruction to execute next, if process registers, which hold its current working variables. It has a stack which contains the executing thread history. The life cycle of thread is **Born state**, **Ready state**, **Running state**, **Blocked State**, **Sleep**, **Dead**.

The number of threads can share **an address space**, **open files** and other resources. Same as the number of the process can share physical memory, disks, printers and the other resources. Because threads have some of the properties of the process. Threads operate in many respects, in the same manner as the process. Threads share the **CPU** and only one thread at a time is active.

Example of Thread:

Word Processor:

A programmer wish to type the text in word processor. Then the programmer opens a file in a word processor and typing the text (It is a **thread**), the text is automatically formatting (It is another **thread**). The text automatically specifies the spelling mistakes (It is another **thread**), and the file is automatically saved in the disk (It is another **thread**).

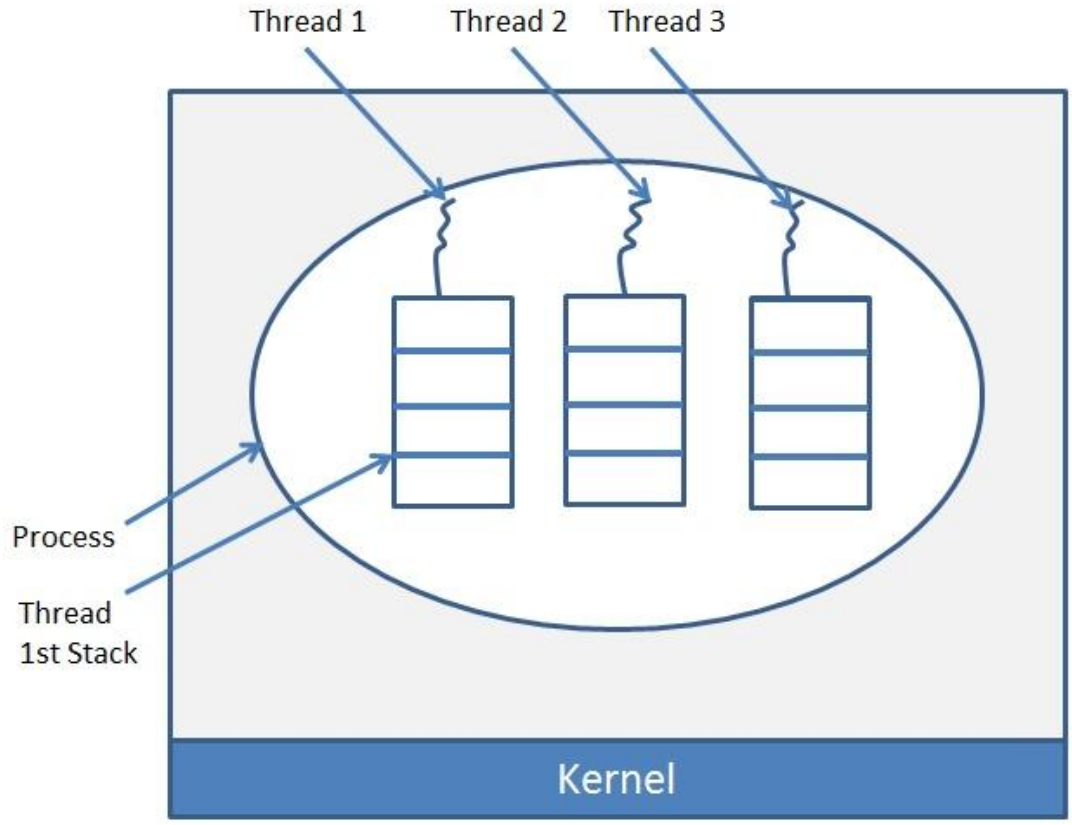


Fig : Threads

Life Cycle of Thread:

1. Born State: A thread that has just created.
2. Ready State: The thread is waiting for the processor (CPU).
3. Running: The System assigns the processor to the thread means that the thread is being executed.
4. Blocked State: The thread is waiting for an event to occur or waiting for an I/O device.

5. Sleep: A sleeping thread becomes ready after the designated sleep time expires.
6. Dead: The execution of the thread is finished.

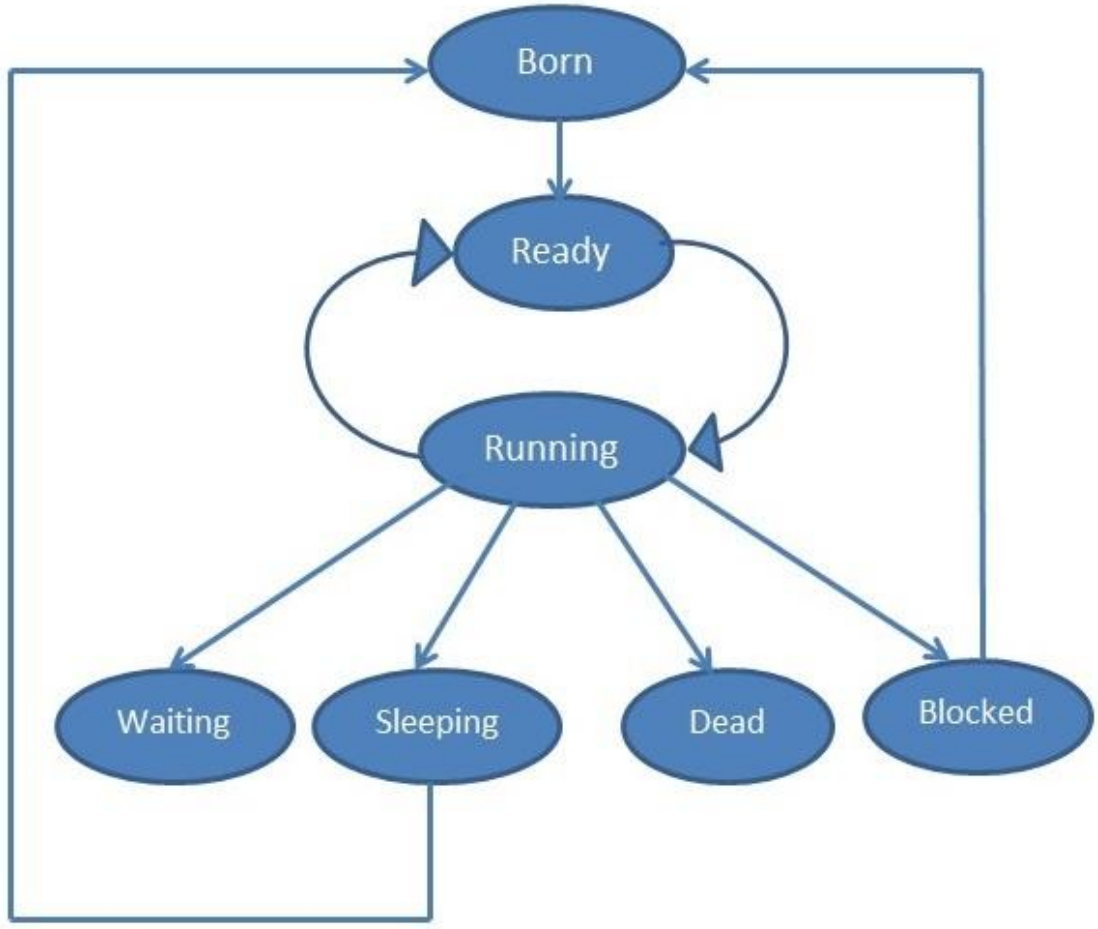


Fig : Life Cycle of a thread

Process vs Thread

Process simply means any program in execution while the thread is a segment of a process. The main differences between process and thread are mentioned below:

Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.

Process	Thread
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can read, write or change another thread's data)
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.

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.

Different Types of Process Schedulers

Process Scheduling handles the selection of a process for the processor on the basis of a scheduling algorithm and also the removal of a process from the processor. It is an important part of multiprogramming operating system.

There are many scheduling queues that are used in process scheduling. When the processes enter the system, they are put into the job queue. The processes that are ready to execute in the main memory are kept in the ready queue. The processes that are waiting for the I/O device are kept in the I/O device queue.

The different schedulers that are used for process scheduling are −

Long Term Scheduler

The job scheduler or long-term scheduler selects processes from the storage pool in the secondary memory and loads them into the ready queue in the main memory for execution.

The long-term scheduler controls the degree of multiprogramming. It must select a careful mixture of I/O bound and CPU bound processes to yield optimum system throughput. If it selects too many CPU bound processes then the I/O devices are idle and if it selects too many I/O bound processes then the processor has nothing to do.

The job of the long-term scheduler is very important and directly affects the system for a long time.

Short Term Scheduler

The short-term scheduler selects one of the processes from the ready queue and schedules them for execution. A scheduling algorithm is used to decide which process will be scheduled for execution next.

The short-term scheduler executes much more frequently than the long-term scheduler as a process may execute only for a few milliseconds.

The choices of the short term scheduler are very important. If it selects a process with a long burst time, then all the processes after that will have to wait for a long time in the ready queue. This is known as starvation and it may happen if a wrong decision is made by the short-term scheduler.

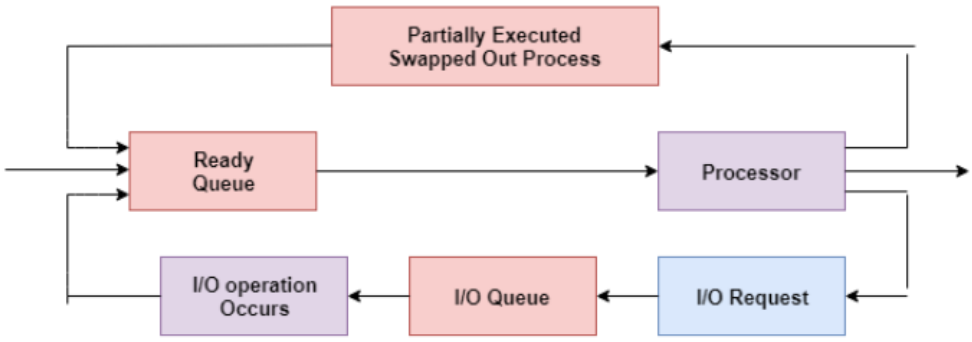
A diagram that demonstrates long-term and short-term schedulers is given as follows −

Medium Term Scheduler

The medium-term scheduler swaps out a process from main memory. It can again swap in the process later from the point it stopped executing. This can also be called as suspending and resuming the process.

This is helpful in reducing the degree of multiprogramming. Swapping is also useful to improve the mix of I/O bound and CPU bound processes in the memory.

A diagram that demonstrates medium-term scheduling is given as follows −

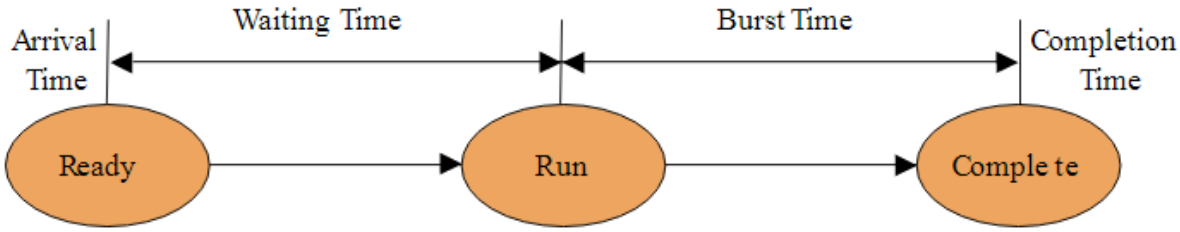


Representation of Medium-Term Scheduling using a Queuing Diagram

Burst time, Arrival time, Exit time, Response time, Waiting time, Turnaround time, and Throughput

Burst time

Every process in a computer system requires some amount of time for its execution. This time is both the CPU time and the I/O time. The CPU time is the time taken by CPU to execute the process. While the I/O time is the time taken by the process to perform some I/O operation. In general, we ignore the I/O time and we consider only the CPU time for a process. So, **Burst time is the total time taken by the process for its execution on the CPU.**



Arrival time

Arrival time is the time when a process enters into the ready state and is ready for its execution.

Here in the above example, the arrival time of all the 3 processes are 0 ms, 1 ms, and 2 ms respectively.

Exit time

Exit time is the time when a process completes its execution and exit from the system.

Response time

Response time is the time spent when the process is in the ready state and gets the CPU for the first time. For example, here we are using the First Come First Serve CPU scheduling algorithm for the below 3 processes:

Process	Arrival time	Burst time
P1	0 ms	8 ms
P2	1 ms	7 ms
P3	2 ms	10 ms

AfterAcademy

Here, the response time of all the 3 processes are:

- **P1:** 0 ms
- **P2:** 7 ms because the process P2 have to wait for 8 ms during the execution of P1 and then after it will get the CPU for the first time. Also, the arrival time of P2 is 1 ms. So, the response time will be 8-1 = 7 ms.
- **P3:** 13 ms because the process P3 have to wait for the execution of P1 and P2 i.e. after 8+7 = 15 ms, the CPU will be allocated to the process P3 for the first time. Also, the arrival of P3 is 2 ms. So, the response time for P3 will be 15-2 = 13 ms.

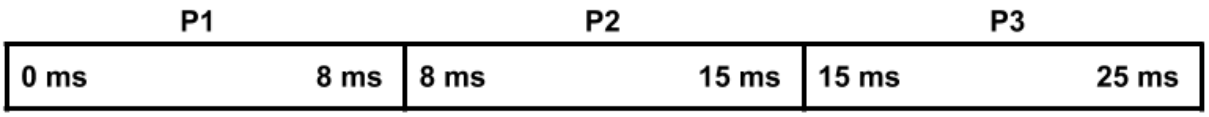
Response time = Time at which the process gets the CPU for the first time - Arrival time

Waiting time

Waiting time is the total time spent by the process in the ready state waiting for CPU. For example, consider the arrival time of all the below 3 processes to be 0 ms, 0 ms, and 2 ms and we are using the First Come First Serve scheduling algorithm.

Process	Arrival time	Burst time
P1	0 ms	8 ms
P2	0 ms	7 ms
P3	2 ms	10 ms

Gantt Chart



Then the waiting time for all the 3 processes will be:

- **P1:** 0 ms
- **P2:** 8 ms because P2 have to wait for the complete execution of P1 and arrival time of P2 is 0 ms.
- **P3:** 13 ms becuase P3 will be executed after P1 and P2 i.e. after 8+7 = 15 ms and the arrival time of P3 is 2 ms. So, the waiting time of P3 will be: 15-2 = 13 ms.

Waiting time = Turnaround time - Burst time

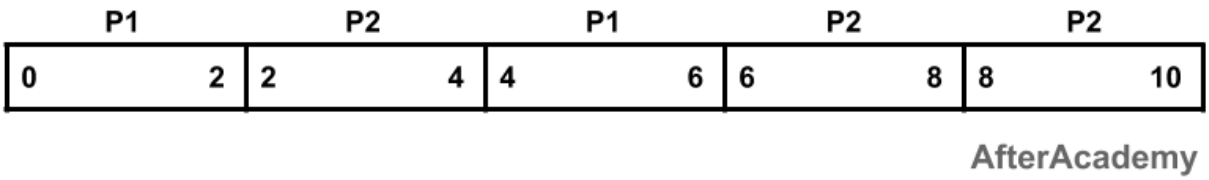
In the above example, the processes have to wait only once. But in many other scheduling algorithms, the CPU may be allocated to the process for some time and then the process will be moved to the waiting state and again after some time, the process will get the CPU and so on.

There is a difference between waiting time and response time. Response time is the time spent between the ready state and getting the CPU for the first time. But the waiting time is the total time taken by the process in the ready state. Let's take an example of a round-robin scheduling algorithm. The time quantum is 2 ms.

Process	Arrival time	Burst time
P1	0 ms	4 ms
P2	0 ms	6 ms

Time Quantum = 2ms

Gantt Chart



In the above example, the response time of the process P2 is 2 ms because after 2 ms, the CPU is allocated to P2 and the waiting time of the process P2 is 4 ms i.e turnaround time - burst time (10 - 6 = 4 ms).

Turnaround time

Turnaround time is the total amount of time spent by the process from coming in the ready state for the first time to its completion.

Turnaround time = Burst time + Waiting time

or

Turnaround time = Exit time - Arrival time

For example, if we take the First Come First Serve scheduling algorithm, and the order of arrival of processes is P1, P2, P3 and each process is taking 2, 5, 10 seconds. Then the turnaround time of P1 is 2 seconds because when it comes at 0th second, then the CPU is allocated to it and so the waiting time of P1 is 0 sec and the turnaround time will be the Burst time only i.e. 2 seconds. The turnaround time of P2 is 7 seconds because the process P2 have to wait for 2 seconds for the execution of P1 and hence the waiting time of P2 will be 2 seconds. After 2 seconds, the CPU will be given to P2 and P2 will execute its task. So, the turnaround time will be 2+5 = 7 seconds. Similarly, the turnaround time for P3 will be 17 seconds because the waiting time of P3 is 2+5 = 7 seconds and the burst time of P3 is 10 seconds. So, turnaround time of P3 is 7+10 = 17 seconds.

Different CPU scheduling algorithms produce different turnaround time for the same set of processes. This is because the waiting time of processes differ when we change the CPU scheduling algorithm.

Throughput

Throughput is a way to find the efficiency of a CPU. It can be defined as the number of processes executed by the CPU in a given amount of time. For example, let's say, the process P1 takes 3 seconds for execution, P2 takes 5 seconds, and P3 takes 10 seconds. So, throughput, in this case, the throughput will be (3+5+10)/3 = 18/3 = 6 seconds.

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time – Burst Time

Preemptive Scheduling is a CPU scheduling technique that works by dividing time slots of CPU to a given process. The time slot given might be able to complete the whole process or might not be able to it. When the burst time of the process is greater than CPU cycle, it is placed back into the ready queue and will execute in the next chance. This scheduling is used when the process switch to ready state.

Algorithms that are backed by preemptive Scheduling are round-robin (RR), priority, SRTF (shortest remaining time first).

Non-preemptive Scheduling is a CPU scheduling technique the process takes the resource (CPU time) and holds it till the process gets terminated or is pushed to the waiting state. No process is interrupted until it is completed, and after that processor switches to another process.

Algorithms that are based on non-preemptive Scheduling are non-preemptive priority, and shortest Job first.

Preemptive Vs Non-Preemptive Scheduling

Preemptive Scheduling	Non-Preemptive Scheduling
Resources are allocated according to the cycles for a limited time.	Resources are used and then held by the process until it gets terminated.
The process can be interrupted, even before the completion.	The process is not interrupted until its life cycle is complete.
Starvation may be caused, due to the insertion of priority process in the queue.	Starvation can occur when a process with large burst time occupies the system.

Preemptive Scheduling	Non-Preemptive Scheduling
Maintaining queue and remaining time needs storage overhead.	No such overheads are required.

First come first serve:- Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.
First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.
In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
Here we are considering that arrival time for all processes is 0.
How to compute below times in Round Robin using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time – Burst Time

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

Important Points:

1. Non-preemptive
2. Average Waiting Time is not optimal
3. Cannot utilize resources in parallel : Results in Convoy effect (Consider a situation when many IO bound processes are there and one CPU bound process. The IO bound processes have to wait for CPU bound process when CPU bound process acquires CPU. The IO bound process could have better taken CPU for some time, then used IO devices).

Shortest Job First Scheduling:-

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

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.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- 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.

How to compute below times in SJF using a program?

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

Program for Non-Preemptive Shortest Job First CPU Scheduling

Non-Preemptive Shortest Job First algorithm can be implemented using Segment Trees data structure. For detailed implementation of Non-

Preemptive **Shortest Job First** scheduling algorithm, please refer: Program for Non-Preemptive Shortest Job First CPU Scheduling.

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.

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.

Round Robin Scheduling

Round Robin(RR) scheduling algorithm is mainly designed for time-sharing systems. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes.

- A fixed time is allotted to each process, called a **quantum**, for execution.
- Once a process is executed for the given time period that process is preempted and another process executes for the given time period.
- Context switching is used to save states of preempted processes.
- This algorithm is simple and easy to implement and the most important is thing is this algorithm is starvation-free as all processes get a fair share of CPU.
- It is important to note here that the length of time quantum is generally from 10 to 100 milliseconds in length.

Some important characteristics of the Round Robin(RR) Algorithm are as follows:

1. Round Robin Scheduling algorithm resides under the category of Preemptive Algorithms.
2. This algorithm is one of the oldest, easiest, and fairest algorithm.
3. This Algorithm is a real-time algorithm because it responds to the event within a specific time limit.
4. In this algorithm, the time slice should be the minimum that is assigned to a specific task that needs to be processed. Though it may vary for different operating systems.
5. This is a hybrid model and is clock-driven in nature.
6. This is a widely used scheduling method in the traditional operating system.

Important terms

1. **Completion Time** It is the time at which any process completes its execution.
2. **Turn Around Time** This mainly indicates the time Difference between completion time and arrival time. The Formula to calculate the same is: **Turn Around Time = Completion Time – Arrival Time**
3. **Waiting Time(W.T):** It Indicates the time Difference between turn around time and burst time. And is calculated as **Waiting Time = Turn Around Time – Burst Time**

Let us now cover an example for the same:

In the above diagram, arrival time is not mentioned so it is taken as 0 for all processes.

Note: If arrival time is not given for any problem statement then it is taken as 0 for all processes; if it is given then the problem can be solved accordingly.

Explanation

The value of time quantum in the above example is 5.Let us now calculate the Turn around time and waiting time for the above example :

Processes	**Burst Time**	**Turn Around Time** **Turn Around Time = Completion Time – Arrival Time**	**Waiting Time** **Waiting Time = Turn Around Time – Burst Time**
P1	21	32-0=32	32-21=11
P2	3	8-0=8	8-3=5
P3	6	21-0=21	21-6=15
P4	2	15-0=15	15-2=13

Average waiting time is calculated by adding the waiting time of all processes and then dividing them by no.of processes.

average waiting time = waiting time of all processes/ no.of processes

average waiting time=11+5+15+13/4 = 44/4= **11ms**

Priority Scheduling

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.

The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

Types of Priority Scheduling

Priority scheduling divided into two main types:

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.

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.

Characteristics of Priority Scheduling

- A CPU algorithm that schedules processes based on priority.
- It used in Operating systems for performing batch processes.
- If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis.
- In priority scheduling, a number is assigned to each process that indicates its priority level.
- Lower the number, higher is the priority.
- In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

Example of Priority Scheduling

Consider following five processes P1 to P5. Each process has its unique priority, burst time, and arrival time.

Process	Priority	Burst time	Arrival time
P1	1	4	0
P2	2	3	0
P3	1	7	6
P4	3	4	11
P5	2	2	12

Multiple Processors Scheduling in Operating System

Multiple processor scheduling or multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is complex as compared to single processor scheduling. In the multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

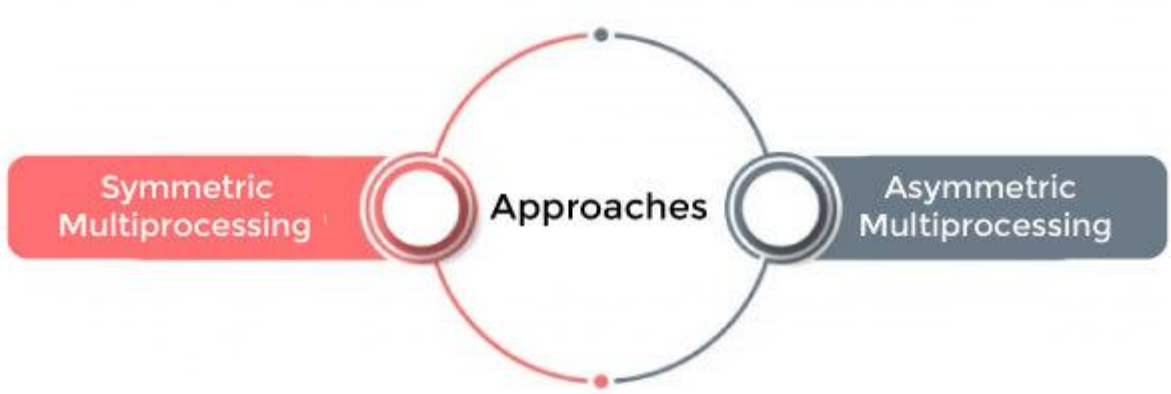
The multiple CPUs in the system are in close communication, which shares a common bus, memory, and other peripheral devices. So we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc.

There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. We can use any processor available to run any process in the queue.

Multiprocessor systems may be ***heterogeneous*** (different kinds of CPUs) or ***homogenous*** (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one

Approaches to Multiple Processor Scheduling

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.

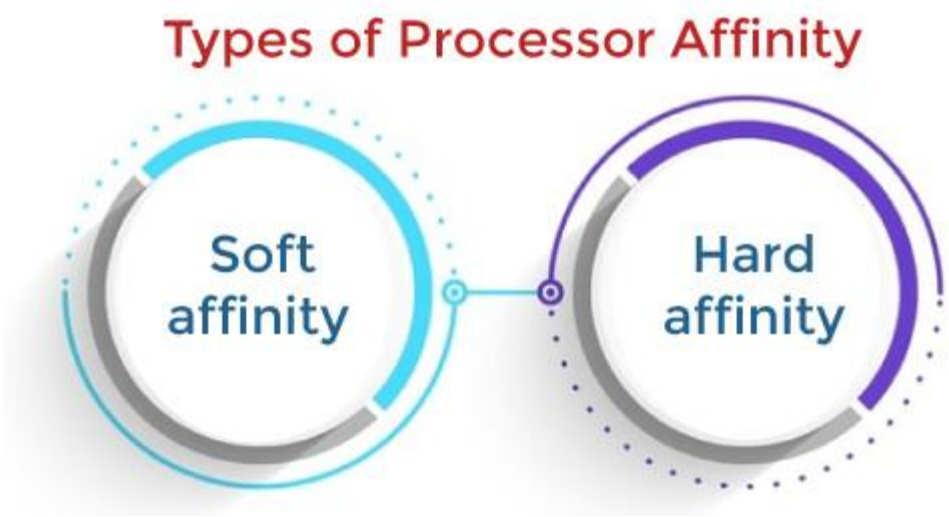


1. **Symmetric Multiprocessing:** It is used where each processor is ***self-scheduling***. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.
2. **Asymmetric Multiprocessing:** It is used when all the scheduling decisions and I/O processing are handled by a single processor called the ***Master Server***. The other processors execute only the ***user code***. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing.

Processor Affinity

Processor Affinity means a process has an ***affinity*** for the processor on which it is currently running. When a process runs on a specific processor, there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory access by the process is often satisfied in the cache memory.

Now, suppose the process migrates to another processor. In that case, the contents of the cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most SMP(symmetric multiprocessing) systems try to avoid migrating processes from one processor to another and keep a process running on the same processor. This is known as processor affinity. There are two types of processor affinity, such as:

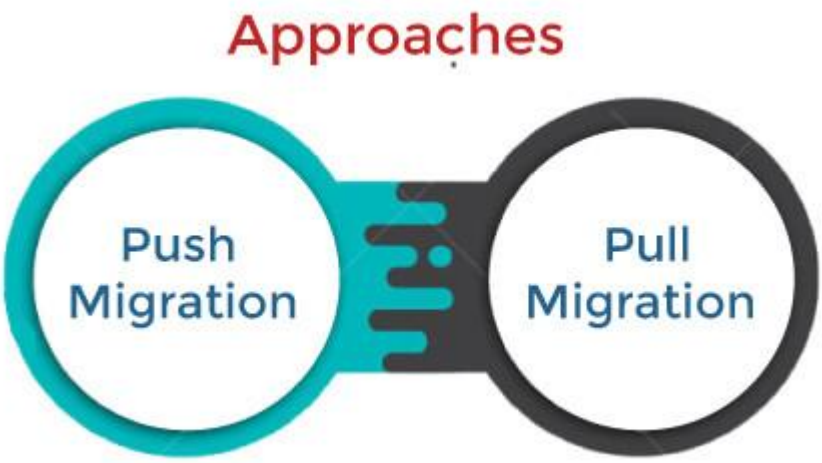


1. **Soft Affinity:** When an operating system has a policy of keeping a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity:** Hard Affinity allows a process to specify a subset of processors on which it may run. Some Linux systems implement soft affinity and provide system calls like ***sched_setaffinity()*** that also support hard affinity.

Load Balancing

Load Balancing is the phenomenon that keeps the workload evenly distributed across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of a process that is eligible to execute.

Load balancing is unnecessary because it immediately extracts a runnable process from the common run queue once a processor becomes idle. On SMP (symmetric multiprocessing), it is important to keep the workload balanced among all processors to utilize the benefits of having more than one processor fully. One or more processors will sit idle while other processors have high workloads along with lists of processors awaiting the CPU. There are two general approaches to load balancing:

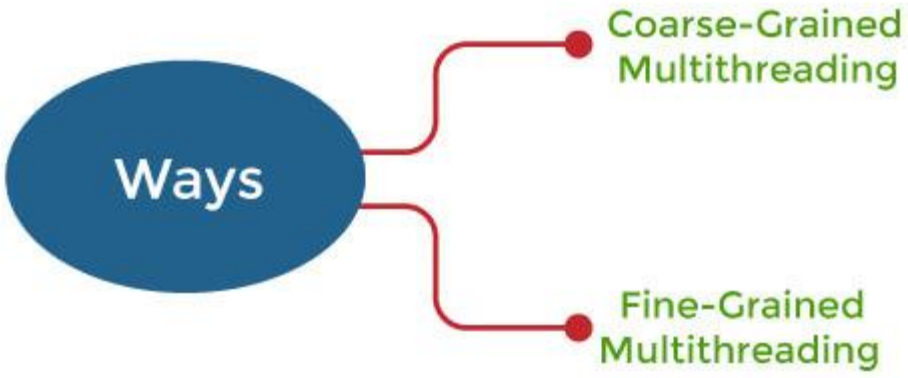


1. **Push Migration:** In push migration, a task routinely checks the load on each processor. If it finds an imbalance, it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration:**Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multi-core Processors

In multi-core processors, multiple processor cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. ***SMP systems*** that use multi-core processors are faster and consume less power than systems in which each processor has its own physical chip.

In such cases, the processor can spend upto 50% of its time waiting for data to become available from memory. To solve this problem, recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, the core can switch to another thread. There are two ways to multithread a processor:



1. **Coarse-Grained Multithreading:** A thread executes on a processor until a long latency event such as a memory stall occurs in coarse-grained multithreading. Because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.
2. **Fine-Grained Multithreading:** This multithreading switches between threads at a much finer level, mainly at the boundary of an instruction cycle. The architectural design of fine-grained systems includes logic for thread switching, and as a result, the cost of switching between threads is small.

Symmetric Multiprocessor

Symmetric Multiprocessors (SMP) is the third model. There is one copy of the OS in memory in this model, but any central processing unit can run it. Now, when a system call is made, the central processing unit on which the system call was made traps the kernel and processed that system call. This model balances processes and memory dynamically. This approach uses Symmetric Multiprocessing, where each processor is self-scheduling.

The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute. In this system, this is possible that all the process may be in a common ready queue or each processor may have its private queue for the ready process. There are mainly three sources of contention that can be found in a multiprocessor operating system.

- **Locking system:** As we know that the resources are shared in the multiprocessor system, there is a need to protect these resources for safe access among the multiple

processors. The main purpose of the locking scheme is to serialize access of the resources by the multiple processors.

- **Shared data:** When the multiple processors access the same data at the same time, then there may be a chance of inconsistency of data, so to protect this, we have to use some protocols or locking schemes.
- **Cache coherence:** It is the shared resource data that is stored in multiple local caches. Suppose two clients have a cached copy of memory and one client change the memory block. The other client could be left with an invalid cache without notification of the change, so this conflict can be resolved by maintaining a coherent view of the data.

Master-Slave Multiprocessor

In this multiprocessor model, there is a single data structure that keeps track of the ready processes. In this model, one central processing unit works as a master and another as a slave. All the processors are handled by a single processor, which is called the master server.

Multiple Processor Scheduling

When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times.

Load sharing revolves around balancing the load between multiple processors.

Multi-processor systems may be heterogeneous, (different kinds of CPUs), or homogenous, (all the same kind of CPU).

1. Approaches to Multiple-Processor Scheduling

Asymmetric multiprocessing: One processor is the master, controlling all activities and running all kernel code, while the other runs only user code.

Symmetric multiprocessing (SMP): Each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor.

2. Processor Affinity

Soft affinity occurs when the system attempts to keep processes on the same processor but makes no guarantees.
Linux and some other OSes support hard affinity, in which a process specifies that it is not to be moved between processors.

3. Load Balancing

Obviously an important goal in a multiprocessor system is to balance the load between processors, so that one processor won't be sitting idle while another is overloaded.

4. Multicore Processors

Traditional SMP required multiple CPU chips to run multiple kernel threads concurrently.
Recent trends are to put multiple CPUs (cores) onto a single chip, which appear to the system as multiple processors.

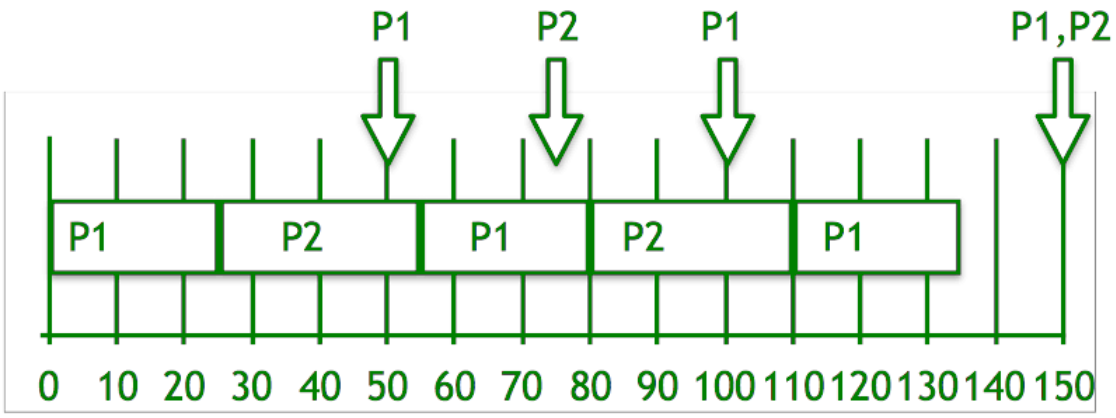
Earliest Deadline First (EDF) CPU scheduling algorithm

Earliest Deadline First (EDF) is an optimal dynamic priority scheduling algorithm used in real-time systems.
It can be used for both static and dynamic real-time scheduling.
EDF uses priorities to the jobs for scheduling. It assigns priorities to the task according to the absolute deadline. The task whose deadline is closest gets the highest priority. The priorities are assigned and changed in a dynamic fashion. EDF is very efficient as compared to other scheduling algorithms in real-time systems. It can make the CPU utilization to about 100% while still guaranteeing the deadlines of all the tasks.

EDF includes the kernel overload. In EDF, if the CPU usage is less than 100%, then it means that all the tasks have met the deadline. EDF finds an optimal feasible schedule. The feasible schedule is one in which all the tasks in the system are executed within the deadline. If EDF is not able to find a feasible schedule for all the tasks in the real-time system, then it means that no other task scheduling algorithms in real-time systems can give a feasible schedule. All the tasks which are ready for execution should announce their deadline to EDF when the task becomes runnable.

EDF scheduling algorithm does not need the tasks or processes to be periodic and also the tasks or processes require a fixed CPU burst time. In EDF, any executing task can be preempted if any other periodic instance with an earlier deadline is ready for execution and becomes active. Preemption is allowed in the Earliest Deadline First scheduling algorithm.

Example:
Consider two processes P1 and P2.
Let the period of P1 be $p_1 = 50$
Let the processing time of P1 be $t_1 = 25$
Let the period of P2 be $period_2 = 75$
Let the processing time of P2 be $t_2 = 30$



Steps for solution:

1. Deadline pf P1 is earlier, so priority of P1>P2.
2. Initially P1 runs and completes its execution of 25 time.
3. After 25 times, P2 starts to execute until 50 times, when P1 is able to execute.
4. Now, comparing the deadline of (P1, P2) = (100, 75), P2 continues to execute.
5. P2 completes its processing at time 55.
6. P1 starts to execute until time 75, when P2 is able to execute.
7. Now, again comparing the deadline of (P1, P2) = (100, 150), P1 continues to execute.
8. Repeat the above steps…
9. Finally at time 150, both P1 and P2 have the same deadline, so P2 will continue to execute till its processing time after which P1 starts to execute.

Limitations of EDF scheduling algorithm:

- Transient Overload Problem
- Resource Sharing Problem
- Efficient Implementation Problem