**CHAPTER 2**

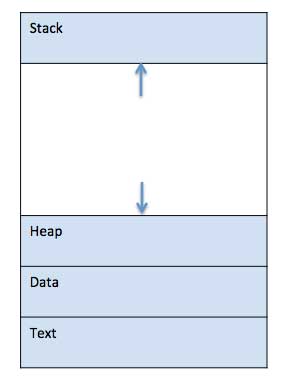
**Process and CPU Scheduling** - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads, and Interposes Communication, Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling.

**System call interface for process management**-fork, exit, wait, waitpid, exec.

**2.1 Process concepts**

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. when we execute 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 layout of a process inside main memory –



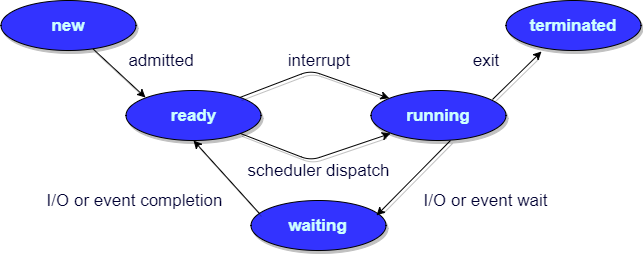
**Process Concept is** divided into four sections for efficient working:

* The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
* The **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
* The **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
* The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.

**2.2 Process States**

Processes in the operating system have following states:

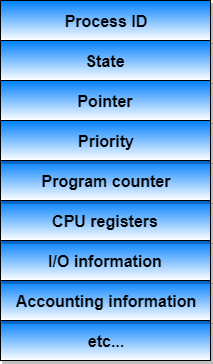
* NEW- The process is being created.
* READY- The process is waiting to be assigned to a processor.
* RUNNING- Instructions are being executed.
* WAITING- The process is waiting for some event to occur(such as an I/O completion or reception of a signal).
* TERMINATED- The process has finished execution.



## 2.3 Process Control Block

There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:

* **Process State**: It can be running, waiting etc.
* **Process ID** and the **parent process ID**.
* CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
* **CPU Scheduling** information: Such as priority information and pointers to scheduling queues.
* **Memory Management information**: For example, page tables or segment tables.
* **Accounting information**: The User and kernel CPU time consumed, account numbers, limits, etc.
* **I/O Status information**: Devices allocated, open file tables, etc.



# 2.4 Process Scheduling

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

The primary aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

**Scheduling fell into one of the two general categories:**

* **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
* **Pre-emptive Scheduling:** When the operating system decides to favor another process, pre-empting the currently executing process.

## 2.4.1 Scheduling Queues

The OS maintains all 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.

**2.4.2 Types of Schedulers**

There are three types of schedulers available:

1. Long Term Scheduler
2. Short Term Scheduler
3. 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.

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

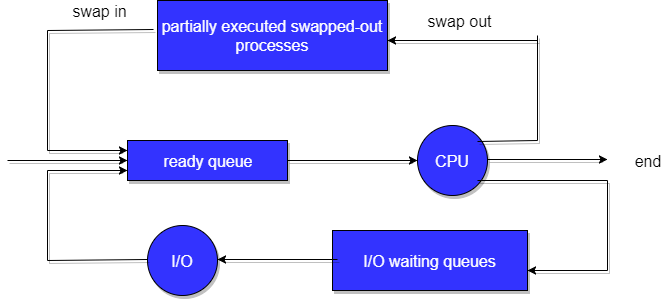
It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU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



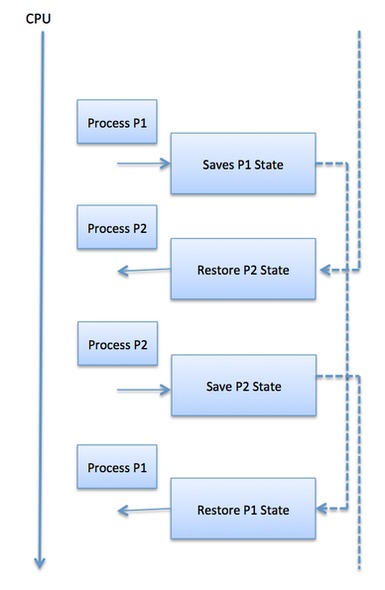
## Comparison among Scheduler

|  |  |  |  |
| --- | --- | --- | --- |
| 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. |

## 2.4.3. Context Switch

A context switch 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.

## 2.5 Operations on Process

Operations on Process are two:

1. Process Creation
2. Process Termination.

**2.5.1 Process Creation**

The process which creates other process is termed the **parent** of the other process, while the created sub-process is termed its **child**.

Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.

There are two options for the parent process after creating the child:

* Wait for the child process to terminate before proceeding. Parent process makes a **wait()** system call, for either a specific child process or for any particular child process, which causes the parent process to block until the wait() returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
* Run concurrently with the child, continuing to process without waiting. When a UNIX shell runs a process as a background task, this is the operation seen. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

**2.5.2 Process Termination.**

By making the exit(system call), typically returning an int, processes may request their own termination. This int is passed along to the parent if it is doing a wait(), and is typically zero on successful completion and some non-zero code in the event of any problem.

Processes may also be terminated by the system for a variety of reasons, including :

* The inability of the system to deliver the necessary system resources.
* In response to a KILL command or other unhandled process interrupts.
* A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
* If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them.)

When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate.

**2.6 Cooperating Processes**

Cooperating processes are those that can affect or are affected by other processes running on the system. Cooperating processes may share data with each other.

Reasons for needing cooperating processes

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

1. **Modularity**

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

1. **Information Sharing**

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

1. **Convenience**

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

1. **Computation Speedup**

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

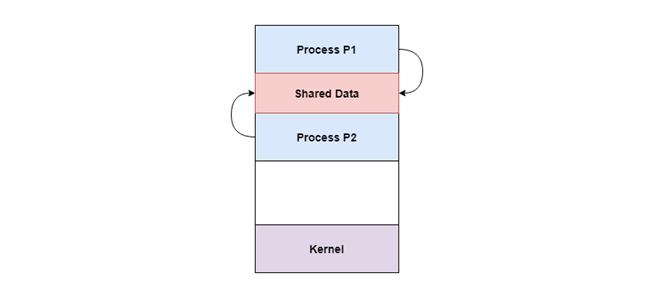
**Methods of Cooperation**

Cooperating processes can coordinate with each other using shared data or messages. Details about these are given as follows:

1. **Cooperation by Sharing**

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

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

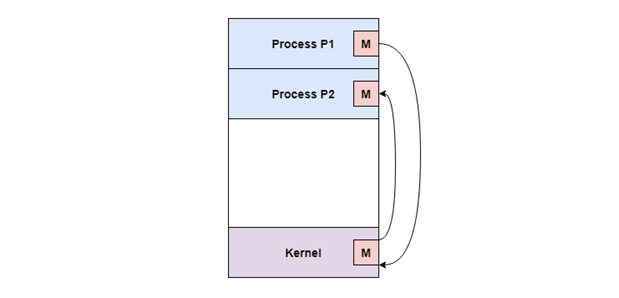


In the above diagram, Process P1 and P2 can cooperate with each other using shared data such as memory, variables, files, databases etc.

1. **Cooperation by Communication**

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

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



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

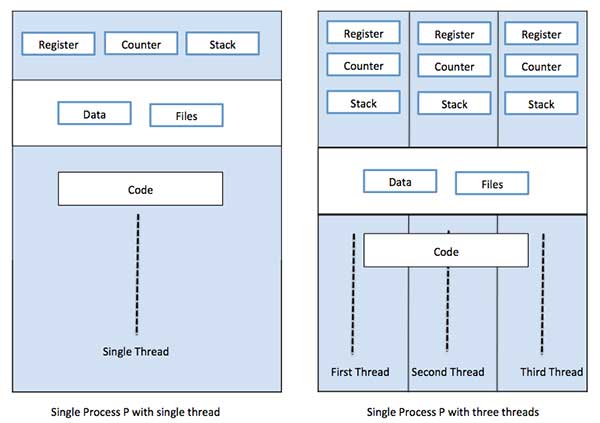
**2.7 Threads**

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

****

**2.7.1 Difference between Process and Thread**

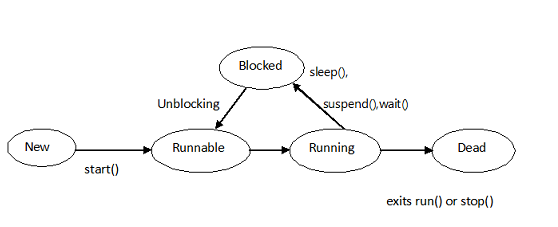
|  |  |  |
| --- | --- | --- |
| **S.N.** | **Process** | **Thread** |
| 1 | Process is heavy weight or resource intensive. | Thread is light weight, taking lesser resources than a process. |
| 2 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 3 | In multiple processing environments, each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 4 | If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, a second thread in the same task can run. |
| 5 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

**Advantages of Thread**

* Threads minimize the context switching time.
* Use of threads provides concurrency within a process.
* Efficient communication.
* It is more economical to create and context switch threads.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

## 2.7.2 Thread Life Cycle

## A thread goes through various stages in its life cycle. The following diagram shows the complete life cycle of a thread.



* 1. **New:** A new thread is created but not working. A thread after creation and before invocation of start() method will be in new state.
  2. **Runnable:** A thread after invocation of start() method will be in runnable state. A thread in runnable state will be available for thread scheduler.
  3. **Running:** A thread in execution after thread scheduler select it, it will be in running state.
  4. **Blocked:** A thread which is alive but not in runnable or running state will be in blocked state. A thread can be in blocked state because of suspend(), sleep(), wait() methods or implicitly by JVM to perform I/O operations.
  5. **Dead:** A thread after exiting from run() method will be in dead state. We can use stop() method to forcefully killed a thread.

**2.7.2.1 Thread Priorities**

Every thread has a priority that helps the operating system determine the order in which threads are scheduled.

Thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

## 2.7.2.2 Create a Thread by Implementing a Runnable Interface

If class is intended to be executed as a thread then by implementing a **Runnable** interface.

### Step 1

As a first step, to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and will put complete business logic inside this method. Following is a simple syntax of the run() method −

**public void run( )**

### Step 2

As a second step, will instantiate a **Thread** object using the following constructor −

**Thread(Runnable threadObj, String threadName);**

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

### Step 3

Once a Thread object is created, it can start by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

**void start();**

## 2.7.2.4. Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

### Step 1

Need to override **run( )** method available in Thread class. This method provides an entry point for the thread and will put your complete business logic inside this method. Following is a simple syntax of run() method −

**public void run( )**

### Step 2

Once Thread object is created, it can start by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

**void start( );**

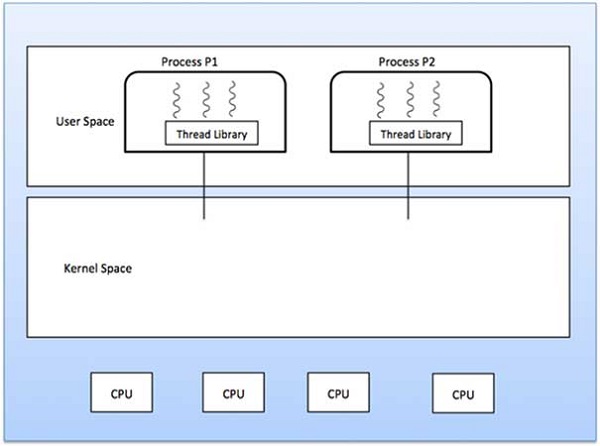
## 2.7.3 Types of Thread

Threads are implemented in following two ways −

* **User Level Threads** − User managed threads.
* **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

## User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



### Advantages

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.

### Disadvantages

* In a typical operating system, most system calls are blocking.
* Multithreaded application cannot take advantage of multiprocessing.

## Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

### Advantages

* Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Kernel routines themselves can be multithreaded.

### Disadvantages

* Kernel threads are generally slower to create and manage than the user threads.
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

## 2.7.4 Multithreading Models

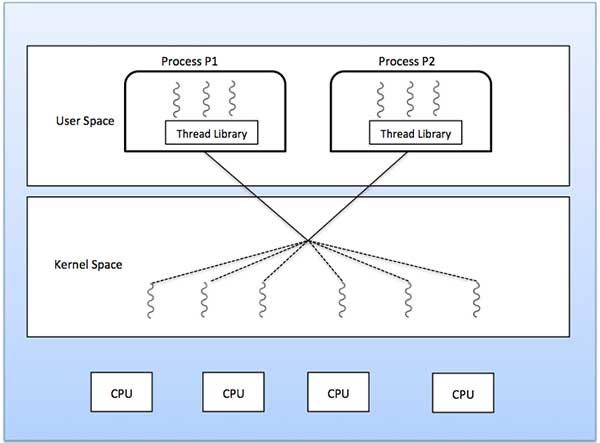
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

* Many to many relationship.
* Many to one relationship.
* One to one relationship.

## Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

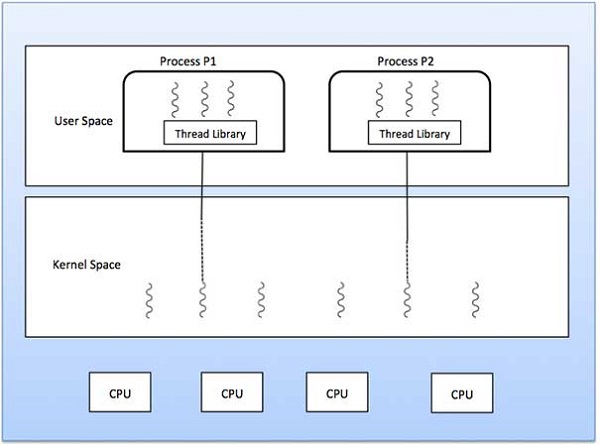
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



## Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

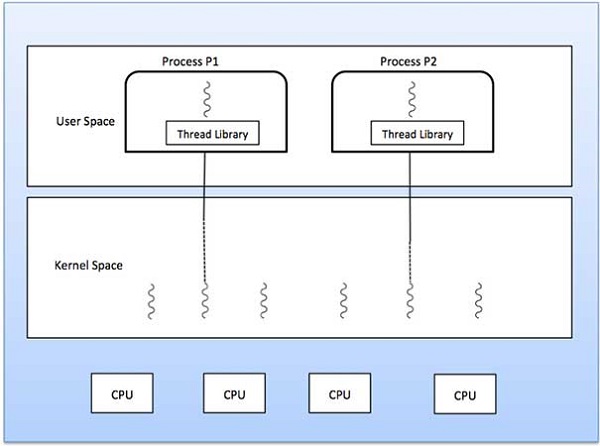
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



## One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



## 2.7.5 Difference between User-Level & Kernel-Level Thread

|  |  |  |
| --- | --- | --- |
| **S.N.** | **User-Level Threads** | **Kernel-Level Thread** |
| 1 | User-level threads are faster to create and manage. | Kernel-level threads are slower to create and manage. |
| 2 | Implementation is by a thread library at the user level. | Operating system supports creation of Kernel threads. |
| 3 | User-level thread is generic and can run on any operating system. | Kernel-level thread is specific to the operating system. |
| 4 | Multi-threaded applications cannot take advantage of multiprocessing. | Kernel routines themselves can be multithreaded. |

## 2. 7.6 Thread Methods

Following is the list of important methods available in the Thread class.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void start()**  Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. |
| 2 | **public void run()**  If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object. |
| 3 | **public final void setName(String name)**  Changes the name of the Thread object. There is also a getName() method for retrieving the name. |
| 4 | **public final void setPriority(int priority)**  Sets the priority of this Thread object. The possible values are between 1 and 10. |
| 5 | **public final void setDaemon(boolean on)**  A parameter of true denotes this Thread as a daemon thread. |
| 6 | **public final void join(long millisec)**  The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | **public void interrupt()**  Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | **public final boolean isAlive()**  Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public static void yield()**  Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled. |
| 2 | **public static void sleep(long millisec)**  Causes the currently running thread to block for at least the specified number of milliseconds. |
| 3 | **public static boolean holdsLock(Object x)**  Returns true if the current thread holds the lock on the given Object. |
| 4 | **public static Thread currentThread()**  Returns a reference to the currently running thread, which is the thread that invokes this method. |
| 5 | **public static void dumpStack()**  Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |

## 2.8 Scheduling Criteria

There are many different criterias to check when considering the scheduling algorithm, they are:

#### CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

#### Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

#### Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

#### Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

#### Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

#### Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

## 2.9 Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilization, computer scientists have defined some algorithms, they are:

1. [First Come First Serve(FCFS) Scheduling](https://www.studytonight.com/operating-system/first-come-first-serve)
2. [Shortest-Job-First(SJF) Scheduling](https://www.studytonight.com/operating-system/shortest-job-first)
3. [Priority Scheduling](https://www.studytonight.com/operating-system/priority-scheduling)
4. [Round Robin(RR) Scheduling](https://www.studytonight.com/operating-system/round-robin-scheduling)
5. [Multilevel Queue Scheduling](https://www.studytonight.com/operating-system/multilevel-queue-scheduling)
6. [Multilevel Feedback Queue Scheduling](https://www.studytonight.com/operating-system/multilevel-feedback-queue-scheduling)

# First Come First Serve Scheduling

In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

* First Come First Serve is just like **FIFO** (First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.
* This is used in [Batch Systems](https://www.studytonight.com/operating-system/types-of-os).
* It's **easy to understand and implement** programmatically, using a Queue data structure, where a new process enters through the **tail** of the queue, and the scheduler selects process from the **head** of the queue.
* A perfect real life example of FCFS scheduling is **buying tickets at ticket counter**.

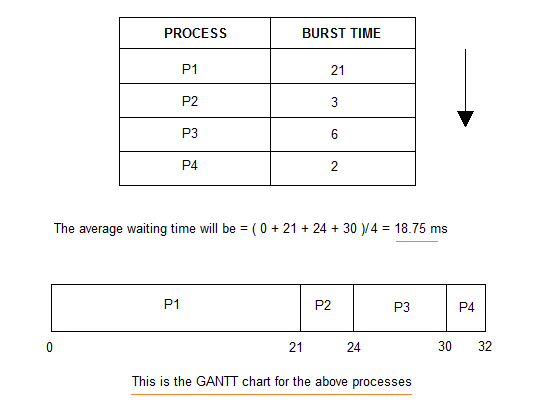
**Calculating Average Waiting Time**

For every scheduling algorithm, **Average waiting time** is a crucial parameter to judge it's performance.

AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.

***Lower the Average Waiting Time, better the scheduling algorithm.***

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with **Arrival Time** 0, and given **Burst Time**, let's find the average waiting time using the FCFS scheduling algorithm.



The average waiting time will be 18.75 ms

For the above given proccesses, first **P1** will be provided with the CPU resources,

* Hence, waiting time for **P1** will be 0
* **P1** requires 21 ms for completion, hence waiting time for **P2** will be 21 ms
* Similarly, waiting time for process **P3** will be execution time of **P1** + execution time for **P2**, which will be (21 + 3) ms = 24 ms.
* For process **P4** it will be the sum of execution times of **P1**, **P2** and **P3**.

The **GANTT chart** above perfectly represents the waiting time for each process.

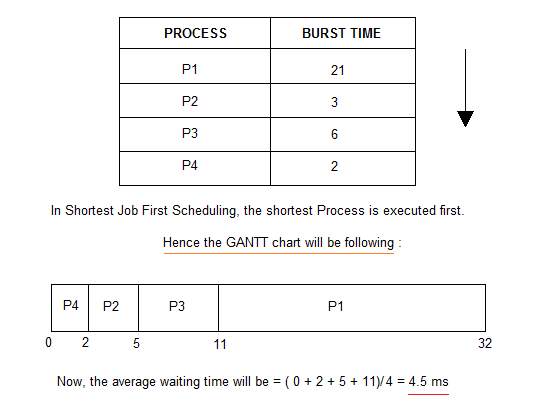
# Shortest Job First(SJF) Scheduling

Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

* This is the best approach to minimize waiting time.
* This is used in [Batch Systems](https://www.studytonight.com/operating-system/types-of-os).
* It is of two types:
  1. Non Pre-emptive
  2. Pre-emptive
* To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.
* 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)

## Non Pre-emptive Shortest Job First

Consider the below processes available in the ready queue for execution, with **arrival time** as 0 for all and given **burst times**.



As you can see in the **GANTT chart** above, the process **P4** will be picked up first as it has the shortest burst time, then **P2**, followed by **P3** and at last **P1**.

We scheduled the same set of processes using the [First come first serve](https://www.studytonight.com/operating-system/first-come-first-serve) algorithm in the previous tutorial, and got average waiting time to be 18.75 ms, whereas with SJF, the average waiting time comes out 4.5 ms.

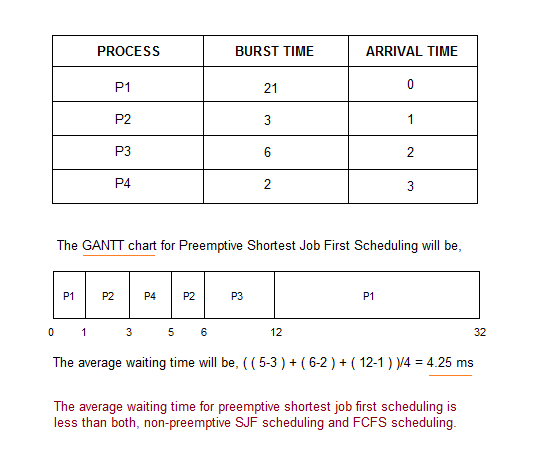
### Problem with Non Pre-emptive SJF

If the **arrival time** for processes are different, which means all the processes are not available in the ready queue at time 0, and some jobs arrive after some time, in such situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of **aging**.

## Pre-emptive Shortest Job First

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with **short burst time** arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.



As you can see in the **GANTT chart** above, as **P1** arrives first, hence it's execution starts immediately, but just after 1 ms, process **P2** arrives with a **burst time** of 3 ms which is less than the burst time of **P1**, hence the process **P1**(1 ms done, 20 ms left) is preemptied and process **P2** is executed.

As **P2** is getting executed, after 1 ms, **P3** arrives, but it has a burst time greater than that of **P2**, hence execution of **P2** continues. But after another millisecond, **P4** arrives with a burst time of 2 ms, as a result **P2**(2 ms done, 1 ms left) is preemptied and **P4** is executed.

After the completion of **P4**, process **P2** is picked up and finishes, then **P2** will get executed and at last **P1**.

The Pre-emptive SJF is also known as **Shortest Remaining Time First**, because at any given point of time, the job with the shortest remaining time is executed first.

# Priority CPU Scheduling

In this tutorial we will understand the priority scheduling algorithm, how it works and its advantages and disadvantages.

In the [Shortest Job First](https://www.studytonight.com/operating-system/shortest-job-first) scheduling algorithm, the priority of a process is generally the inverse of the CPU burst time, i.e. the larger the burst time the lower is the priority of that process.

In case of priority scheduling the priority is not always set as the inverse of the CPU burst time, rather it can be internally or externally set, but yes the scheduling is done on the basis of priority of the process where the process which is most urgent is processed first, followed by the ones with lesser priority in order.

Processes with same priority are executed in FCFS manner.

The priority of process, when internally defined, can be decided based on **memory requirements**, **time limits** ,**number of open files**, **ratio of I/O burst to CPU burst** etc.

Whereas, external priorities are set based on criteria outside the operating system, like the importance of the process, funds paid for the computer resource use, makrte factor etc.

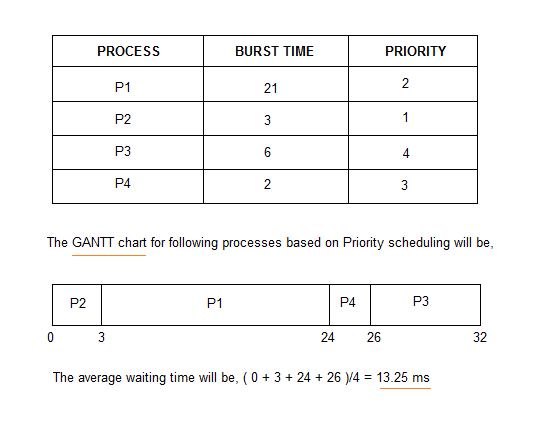
### Types of Priority Scheduling Algorithm

Priority scheduling can be of two types:

1. **Preemptive Priority Scheduling**: If the new process arrived at the ready queue has a higher priority than the currently running process, the CPU is preempted, which means the processing of the current process is stoped and the incoming new process with higher priority gets the CPU for its execution.
2. **Non-Preemptive Priority Scheduling**: In case of non-preemptive priority scheduling algorithm if a new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue, which means after the execution of the current process it will be processed.

## Example of Priority Scheduling Algorithm

Consider the below table fo processes with their respective CPU burst times and the priorities.



### Problem with Priority Scheduling Algorithm

In priority scheduling algorithm, the chances of **indefinite blocking** or **starvation**.

A process is considered blocked when it is ready to run but has to wait for the CPU as some other process is running currently.

But in case of priority scheduling if new higher priority processes keeps coming in the ready queue then the processes waiting in the ready queue with lower priority may have to wait for long durations before getting the CPU for execution.

In 1973, when the IBM 7904 machine was shut down at MIT, a low-priority process was found which was submitted in 1967 and had not yet been run.

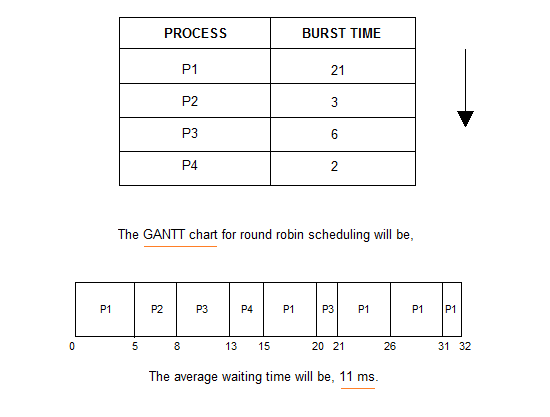
### Using Aging Technique with Priority Scheduling

To prevent starvation of any process, we can use the concept of **aging** where we keep on increasing the priority of low-priority process based on the its waiting time.

For example, if we decide the aging factor to be **0.5** for each day of waiting, then if a process with priority **20**(which is comparitively low priority) comes in the ready queue. After one day of waiting, its priority is increased to **19.5** and so on.

# 4. Round Robin Scheduling

* A fixed time is allotted to each process, called **quantum**, for execution.
* Once a process is executed for given time period that process is preemptied and other process executes for given time period.
* Context switching is used to save states of preemptied processes.

****

# 5. Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

**For example:** A common division is made between foreground(or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A multi-level queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

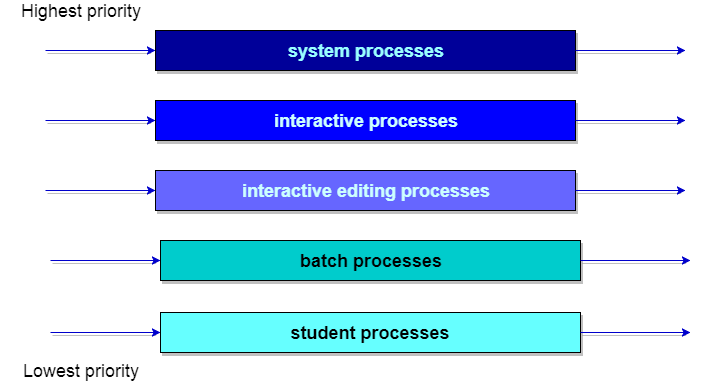
**For example:** separate queues might be used for foreground and background processes. The foreground queue might be scheduled by Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

n addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. **For example:** The foreground queue may have absolute priority over the background queue.

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

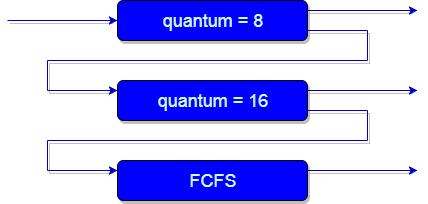
Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process will be preempted.



# Multilevel Feedback Queue Scheduling

In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.



**An example of a multilevel feedback queue can be seen in the below figure.**

In general, a multilevel feedback queue scheduler is defined by the following parameters:

* The number of queues.
* The scheduling algorithm for each queue.
* The method used to determine when to upgrade a process to a higher-priority queue.
* The method used to determine when to demote a process to a lower-priority queue.
* The method used to determine which queue a process will enter when that process needs service.

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the **most general scheme**, it is also the **most complex**.

**2.10 Multiple -Processor Scheduling.**

In multiple-processor scheduling **multiple CPU’s** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

**Approaches to Multiple-Processor Scheduling –**

**One approach** is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A **second approach** uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own 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.