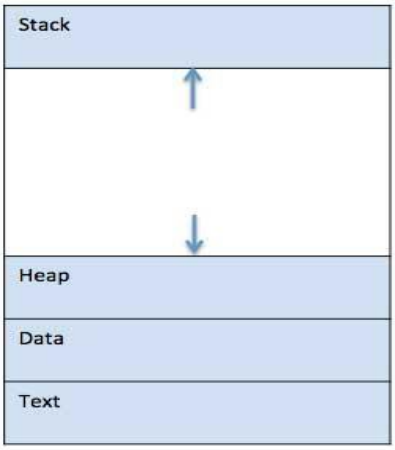
unit-2

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

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 −



Process Components

1. Stack

The process Stack contains the temporary data such as method/function parameters, return address and local variables.

2. Heap

This is dynamically allocated memory to a process during its run time.

3 .Text

This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

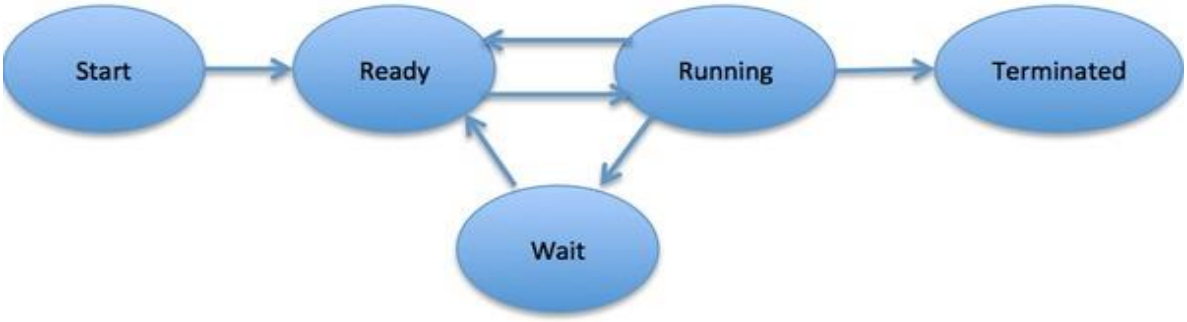
4 .Data

This section contains the global and static variables.

Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.



S.No	State & Description
1	**Start:-**This is the initial state when a process is first started/created.
2	**Ready:-**The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it but interrupted by the scheduler to assign CPU to some other process.
3	**Running:-**Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting:-**Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

5	**Terminated or Exit:-**Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.No	Information & Description
1	**Process State:-**The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges:-**This is required to allow/disallow access to system resources.
3	**Process ID:-**Unique identification for each of the process in the operating system.
4	**Pointer:-**points to parent process.

5	**Priority:-** The process is assigned the priority at the time of its creation. The priority of the process may get changed over its lifetime depending on the various parameters.
5	**Program Counter:-**Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers:-**Various CPU registers where processes need to be stored for execution for running state.
7	**CPU Scheduling Information:-**Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information:-**This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information:-**This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information:-**This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on the Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −

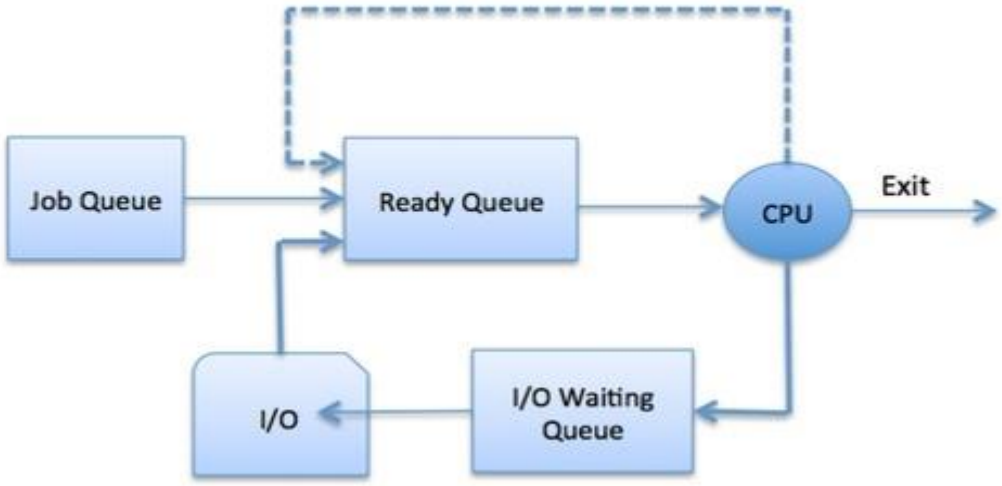
The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Process 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.



Schedulers

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

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

Long Term Scheduler

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

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

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

Short Term Scheduler

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

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

Medium Term Scheduler

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

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

Comparison among Scheduler

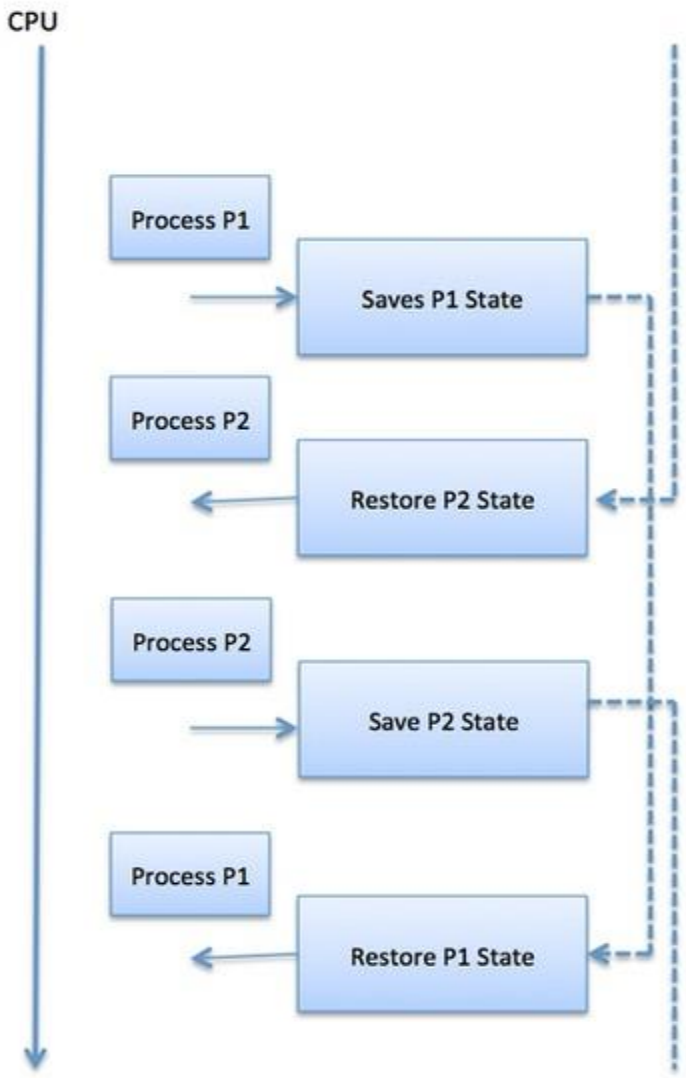
S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.

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

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in the 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 multitasking operating system features.

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



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

Scheduling algorithms

- **Completion Time** is the time required by the process to complete its execution
- **Turnaround Time** is the time interval between the submission of a process and its completion.
  Turnaround Time = completion of a process – submission of a process
- **Waiting Time** is the difference between turnaround time and burst time
  Waiting Time = turnaround time – burst time

Cpu scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-first (SJF) Scheduling
3. Priority Scheduling
4. Shortest Remaining Time
5. Round Robin(RR) Scheduling
6. Multiple-Level Queues Scheduling

FCFS

First Come, First Served (FCFS) also known as First In, First Out(FIFO) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue.

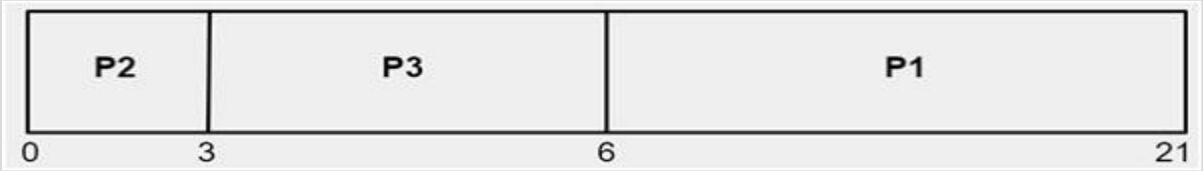
FCFS follows non-preemptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not get terminated or may get halted due to some I/O interrupt.

Example

Let's say, there are four processes arriving in the sequence as P2, P3, P1 with their corresponding execution time as shown in the table below. Also, taking their arrival time to be 0.

Process	Order of arrival	Execution time in msec
P1	3	15
P2	1	3
P3	2	3

Gantt chart showing the waiting time of processes P1, P2 and P3 in the system



As shown above,

The waiting time of process P2 is 0

The waiting time of process P3 is 3

The waiting time of process P1 is 6

Average time = (0 + 3 + 6) / 3 = 3 msec.

As we have taken arrival time to be 0 therefore turn around time and completion time will be the same.

shortest job first scheduling

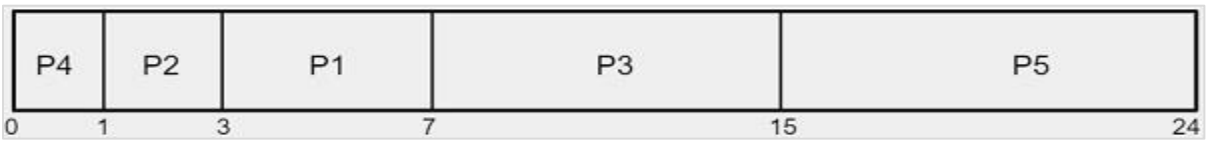
scheduler selects the process from the waiting queue with the least completion time and allocate the CPU to that job or process. Shortest Job First is more desirable than FCFS algorithm because SJF is more optimal as it reduces average wait time which will increase the throughput**.**

Example

We are given with the processes P1, P2, P3, P4 and P5 having their corresponding burst time given below

Process	Burst Time
P1	4
P2	2
P3	8
P4	1
P5	9

Since the burst time of process P4 is minimum amongst all the processes it will be allocated CPU first. After that, P2 will be queued then P1, P3 and P5.



Average waiting time is calculated on the basis of the gantt chart. P1 have to wait for 3, P2 have to wait for 1, P3 have to wait for 7, P4 have to wait for 0 and P5 have to wait for 15. So, their average waiting time will be −

```
Average Waiting Time = (0 + 1 + 3 + 7 + 15) / 5
                     =  26 / 5
                     =  5
```

shortest-remaining-time-first scheduling

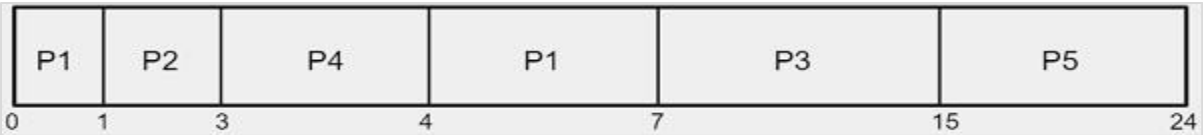
SJF algorithms can be preemptive as well as non-preemptive. Preemptive scheduling is also known as **shortest-remaining-time-first scheduling.** In Preemptive approach, the new process arises when there is already an executing process. If the burst of a newly arriving process is lesser than the burst time of the executing process, the scheduler will preempt the execution of the process with lesser burst time.

Example

We are given with the processes P1, P2, P3, P4 and P5 having their corresponding burst time given below

Process	Burst Time	Arrival Time
P1	4	0
P2	2	1
P3	8	2
P4	1	3
P5	9	4

Since the arrival time of P1 is 0 it will be the first one to get executed till the arrival of another process. When at 1 the process P2 enters and the burst time of P2 is less than the burst time of P1 therefore scheduler will dispatch the CPU with the process P2 and so on.



Average waiting time =(4-1)+1+7+3+15/5

=(3+1+7+3+15)/5

=29/5=5.8ms

Priority Scheduling

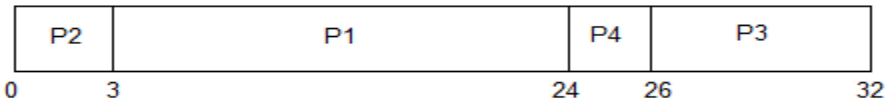
Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

The difference between preemptive priority scheduling and non preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

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

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

problem:-

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

Solution:-

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

For example, if we decide the aging factor to be 1 for each execution of another process and waiting, then if a process with priority 20(which is comparatively low priority) comes in the ready queue. After one execution of another process and waiting, its priority is increased to 19

and so on.

Round Robin(RR):-

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.

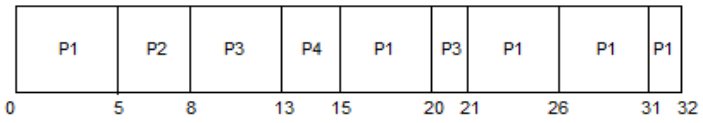
- A fixed time is allotted to each process, called a quantum, for execution.
- Once a process is executed for the given time period that process is preempted and another process executes for the given time period.
- Context switching is used to save states of preempted processes.
- This algorithm is simple and easy to implement and the most important 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 algorithms.
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.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



Waiting time p1= 0+(15-5)+(21-20)+(26-26)+(31-31)

0 + 10 + 1 + 0 + 0 =11

p2=5

p3=8

p4=13

Average waiting time=11+5+8+13/4=9.25 ms

A multi-level queue scheduling algorithm

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 the Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

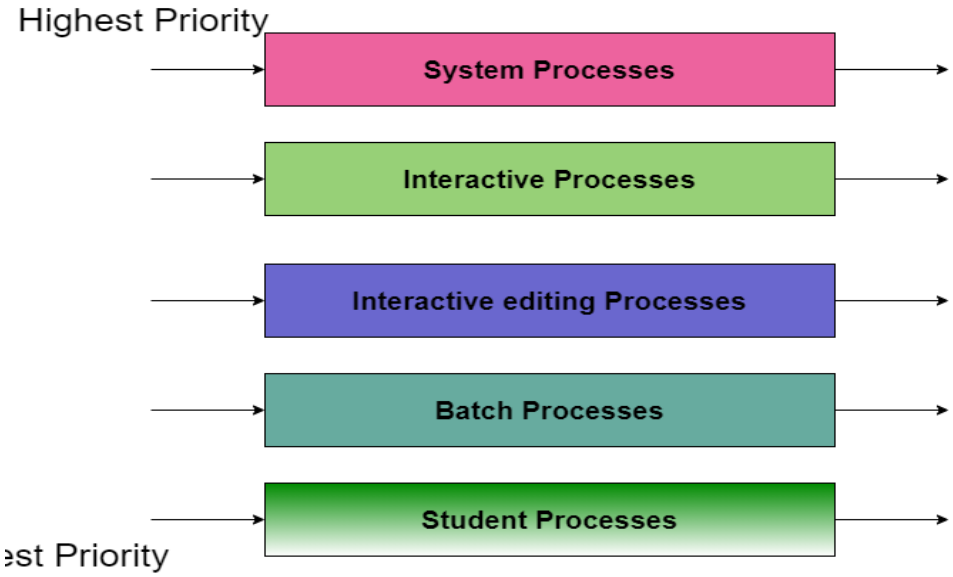
In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, The foreground queue may have absolute priority over the background queue.

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.



Lowest priority

In this case, if there are no processes on the higher priority queue only then the processes on the low priority queues will run. For Example: Once processes on the system queue, the Interactive queue, and Interactive editing queue become empty, only then the processes on the batch queue will run.

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

- **System Process** The Operating system itself has its own process to run and is termed as System Process.
- **Interactive Process** The Interactive Process is a process in which there should be the same kind of interaction (basically an online game).
- **Batch Processes** Batch processing is basically a technique in the Operating system that collects the programs and data together in the form of the batch before the processing starts.

- **Student Process** The system process always gets the highest priority while the student processes always get the lowest priority.

In an operating system, there are many processes, in order to obtain the result we cannot put all processes in a queue; thus this process is solved by Multilevel queue scheduling.

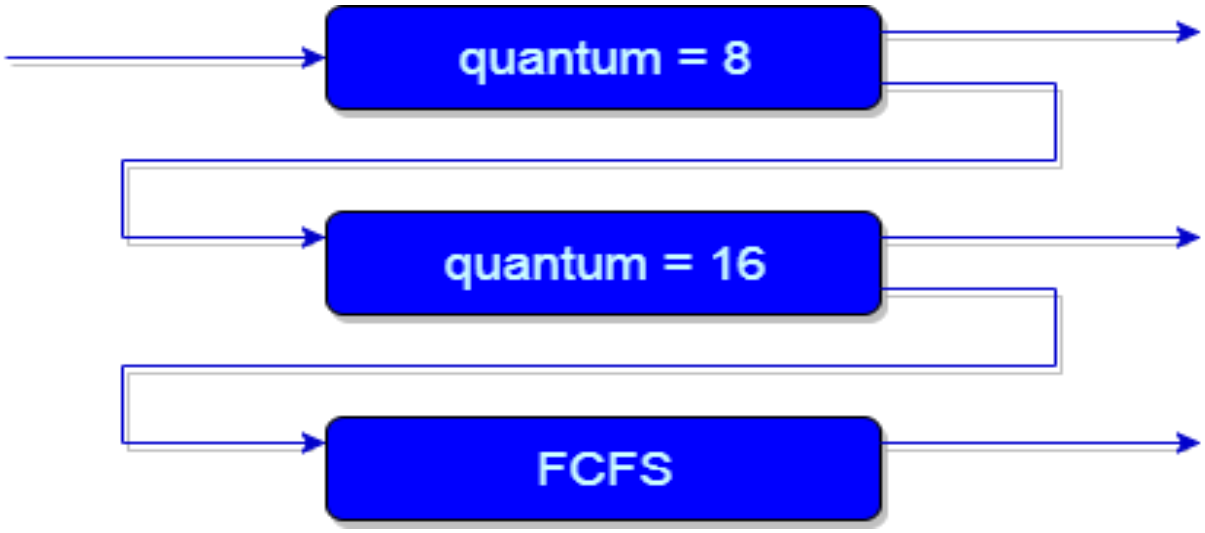
utu.be/P8XHqYnzux0https://yo

Multilevel feedback queue scheduling

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.
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.



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

Explanation:

First of all, Suppose that queues 1 and 2 follow round robin with time quantum 8 and 16 respectively and queue 3 follows FCFS. One of the implementations of Multilevel Feedback Queue Scheduling is as follows:

1. If any process starts executing then firstly it enters queue 1.
2. In queue 1, the process executes for 8 unit and if it completes in these 8 units or it gives CPU for I/O operation in these 8 units unit than the priority of this process does not change, and if for some reasons it again comes in the ready queue than it again starts its execution in the Queue 1.
3. If a process that is in queue 1 does not complete in 8 units then its priority gets reduced and it gets shifted to queue 2.
4. Above points 2 and 3 are also true for processes in queue 2 but the time quantum is 16 units. Generally, if any process does not complete in a given time quantum then it gets shifted to the lower priority queue.
5. After that in the last queue, all processes are scheduled in an FCFS manner.
6. It is important to note that a process that is in a lower priority queue can only execute only when the higher priority queues are empty.
7. Any running process in the lower priority queue can be interrupted by a process arriving in the higher priority queue.

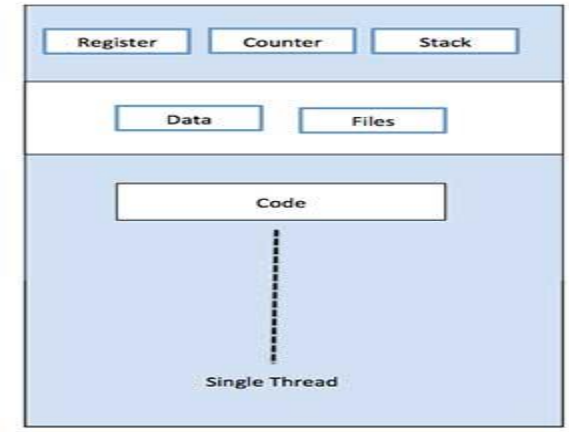
Also, the above implementation may differ for the example in which the last queue will follow Round-robin Scheduling.

problem ; Any process that is in the lower priority queue has to suffer starvation due to some short processes that are taking all the CPU time.

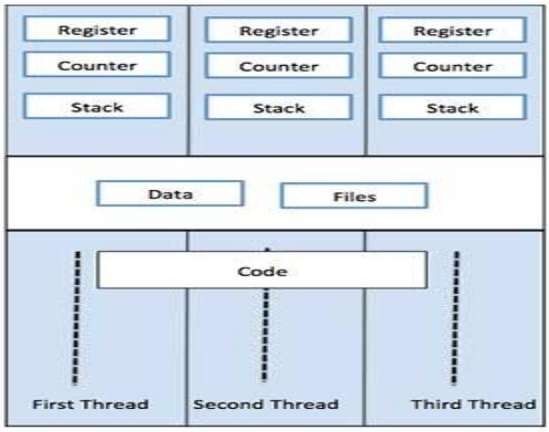
solution : There is a solution that is to boost the priority of all the processes after regular intervals then place all the processes in the highest priority queue.

THREADS

A thread is also called a lightweight process**.**A thread is a path of execution within a process. A process can contain multiple threads,The idea is to achieve parallelism by dividing a process into multiple threads



Single Process P with single thread



Single Process P with three threads

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.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is lightweight, taking lesser resources than a process.

2	Process switching needs interaction with the operating system.	Thread switching does not need to interact with the 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 the 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.

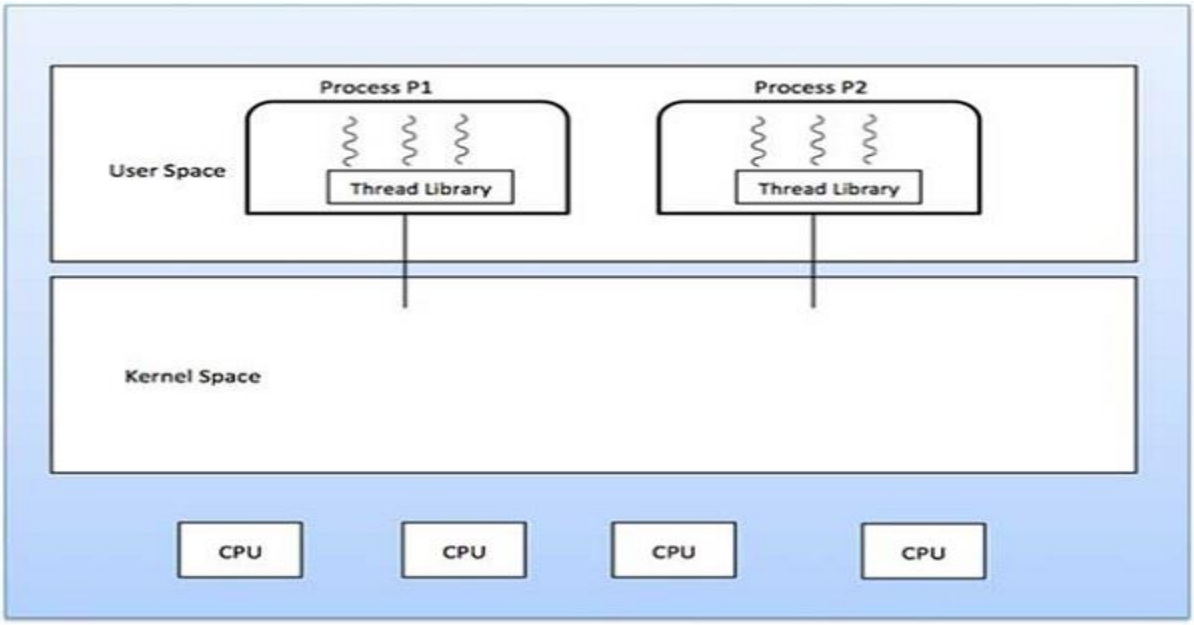
Types of Thread

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 applications 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

- Kernels 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.

Multithreading Models

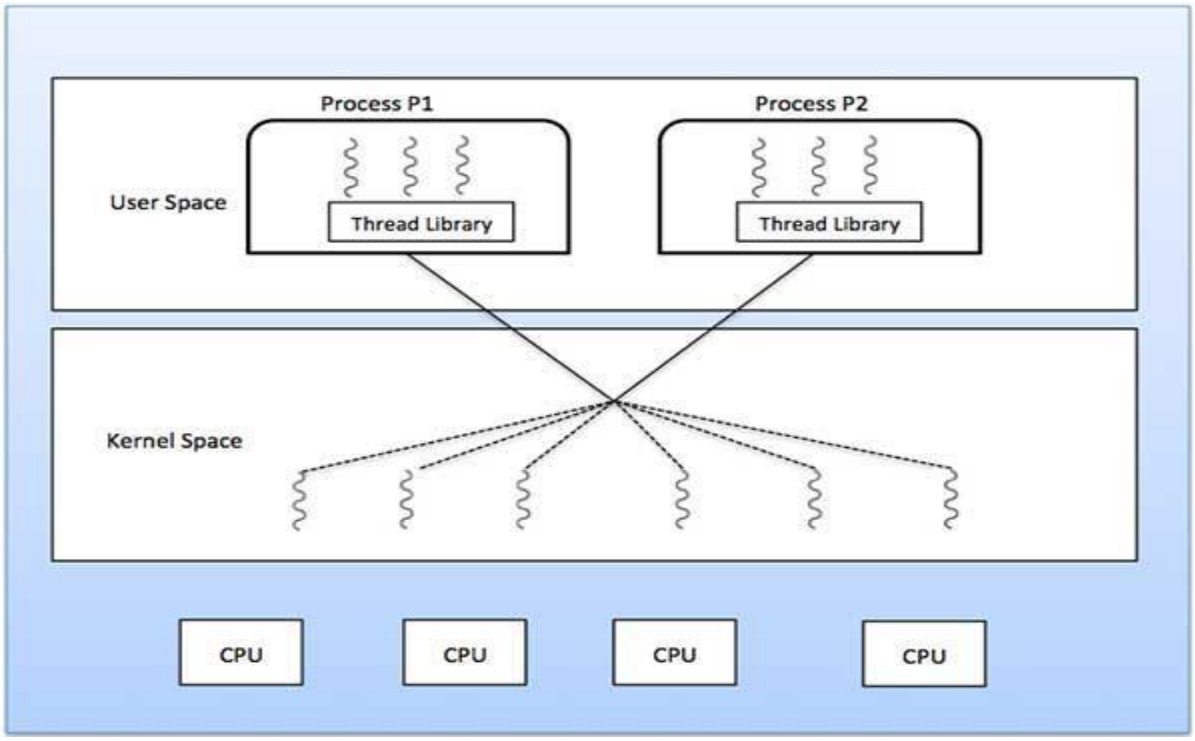
Some operating systems provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

- Many to many relationships.
- 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.

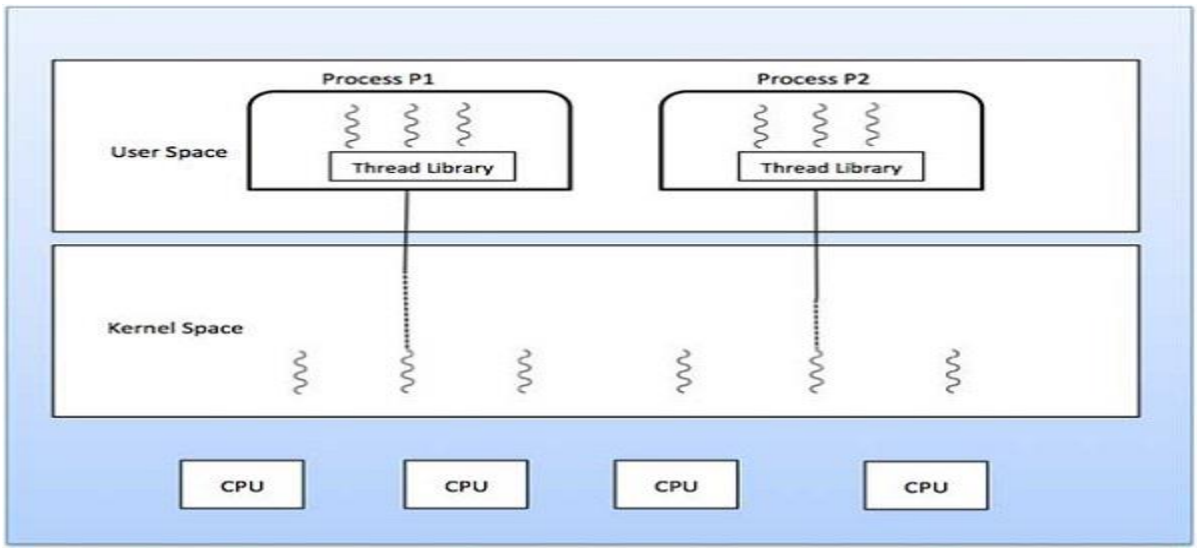
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model 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.

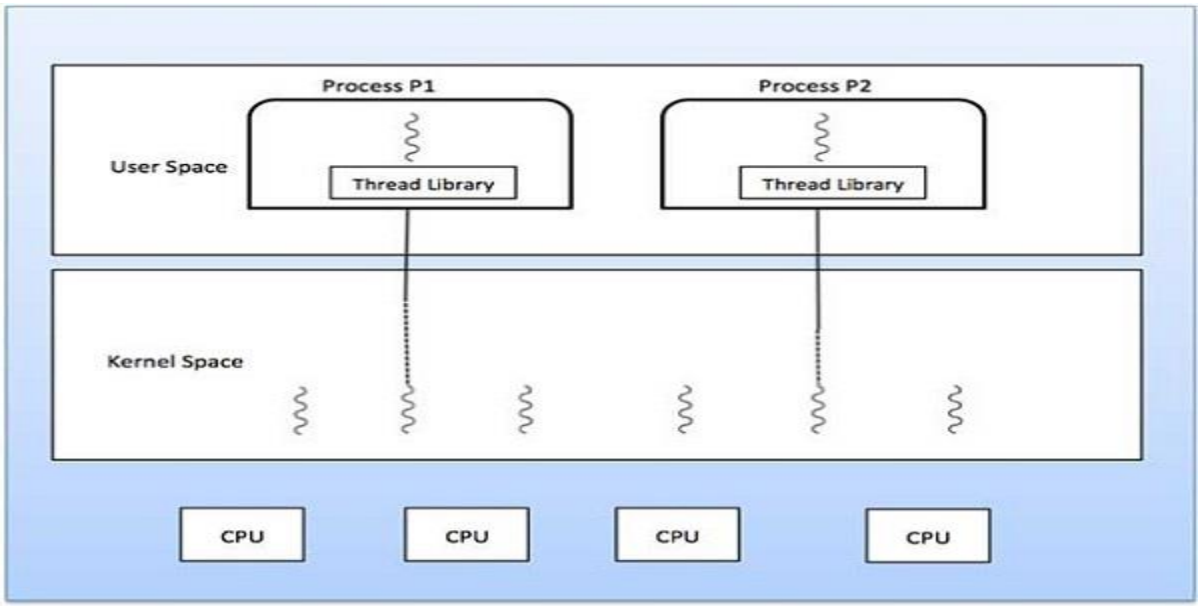
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

There is a 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.

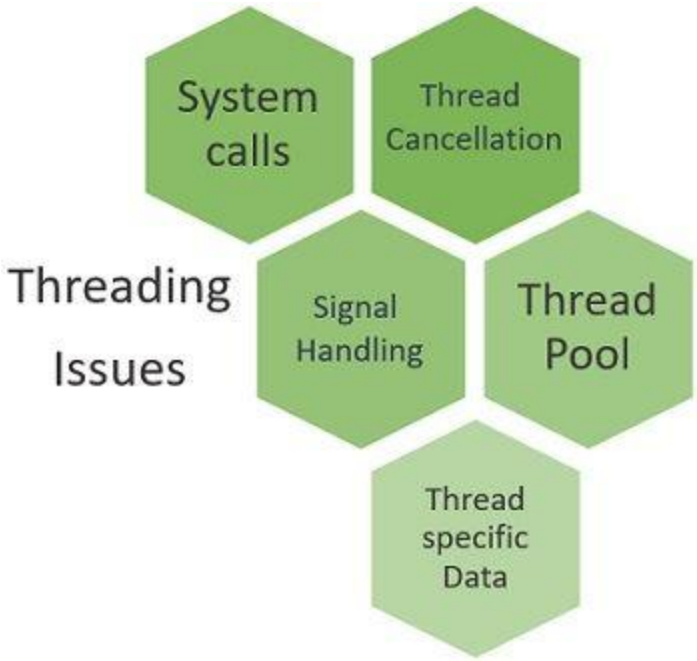


Difference between User-Level & Kernel-Level Thread

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.

Threading Issues

- The fork() and exec() system call
- Signal handling
- Thread cancelation
- Thread local storage



The fork() and exec() system call

The fork() and exec() are the system calls. The fork() call creates a duplicate process of the process that invokes fork(). The new duplicate process is called child process and process invoking the fork() is called the parent process. Both the parent process and the child process continue their execution from the instruction that is just after the fork().

Let us now discuss the issue with the fork() system call. Consider that a thread of the multithreaded program has invoked the fork(). So, the fork() would create a new duplicate process. Here the issue is whether the new duplicate process created by fork() will duplicate all the threads of the parent process or the duplicate process would be single-threaded.

Well, there are two versions of fork() in some of the UNIX systems. Either the fork() can duplicate all the threads of the parent process in the child process or the fork() would only duplicate that thread from parent process that has invoked it.

Which version of fork() must be used totally depends upon the application.

Next system call i.e. exec() system call when invoked replaces the program along with all its threads with the program that is specified in the parameter to exec(). Typically the exec() system call is lined up after the fork() system call.

Here the issue is if the exec() system call is lined up just after the fork() system call then

duplicating all the threads of parent process in the child process by fork() is useless. As the exec() system call will replace the entire process with the process provided to exec() in the parameter.

In such case, the version of fork() that duplicates only the thread that invoked the fork() would be appropriate

Signal Handling

Whenever a multithreaded process receives a signal then to what thread should that signal be conveyed? There are following four main option for signal distribution:

1. Signal deliver to the thread to which the signal applies.
2. Signal deliver to each and every thread in the process.
3. Signal deliver to some of the threads in the process.
4. Assign a particular thread to receive all the signals in a process.

how the signal would be delivered to the thread would be decided, depending upon the type of generated signal. The generated signal can be classified into two type's synchronous signal and asynchronous signal.

Synchronous signals are forwarded to the same process that leads to the generation of the signal. Asynchronous signals are generated by the event external to the running process thus the running process receives the signals asynchronously.

Thread Cancellation

Threads that are no-longer required can be cancelled by another thread in one of two techniques:

1. Asynchronies cancellation
2. Deferred cancellation

Asynchronies Cancellation

It means cancellation of thread immediately. Allocation of resources and inter thread data transfer may be challenging for asynchronies cancellation.

Deferred Cancellation

In this method a flag is set that indicates the thread should cancel itself when it is feasible. It's upon the cancelled thread to check this flag intermittently and exit nicely when it sees the set flag.

Thread Local Storage or thread pool

When a user requests for a webpage to the server, the server creates a separate thread to service the request. Although the server also has some potential issues. Consider if we do not have a bound on the number of actives thread in a system and would create a new thread for every new request then it would finally result in exhaustion of system resources.

We are also concerned about the time it will take to create a new thread. It must not be that case that the time require to create a new thread is more than the time required by the thread to service the request and then getting discarded as it would result in wastage of CPU time.

The solution to this issue is the **thread pool**. The idea is to create a finite amount of threads when the process starts. This collection of threads is referred to as the thread pool. The threads stay in the thread pool and wait till they are assigned any request to be serviced.

Whenever the request arrives at the server, it invokes a thread from the pool and assigns it the request to be serviced. The thread completes its service and return back to the pool and wait for the next request.

If the server receives a request and it does not find any thread in the thread pool it waits for some or the other thread to become free and return to the pool. This much better than creating a new thread each time a request arrives and convenient for the system that cannot handle a large number of concurrent threads.