UNIT-II

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

----
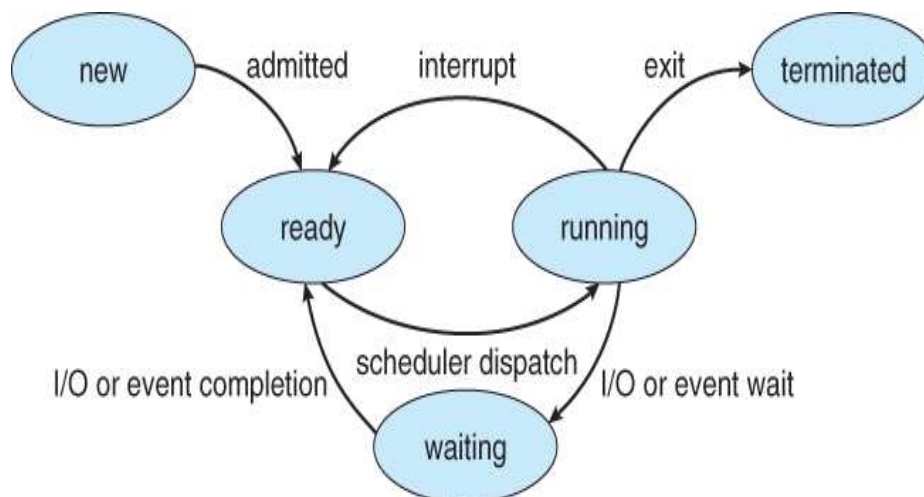
Process

Definition: "A program in the execution is called a Process .it is an entity which represents the basic unit of work to be implemented in the system."

Program:

- A program by itself is not a process. It is a static entity made up of program statement while process is a dynamic entity. Program contains the instructions to be executed by processor.
- A program takes a space at single place in main memory and continues to stay there. A program does not perform any action by itself.

Process States

- As a process executes, it changes state. The state of a process is defined as the current activity of the process.
- Process can have one of the following five states at a time.

Processes can be any of the following states:

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

- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur.
- **Terminated** - The process has completed.

Process Control Block, PCB

- Each process is represented in the operating system by a process control block (PCB) also called a task control block. PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process.

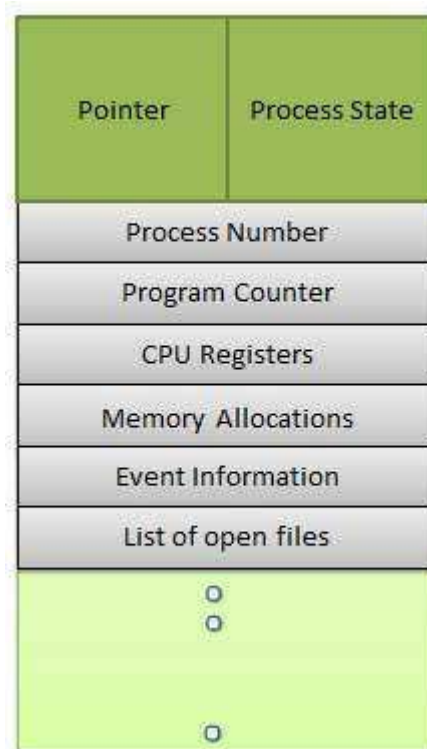- PCB contains many pieces of information associated with a specific process which is described below.

Figure: Process control block (PCB).

> **Pointer**

Pointer points to another process control block. Pointer is used for maintaining the scheduling list.

> **Process State**

Process state may be new, ready, running, waiting and so on.

> **Process Number**

A unique identification number for each process in the operating system.

> **Program Counter**

Program Counter indicates the address of the next instruction to be executed for this process.

> **CPU registers**

CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

> **Memory management information**

This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for deallocating the memory when the process terminates.

> **Accounting information**

This information includes the amount of CPU and real time used time limits, job or process numbers, account numbers etc.

- Process control block includes CPU scheduling, I/O resource management, file management information etc.

Two State Process Model:

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

1. **Running**

when new process is created by Operating System that process enters into the system as in the running state.

2. **Not Running**

Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

Schedulers

Schedulers are special system software's which handles 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 job scheduler. Long term scheduler determines which programs are admitted to the system for processing.
- Job scheduler 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 process changes the state from new to ready, then there is use of long term scheduler.

Short Term Scheduler

- It is also called CPU scheduler.
- Main objective is increasing 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 process among the processes that are ready to execute and allocates CPU to one of them.

➢ Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

- Medium term scheduling is part of the 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.
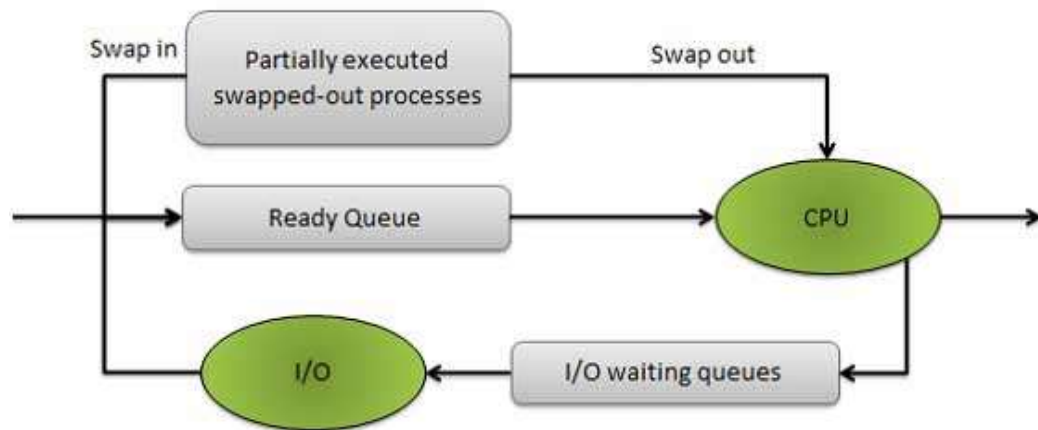
Figure: Addition of medium-term scheduling to the queuing diagram.

- Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion.

- In this condition, to remove the process from memory and make space for other process, 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 between 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

- When an interrupt occurs, the system needs to save the current of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **Context switch.**

- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- Context-switch time is pure overhead, because the system does no useful work while switching.

- Typical speeds are a few milliseconds. Context-switch times are highly dependent on hardware support.

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.

2. **Information Sharing**

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

3. **Convenience**

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

4. **Computation Speedup**

   Subtasks of a single task can be performed 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:
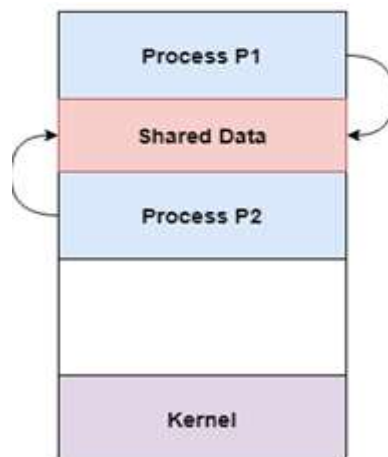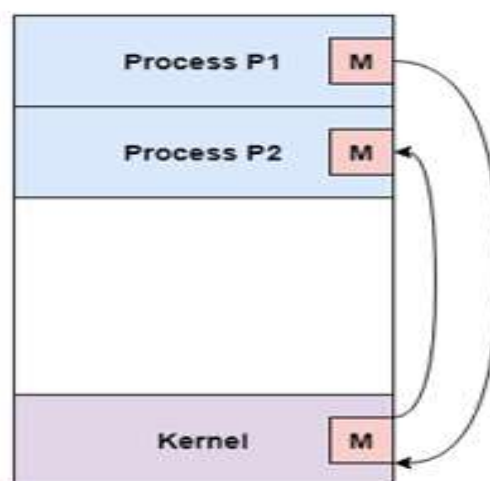

   In the above diagram, Process P1 and P2 can cooperate with each other using shared data such as memory, variables, files, databases etc.

2. **Cooperation by Communication**

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

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

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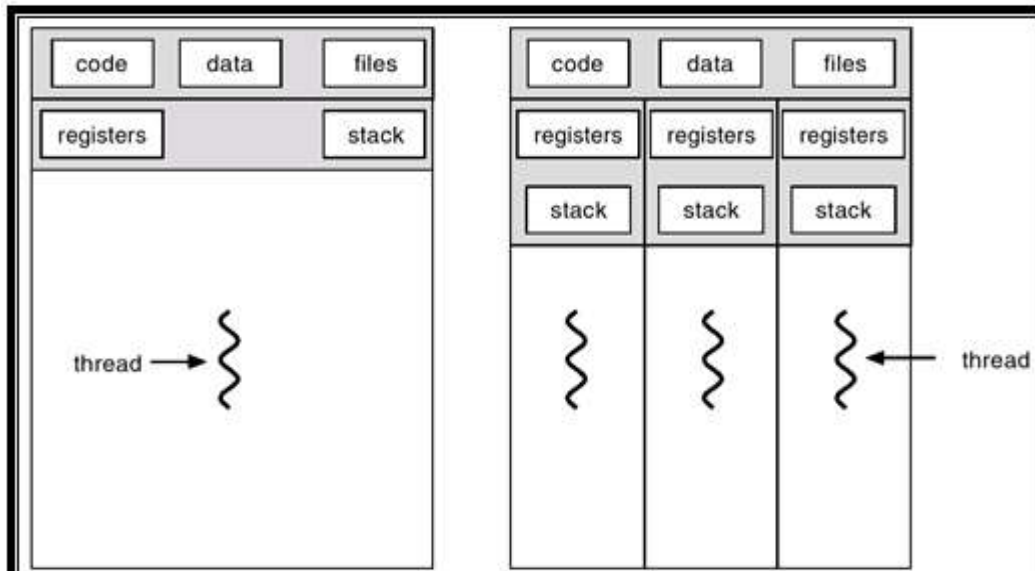
Threads

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.

- A thread is also called a light weight process (LWP). Threads provide a way to improve application performance through parallelism.

- 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.

- Following figure shows the working of the single and multithreaded processes

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, 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

- Thread minimizes context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency.

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, application manages 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 on texts. The application begins with a single thread and begins running in that thread.
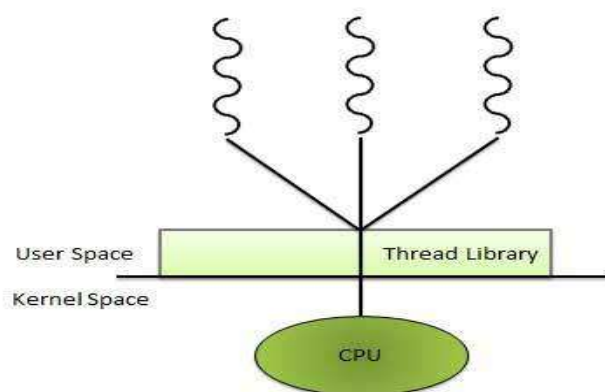
Advantages

- ➢ 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 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 multithreaded.

Disadvantages

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

Multithreading Models:

Some operating system provides 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

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular Machine.

Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.
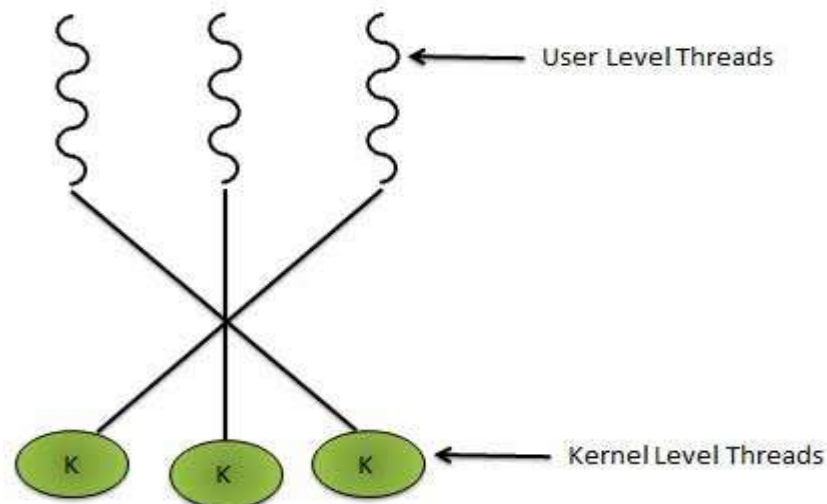
Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. 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 system does not support them then Kernel threads use the many to one relationship modes.
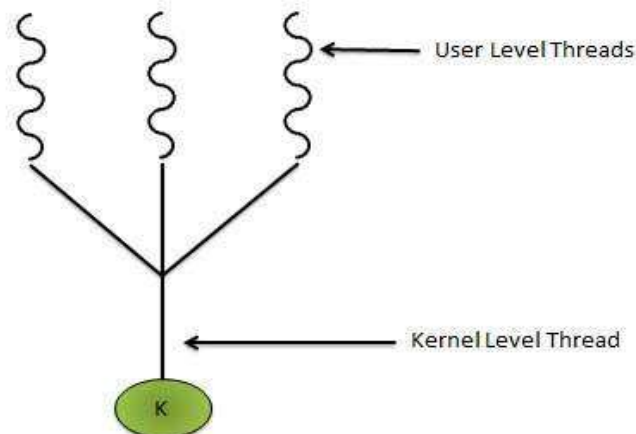
One to One Model

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 another thread to run when a thread makes a blocking system call. It support multiple thread 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 application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

CPU Scheduling

❖ Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler** (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

➢ Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.

➢ As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

CPU scheduling decisions take place under one of four conditions:

1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait ( ) system call.
2. When a process switches from the running state to the ready state, for example in response to an interrupt.
3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait ( ).
4. When a process terminates.

➢ For conditions 1 and 4 there is no choice - A new process must be selected.

➢ For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.

➢ If scheduling takes place only under conditions 1 and 4, the system is said to be non-pre-emptive, or cooperative. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be pre-emptive.

Scheduling Queues

➢ As processes enter the system, they are put into a job queue, which consists of all processes in the system.

➢ The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**

➢ The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device is called **device queue.**

➢ Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.

- The circles represent the resources that serve the queues.

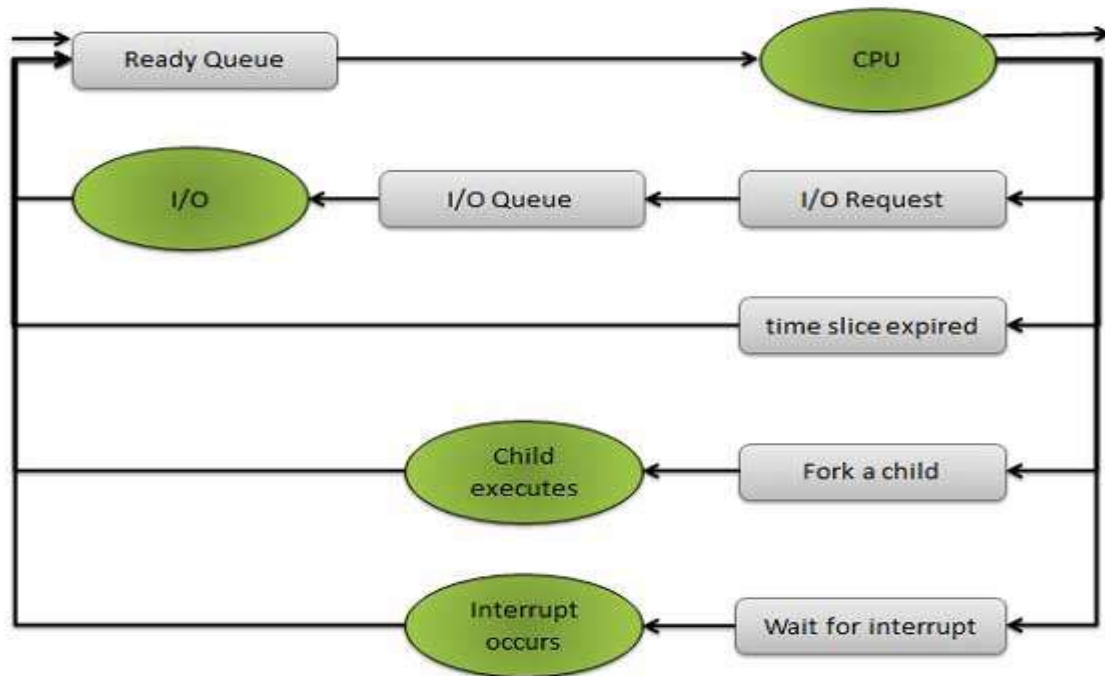- The arrows indicate the process flow in the system.

Figure: Queuing-diagram representation of process scheduling

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

1. The process could issue an I/O request and then be placed in an I/0 queue.
2. The process could create a new sub process and wait for the sub process's termination.
3. The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

   In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Dispatcher:

The dispatcher is the module that gives control of the CPU to the process selected by the scheduler. This function involves:

- ❖ Switching context.
- ❖ Switching to user mode.

❖ Jumping to the proper location in the newly loaded program.

The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency.**

Scheduling Criteria

There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

- ❖ **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- ❖ **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
- ❖ **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.) or The total amount of time spent by the process in system is called Turnaround time
- ❖ **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
- ❖ **Completion / Finishing time:** at what time the process going to be completed is called Completion / Finishing time
- ❖ **Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU.
- ❖ **Response time** – if the process are arrive at the system and it gets the <u>scheduled for the first time</u> is called response time.

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

  Optimization

  ❖ Maximum CPU Utilization
  ❖ Maximum Throughput
  ❖ Minimum Turnaround Time
  ❖ Minimum Waiting Time
  ❖ Minimum Response Time

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

There are different CPU-scheduling algorithms

1. First-come, first-served scheduling (FCFS) algorithm
2. Shortest Job First Scheduling (SJF) algorithm
3. Priority Scheduling algorithm
4. Round-Robin Scheduling algorithm
5. Multilevel Queue Scheduling algorithm

1.   First-come, first-served scheduling (FCFS) algorithm:

➤ This is one of the simplest scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first.

➤ FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office.

➤ A FCFS scheduling is **non preemptive** which usually results in poor performance and average waiting time is high.

➤ The FCFS scheduling algorithm is non pre-emptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

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

Process	Burst/Execution time (BT)
P1	24
P2	3
P3	3

➤ If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart,

Gantt chart

P1	P2	P3
0 24	27	30
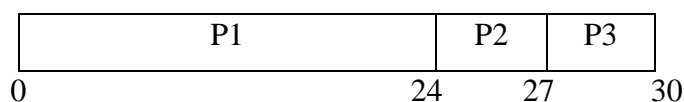
➢ **The average waiting time:**

The waiting time of process P1 is 0 milliseconds

The waiting time of process P2 is 24 milliseconds

The waiting time of process P3 is 27 milliseconds

Thus, the average waiting time is (0+ 24 + 27)/3 = 17 milliseconds.

➢ **The average turnaround time:**

The turnaround time (TAT) of process P1 is 24

The turnaround time (TAT) of process P2 is 27

The turnaround time (TAT) of process P3 is 30

Thus, average turnaround time (TAT) is (24+27+30)/3=27 milliseconds.

➢ **The completion Time(CT):**

The completion Time of process P1 is 24

The completion Time of process P2 is 27

The completion Time of process P3 is 30

❖ In non pre-emptive scheduling , Response Time =Waiting Time

Process	Burst time (BT)	TAT	Waiting time	CT
P1	24	24	0	24
P2	3	27	24	27
P3	3	30	27	30

➢ If the processes arrive in the order P2, P3 , P1,however, the results will be as shown in the following Gantt chart:

P2	P3	P1
0 3	6	30
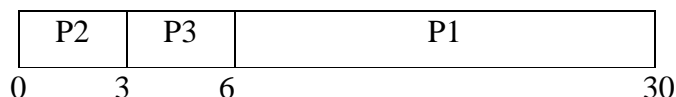
The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds

Example 2: with Arrival Time –Non pre-emptive

Process	Arrival Time(AT)	Burst (BT)
P1	0	4
P2	1	3
P3	2	1
P4	3	2
P5	4	5

Gantt chart

P1	P2	P3	P4	P5

0   4   7   8   10   15

Calculating   average waiting time and Turnaround time and completion time

➢ **The average waiting time:**

  The waiting time of process P1 is 0 (as per Gantt chart it is 0 but it arrives at time0 so 0-0=0)

  The waiting time of process P2 is 3 (as per Gantt chart it is 4 but it arrives at time 1 so 4-1=3)

  The waiting time of process P3 is   5 (as per Gantt chart it is 7 but it arrives at time 2 so 7-2=5)

  The waiting time of process P4 is 5 (as per Gantt chart it is 8 but it arrives at time 3 so 8-3=5)

  The waiting time of process P5 is   6 (as per Gantt chart it is 10 but it arrives at time 4 so 10-5=6)

   Thus, the average waiting time is (0+3+5+5+6)/5=14.2 milliseconds.

➢ **The average turnaround time:**

  The turnaround time (TAT) of process P1 is 4

  The turnaround time (TAT) of process P2 is 6

  The turnaround time (TAT) of process P3 is 6

  The turnaround time (TAT) of process P4 is 7

  The turnaround time (TAT) of process P5 is 11

   Thus, average turnaround time (TAT) is 4+6+6+7+11/5=6.8 milliseconds.

   Simple way to find TAT = CT- AT or TAT=WT+BT

➢ **The completion Time(CT):**

   The completion Time of process P1 is 0

   The completion Time of process P2 is 7
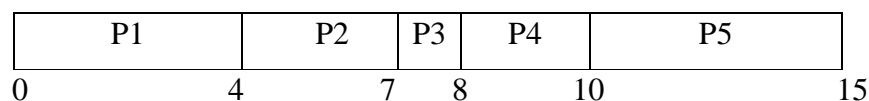
   The completion Time of process P3 is 8

The completion Time of process P4 is 10

The completion Time of process P5 is 15

Process Name	AT	BT	TAT	WT	CT
P1	0	4	4	0	4
P2	1	3	6	3	7
P3	2	1	6	5	8
P4	3	2	7	5	10
P5	4	5	11	6	15

2. Shortest Job First Scheduling (SJF) algorithm:

➢ A different approach to CPU scheduling is the Shortest-Job-First where the scheduling of a job or a process is done on the basis of its having shortest next CPU burst (execution time).

➢ When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

➢ If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

➢ SJF is optimal – gives minimum average waiting time for a given set of processes**.**

➢ The SJF algorithm can be either preemptive or no preemptive

1. Non pre- emptive – once CPU given to the process it cannot be preempted until completes its CPU burst. This scheme is known as the **shortest-next-CPU-burst algorithm**.

2. Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the **Shortest-Remaining-Time-First (SRTF).**

❖ The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

❖ Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. SJF scheduling is used frequently in long-term scheduling. For long-term scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.

SJF may be implemented in either non-preemptive or preemptive varieties. When a new process arrives at the ready queue with a shorter next CPU burst than what is left of the currently executing

process, then a preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is also called **shortest-remaining-time-first** scheduling.

Example 1: Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst/Execution time (BT)
P1	6
P2	8
P3	7
P4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

P4	P1	P3	P2

0   3   9   16   24 .
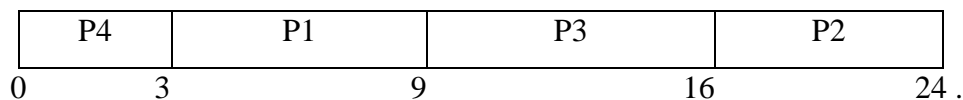
➢ **The average waiting time:**

  The waiting time of process P1 is 3 milliseconds

  The waiting time of process P2 is 16 milliseconds

  The waiting time of process P3 is 9 milliseconds

  The waiting time of process P4 is 0 milliseconds

 Thus, the average waiting time is (3 + 16 + 9 + 0) I 4 = 7 milliseconds

By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

➢ **The average turnaround time:**

  The turnaround time (TAT) of process P1 is 9

  The turnaround time (TAT) of process P2 is 24

  The turnaround time (TAT) of process P3 is 16

  The turnaround time (TAT) of process P4 is 3

 Thus, average turnaround time (TAT) is (9+24+16+3) / 4=13 milliseconds

➢ **The completion Time:**

 The completion Time of process P1 is 9

The completion Time of process P2 is 24

The completion Time of process P3 is 16

The completion Time of process P4 is 3

Process Name	BT	TAT	WT	CT
P1	6	9	3	9
P2	8	24	16	24
P3	7	16	9	16
P4	3	3	0	3

Example 2: Consider the following five processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	1	7
P2	2	5
P3	3	1
P4	4	2
P5	5	8

Using SJF scheduling, we would schedule these processes according to the following

Gantt chart:

	P1	P3	P4	P2	P5
0 1	8	9	11	16	24
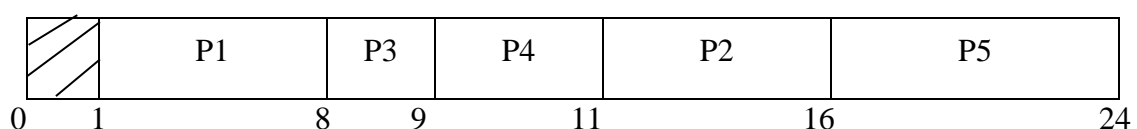
- Process P1 is started at time 1, since it is the only process in the queue.

- So after P1 scheduled find out the short burst time for remaining processes. So that process P3 is short burst time i.e. 1.so on….

➤ **The average waiting time:**

The waiting time of process P1 is 0 (as per Gantt chart it is 1 but it arrives at time 1 so 1-1=0)

The waiting time of process P2 is 9 (as per Gantt chart it is 11 but it arrives at time 2 so 11-2=9)

The waiting time of process P3 is 5 (as per Gantt chart it is 8 but it arrives at time 3 so 8-3=5)

The waiting time of process P4 is 5 (as per Gantt chart it is 9 but it arrives at time 4 so 9-4=5)

The waiting time of process P5 is 11 (as per Gantt chart it is 16 but it arrives at time 5 so 16-5=11)

Thus, <u>the average waiting time</u> is (3 + 16 + 9 + 0) / 4 = 6 milliseconds

➢ **The average turnaround time:**

The turnaround time (TAT) of process P1 is 7

The turnaround time (TAT) of process P2 is 14

The turnaround time (TAT) of process P3 is 6

The turnaround time (TAT) of process P4 is 7

The turnaround time (TAT) of process P5 is 19

Thus, average turnaround time (TAT) is (7+14+6+7+19) / 5=10.6 milliseconds

Simple way to find TAT = CT- AT or TAT=WT+BT

➢ **The completion Time:**

The completion Time of process P1 is 8

The completion Time of process P2 is 16

The completion Time of process P3 is 9

The completion Time of process P4 is 11

The completion Time of process P5 is 24

Process Name	AT	BT	TAT	WT	CT
P1	1	7	7	0	8
P2	2	5	14	9	16
P3	3	1	6	5	9
P4	4	2	7	5	11
P5	5	8	19	11	24

Example 3: Shortest-Remaining-Time-First: Consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:

P1	P2	P4	P1	P3
0   1   5   10   17   26
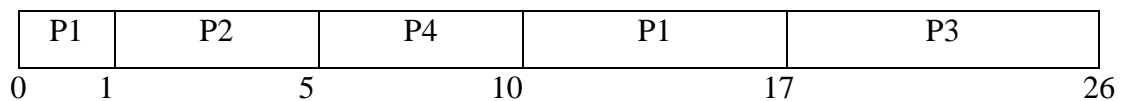
- Process P1 is started at time 0, since it is the only process in the queue.
- Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

➢ **The average waiting time:**

 The waiting time of process P1 is 9 (as per Gantt chart it is 0 but process could not complete so again process p1 started at time 10. First case p1 executing time is 1. So 10-1=9.and process arrive at 0 so 9-0=9)

 The waiting time of process P2 is 0 (as per Gantt chart it is 1 but it arrives at time 1 so 1-1=0)

  The waiting time of process P3 is 15 (as per Gantt chart it is 17 but it arrives at time 2 so 17-2=15)

 The waiting time of process P4 is 2 (as per Gantt chart it is 5 but it arrives at time 3 so 5-3=2)

    The average waiting time for this example is

   [(10- 1) + (1 - 1) + (17- 2) + (5-3)] / 4 = 26/4 = 6.5 milliseconds.

The average turnaround time:

            The turnaround time (TAT) of process P1 is 17
            The turnaround time (TAT) of process P2 is 4
            The turnaround time (TAT) of process P3 is 24
            The turnaround time (TAT) of process P4 is 7

   Thus, average turnaround time (TAT) is (17+4+24+7)/4=13 milliseconds.

➢ **The completion Time:**

The completion Time of process P1 is 17

The completion Time of process P2 is 5

The completion Time of process P3 is 26

The completion Time of process P4 is 10

- No preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Process Name	AT	BT	TAT	WT	CT
P1	0	8	17	9	17
P2	1	4	4	0	5
P3	2	9	24	15	26
P4	3	5	7	2	10

Another examples of SJF :

SJF (Non preemptive)

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Gantt chart:

P1	P3	P2	P4

0   7   8   12   16

Average waiting time = [0 +(8-2)+(7-4) +(12-5)] /4 =4

SJF (Preemptive)

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

P1	P2	P3	P2	P4	P1

0   2   4   5   7   11   16

Average waiting time = (9 + 1 + 0 +2)/4 =3

SJF (Preemptive)

Process	Arrival Time	Burst Time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

Calculate   average waiting time and TAT

Gantt chart

P1	P2	P3	P4	P3	P3	P6	P5	P2	P1
0	1	2	3	4	5	6	7	9	13 19

| Process Name	AT	BT	TAT	WT	CT
P1	0	7	19	12	19
P2	1	5	12	7	13
P3	2	3	4	1	6
P4	3	1	1	0	4
P5	4	2	5	3	9
P6	5	1	2	1	7

3. Priority Scheduling algorithm:

- The SJF algorithm is a special case of the general priority scheduling algorithm.

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

- Equal-priority processes are scheduled in FCFS order.

- Note that we discuss scheduling in terms of high priority and low priority.

- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority.

- Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. We assume that high numbers represent high priority.

Example 1: Without Arrival time- Consider the following set of processes, assumed to have <u>arrived at time 0</u> in the order P1, P2, - - -, Ps, with the length of the CPU burst given in milliseconds.

Process	Burst Tim	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Using priority scheduling, we would schedule these processes according to the Following

Gantt chart:

P4	P3	P1	P5	P2
0 1	3	13	18	19
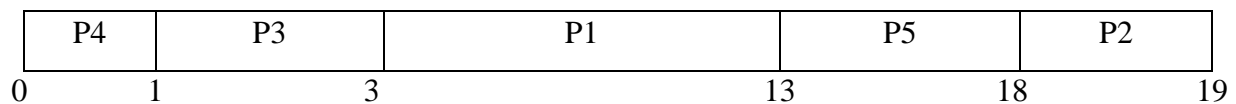
➢ **The average waiting time**

      The waiting time of process P1 is 3 milliseconds

      The waiting time of process P2 is 18 milliseconds

      The waiting time of process P3 is 1 milliseconds

      The waiting time of process P4 is 0 milliseconds

      The waiting time of process P5 is 13 milliseconds

The average waiting time is (3+18+1+0+13)/5= 7 milliseconds.

➢ **The average turnaround time:**

      The turnaround time (TAT) of process P1 is 13

      The turnaround time (TAT) of process P2 is 19

      The turnaround time (TAT) of process P3 is 3

      The turnaround time (TAT) of process P4 is 1

      The turnaround time (TAT) of process P5 is 18

     Thus, average turnaround time (TAT) is (16+1+18+19+6) / 5=12 milliseconds

     Simple way to find TAT = CT- AT or TAT=WT+BT

➢ **The completion Time:**

     The completion Time of process P1 is 13

     The completion Time of process P2 is 19

The completion Time of process P3 is 3

The completion Time of process P4 is 1

The completion Time of process P5 is 18

Process Name	BT	Priority	TAT	WT	CT
P1	10	3	13	3	13
P2	1	1	19	18	19
P3	2	4	3	1	3
P4	1	5	1	0	1
P5	5	2	18	13	18

❖ Priorities can be defined either internally or externally.

Internally: priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files.

Externally: priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use.

❖ Priority scheduling can be either preemptive or non preemptive.

➢ When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

➢ A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

➢ A non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A Major problem with priority scheduling algorithms:

• A Major problem with priority scheduling algorithms is indefinite blocking, or starvation.

• A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

• In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Solution:

A solution to the problem of indefinite blockage of low-priority processes is aging.

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Starvation:

Starvation is a resource management problem where a process does not get the resources it needs for a long time because the resources are being allocated to other processes.

(in Simple words, low priority processes may never execute)

Aging:

Aging is a technique to avoid starvation in a scheduling system. It works by adding an aging factor to the priority of each request. The aging factor must increase the requests priority as time passes and must ensure that a request will eventually be the highest priority request (after it has waited long enough)

(In simple words, as time progresses increase the priority of the process.)

Example 2: With Arrival Time – **Non Preemptive** Consider the following set of processes, with the length of the CPU burst given in milliseconds

Process	Arrival time	Burst Tim	Priority
P1	0	4	2
P2	1	2	4
P3	2	3	6
P4	3	5	10
P5	4	1	8
P6	5	4	12
P7	6	6	9

Using priority scheduling, we would schedule these processes according to the Following

Gantt chart:

P1	P4	P6	P7	P5	P3	P2

0   4   9   13   19   20   23   25
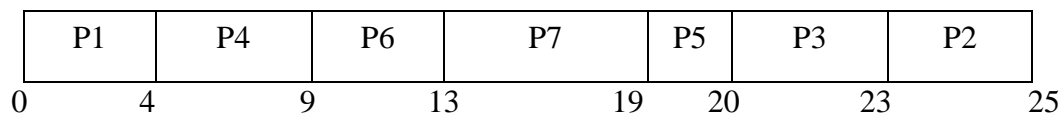
➢ **The average waiting time:**

The waiting time of process P1 is 0 (as per Gantt chart it is 0 and process arrive at o so 0-0=0)

The waiting time of process P2 is 22 (as per Gantt chart it is 23 but it arrives at time 1 so 23-1=22)

The waiting time of process P3 is 18 (as per Gantt chart it is 20 but it arrives at time 2 so 20-2=18)

The waiting time of process P4 is 1 (as per Gantt chart it is 4 but it arrives at time 3 so 4-3=1)

The waiting time of process P5 is 15 (as per Gantt chart it is 19 but it arrives at time 4 so 19-4=15)

The waiting time of process P6 is 4 (as per Gantt chart it is 9 but it arrives at time 5 so 9-5=4)

The waiting time of process P7 is 7 (as per Gantt chart it is 13 but it arrives at time 6 so 13-6=7)

The average waiting time for this example is

[(0-0) + (23- 1) + (20 - 2) + (4- 3) + (19-4) + (9-5) + (13-6)] / 7 = 45/7 = 9.5 milliseconds.

➢ **The average turnaround time:**

The turnaround time (TAT) of process P1 is 4

The turnaround time (TAT) of process P2 is 24

The turnaround time (TAT) of process P3 is 21

The turnaround time (TAT) of process P4 is 6

The turnaround time (TAT) of process P5 is 16

The turnaround time (TAT) of process P6 is 8

The turnaround time (TAT) of process P7 is 13

Thus, average turnaround time (TAT) is (4+24+21+6+16+8+13)/7=13.1 milliseconds.

➢ **The completion Time:**

The completion Time of process P1 is 4

The completion Time of process P2 is 25

The completion Time of process P3 is 23

The completion Time of process P4 is 9

The completion Time of process P5 is 20

The completion Time of process P6 is 13

The completion Time of process P7 is 19

Process Name	Arrival Time	BT	Priority	TAT	WT	CT
P1	0	4	2	4	0	4
P2	1	2	4	24	22	25
P3	2	3	6	21	18	23
P4	3	5	10	6	1	9
P5	4	1	8	16	15	20
P6	5	4	12	8	4	13
P7	6	6	9	13	7	19

Example 3: Pre emptive - Consider the following set of processes, with the length of the CPU burst given in milliseconds

Process	Arrival time	Burst Time	Priority
P1	0	4	2
P2	1	2	4
P3	2	3	6
P4	3	5	10
P5	4	1	8
P6	5	4	12
P7	6	6	9

Gantt chart:

P1	P2	P3	P4	P6	P4	P7	P5	P3	P2	P1

0   1   2   3   5   9   12   18 19   21 22   25

> **The average waiting time:**

The waiting time of process P1 is 21 (as per Gantt chart initially it is 0 but process could not complete so again process p1 started at time 22. First case P1 executing time is 1. So 22-1=21.and process arrive at 0 so 21-0=21)

The waiting time of process P2 is 19 (as per Gantt chart initially it is 1 but process could not complete so again process P2 started at time 21. First case P2 executing time is 1. So 21-1=20 .and process arrive at 1 so 21-1=19)

The waiting time of process P3 is 16 (as per Gantt chart initially it is 2 but process could not complete so again process P3 started at time 19. First case P3 executing time is 1. So 19-1=18 .and process arrive at 2 so 18-2=16).

The waiting time of process P4 is 4 (as as per Gantt chart initially it is 3 but process could not complete so again process P4 started at time 9. First case P4 executing time is 2. So 9-2=7 .and process arrive at 3 so 7-3=4)

The waiting time of process P5 is 14 (as per Gantt chart it is 18 but it arrives at time 4 so 18-4=14)

The waiting time of process P6 is 0 (as per Gantt chart it is 5 but it arrives at time 5 so 5-5=0)

The waiting time of process P7 is 6 (as per Gantt chart it is 12 but it arrives at time 6 so 12-6=6)

The average waiting time for this example is

(21+19+16+4+14+0+6)] / 7 = 45/7 = 11.4 milliseconds.

> ➤ **The average turnaround time:**

The turnaround time (TAT) of process P1 is 25

The turnaround time (TAT) of process P2 is 21

The turnaround time (TAT) of process P3 is 19

The turnaround time (TAT) of process P4 is 9

The turnaround time (TAT) of process P5 is 15

The turnaround time (TAT) of process P6 is 4

The turnaround time (TAT) of process P7 is 12

Thus, average turnaround time (TAT) is (25+21+19+9+15+4+12)/7=15.1 milliseconds.

> ➤ **The completion Time:**

The completion Time of process P1 is 25

The completion Time of process P2 is 22

The completion Time of process P3 is 21

The completion Time of process P4 is 12

The completion Time of process P5 is 19

The completion Time of process P6 is 9

The completion Time of process P7 is 18

Process Name	Arrival Time	BT	Priority	TAT	WT	CT
P1	0	4	2	25	21	25
P2	1	2	4	21	19	22
P3	2	3	6	19	16	21
P4	3	5	10	9	4	12
P5	4	1	8	15	14	19
P6	5	4	12	4	0	9
P7	6	6	9	12	6	18

4. Round-Robin Scheduling algorithm:

- The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems.

- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined.

- A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue o£ processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen-

- ➢ The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

- ➢ Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.

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

Process	Burst Time
P1	24
P2	3
P3	3

- ➢ If we use a time-slice/ time quantum of 4 units of time, then P1 gets the first 4 units of time.

- ➢ Since it requires another 20 units of time, it is pre-empted after the first time slice / time quantum and the CPU is given to the next process i.e. P2.

- ➢ Since P2 just needs 4 units of time, it terminates as time-slice/ time quantum expires. The CPU is then given to the next process

- ➢ P3. Once each process has received one time slice, the CPU is returned to P1 for an additional time-slice.

Thus the resulting round robin schedule is:

P1	P2	P3	P1	P1	P1	P1	P1
0 4	7	10	14	18	22	26	30

Let's calculate the average waiting time for the above schedule.

➢ **The average waiting time:**

The waiting time of process P1 is 6 ms (as per Gantt chart initially it is 0 but process could not complete so again process p1 started at time 10. First case P1 executing time is 4. So 10-4=6.)

The waiting time of process P2 is 4 ms.

The waiting time of process P3 is 7 ms

Thus, the average waiting time is 17/3 = 5.66 milliseconds.

➢ **The average turnaround time:**

The turnaround time (TAT) of process P1 is 6

The turnaround time (TAT) of process P2 is 7

The turnaround time (TAT) of process P3 is 7

Thus, average turnaround time (TAT) is (6+7+7) / 3=6.6 milliseconds

Simple way to find TAT = CT- AT or TAT=WT+BT

➢ **The completion Time:**

The completion Time of process P1 is 30

The completion Time of process P2 is 7

The completion Time of process P3 is 7

Process	Burst Time	WT	TAT	CT
P1	24	6	30	30
P2	3	4	7	7
P3	3	7	7	10

➢ The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing

In software, we need also to consider the effect of context switching on the performance of RR scheduling.

❖ **Assume, for example**, that we have only one process of 10 time units.

• If the quantum is 12 time units, the process finishes in. less than 1 time quantum, with no overhead.

- If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

Example 2: With Arrival Time : Consider the following set of processes with the length of the CPU burst given in milliseconds:

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

Here time quantum is =2. i.e the maximum allowable time when the process is scheduled

Queue

P1

Gantt chart

P1	

0   2

- Initially at time 0 only P1 occur at Ready queue so that P1 scheduled first. here time quantum is 2 it only executes at each time 2 sec only. it means it pre empted because the burst time of P1 is 4. (Remaining BT is 2 sec).

- But observe queue at time 2, P2 and P3 process are occur so that P1 is pre empted after P2 and P3 scheduled.(after 2sec already P2 and P3 arrived at queue so after P2 and p3 only P1 Preempted.)

Queue

P1 P2 P3 **P1**	

Gantt chart
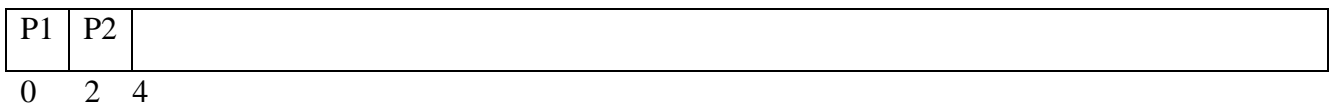
P1	P2	
0	2	4

- Now here Next process i.e P2 executed only 2 sec so total time is 4 now. At time 4 P4 and P5 process arrived at queue. P2 burst time is 5 it is pre empted after P4 and P5 scheduled and P2 reaming BT is 3.
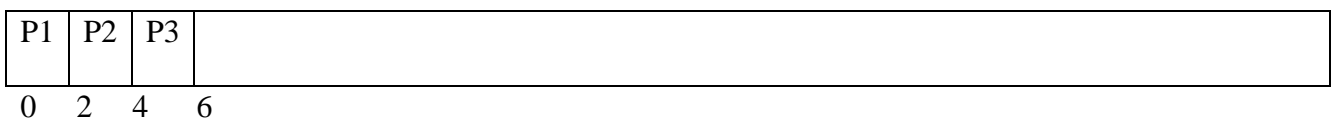
Queue

P1 P2 P3 **P1** P4 P5 **P2**	

Gantt chart
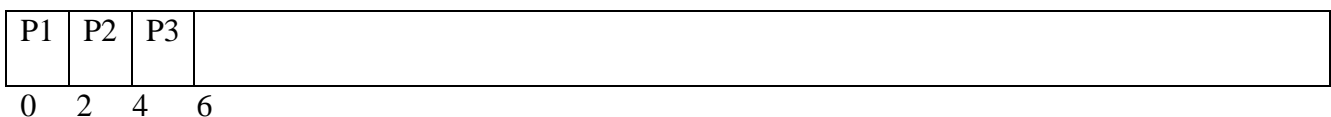
P1	P2	P3	
0	2	4	6

- Now here next process P3 scheduled only 2 sec.. at Time 6sec P6 process arrived. Here we need not to preempted because the BT of P3 is exactly 2sec

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6	

Gantt chart

P1	P2	P3	
0	2	4	6

Now observe at ready queue the next scheduled process is P1.the reaming BT of P1 is 2. So P1 is completely executed the total BT of P1.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6	

Gantt chart
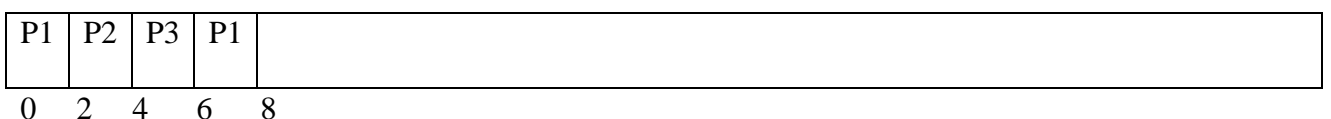
P1	P2	P3	P1	
0	2	4	6	8

Now observe at queue the next scheduled process is P4. Now scheduled P4 with 2sec . The total BT of P4 is only 1sec. it is completely executed within the time quantum.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6

Gantt chart

P1	P2	P3	P1	P4	
0	2	4	6	8	9

Now observe at queue the next scheduled process is P5. Now scheduled P4 with 2sec . The total BT of P5 is 6 sec. i.e. the reaming time is 4 sec. it is pre empted after remaining processes P2 and P6.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5**

Gantt chart

P1	P2	P3	P1	P4	P5	
0	2	4	6	8	9	11

- Now observe at queue the next scheduled process is P2 (it is preempted process). Now scheduled P2 with 2sec. The total BT of P2 is 5 sec.
- Still Process P2 remaining time is 1 sec. so it is again pre empted after remaining processes that are P6 and P5.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5 P2**

Gantt Chart

P1	P2	P3	P1	P4	P5	P2	
0	2	4	6	8	9	11	13

- Now observe at queue the next scheduled process is P6. Now scheduled P6 with 2sec. The total BT of P6 is 3 sec. the remaining time is 1sec. it is pre empted after remaining process that are P5 and P2.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5 P2 P6**

Gantt chart

P1	P2	P3	P1	P4	P5	P2	P6			

0   2   4   6   8   9   11   13   15

- Now observe at queue the next scheduled process is P5 ((it is preempted process). Now scheduled P5 with 2sec. The total BT of P5 is 6 sec.

- Still process P5 remaining time is 2 sec .so it is again pre empted after remaining processes that are P2 and P6 .

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5 P2 P6 P5**

Gantt chart

P1	P2	P3	P1	P4	P5	P2	P6	P5		

0   2   4   6   8   9   11   13   15   17

- Now observe at queue the next scheduled process is P2. The reaming time of P2 is only 1 sec. it is completely executed.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5 P2 P6 P5**

Gantt chart

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	

0   2   4   6   8   9   11   13   15   17   18

- Now observe at queue the next scheduled process is P6. The reaming time of P6 is only 1 sec. it is completely executed.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5 P2 P6 P5**

Gantt chart

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	P6

0   2   4   6   8   9   11   13   15   17   18   19

- Now observe at queue the next scheduled process is P5. The reaming time of P5 is only 2 sec. it is completely executed.

Queue

P1 P2 P3 **P1** P4 P5 **P2** P6 **P5 P2 P6 P5**
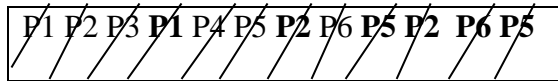
Gantt chart

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	P6	P5	
0	2	4	6	8	9	11	13	15	17	18	19	21

Let's calculate the average waiting time for the above schedule.

➢ **The average waiting time:**

The waiting time of process P1 is 4 ms (as per Gantt chart initially it is 0 but process could not complete so again process P1 started at time 6. First case P1 executing time is 2. So 6-2=4.)

The waiting time of process P2 is 12 ms. (as per Gantt chart initially process start 2 but process could not complete so again process P2 started at time 11 and 17. First case P2 executing time is 2.second case it is 2 . So 17-4=13.but arrival time of P2 is 1 so 13-1=12)

The waiting time of process P3 is 2 ms. (as per Gantt chart initially process start at 4. But arrival time process p3 is 2 . so 4-2=2)

The waiting time of process P4 is 5 ms. (as per Gantt chart initially process start at 8. But arrival time process p4 is 3 . so 8-3=5).

The waiting time of process P5 is 11 ms. (as per Gantt chart initially process start 9 but process could not complete so again process P5 started at time 15 and 19. First case P5 executing time is 2.second case it is 2 . So 19-4=15.but arrival time of P5 is 4 so 15-4=11)

The waiting time of process P6 is 10 ms. (as per Gantt chart initially process start 13 but process could not complete so again process P6 started at time 18. First case P6 executing time is 2.second. So 18-2=16.but arrival time of P6 is 6 so 16-6=10)

Thus, the average waiting time is (4+12+2+5+11+10)/6 = -------milliseconds.

➢ **The average turnaround time:**

The turnaround time (TAT) of process P1 is 8

The turnaround time (TAT) of process P2 is 17

The turnaround time (TAT) of process P3 is 4

The turnaround time (TAT) of process P4 is 6

The turnaround time (TAT) of process P5 is 17

The turnaround time (TAT) of process P6 is 13

Thus, average turnaround time (TAT) is (8+17+4+6+17+13) /6=------ milliseconds

Simple way to find TAT = CT- AT or TAT=WT+BT

> **The completion Time:**
   The completion Time of process P1 is 8
   The completion Time of process P2 is 18
   The completion Time of process P3 is 6
   The completion Time of process P4 is 9
   The completion Time of process P5 is 21
   The completion Time of process P6 is 19

Process Name	Arrival Time	BT	TAT	WT	CT
P1	0	4	18	4	8
P2	1	5	17	12	18
P3	2	2	4	2	6
P4	3	1	6	5	9
P5	4	6	17	11	21
P6	6	3	13	10	19

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** (interactive) processes and **background** (batch) processes.

- These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

- ❖ A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues (Figure).

- ❖ 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 an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

❖ In addition, there <u>must be scheduling among the queues</u>, which is commonly implemented as fixed-priority preemptive scheduling.

❖ For example, the foreground queue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
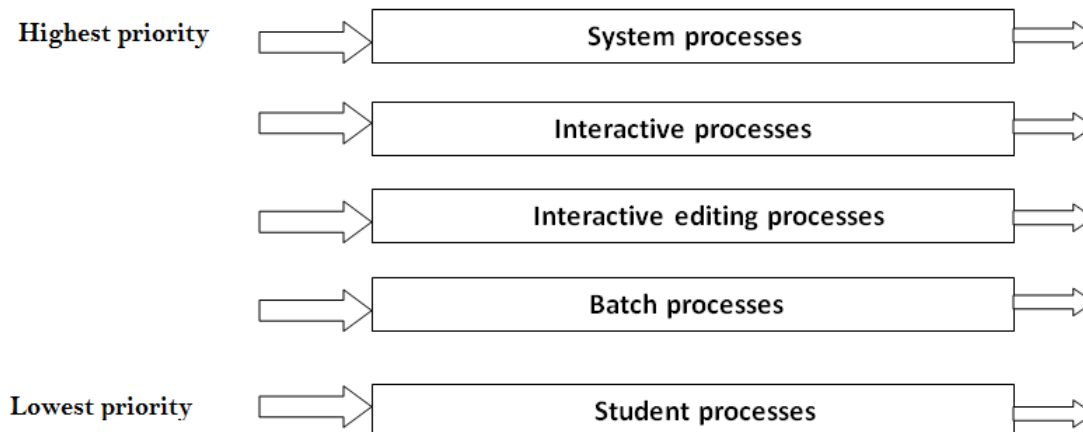3. Interactive editing processes
4. Batch processes
5. Student processes

Figure : Multilevel queue scheduling

• 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 would be pre-empted.

Multilevel Feedback Queue Scheduling:

❖ Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

➢ The **multilevel feedback queue scheduling** algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts.

➢ If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, 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.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure).

• The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.

• A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.
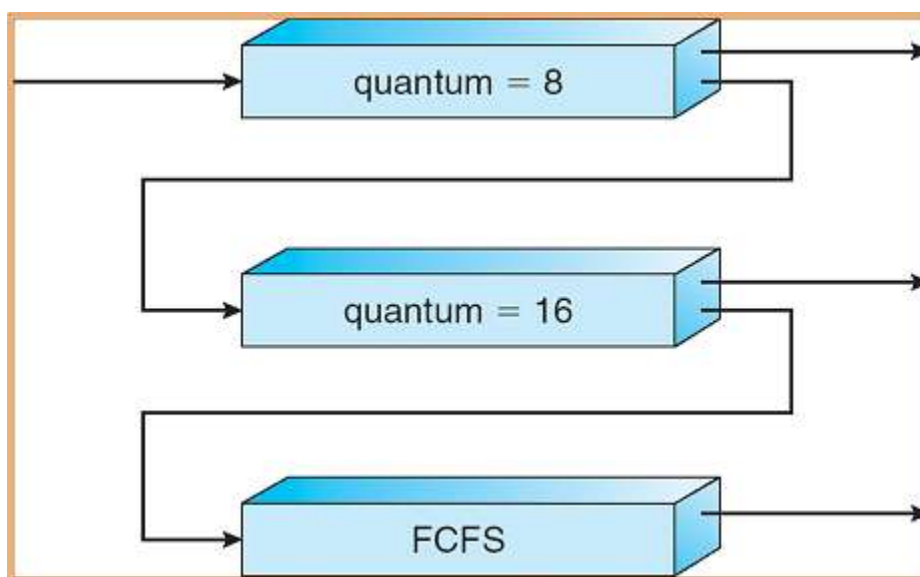

Figure: Multilevel feedback queues.

- A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1.

- If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is pre-empted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

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 is also the most complex algorithm since defining the best scheduler requires some means by which to select values for all the parameters.

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).

Approaches to Multiple-Processor Scheduling:

➢ One approach to multi-processor scheduling is **asymmetric multiprocessing,** in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple, as there is no need to share critical system data.

➢ Another approach is **symmetric multiprocessing, SMP,** where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor.

➢ Virtually all modern OSes support SMP, including XP, Win 2000, Solaris, Linux, and Mac OSX.

Processor Affinity:

➢ Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor; and as a result, successive memory accesses by the process are often satisfied in cache memory.

➢ Now consider what happens if the process migrates to another processor. The contents of 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 systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.

➢ This is known as **processor affinity**-that is, a process has an affinity for the processor on which it is currently running.

❖ Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor-but not guaranteeing that it will do so-we have a situation known as **soft affinity**.

❖ Here, it is possible for a process to migrate between processors. Some systems-such as Linux - also provide system calls that support **hard affinity**, thereby allowing a process to specify that it is not to migrate to other processors.
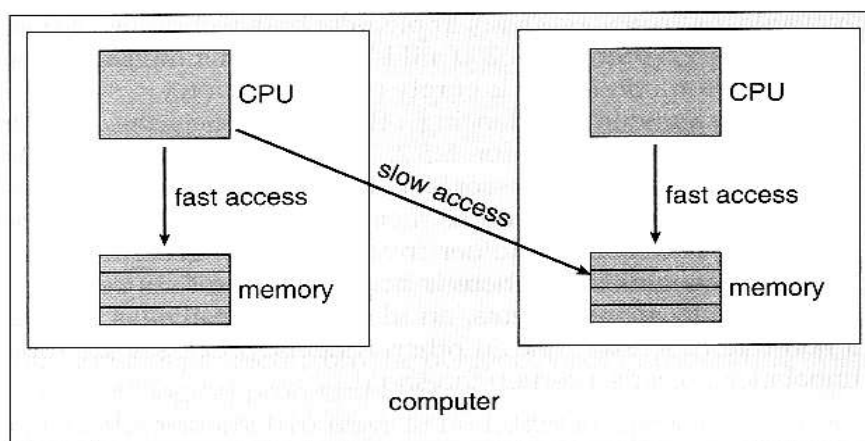

Figure 5.9   NUMA and CPU scheduling.

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.

➢ Systems using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor.

➤ Balancing can be achieved through either <u>push migration</u> or <u>pull migration:</u>

    ✓ Push migration involves a separate process that runs periodically, (e.g. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.

    ✓ Pull migration involves idle processors taking processes from the ready queues of other processors.

    ✓ Push and pull migration are not mutually exclusive.

Operations on Processes:

There are two major operations:

1. **Process Creation**

2. **Process Termination**

How Does Linux Identify Processes?

➤ Because Linux is a multi-user system, meaning different users can be running various programs on the system, each running instance of a program must be identified uniquely by the kernel.

➤ And a program is identified by its process ID (PID) as well as it's parent processes ID (PPID).

The Init Process

❖ **Init** process is the mother (parent) of all processes on the system, it's the first program that is executed when the <u>Linux system boots up</u>; it manages all other processes on the system. It is started by the kernel itself, so in principle it does not have a parent process.

❖ The init process always has process ID of **1**. It functions as an adoptive parent for all orphaned processes.

> ➤ You can use the pidof command to find the ID of a process:

```
[root@tecmint ~]# pidof systemd
1
[root@tecmint ~]# pidof top
2160
[root@tecmint ~]# pidof httpd
2103 2102 2101 2100 2099 1076
[root@tecmint ~]#
```

> ➤ To find the process ID and parent process ID of the current shell, run:

```
[root@tecmint ~]# echo $$
2109
[root@tecmint ~]# echo $PPID
2106
[root@tecmint ~]#
```

1. Process creation:

The fork ( ) Function:

❖ The fork ( ) function is used to create a new process by duplicating the existing process from which it is called.

❖ The existing process from which this function is called becomes the parent process and the newly created process becomes the child process. As already stated that child is a duplicate copy of the parent but there are some exceptions to it.

- The child has a unique PID like any other process running in the operating system.
- The child has a parent process ID which is same as the PID of the process that created it.
- Resource utilization and CPU time counters are reset to zero in child process.
- Set of pending signals in child is empty.
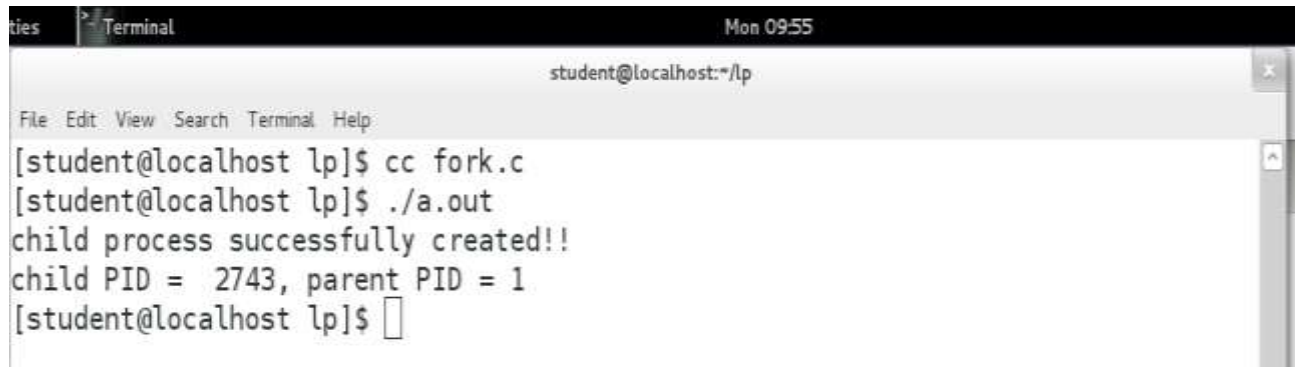- Child does not inherit any timers from its parent

Return Type:

➢ *Negative Value*: creation of a child process was unsuccessful.

➢ *Zero*: Returned to the newly created child process.

➢ *Positive value*: Returned to parent or caller. The value contains process ID of newly created child process.

Example:

```c
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
int main( )
{
  pid_t child_pid;
  child_pid = fork ( );                          // Create a new child process;
  if (child_pid >= 0)
  {
     if (child_pid == 0)
     {
        printf ("child process successfully created!!\n");
 printf ("child PID =  %d, parent PID = %d\n", getpid( ), getppid( ) );
        exit(0);
     }
  }
  else
  {
     perror("fork");
     exit(0);
  } }
```

Output:


```
ties   Terminal                              Mon 09:55

                          student@localhost:~/lp                          x

File Edit View Search Terminal Help
[student@localhost lp]$ cc fork.c
[student@localhost lp]$ ./a.out
child process successfully created!!
child PID =  2743, parent PID = 1
[student@localhost lp]$ 
```

2. Process termination:

- The exit ( ) system call terminates the process normally.

- Status: Status value returned to the parent process. Generally, a status value of 0 or EXIT_SUCCESS indicates success, and any other value or the constant EXIT_FAILURE is used to indicate an error.

Example:

```c
/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  FILE * fd;
  fd = fopen ("myfile.txt", "r");
  if (fd == NULL)
  {
    printf ("Error opening file");
    exit (1);
  }
  else
  {
    /* file operations here */
  }
  return 0;
}
```

❖ When exit( ) is called, any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal.

3. exec( ) system call - Replacing a process image:

"The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program"

❖ More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.

❖ The user data segment which executes the exec( ) system call is replaced with the data file whose name is provided in the argument while calling exec( ).

❖ The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec.

❖ If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process.

❖ PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

➢ **exec system** call is a collection of functions as follows:
  1. execl
  2. execv.... etc.

Synopsis:

int execl(const char *path*, const char *arg0*, ..., const char *argn*, (char *)0);

int execv(const char *path*, char *const *argv*[]);

Description:

➢ The **exec**( ) family of functions replaces the current process image with a new process image.

➢ The initial argument for these functions is the name of a file that is to be executed.

➢ The *const char *arg* and subsequent ellipses in the **execl**( ) function can be thought of as *arg0*, *arg1*, ..., *argn*. It describes a list of one or more pointers to null- terminated strings that represent the argument list available to the executed program.

➢ The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer.

➢ The **execv** function provides an array of pointers to null-terminated string that represent the argument list available to the new program.

> ➤ The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

Return Value:

The **exec** ( ) functions return only if an error has occurred. The return value is -1, and *errno* is set to indicate the error.

Example:

Consider the following example in which we have used exec system call in C programming in Linux.

We have two c files here example.c and hello.c:

example.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("PID of example.c = %d\n", getpid());
   char *args[] = {"Hello", "C", "Programming", NULL};
   execv("./hello", args);
   printf("Back to example.c");
   return 0;
}
```
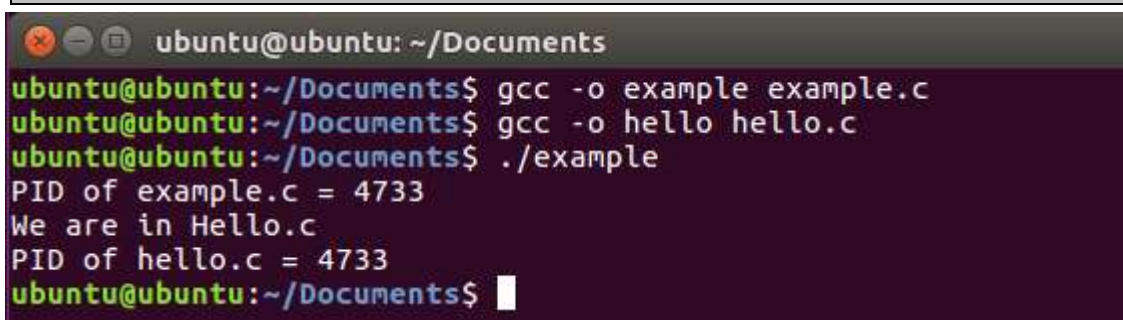
hello.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("We are in Hello.c\n");
   printf("PID of hello.c = %d\n", getpid());
   return 0;
}
```

OUTPUT:

```
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
```



```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```

➢ In the above example we have an example.c file and hello.c file. In the example .c file first of all we have printed the ID of the current process (file example.c is running in current process). Then in the next line we have created an array of character pointers. The last element of this array should be NULL as the terminating point.

➢ Then we have used the function execv( ) which takes the file name and the character pointer array as its argument. It should be noted here that we have used ./ with the name of file, it specifies the path of the file. As the file is in the folder where example.c resides so there is no need to specify the full path.

➢ When execv( ) function is called, our process image will be replaced now the file example.c is not in the process but the file hello.c is in the process. It can be seen that the process ID is same whether hello.c is process image or example.c is process image because process is same and process image is only replaced.

➢ Then we have another thing to note here which is the printf( ) statement after execv( ) is not executed. This is because control is never returned back to old process image once new process image replaces it. The control only comes back to calling function when replacing process image is unsuccessful. (The return value is -1 in this case).

Difference between fork( ) and exec( ) system calls:

➢ The fork( ) system call is used to create an exact copy of a running process and the created copy is the child process and the running process is the parent process. Whereas, exec( ) system call is used to replace a process image with a new process image. Hence there is no concept of parent and child processes in exec( ) system call.

➢ In fork( ) system call the parent and child processes are executed at the same time. But in exec( ) system call, if the replacement of process image is successful, the control does not return to where the exec function was called rather it will execute the new process. The control will only be transferred back if there is any error.

Example: Combining fork( ) and exec( ) system calls

Consider the following example in which we have used both fork( ) and exec( ) system calls in the same program:

example.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
   printf("PID of example.c = %d\n", getpid());
   pid_t p;
```

```
   p = fork();
   if(p = = -1)
   {
      printf("There is an error while calling fork()");
   }
   if(p = = 0)
   {
   printf("We are in the child process\n");
   printf("Calling hello.c from child process\n");
   char *args[] = {"Hello", "C", "Programming", NULL};
   execv("./hello", args);
   }
   else
   {
      printf("We are in the parent process");
   }
   return 0;
}
```

hello.c:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("We are in Hello.c\n");
   printf("PID of hello.c = %d\n", getpid());
   return 0;
}
```
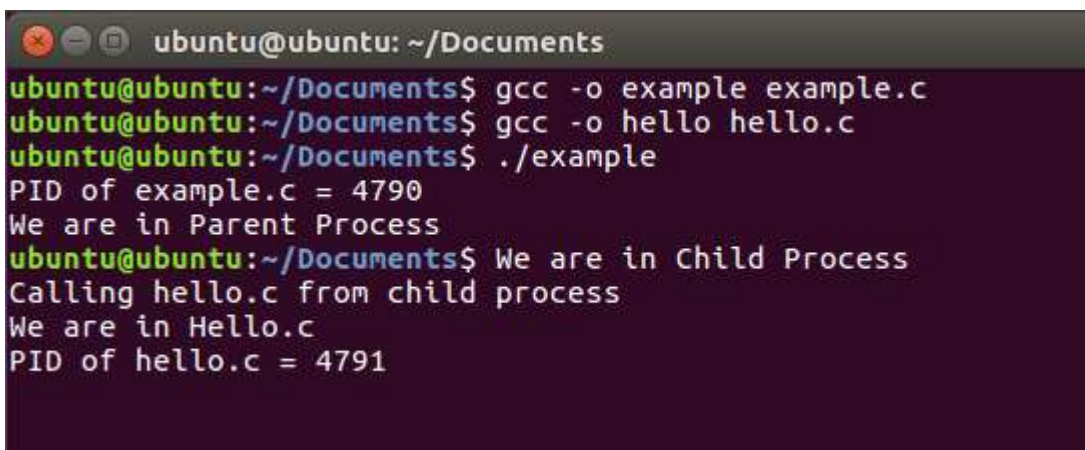
OUTPUT:

```
PID of example.c = 4790
We are in Parent Process
We are in Child Process
Calling hello.c from child process
We are in hello.c
PID of hello.c = 4791
```

❖ In this example we have used fork( ) system call. When the child process is created 0 will be assigned to p and then we will move to the child process. Now the block of statements with if(p = = 0) will be executed. A message is displayed and we have used execv ( ) system call and the current child process image which is example.c will be replaces with hello.c. Before execv( ) call child and parent processes were same.

❖ It can be seen that the PID of example.c and hello.c is different now. This is because example.c is the parent process image and hello.c is the child process image.

Waiting for a process:

wait( ) and waitpid( )

wait, waitpid - wait for process to change state

Synopsis:

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int **status***);**

pid_t waitpid(pid_t *pid***, int ****status***, int** *options***);**

Description:

❖ All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

A state change is considered to be:

- ✓ the child terminated;
- ✓ the child was stopped by a signal;
- ✓ or the child was resumed by a signal.

❖ In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state.

❖ The **wait**( ) system call suspends execution of the current process until one of its children terminates. The call *wait(&status)* is equivalent to: waitpid(-1, &status, 0);

❖ The **waitpid**( ) system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**( ) waits only for terminated children, but this behaviour is modifiable via the *options* argument, as described below.

The value of pid can be:

Tag	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of pid.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of pid.

Return Value:

➢ **wait**( ): on success, returns the process ID of the terminated child; on error, -1 is returned.
➢ **waitpid**( ): on success, returns the process ID of the child whose state has changed; on error, -1 is returned; if **WNOHANG** was specified and no child(ren) specified by pid has yet changed state, then 0 is returned.

Errors:

Tag	Description
ECHILD	(for **wait**()) The calling process does not have any unwaited-for children.
ECHILD	(for **waitpid**( ) or **waitid**( )) The process specified by pid (**waitpid**( )) or idtype and id (**waitid**( )) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for SIGCHLD is set to SIG_IGN. See also the LINUX NOTES section about threads.)
EINTR	**WNOHANG** was not set and an unblocked signal or a **SIGCHLD** was caught.
EINVAL	The options argument was invalid.

CPU Scheduling - Practice Exercises

1. **A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given *n* processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of *n*.**

   Answer:

   n! (*n* factorial $= n \times n - 1 \times n - 2 \times ... \times 2 \times 1$).

2. **Explain the difference between preemptive and nonpreemptive scheduling.**

   Answer:

   Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

3. **Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made**.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

   a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

   b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

   c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P1 and P2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.

   Answer:
   a. 10.53
   b. 9.53
   c. 6.86

Remember that turnaround time is finishing time minus arrival time, so you have to subtract the arrival times to compute the turnaround times. FCFS is 11 if you forget to subtract arrival time.

4. **What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?**

   Answer:

   Processes that need more frequent servicing, for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, and thus making more efficient use of the computer.

5. Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to moveprocesses between queues, and so on.

   These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum).What (if any) relation holds between the following pairs of algorithm sets?

   a. Priority and SJF

   b. Multilevel feedback queues and FCFS

   c. Priority and FCFS

   d. RR and SJF

   Answer:

   a. The shortest job has the highest priority.

   b. The lowest level of MLFQ is FCFS.

   c. FCFS gives the highest priority to the job having been in existence the longest.

   d. None.

6. Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

   Answer:

   It will favor the I/O-bound programs because of the relatively short CPU burst request by them; however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.

7. **Discuss how the following pairs of scheduling criteria conflict in certain settings.**

a. CPU utilization and response time

b. Average turnaround time and maximum waiting time

c. I/O device utilization and CPU utilization

8. **Consider the following set of processes, with the length of the CPU burst given in milliseconds:**

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b) What is the turnaround time of each process for each of the scheduling algorithms in part a?

c) What is the waiting time of each process for each of these scheduling algorithms?

d) Which of the algorithms results in the minimum average waiting time (over all processes)?

9. The following processes are being scheduled using a preemptive, roundrobin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an *idle task* (which consumes no CPU resources and is identified as *Pidle*). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

Thread	Priority	Burst	Arrival
P1	40	20	0
P2	30	25	25
P3	30	25	30
P4	35	15	60
P5	5	10	100
P6	10	10	105

a) Show the scheduling order of the processes using a Gantt chart.

b) What is the turnaround time for each process?

c) What is the waiting time for each process?

d) What is the CPU utilization rate?