CPU Scheduling Algorithms:

FCFS Scheduling Algorithm: **First come first serve** (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

Advantages of FCFS

* Simple
* Easy
* First come, First serve

Disadvantages of FCFS

1. The scheduling method is non-pre-emptive, the process will run to the completion.
2. Due to the non-pre-emptive nature of the algorithm, the problem of starvation may occur.
3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

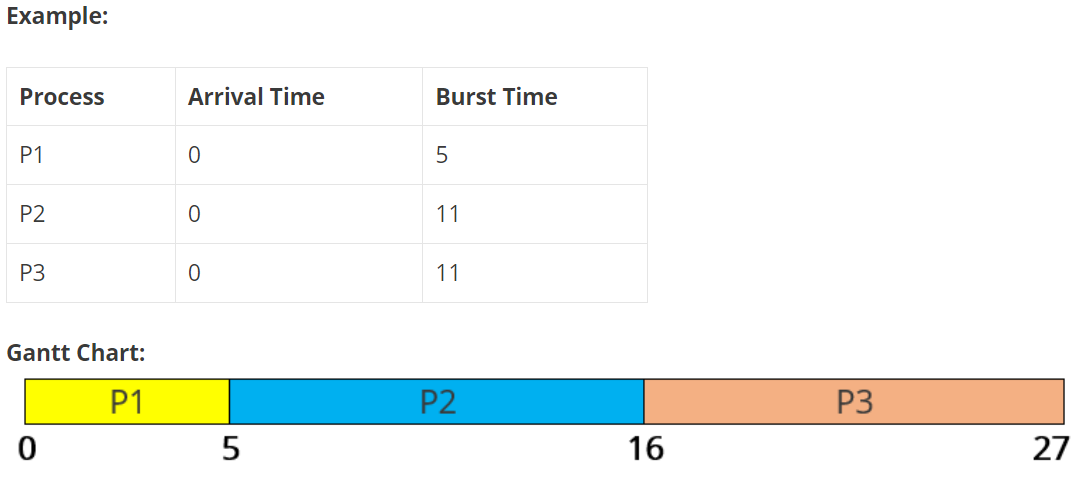
Example

Let's take an example of The FCFS scheduling algorithm. In the Following schedule, there are 5 processes with process ID **P0, P1, P2, P3 and P4**. P0 arrives at time 0, P1 at time 1, P2 at time 2, P3 arrives at time 3 and Process P4 arrives at time 4 in the ready queue. The processes and their respective Arrival and Burst time are given in the following table.

The Turnaround time and the waiting time are calculated by using the following formula.

1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turnaround time - Burst Time

The average waiting Time is determined by summing the respective waiting time of all the processes and divided the sum by the total number of processes.



**Waiting Time:** Time Difference between turnaround time and burst time.

**Waiting Time = Turnaround Time – Burst Time**

P1 waiting time: 0  
P2 waiting time: 5  
P3 waiting time: 16

**Average Waiting Time** = (0 + 5 + 16)/3 = 21/3 = 7

**Turnaround Time:** Difference between completion time and arrival time.

**Turnaround Time = Completion Time – Arrival Time**

P1 turnaround time: 5-0 = 5  
P2 turnaround time: 16-0 = 16  
P3 turnaround time: 27-0 = 27

**Average Turnaround Time** = (5+16+27)/3 = 16

**Program/Source Code**

Here is the source code of the C program for the FCFS Scheduling. The C program is successfully compiled and run on a Linux system. The program output is also shown below.

Program 1:

//Program for FCFS

#include<stdio.h>

int main()

{

int i,n,bt[10],at[10],tat[10],wt[10],complt=0;

float avgtat=0,avgwt=0,totaltat=0,totalwt=0;

printf("enter the no. of processes:\t");

scanf("%d",&n);

printf("enter burst time");

for(i=0;i<n;i++)

{printf("Burst time of process-%d:\t",i+1);

scanf("%d", &bt[i]);

printf("\n");

}

//set arrival time to zero for all process

for(i=0;i<n;i++)

at[i]=0;

//caalculate TAT

for(i=0;i<n;i++)

{

complt=complt+bt[i];

tat[i]=complt-at[i];

totaltat=totaltat+tat[i];

}

avgtat=(float)totaltat/n;

// calculate wait time

for(i=0;i<n;i++)

{

wt[i]=tat[i]-bt[i];

totalwt=totalwt+wt[i];

}

avgwt=(float)totalwt/n;

printf("ProcessNo Burst\_time Wait\_time Turnaround\_time\n");

for(i=0;i<n;i++)

{

printf("%d\t\t\t",i+1);

printf("%d\t\t\t",bt[i]);

printf("%d\t\t\t",wt[i]);

printf("%d\t\t\t",tat[i]);

printf("\n");

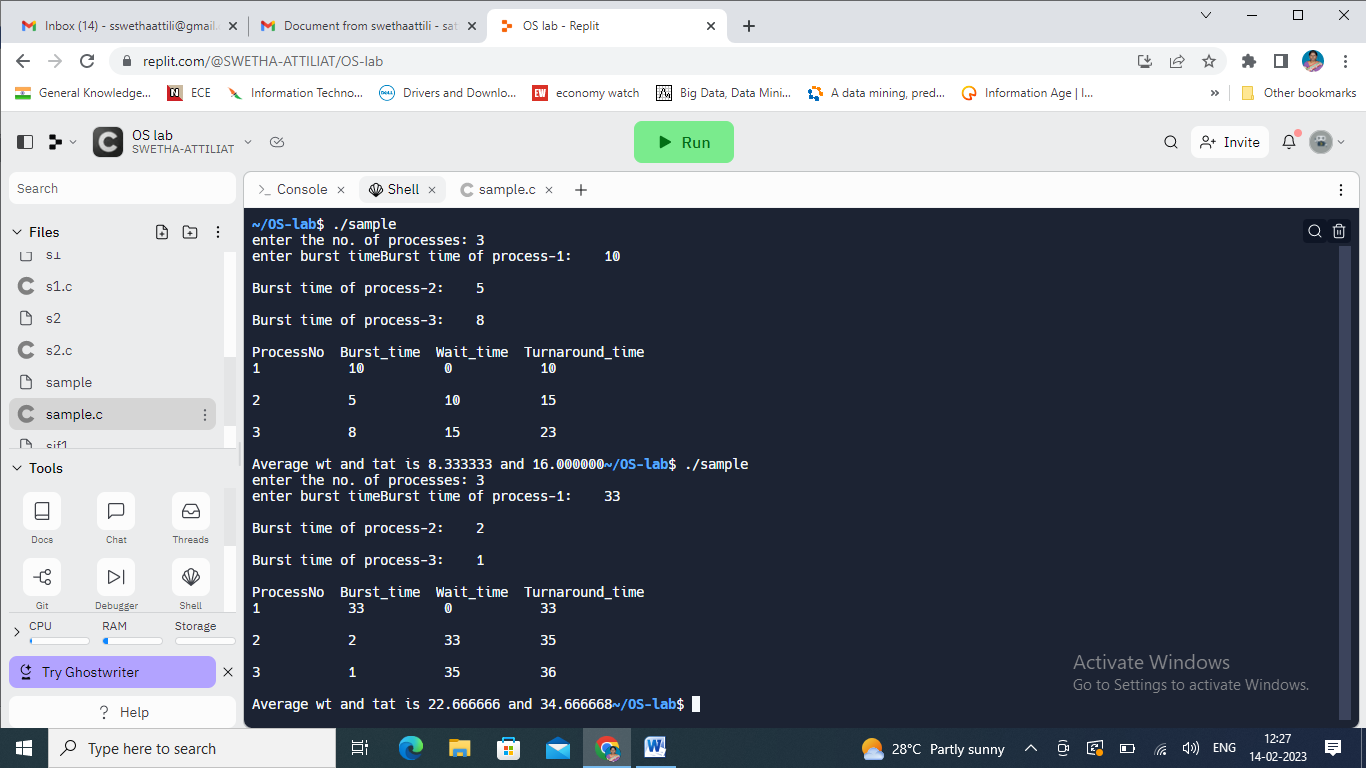
printf("\n");

}

printf("Average wt and tat is %f and %f",avgwt,avgtat);

}

Output:



**SJF Scheduling Algorithm:** The CPU scheduling algorithm Shortest Job First (**SJF**), allocates the CPU to the processes according to the process with smallest execution time.

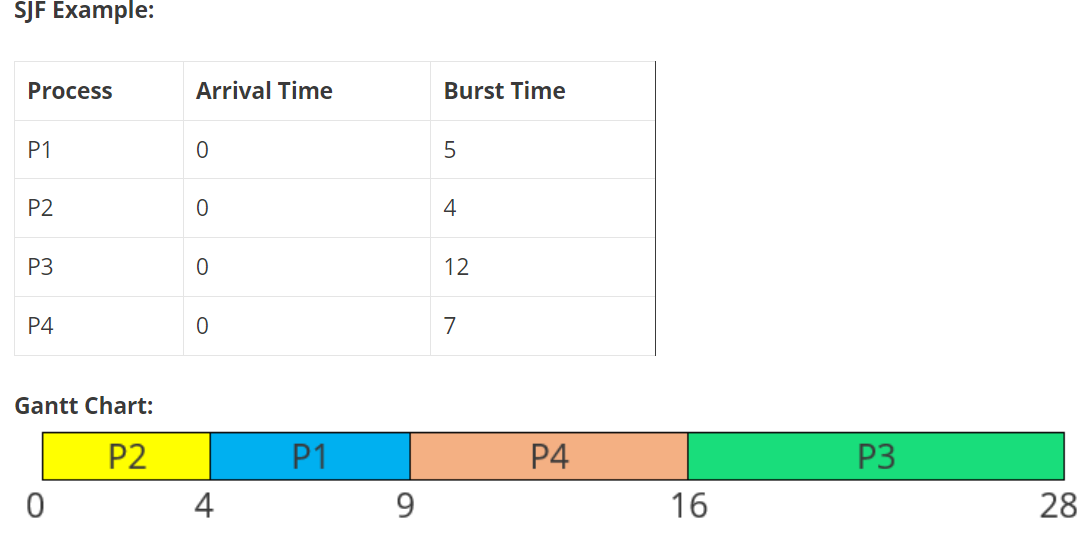
**SJF** uses both pre-emptive and non-pre-emptive scheduling. The pre-emptive version of SJF is called **SRTF** (Shortest Remaining Time First). Here we will discuss about SJF i.e., the **non-pre-emptive scheduling**.

**Advantages of SJF:**

* It has the minimum waiting time among all the scheduling algorithms.
* A process having larger burst time may get into starvation but the problem can be solved using concept of Ageing.
* It is a greedy algorithm and provides optimal scheduling.

**Procedure:**

1. Enter number of processes.  
2. Enter the **burst time** of all the processes.  
3. Sort all the processes according to their **burst time**.  
4. Find waiting time, **WT** of all the processes.  
5. For the smallest process, **WT = 0**.  
6. For all the next processes **i**, find waiting time by adding burst time of all the previously completed process.  
7. Calculate **Turnaround time = WT + BT** for all the processes.  
8. Calculate **average waiting time = total waiting time / no. of processes**.  
9. Calculate **average turnaround time= total turnaround time / no. of processes**.



**Waiting Time:** Time Difference between turnaround time and burst time.

**Waiting Time = Turnaround Time – Burst Time**

P1 waiting time: 4  
P2 waiting time: 0  
P3 waiting time: 16  
P4 waiting time: 9

**Average Waiting Time** = (4 + 0 + 16 + 9)/4 = 29/4 = 7.25

**Turnaround Time:** Difference between completion time and arrival time.

**Turnaround Time = Completion Time – Arrival Time**

P1 turnaround time: 9-0 = 9  
P2 turnaround time: 4-0 = 4  
P3 turnaround time: 28-0 = 28  
P4 turnaround time: 16-0 = 16

**Average Turnaround Time** = (9 + 4 + 28 + 16)/4 = 14.25

**OUTPUT:** Enter number of process: 4

Enter Burst Time:

p1:5

p2:4

p3:12

p4:7

Process Burst Time Waiting Time Turnaround Time

p2 4 0 4

p1 5 4 9

p4 7 9 16

p3 12 16 28

Average Waiting Time=7.250000

Average Turnaround Time=14.250000

**Time Complexity: O(n2)**  
The above program for implementing SJF Scheduling has a time complexity of O(n2), as the for loop runs for n^2 times for calculating waiting time of each process.

**Space Complexity: O(n)**  
In the SJF Scheduling program, space complexity is O(n) as arrays of size n have been initialized to store the values in it.

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("\nEnter Burst Time:\n");

for(i=0;i<n;i++)

{

printf("p%d:",i+1);

scanf("%d",&bt[i]);

p[i]=i+1; //creating a process id array

}

//sorting of burst times

for(i=0;i<n;i++)

{

pos=i;

for(j=i+1;j<n;j++)

{

if(bt[j]<bt[pos])

pos=j;

}

//arranging burst times

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

//arranging proceess ids

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

//finding the waiting time of all the processes

for(i=1;i<n;i++)

{

wt[i]=0;

for(j=0;j<i;j++)

//individual WT by adding BT of all previous completed processes

wt[i]+=bt[j];

//total waiting time

total+=wt[i];

}

//average waiting time

avg\_wt=(float)total/n;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

for(i=0;i<n;i++)

{

//turnaround time of individual processes

tat[i]=bt[i]+wt[i];

//total turnaround time

totalT+=tat[i];

printf("\np%d\t\t\t %d\t\t\t %d\t\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

//average turnaround time

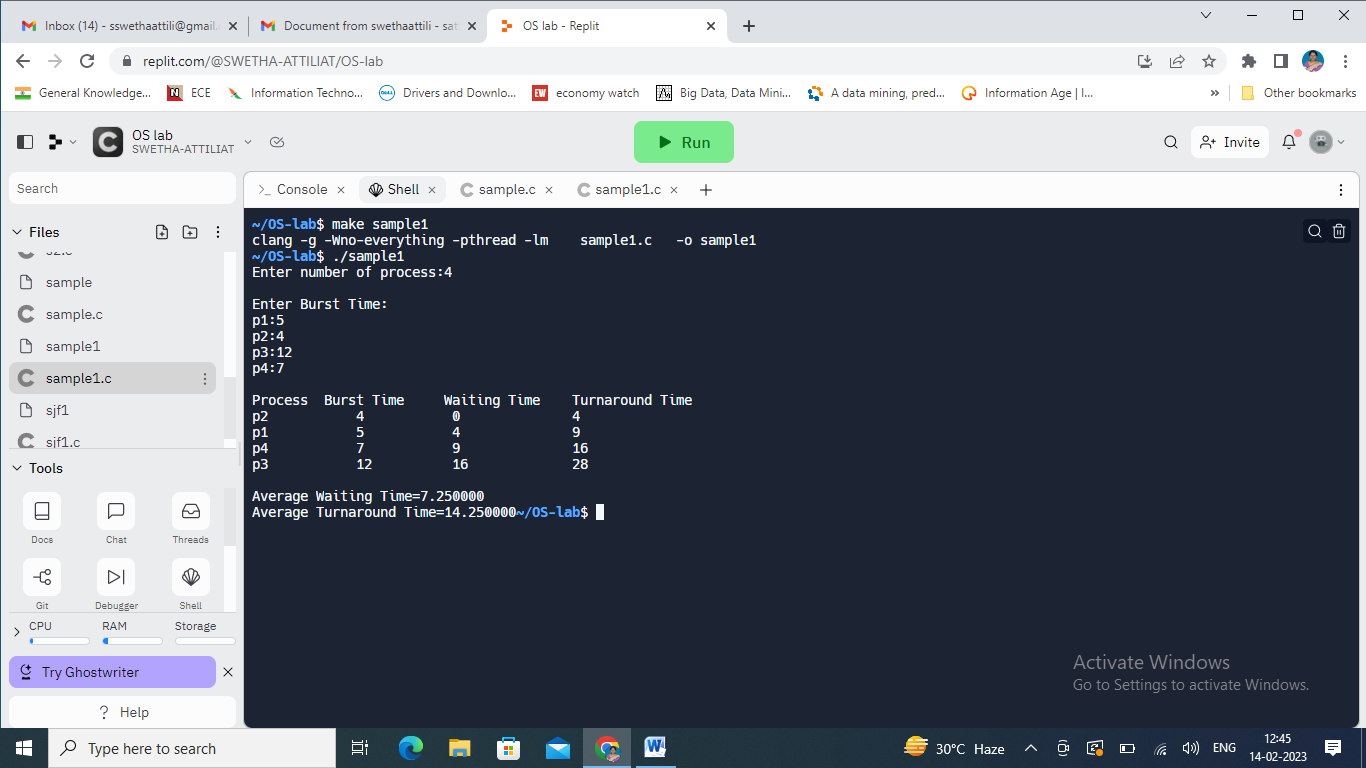
avg\_tat=(float)totalT/n;

printf("\n\nAverage Waiting Time=%f",avg\_wt);

printf("\nAverage Turnaround Time=%f",avg\_tat);

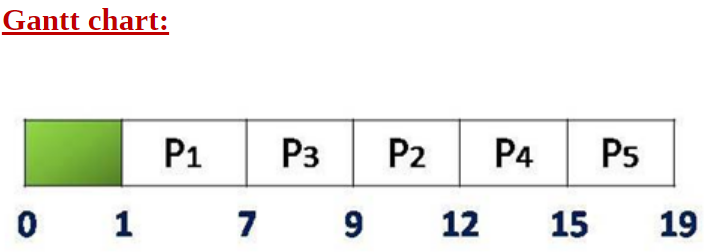
}

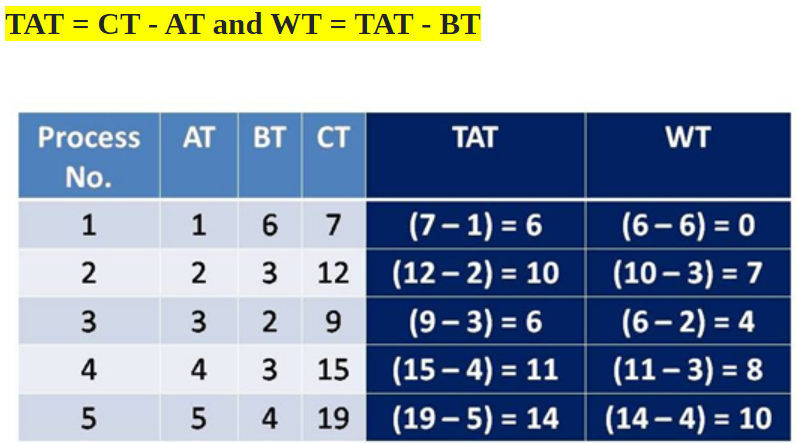
**OUTPUT:**

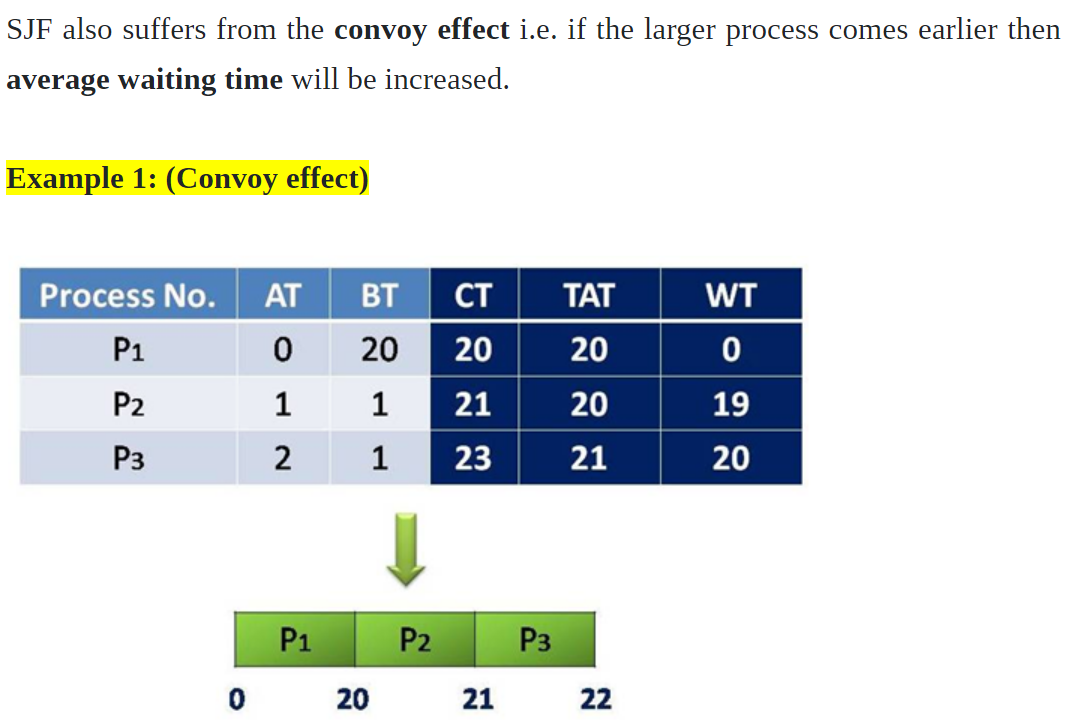


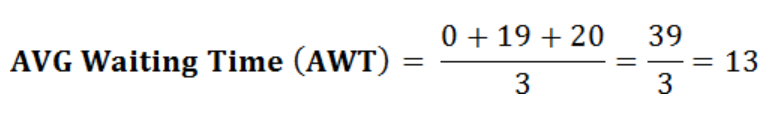
Program-2: With Arrival time

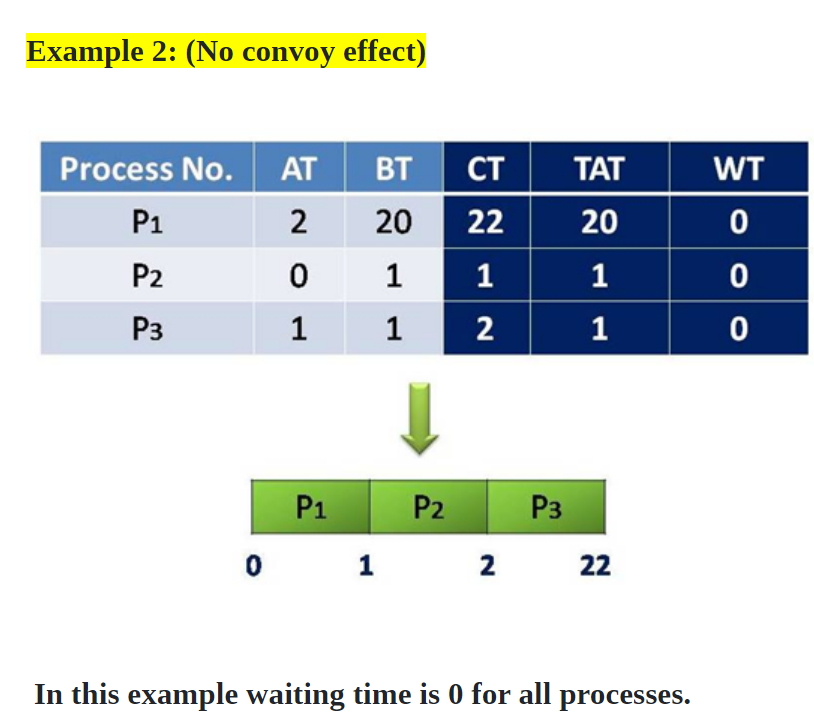












**Round Robin CPU Scheduling Algorithm:**

Round Robin Scheduling is a pre-emptive scheduling algorithm used by the system to schedule CPU utilization. This is a pre-emptive algorithm. There exists a fixed time slice associated with each request called the quantum. The job scheduler saves the progress of the job that is being executed currently and moves to the next job present in the queue when a particular process is executed for a given time quantum.

**ROUND ROBIN SCHEDULING ALGORITHM**

Step 1: We first have a queue where the processes are arranged in first come first serve order.

Step 2: A quantum value is allocated to execute each process.

Step 3: The first process is executed until the end of the quantum value. After this, an interrupt is generated and the state is saved.

Step 4: The CPU then moves to the next process and the same method is followed.

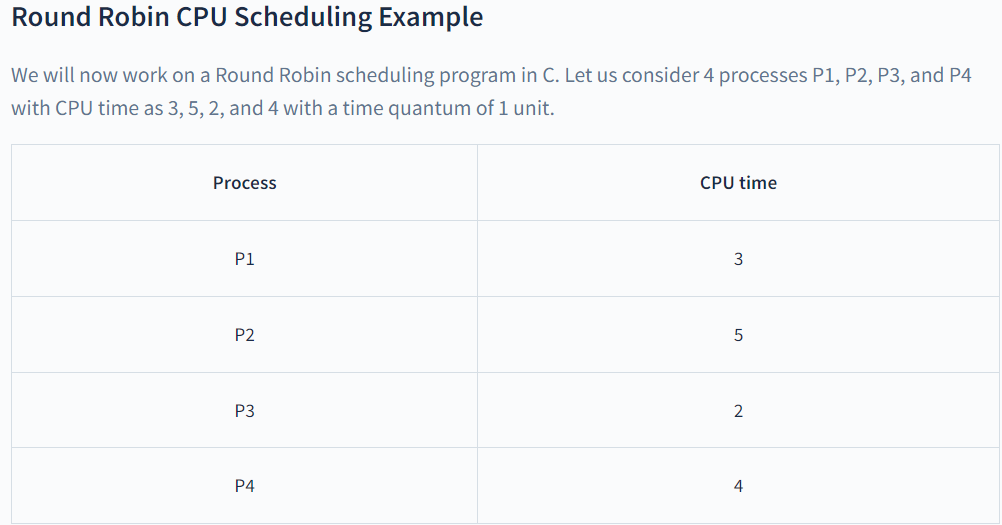
Step5: Same steps are repeated till all the processes are over.

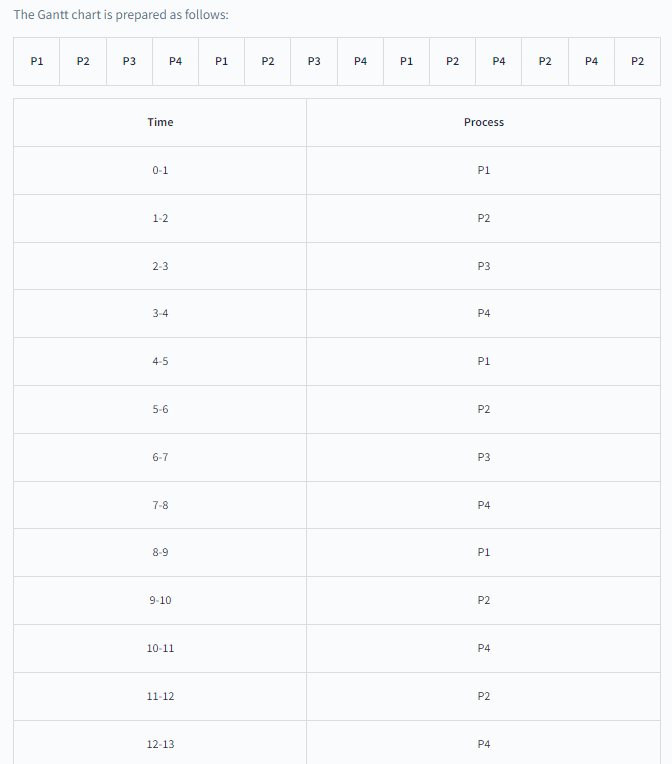
ADVANTAGES:

* Low overhead for decision making.
* Unlike other algorithms, it gives equal priority to all processes.
* Starvation rarely occurs in this process.

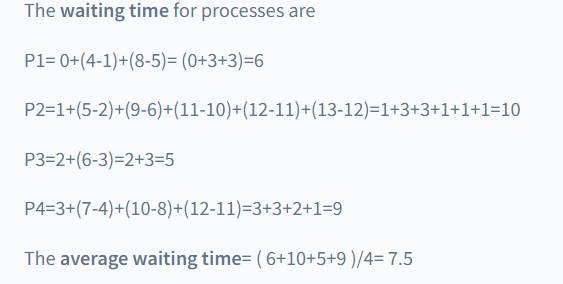
DISADVANTAGES:

* The efficiency of the system is decreased if the quantum value is low as frequent switching takes place.
* The system may become unresponsive if the quantum value is high.









**Program:**

#include<stdio.h>

int main()

{

int i,j,n,t,remain,flag=0,tq;

int wt=0,tat=0,at[10],bt[10],rt[10];

printf("Enter Total Process:\t ");

scanf("%d",&n);

remain=n;

for(i=0;i<n;i++)

{

printf("Enter Arrival Time and Burst Time for Process Process Number %d :",i+1);

scanf("%d",&at[i]);

scanf("%d",&bt[i]);

rt[i]=bt[i];

}

printf("Enter Time Quantum:\t");

scanf("%d",&tq);

printf("\n\nProcess\t|Burst Time|Turnaround Time|Waiting Time\n\n");

for(t=0,i=0;remain!=0;)

{

if(rt[i]<=tq && rt[i]>0)

{

t+=rt[i];

rt[i]=0;

flag=1;

}

else if(rt[i]>0)

{

rt[i]-=tq;

t+=tq;

}

if(rt[i]==0 && flag==1)

{

remain--;

printf("P[%d]\t|\t%d\t|\t%d\t|\t%d\n",i+1,bt[i],t-at[i],t-at[i]-bt[i]);

wt+=t-at[i]-bt[i];

tat+=t-at[i];

flag=0;

}

if(i==n-1)

i=0;

else if(at[i+1]<=t)

i++;

else

i=0;

}

printf("\nAverage Waiting Time= %f\n",wt\*1.0/n);

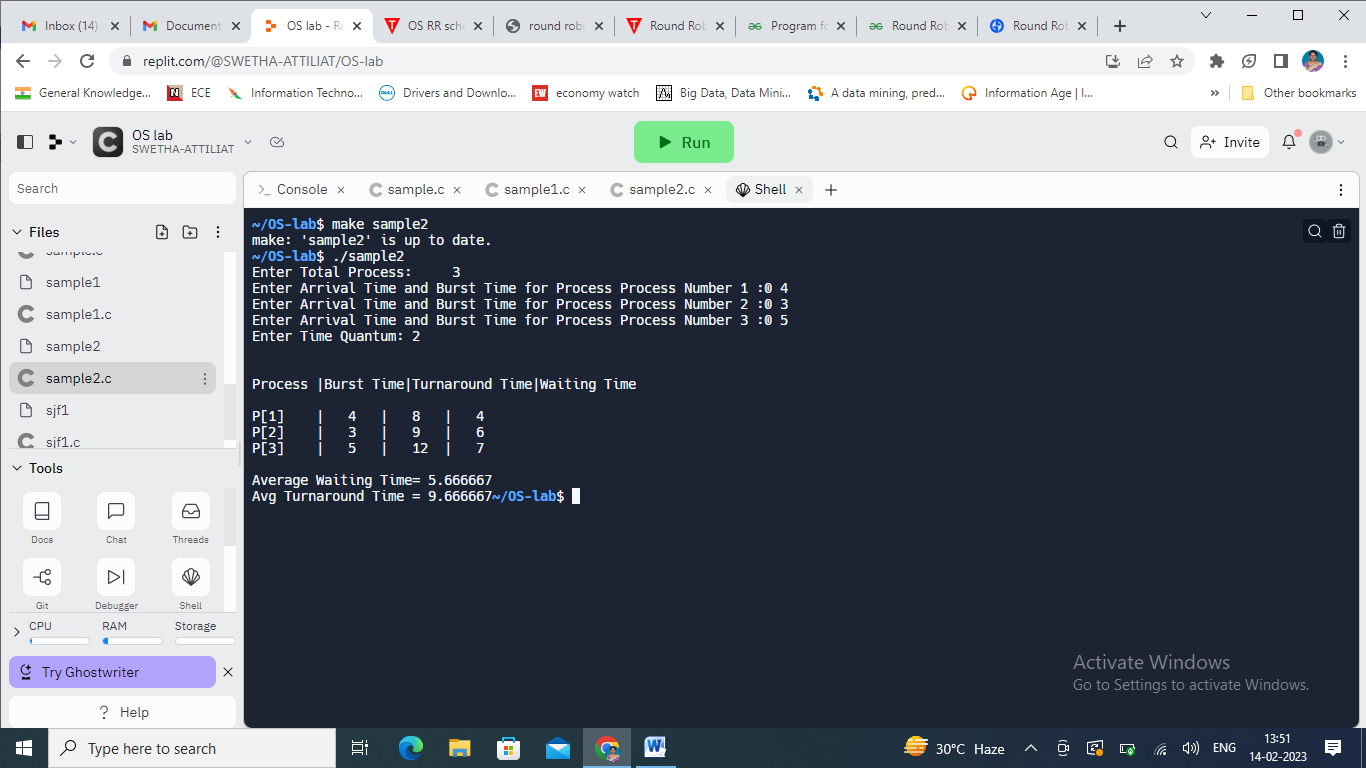
printf("Avg Turnaround Time = %f",tat\*1.0/n);

return 0;

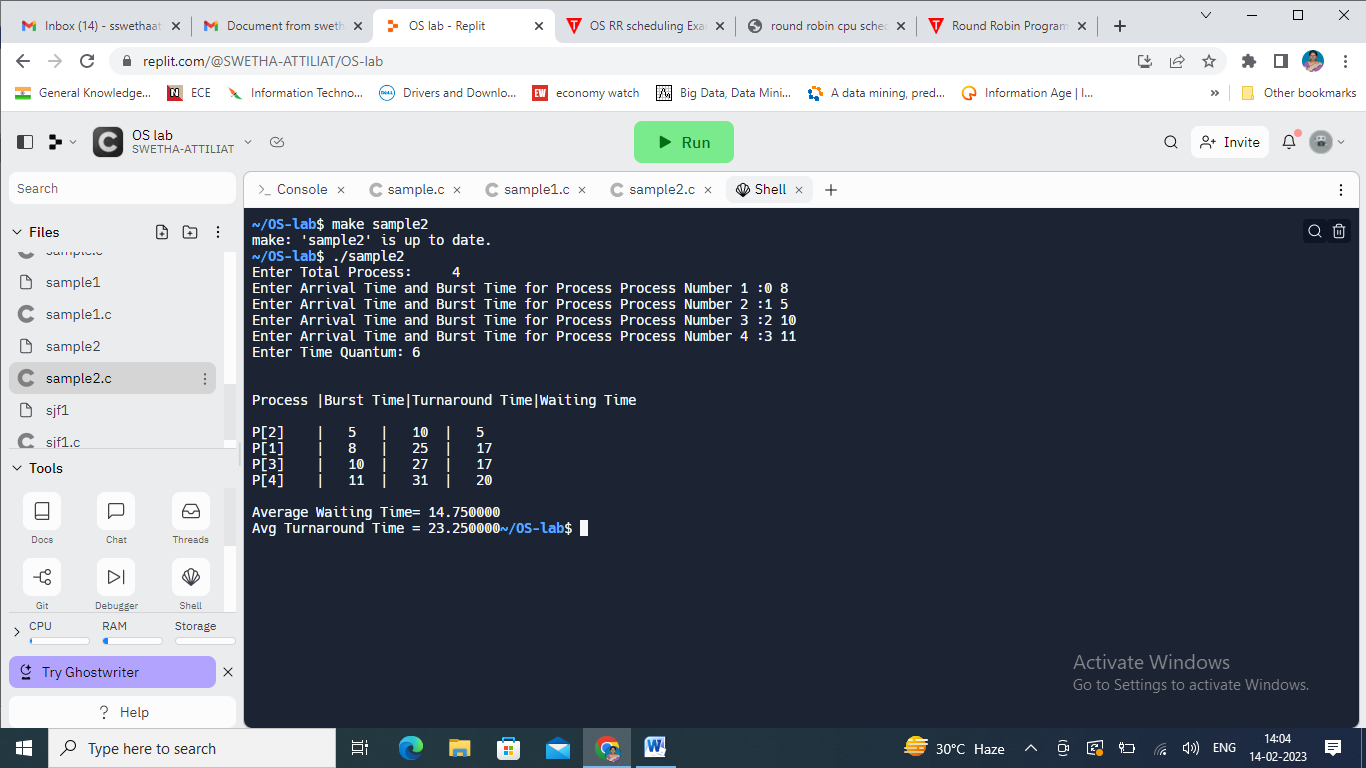
}

**Output:**

**Case-1: With zero arrival time**



**Case-2: With sequential arrival time**



**Priority Scheduling** is a CPU scheduling algorithm in which the CPU performs the task having higher priority at first. If two processes have the same priority then scheduling is done on **FCFS** basis (first come first serve). Priority Scheduling is of two types **Pre-emptive** and **Non-Pre-emptive**.

**Characteristics:-**

1. It schedules the process based on the priority of the processes.
2. Lower the number higher the priority.
3. If the two or more processes have the same priority then we schedules on the basis of FCFS.
4. Major problem with priority scheduling is problem of starvation.
5. Solution of the problem of the starvation is aging ,where aging is a technique of gradually increasing the priority of the processes that wait in the system from long time.

**Drawbacks:-**

1. Major problem with priority scheduling is problem of starvation.
2. Solution of the problem of the starvation is aging ,where aging is a technique of gradually increasing the priority of the processes that wait in the system from long time.

**Non-Preemptive Scheduling algorithm with zero arrival time:**

Program:

#include <stdio.h>

//Function to swap two variables

void swap(int \*a,int \*b)

{

int temp=\*a;

\*a=\*b;

\*b=temp;

}

int main()

{

int n;

printf("Enter Number of Processes: ");

scanf("%d",&n);

// b is array for burst time, p for priority and index for process id

int b[n],p[n],index[n];

for(int i=0;i<n;i++)

{

printf("Enter Burst Time and Priority Value for Process %d: ",i+1);

scanf("%d %d",&b[i],&p[i]);

index[i]=i+1;

}

for(int i=0;i<n;i++)

{

int a=p[i],m=i;

//Finding out highest priority element and placing it at its desired position

for(int j=i;j<n;j++)

{

if(p[j] > a)

{

a=p[j];

m=j;

}

}

//Swapping processes

swap(&p[i], &p[m]);

swap(&b[i], &b[m]);

swap(&index[i],&index[m]);

}

// T stores the starting time of process

int t=0;

//Printing scheduled process

printf("Order of process Execution is\n");

for(int i=0;i<n;i++)

{

printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);

t+=b[i];

}

printf("\n");

printf("Process Id Burst Time Wait Time TurnAround Time\n");

int wait\_time=0;

for(int i=0;i<n;i++)

{

printf("P%d\t\t\t|\t%d\t\t\t|\t%d\t\t\t|\t%d\n",index[i],b[i],wait\_time,wait\_time + b[i]);

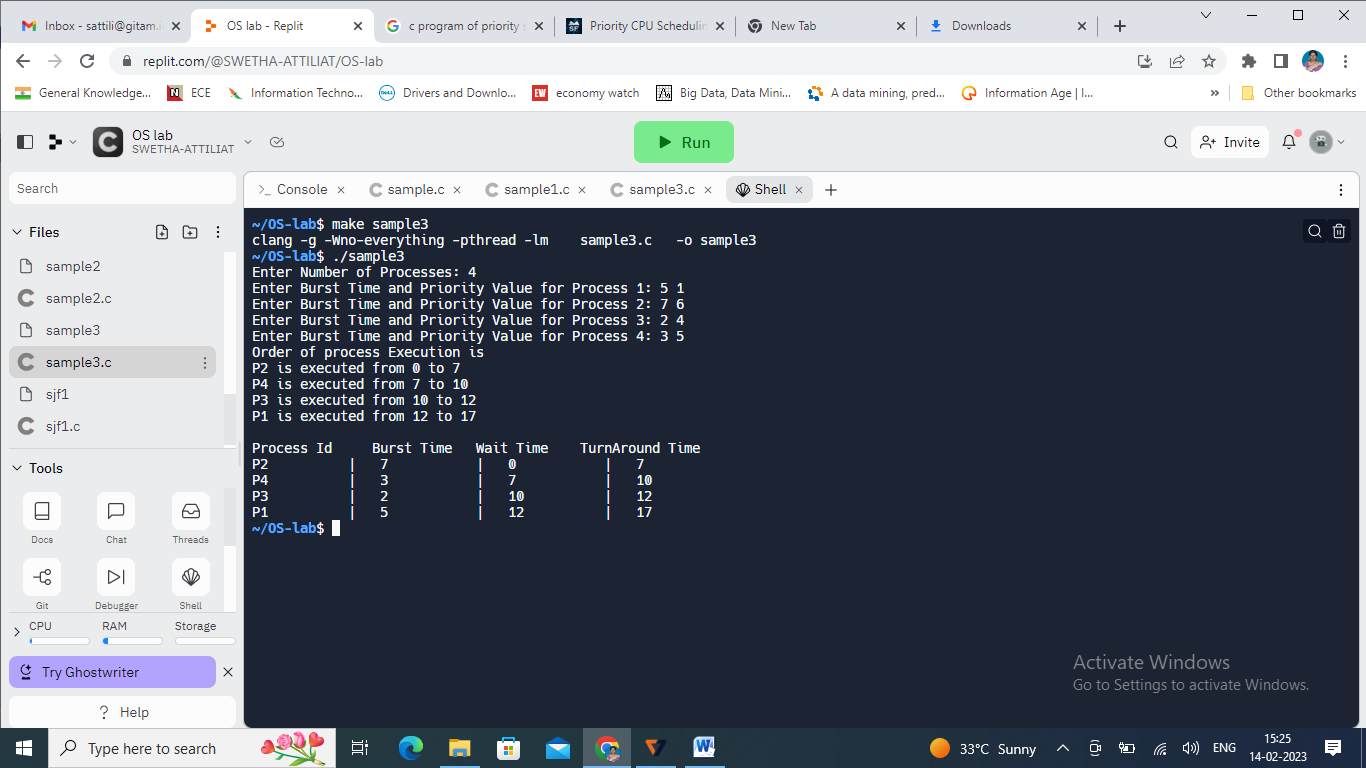
wait\_time += b[i];

}

return 0;

}

**Output:**



### ****Preemptive Priority CPU Scheduling Algorithm:****

* **Step-1:**Select the first process whose [arrival time](https://www.geeksforgeeks.org/difference-between-arrival-time-and-burst-time-in-cpu-scheduling/) will be 0, we need to select that process because that process is only executing at time t=0.
* **Step-2:**Check the priority of the next available process. Here we need to check for 3 conditions.
  + if **priority(current\_process)> priority(prior\_process)**:- then execute the current process.
  + if **priority(current\_process) < priority(prior\_process)** :- then execute the prior process.
  + if **priority(current\_process) = priority(prior\_process)** :- then execute the process which arrives first i.e., arrival time should be first.
* **Step-3:** Repeat **Step-2** until it reaches the final process.
* **Step-4:**When it reaches the final process, choose the process which is having the highest priority & execute it. Repeat the same step until all processes complete their execution