**PRE-EMPTIVE CPU SCHEDULING ALGORITHM**

**Experiment No: 4 Date: 06/09/23**

**Aim:** **a)** To implement Shortest Remaining Time Next (SRTN) CPU scheduling algorithm.

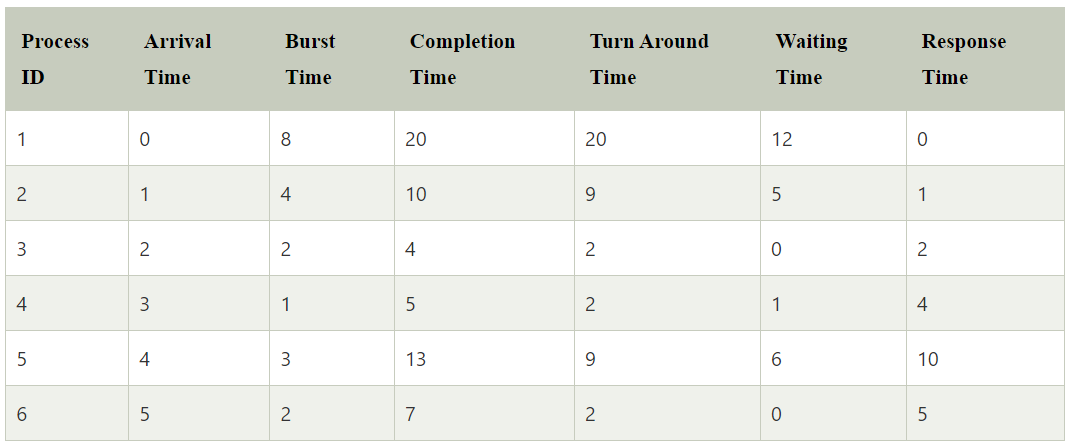
**Theory:**

The Shortest Remaining Time Next (SRTN) algorithm, also known as Shortest Remaining Time First (SRTF) or Pre-emptive Shortest Job Next (PSJN), is a CPU scheduling algorithm used in operating systems. It is a variation of the Shortest Job First (SJF) algorithm, but it can pre-emptively switch between processes if a new process with a shorter burst time arrives.

Here's how the SRTN algorithm works:

1. When a process enters the ready queue or is currently running, the algorithm keeps track of the remaining time required to complete its execution.
2. The algorithm selects the process with the shortest remaining time to execute next. If a new process arrives with a shorter burst time than the currently running process, it preempts the running process and starts executing the new process.
3. When a process finishes execution, it is removed from the queue.
4. If two processes have the same remaining time, the one that arrived first is selected.
5. The algorithm continues to select processes based on their remaining time until all processes have completed their execution.

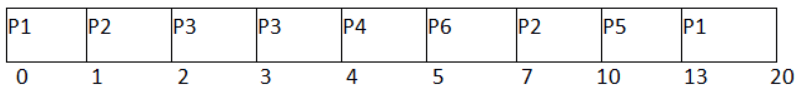
Example:



Average waiting time = 4

Average Turn Around Time = 7.33

Gantt Chart:



**CODE:**

#include<bits/stdc++.h>

using namespace std;

class Process

{

public:

static int count;

char id;

int at , bt , ct , tat , wt , orig\_bt;

Process()

{

cin >> this -> at >> this -> bt ;

this -> id = 'A' + count++ ;

this -> orig\_bt = this -> bt ;

}

};

int Process :: count = 0;

class compare

{

public:

bool operator() (Process &a , Process &b)

{

return a.bt > b.bt;

}

};

void display(vector<Process> &p , vector<pair<char , int >> & gantte)

{

int n = p.size();

int m = gantte.size();

cout<<"\nGantte Chart : \n";

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

{

cout<<left<<setw(6)<<"|"<<gantte[i].first<<" ";

}

cout<<"|"<<endl;

cout<<" ";

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

{

cout << left << setw(12) << gantte[i].second;

}

cout<<endl<<endl;

sort(p.begin() , p.end() , [&](Process a , Process b){return a.id < b.id;});// for printing purpose

float avg\_wt = 0 , avg\_tat = 0;

cout<<left<<setw(20)<<"Process Id"<<setw(20)<<"Arrival Time"<<setw(20)<<"Burst Time"<<setw(20)<<"Completion Time"<<setw(20)<<"Turn Around Time"<<setw(20)<<"Waiting Time"<<endl;

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

{

cout<<setw(20)<<p[i].id<<setw(20)<<p[i].at<<setw(20)<<p[i].bt<<setw(20)<<p[i].ct<<setw(20)<<p[i].tat<<setw(20)<<p[i].wt<<endl;

avg\_wt += p[i].wt;

avg\_tat += p[i].tat;

}

avg\_tat /= n ;

avg\_wt /= n ;

cout<<"Avg\_WT : "<<avg\_wt<<endl;

cout<<"Avg\_TAT : "<<avg\_tat<<endl;

}

void SRTN(vector<Process>& p)

{

int n = p.size();

vector<pair<char, int>> gantte;

sort(p.begin() , p.end() , [&](Process a , Process b){return a.at < b.at;});

vector<Process> ans;

priority\_queue<Process , vector<Process> , compare> pq;

//initialize the start timer and index

int index = 0 , start = p[index].at;

while(index < n or !pq.empty())

{

while( index < n and p[index].at <= start )

{

pq.push(p[index]);

index++;

}

if(!pq.empty())

{

Process curr = pq.top(); pq.pop();

int time\_for\_nxt\_arr = curr.bt ;

if(index < n )

time\_for\_nxt\_arr = p[index].at - start;

if( curr.bt <= time\_for\_nxt\_arr )

{

curr.ct = start + curr.bt;

curr.tat = curr.ct - curr.at;

curr.wt = curr.tat - curr.orig\_bt;

start = start + curr.bt;

curr.bt = curr.orig\_bt;

gantte.push\_back({curr.id , curr.ct});

ans.push\_back(curr);

}

else

{

curr.ct = start + time\_for\_nxt\_arr;

gantte.push\_back({curr.id , curr.ct});

curr.bt = curr.bt - time\_for\_nxt\_arr;

start = start + time\_for\_nxt\_arr;

pq.push(curr);

}

if(pq.empty() and index < n and p[index].at > start )

start = p[index].at;

}

}

display(ans , gantte);

}

int main()

{

int n;

cout<<"Enter the Number of Processes\n";

cin>>n;

cout<<"Enter the Arrival Time and Burst Time Of Each Process\n";

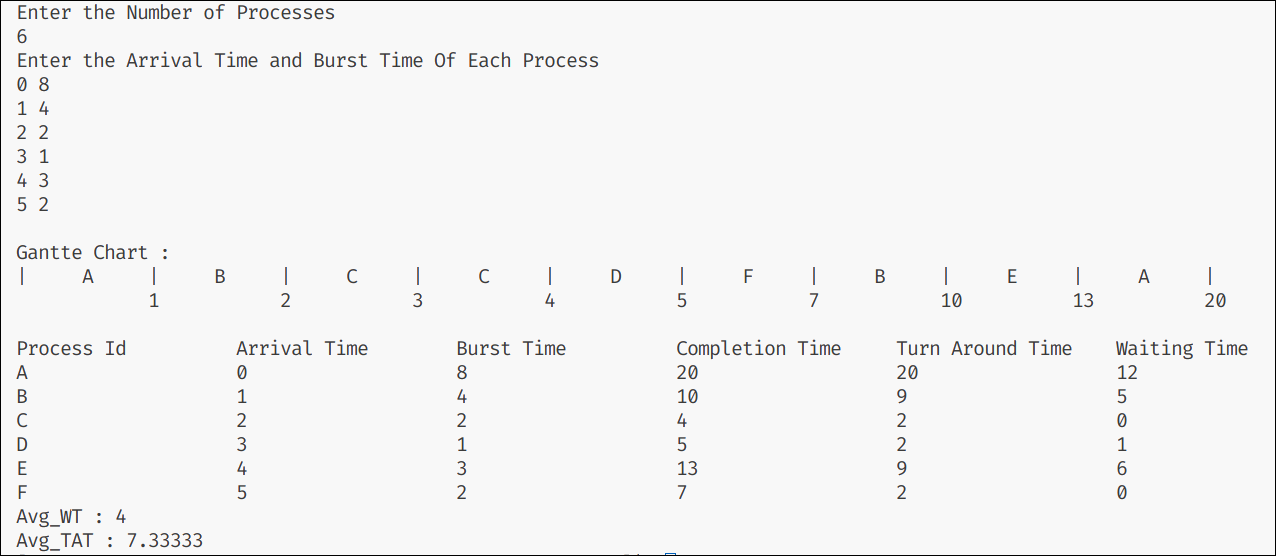
vector<Process> p(n);

SRTN(p);

return 0;

}

**OUTPUT:**

****

**Conclusion:**

The Pre-emptive following CPU scheduling algorithm - Shortest Remaining Time Normal/First was successfully implemented in this experiment

**PRE-EMPTIVE CPU SCEDULING ALGORITHM**

**Experiment No: 4 Date: 13/09/23**

**Aim:** **b)** To implement pre emptive priority CPU scheduling algorithm.

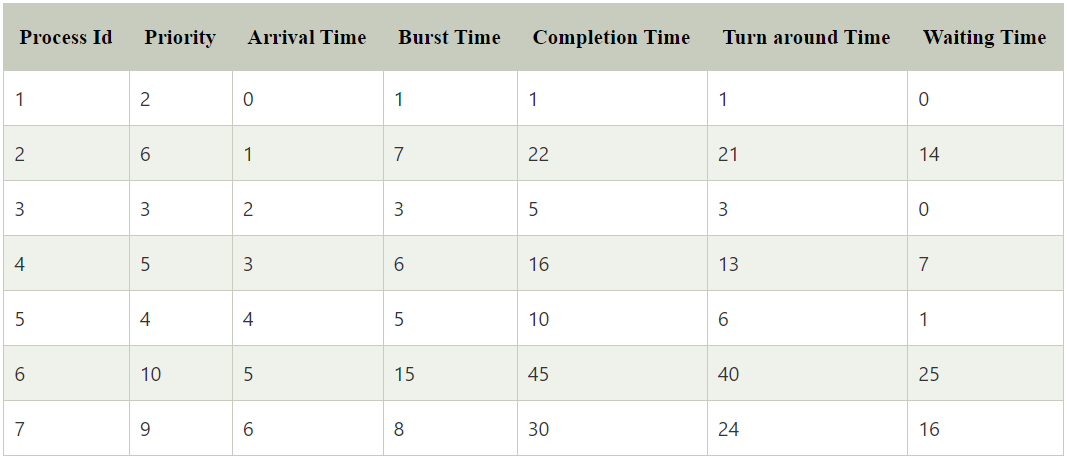
**Theory:**

Preemptive Priority Scheduling is a CPU scheduling algorithm used in operating systems to determine the order in which processes are executed on a CPU. In this algorithm, each process is assigned a priority, and the process with the highest priority that is ready to execute is given access to the CPU. If a higher-priority process becomes available while a lower-priority process is running, the lower-priority process is preempted, and the higher-priority process is allowed to execute.

Here's how the Preemptive Priority Scheduling algorithm works:

1. Assign a priority value to each process in the system. A lower number typically represents a higher priority.
2. When a process enters the ready queue (either due to its arrival or because it has been waiting), the scheduler checks its priority.
3. If the newly arrived process has a higher priority than the currently executing process, the scheduler preempts the running process and allows the higher-priority process to execute.
4. If two processes have the same priority, the scheduler can use other criteria, such as First-Come-First-Served (FCFS) or Round Robin, to determine the execution order among processes with the same priority.
5. When a process completes its execution, it is removed from the queue.
6. The scheduler continues to select the highest-priority process that is ready to execute until all processes have completed.

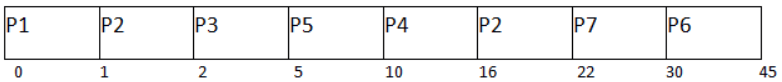
Example:



Average waiting time = 9

Average Turn Around Time = 15.42

Gantt Chart:



**CODE:**

#include<bits/stdc++.h>

using namespace std;

class Process

{

public:

static int count;

char id;

int at , bt , ct , tat , wt , orig\_bt , prior;

Process()

{

cin >> this -> at >> this -> bt >> this -> prior ;

this -> id = 'A' + count++ ;

this -> orig\_bt = this -> bt ;

}

};

int Process :: count = 0;

class compare

{

public:

bool operator() (Process &a , Process &b)

{

return a.prior > b.prior;

}

};

void display(vector<Process> &p , vector<pair<char , int >> & gantte)

{

int n = p.size();

int m = gantte.size();

cout<<"\nGantte Chart : \n";

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

{

cout<<left<<setw(6)<<"|"<<gantte[i].first<<" ";

}

cout<<"|"<<endl;

cout<<" ";

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

{

cout << left << setw(12) << gantte[i].second;

}

cout<<endl<<endl;

sort(p.begin() , p.end() , [&](Process a , Process b){return a.id < b.id;});// for printing purpose

float avg\_wt = 0 , avg\_tat = 0;

cout<<left<<setw(20)<<"Process Id"<<setw(20)<<"Arrival Time"<<setw(20)<<"Burst Time"<<setw(20)<<"Completion Time"<<setw(20)<<"Turn Around Time"<<setw(20)<<"Waiting Time"<<endl;

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

{

cout<<setw(20)<<p[i].id<<setw(20)<<p[i].at<<setw(20)<<p[i].bt<<setw(20)<<p[i].ct<<setw(20)<<p[i].tat<<setw(20)<<p[i].wt<<endl;

avg\_wt += p[i].wt;

avg\_tat += p[i].tat;

}

avg\_tat /= n ;

avg\_wt /= n ;

cout<<"Avg\_WT : "<<avg\_wt<<endl;

cout<<"Avg\_TAT : "<<avg\_tat<<endl;

}

void Priority(vector<Process>& p)

{

int n = p.size();

vector<pair<char, int>> gantte;

sort(p.begin() , p.end() , [&](Process a , Process b){return a.at < b.at;});

vector<Process> ans;

priority\_queue<Process , vector<Process> , compare> pq;

int index = 0 , start = p[index].at;

while(index < n or !pq.empty())

{

while( index < n and p[index].at <= start )

{

pq.push(p[index]);

index++;

}

if(!pq.empty())

{

Process curr = pq.top(); pq.pop();

int time\_for\_nxt\_arr = curr.bt ;

if(index < n )

time\_for\_nxt\_arr = p[index].at - start;

if( curr.bt <= time\_for\_nxt\_arr )

{

curr.ct = start + curr.bt;

curr.tat = curr.ct - curr.at;

curr.wt = curr.tat - curr.orig\_bt;

start = start + curr.bt;

curr.bt = curr.orig\_bt;

gantte.push\_back({curr.id , curr.ct});

ans.push\_back(curr);

}

else

{

curr.ct = start + time\_for\_nxt\_arr;

gantte.push\_back({curr.id , curr.ct});

curr.bt = curr.bt - time\_for\_nxt\_arr;

start = start + time\_for\_nxt\_arr;

pq.push(curr);

}

if(pq.empty() and index < n and p[index].at > start )

start = p[index].at;

}

}

display(ans , gantte);

}

int main()

{

int n;

cout<<"Enter the Number of Processes\n";

cin>>n;

cout<<"Enter the Arrival Time and Burst Time and Priority Of Each Process\n";

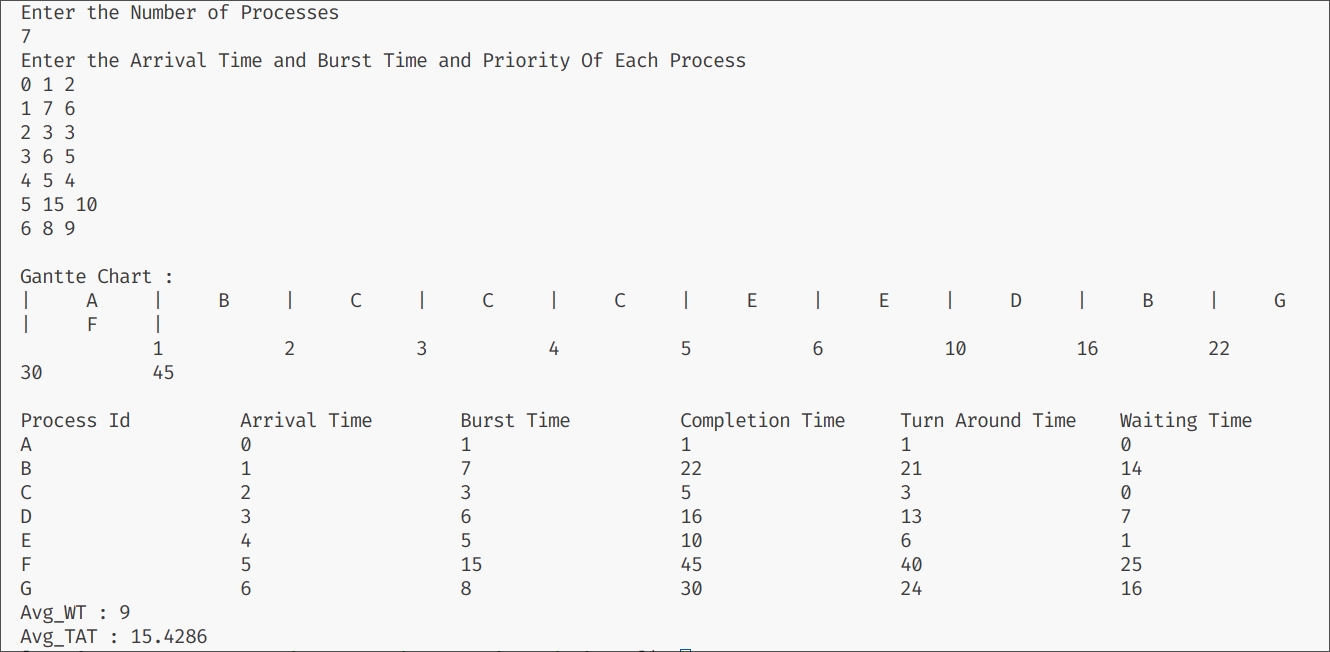
vector<Process> p(n);

Priority(p);

return 0;

}

**OUTPUT:**



**Conclusion:**

The Pre-emptive following CPU scheduling algorithm - Pre-emptive Priority Scheduling Algorithm was successfully implemented in this experiment

**PRE-EMPTIVE CPU SCHEDULING ALGORITHM**

**Experiment No: 4 Date: 20/09/23**

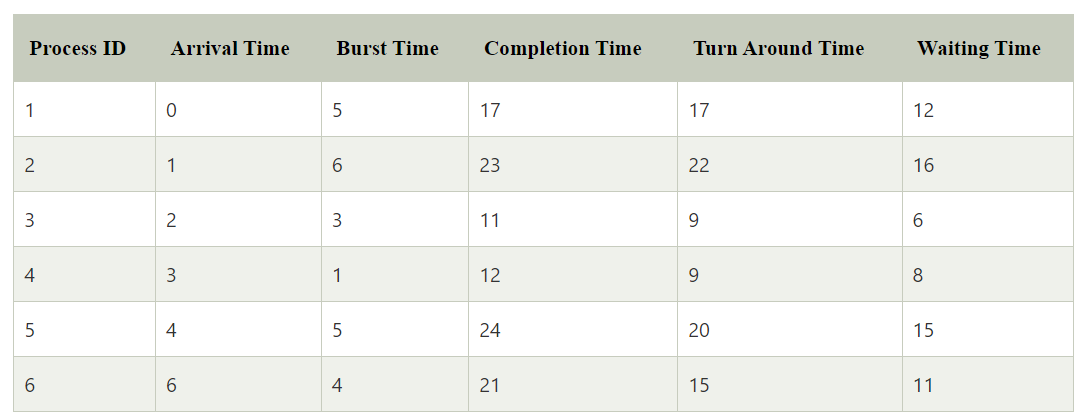
**Aim:** **c)** To implement Round Robin (RR) CPU scheduling algorithm.

**Theory:**

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added 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. 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 of 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. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue. The average waiting time under the RR policy is often long.

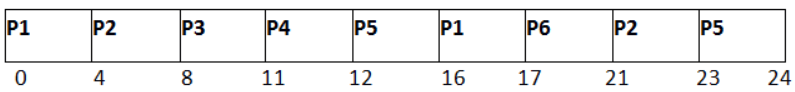
Example:



Average waiting time = 11.33

Average Turn Around Time = 15.33

Gantt Chart:



**CODE:**

#include<bits/stdc++.h>

using namespace std;

class Process

{

public:

static int count;

char id;

int at , bt , ct , tat , wt , orig\_bt;

Process()

{

cin >> this -> at >> this -> bt;

this -> id = 'A' + count++ ;

this -> orig\_bt = this -> bt ;

}

};

int Process :: count = 0;

bool cmp(Process &a , Process &b)

{

if(a.at == b.at)

return a.bt < b.bt;

return a.at < b.at ;

}

void display(vector<Process> &p , vector<pair<char , int >> & gantte)

{

int n = p.size();

int m = gantte.size();

cout<<"\nGantte Chart : \n";

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

{

cout<<left<<setw(6)<<"|"<<gantte[i].first<<" ";

}

cout<<"|"<<endl;

cout<<" ";

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

{

cout << left << setw(12) << gantte[i].second;

}

cout<<endl<<endl;

sort(p.begin() , p.end() , [&](Process a , Process b){return a.id < b.id;});

float avg\_wt = 0 , avg\_tat = 0;

cout<<left<<setw(20)<<"Process Id"<<setw(20)<<"Arrival Time"<<setw(20)<<"Burst Time"<<setw(20)<<"Completion Time"<<setw(20)<<"Turn Around Time"<<setw(20)<<"Waiting Time"<<endl;

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

{

cout<<setw(20)<<p[i].id<<setw(20)<<p[i].at<<setw(20)<<p[i].bt<<setw(20)<<p[i].ct<<setw(20)<<p[i].tat<<setw(20)<<p[i].wt<<endl;

avg\_wt += p[i].wt;

avg\_tat += p[i].tat;

}

avg\_tat /= n ;

avg\_wt /= n ;

cout<<"Avg\_WT : "<<avg\_wt<<endl;

cout<<"Avg\_TAT : "<<avg\_tat<<endl;

}

void RoundRobin(vector<Process>& p , int TQ)

{

int n = p.size();

sort(p.begin() , p.end() , cmp);

queue<Process> q;

vector<Process> ans;

vector<pair<char,int>> gantte;

int index = 0 , start = p[index].at;

q.push(p[index++]);

while( !q.empty() or index < n )

{

Process curr = q.front() ; q.pop() ;

int temp\_bt = curr.bt;

if(curr.bt <= TQ)

{

curr.ct = start + curr.bt;

curr.tat = curr.ct - curr.at;

curr.wt = curr.tat - curr.orig\_bt;

start = start + curr.bt;

curr.bt = curr.orig\_bt;

gantte.push\_back({curr.id , curr.ct});

ans.push\_back(curr);

}

else

{

curr.ct = start + TQ;

curr.bt = curr.bt - TQ;

gantte.push\_back({curr.id, curr.ct});

start = start + TQ;

}

while( index < n and p[index].at <= start )

{

q.push(p[index]);

index++;

}

if(temp\_bt > TQ)

q.push(curr);

if(q.empty() and index < n and p[index].at > start )

start = p[index].at;

}

display(ans , gantte);

}

int main()

{

int n , TQ;

cout<<"Enter the Number of Processes and Time Quantum\n";

cin>> n >> TQ;

cout<<"Enter the Arrival Time and Burst Time Of Each Process\n";

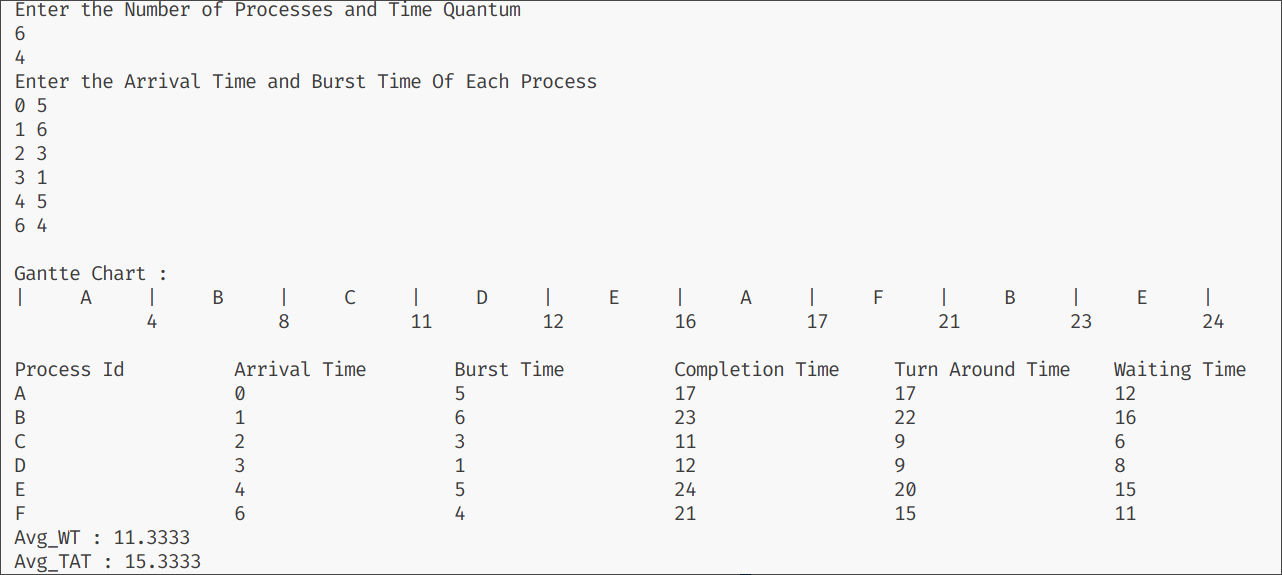
vector<Process> p(n);

RoundRobin(p , TQ);

return 0;

}

**OUTPUT:**

****

**Conclusion:**

The Pre-emptive following CPU scheduling algorithm - Round Robin Scheduling Algorithm

was successfully implemented in this experiment.