**NON-PREEMPTIVE CPU SCHEDULING ALGORITHM**

**Experiment No. 3 Date:23/08/2023**

**Aim:**  To implement the following non-preemptive CPU scheduling algorithms:

1. FCFS- First Come First Serve
2. SJF- Shortest Job First
3. Non-Preemptive Priority

**Theory:**

(i). First Come First Served (FCFS)

First Come First Served (FCFS) is a non-pre-emptive scheduling algorithm. FIFO (First In First Out) strategy assigns priority to process in the order in which they request the processor. The process that requests the CPU first is allocated the CPU first. This is easily implemented with a FIFO queue for managing the tasks. As the process come in, they are put at the end of the queue. As the CPU finishes each task, it removes it from the start of the queue and heads on to the next task.

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm. First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. Here we are considering that arrival time for all processes is 0.

Scheduling of processes/work is done to finish the work on time. A typical process involves both I/O time and CPU time. In a programming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multi programming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

Arrival Time: ​ Time at which the process arrives in the ready queue.

Completion Time: ​ Time at which process completes its execution.

Burst Time: ​ Time required by a process for CPU execution.

Turn Around Time: ​ Time Difference between completion time and arrival time.

Turn Around Time = ​ Completion Time – Arrival Time

Waiting Time(W.T): ​ Time Difference between turn around time and burst time.

Waiting Time = ​ Turn Around Time – Burst Time

Min turnaround time: ​ Time taken by a process to finish execution

Min waiting time: ​ Time a process waits in ready queue

Min response time: ​ Time when a process produces first response

Example:

(ii). SJF Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process & next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution.

Non-preemptive SJF (Shortest Job First) scheduling, also known as Shortest Job Next (SJN), is a CPU scheduling algorithm where the operating system selects the process with the shortest burst time from the ready queue to execute. Once a process starts running, it continues until it completes its execution or blocks (e.g., for I/O operations). The scheduler does not preempt a running process.

Here's how non-preemptive SJF scheduling works:

1. When a new process arrives or the currently running process finishes, the scheduler selects the process with the shortest burst time from the ready queue.
2. The selected process is then executed until it finishes or blocks for I/O.
3. After a process completes or blocks, the scheduler selects the next process based on the shortest remaining burst time among the processes in the ready queue.
4. This process continues until all processes have completed their execution.

Non-preemptive SJF scheduling is efficient in minimizing the average waiting time and average turnaround time because it prioritizes shorter jobs. However, it requires knowledge of the burst times of all processes in advance, which may not always be available in practice. Additionally, it may lead to starvation for longer processes if they are constantly preempted by short processes.

Example:

Top of Form

(iii).Non-preemptive Priority

Non-preemptive Priority Scheduling is a CPU scheduling algorithm in which processes are assigned priorities, and the process with the highest priority is selected to execute first. Once a process starts running, it continues until it completes its execution or blocks for I/O. Unlike preemptive priority scheduling, the scheduler does not interrupt a running process to switch to another process with higher priority.

Here's how non-preemptive priority scheduling works:

1. Each process is assigned a priority value, with lower numbers indicating higher priority. The operating system typically assigns these priorities based on certain criteria, such as the importance of the process or its specific requirements.
2. When a new process arrives or the currently running process completes or blocks, the scheduler selects the process with the highest priority from the ready queue.
3. The selected process is then executed until it finishes or blocks for I/O.
4. After a process completes or blocks, the scheduler selects the next process with the highest priority among the processes in the ready queue.
5. This process continues until all processes have completed their execution.

Non-preemptive priority scheduling is straightforward and can be effective in situations where processes have distinct priorities that need to be honored. However, it may lead to starvation for lower-priority processes if higher-priority processes continually arrive or require extended execution time. To address this, some systems may incorporate aging mechanisms to gradually increase the priority of waiting processes, preventing starvation.

Example:

**PROGRAMS:**

**(i). FCFS- First Come First Serve**

**CODE:**

#include <iostream>

#include <string>

using namespace std;

class process

{

public:

int burst\_time,cb,arrival\_time,index,waiting\_time,turn\_around\_time;

};

void print(int n,process a[])

{

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

cout<<"ID:"<<a[i].index<<"|Burst:"<<a[i].burst\_time<<"|Arrival:"<<a[i].arrival\_time<<endl;

}

int length(int n, process p[])

{

int l=0;

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

if(p[i].burst\_time>0)

l++;

return l;

}

int main()

{

int n;

cout<<"Number of Processes :";

cin>>n;

process processes[n];

//----------------------------------------input

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

{

cout<<"Job "<<i+1<<endl;

cout<<"Burst Time = ";

cin>>processes[i].burst\_time;

processes[i].cb = processes[i].burst\_time;

cout<<"Arrival Time = ";

cin>>processes[i].arrival\_time;

processes[i].index=i+1;

}

cout<<"======================================\nProcesses before sorting\n"<<endl;

print(n,processes);

//--------------------------------------sorting according to the arrival times

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

for(int j=0;j<n-1;j++)

if(processes[j].arrival\_time>processes[j+1].arrival\_time)

{

process temp = processes[j];

processes[j]=processes[j+1];

processes[j+1]=temp;

}

cout<<"======================================\nProcesses after sorting\n"<<endl;

print(n,processes);

//---------------------------------------declaration of arrays to store the processes execution to print

int time = processes[0].arrival\_time,current\_process = 0;

bool idel\_state = false;

int count=1,times[100]={processes[0].arrival\_time},ids[100]={};

//---------------------------------------simulation for each second

while(length(n,processes)>0)

{

if(time>=processes[current\_process].arrival\_time)

{

if(idel\_state)

{

times[count]=time;

count++;

}

idel\_state=false;

processes[current\_process].waiting\_time = time - processes[current\_process].arrival\_time;

time+=processes[current\_process].burst\_time;

processes[current\_process].turn\_around\_time = time - processes[current\_process].arrival\_time;

processes[current\_process].burst\_time=0;

ids[count]=processes[current\_process].index;

times[count]=time;

current\_process+=1;

count++;

}

else

{

time++;

idel\_state=true;

}

}

//--------------------------------------printing of the collected data

cout<<"======================================\nGantt Chart\n"<<endl;

cout<<"+";

for(int i=1;i<count;i++)

cout<<"--------+";

cout<<endl<<"|";

for(int i=1;i<count;i++)

{

cout.width(8);

if(ids[i]==0)

cout<<std::left<<"Idle"<<"|";

else

cout<<std::left<<"P"+to\_string(ids[i])<<"|";

}

cout<<endl<<"+";

for(int i=1;i<count;i++)

cout<<"--------+";

cout<<endl;

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

{

cout.width(9);

cout<<std::left<<times[i];

}

cout << endl<<endl;

cout<<"======================================\nTable\n"<<endl;

cout << "Process ID |";

cout << "Process Burst\_time |";

cout << "Process Arrival\_time |";

cout << "Process Waiting\_time |";

cout << "Process Turn\_around\_time"<<endl;

cout.width(100);

cout.fill('-');

cout << " "<<endl;

cout.fill(' ');

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

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

{

cout.width(11);

cout<<std::left<<"P"+to\_string(processes[i].index)<<"|";

cout.width(19);

cout << std::left << processes[i].cb << "|";

cout.width(21);

cout << std::left << processes[i].arrival\_time << "|";

cout.width(21);

cout << std::left << processes[i].waiting\_time << "|";

cout.width(25);

cout << std::left << processes[i].turn\_around\_time << endl;

avg\_wt += processes[i].waiting\_time;

avg\_tat += processes[i].turn\_around\_time;

}

cout.width(100);

cout.fill('-');

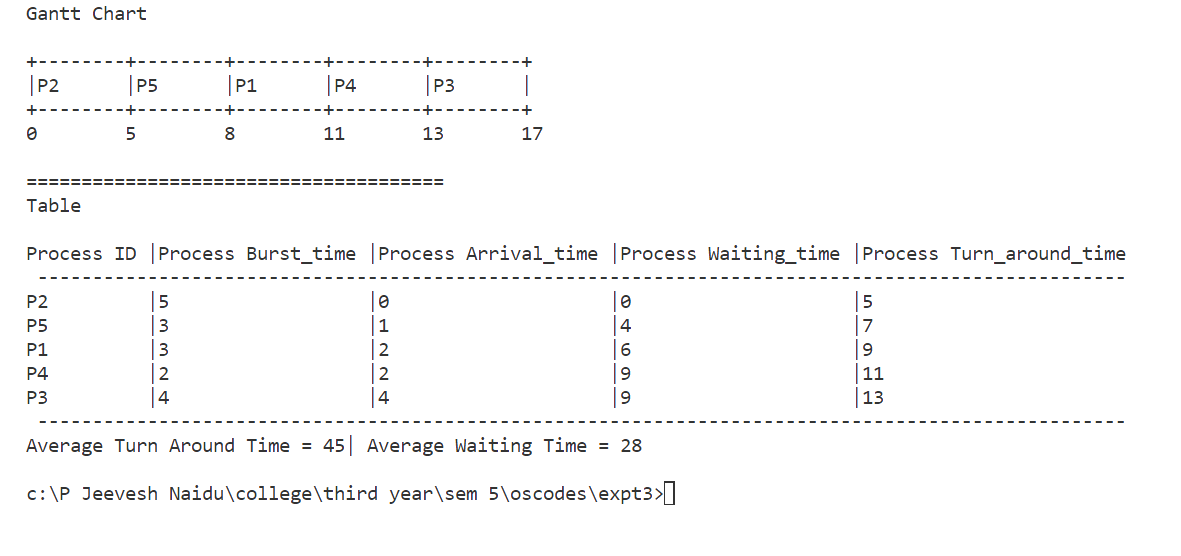
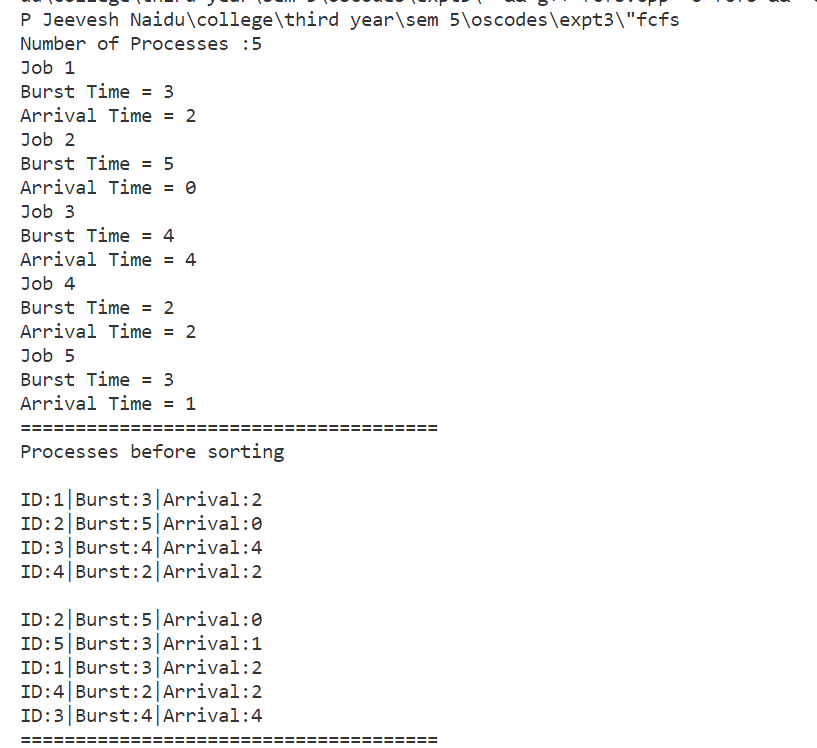
cout << " "<<endl;

cout.fill(' ');

cout << "Average Turn Around Time = " << avg\_tat << "| Average Waiting Time = "<< avg\_wt<<endl;

}

**OUTPUT:**



**(ii). SJF- Shortest Job First**

**CODE:**

#include <iostream>

using namespace std;

class process

{

public:

int burst\_time, cbt, arrival\_time, index, turn\_around\_time, waiting\_time;

};

int length(int n, process p[])

{

int l = 0;

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

if (p[i].burst\_time > 0)

l++;

return l;

}

int main()

{

bool pre\_def = false;

// bool pre\_def = true;

int n = 4;

if (!pre\_def)

{

cout << "Number of Processes :";

cin >> n;

}

int arrival[] = {8, 2, 1, 0};

int burst[] = {1, 2, 1, 4};

process processes[n];

//-------------------------------------------------------------------input

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

{

if (!pre\_def)

{

cout << "Job " << i + 1 << endl;

cout << "Burst Time = ";

cin >> processes[i].burst\_time;

cout << "Arrival Time = ";

cin >> processes[i].arrival\_time;

// processes[i].arrival\_time = 0;

}

else

{

processes[i].burst\_time = burst[i];

processes[i].arrival\_time = arrival[i];

// processes[i].arrival\_time = 0;

}

processes[i].cbt = processes[i].burst\_time;

processes[i].index = i + 1;

}

// sort wrt arrival,priority

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

{

for (int j = 0; j < n - 1; j++)

if (processes[j].arrival\_time > processes[j + 1].arrival\_time)

{

process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

int count = 0;

int time = processes[0].arrival\_time;

int times[100] = {time}, p\_ids[100];

while (length(n, processes) > 0)

{

// finding the next process to execute

int p;

bool idle = true;

for (p = 0; p < n && processes[p].burst\_time < 0; p++)

;

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

if (processes[i].burst\_time >= 0 && processes[i].arrival\_time <= time && processes[i].burst\_time <= processes[p].burst\_time)

{

idle = false;

if (processes[i].burst\_time < processes[p].burst\_time)

p = i;

}

if (idle)

{

int min = 9e8;

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

if (processes[i].arrival\_time >= time && processes[i].arrival\_time < min)

min = i;

p\_ids[count] = -1;

time = processes[min].arrival\_time;

}

else

{

p\_ids[count] = p;

processes[p].waiting\_time = time - processes[p].arrival\_time;

time += processes[p].burst\_time;

processes[p].burst\_time = -1;

processes[p].turn\_around\_time = time - processes[p].arrival\_time;

}

times[count + 1] = time;

count++;

}

cout << "\n\nGantt Chart\n+";

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

cout << "-------+";

cout << endl;

cout << "|";

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

{

cout.width(7);

if (p\_ids[i] != -1)

cout << std::left << "P" + to\_string(processes[p\_ids[i]].index) << "|";

else

cout << std::left << "Idle"

<< "|";

}

cout << "\n+";

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

cout << "-------+";

cout << endl;

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

{

cout.width(8);

cout << times[i];

}

cout << "\n\n\t\t\tAnalysis Table\n";

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

for (int j = 0; j < n - 1; j++)

if (processes[j].index > processes[j + 1].index)

{

process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

cout << "\nID | ARRIVAL TIME | BURST TIME | WAITING TIME | TURN AROUND TIME |\n";

cout.fill('-');

cout.width(78);

cout << std::right << " " << endl;

cout.fill(' ');

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

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

{

cout.width(2);

cout << std::left << "P" + to\_string(processes[i].index) << " | ";

cout.width(12);

cout << std::left << processes[i].arrival\_time << " | ";

cout.width(10);

cout << std::left << processes[i].cbt << " | ";

cout.width(12);

cout << std::left << processes[i].waiting\_time << " | ";

cout.width(16);

cout << std::left << processes[i].turn\_around\_time << " | ";

cout << endl;

avg\_wt += processes[i].waiting\_time;

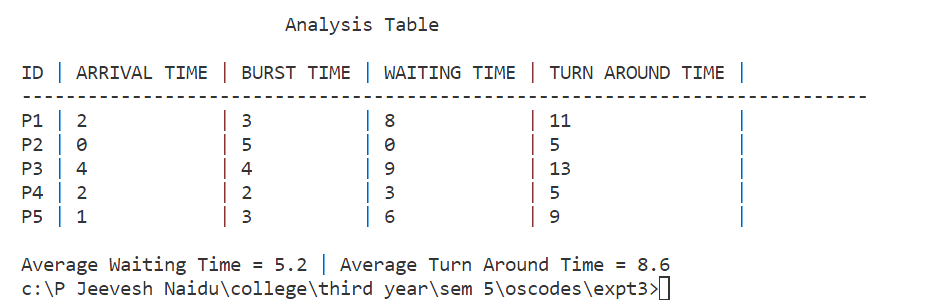
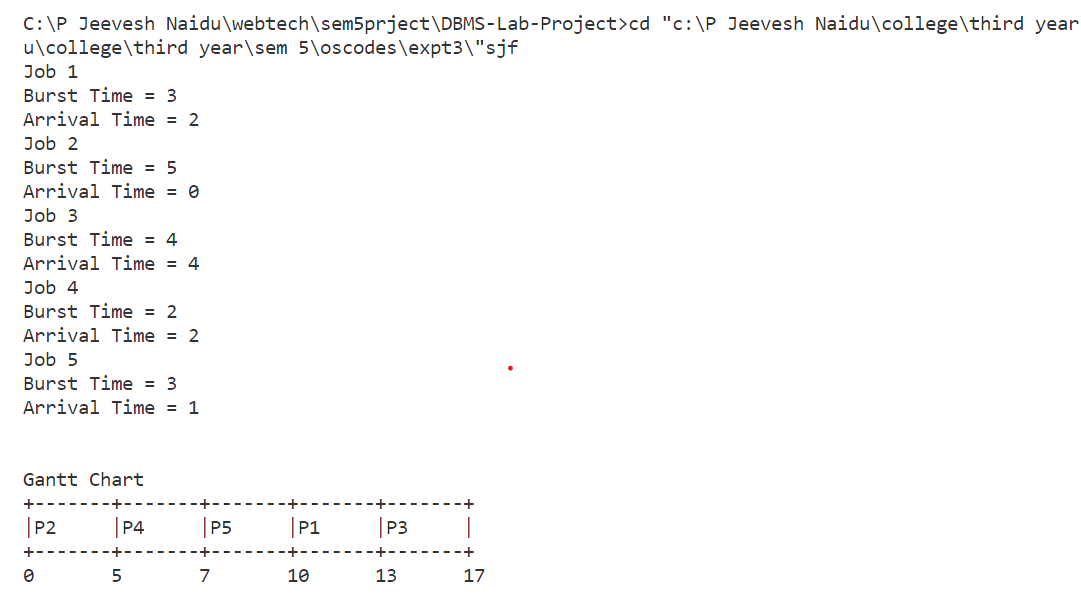
avg\_tat += processes[i].turn\_around\_time;

}

cout << "\nAverage Waiting Time = " << avg\_wt / n << " | Average Turn Around Time = " << avg\_tat / n;

}

**OUTPUT:**

****

**(iii). Non-Preemptive Priority**

**CODE:**

#include <iostream>

using namespace std;

class process

{

public:

int burst\_time, cbt, arrival\_time, priority, index, turn\_around\_time, waiting\_time;

};

int length(int n, process p[])

{

int l = 0;

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

if (p[i].burst\_time > 0)

l++;

return l;

}

int main()

{

bool pre\_def = false;

// bool pre\_def = true;

int n = 4;

if (!pre\_def)

{

cout << "Number of Processes :";

cin >> n;

}

int arrival[] = {8, 2, 1, 0};

int burst[] = {1, 2, 1, 4};

int priority[] = {1, 3, 2, 1};

process processes[n];

//-------------------------------------------------------------------input

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

{

if (!pre\_def)

{

cout << "Job " << i + 1 << endl;

cout << "Burst Time = ";

cin >> processes[i].burst\_time;

cout << "Arrival Time = ";

cin >> processes[i].arrival\_time;

cout << "Priority = ";

cin >> processes[i].priority;

// processes[i].arrival\_time = 0;

}

else

{

processes[i].burst\_time = burst[i];

processes[i].arrival\_time = arrival[i];

processes[i].priority = priority[i];

// processes[i].arrival\_time = 0;

}

processes[i].cbt = processes[i].burst\_time;

processes[i].index = i + 1;

}

// sort wrt arrival,priority

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

{

for (int j = 0; j < n - 1; j++)

if (processes[j].arrival\_time > processes[j + 1].arrival\_time)

{

process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

int count = 0;

int time = processes[0].arrival\_time;

int times[100] = {time}, p\_ids[100];

while (length(n, processes) > 0)

{

// finding the next process to execute

int p;

bool idle = true;

for (p = 0; p < n && processes[p].burst\_time < 0; p++)

;

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

if (processes[i].burst\_time >= 0 && processes[i].arrival\_time <= time && processes[i].priority <= processes[p].priority)

{

idle = false;

if (processes[i].priority < processes[p].priority)

p = i;

}

if (idle)

{

int min = 9e8;

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

if (processes[i].arrival\_time >= time && processes[i].arrival\_time < min)

min = i;

p\_ids[count] = -1;

time = processes[min].arrival\_time;

}

else

{

p\_ids[count] = p;

processes[p].waiting\_time = time - processes[p].arrival\_time;

time += processes[p].burst\_time;

processes[p].burst\_time = -1;

processes[p].turn\_around\_time = time - processes[p].arrival\_time;

}

times[count + 1] = time;

count++;

}

cout << "\n\nGantt Chart\n+";

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

cout << "-------+";

cout << endl;

cout << "|";

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

{

cout.width(7);

if (p\_ids[i] != -1)

cout << std::left << "P" + to\_string(processes[p\_ids[i]].index) << "|";

else

cout << std::left << "Idel"

<< "|";

}

cout << "\n+";

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

cout << "-------+";

cout << endl;

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

{

cout.width(8);

cout << times[i];

}

cout << "\n\n\t\t\tAnalysis Table\n";

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

for (int j = 0; j < n - 1; j++)

if (processes[j].index > processes[j + 1].index)

{

process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

cout << "\nID | ARRIVAL TIME | PRIORITY | BURST TIME | WAITING TIME | TURN AROUND TIME |\n";

cout.fill('-');

cout.width(78);

cout << std::right << " " << endl;

cout.fill(' ');

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

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

{

cout.width(2);

cout << std::left << "P" + to\_string(processes[i].index) << " | ";

cout.width(12);

cout << std::left << processes[i].arrival\_time << " | ";

cout.width(8);

cout << std::left << processes[i].priority << " | ";

cout.width(10);

cout << std::left << processes[i].cbt << " | ";

cout.width(12);

cout << std::left << processes[i].waiting\_time << " | ";

cout.width(16);

cout << std::left << processes[i].turn\_around\_time << " | ";

cout << endl;

avg\_wt += processes[i].waiting\_time;

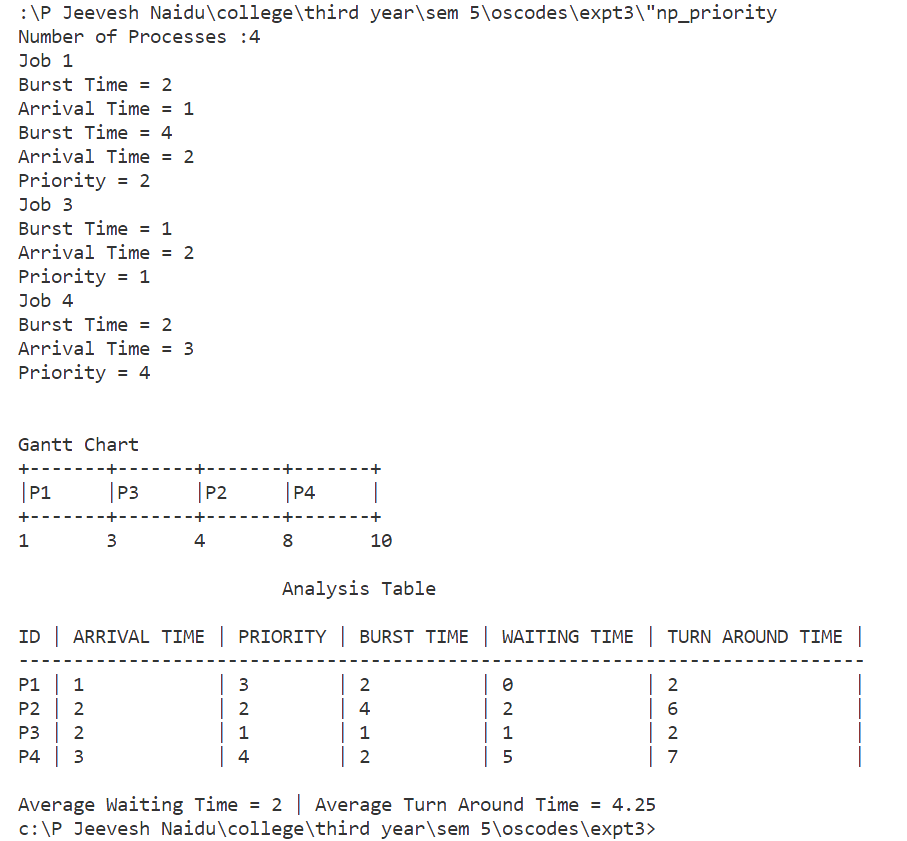
avg\_tat += processes[i].turn\_around\_time;

}

cout << "\nAverage Waiting Time = " << avg\_wt / n << " | Average Turn Around Time = " << avg\_tat / n;

}

**OUTPUT:**

****

**Conclusion:**

The non-preemptive following CPU scheduling algorithms:

1. FCFS- First Come First Serve
2. SJF- Shortest Job First
3. Non-Preemptive Priority

were successfully implemented in this experiment.