Experiment No: 3 **Date:**

NON – PRE-EMPTIVE CPU SCHEDULING ALGORITHM

Aim: a) To implement First come First Served CPU scheduling algorithm.

Theory:

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 turnaround 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. 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. 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.

Example:

Process	AT	BT
P1	1	2
P2	2	4
P3	2	1
P4	3	2

Gantt Chart:

P1	P2	P3	P4

1 3 7 8 10

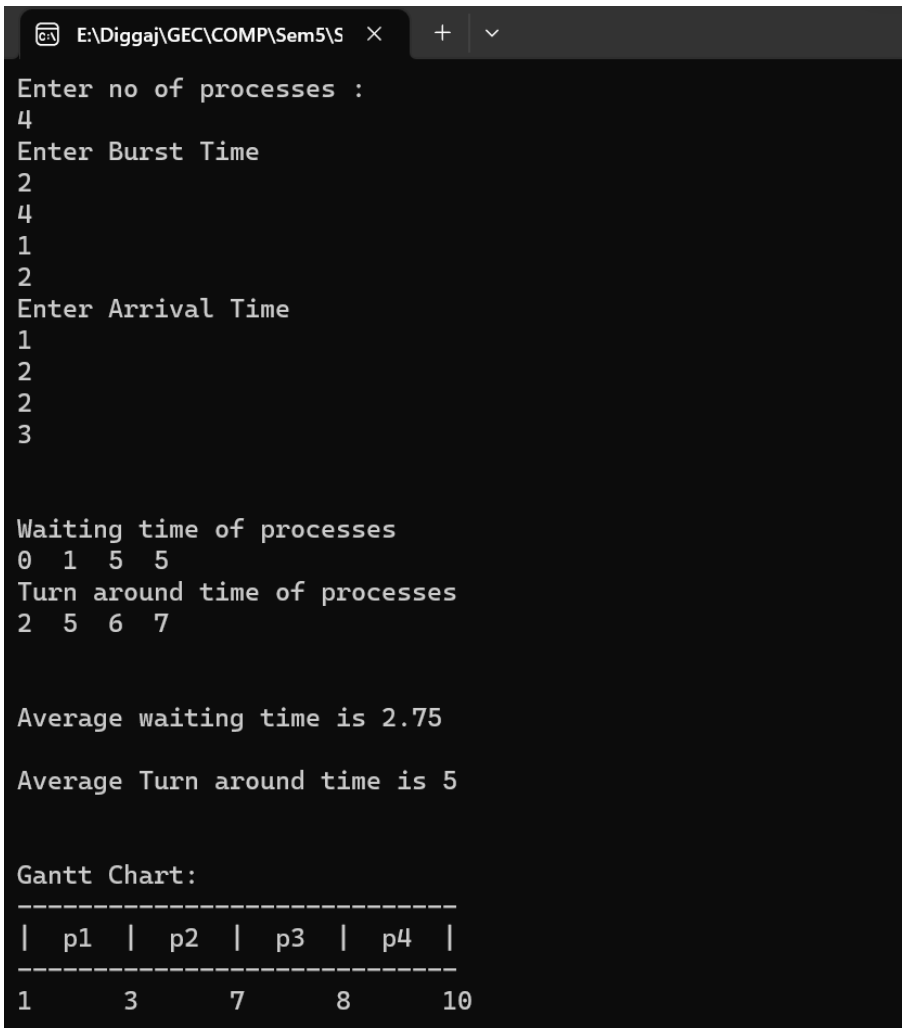
Code:

```
#include<bits/stdc++.h>
using namespace std;
void input_burst_time(int n ,int burst_time[]){
cout<<"Enter Burst Time"<<endl;
   for (int i = 0; i < n; i++){
      cin>>burst_time[i];
   }
}
void input_arrival_time(int n ,int arrival_time[]){
cout<<"Enter Arrival Time"<<endl;
   for (int i = 0; i < n; i++){
      cin>>arrival_time[i];
   }
}
void calculate_cp(int n,int burst_time[],int arrival_time[],int cp_time[]){
   cp_time[0] = arrival_time[0];
   cp_time[1] = arrival_time[0]+ burst_time[0];
   for(int i = 2; i <= n; i++){
   cp_time[i] = cp_time[i-1] + burst_time[i-1];
   }
}
void calculate_waiting(int n ,int wt_time[],int cp_time[],int arrival_time[]){
   wt_time[0]=0;
   for (int i = 1; i < n; i++){
      wt_time[i]= cp_time[i] - arrival_time[i];
   }
   cout<<endl;
}
void calulate_turn_around(int n , int wt_time[],int burst_time[],int turn_around[]){
   for (int i = 0; i < n; i++){
      turn_around[i] = wt_time[i] + burst_time[i];
   }
   cout<<endl;
}
void display(int n,int wt_time[],int turn_around[]){
   cout<<"Waiting time of processes"<<endl;
   for (int i = 0; i < n; i++){
      cout<<wt_time[i]<<"  ";
   }
    cout<<"\nTurn around time of processes"<<endl;
     for (int i = 0; i < n; i++){
      cout<<turn_around[i]<<"  ";
   }
   cout<<"\n";
}
void gchart(int n ,int burst_time[],int cp_time[]){
   cout<<"\nGantt Chart:"<<endl;
   int current_time = 0;
```

```
    int size=3;
    for (int i = 0; i < n*7; i++){
        cout<<'-';
    }
    cout<<'-';
    cout<<endl;
    cout<<"|  ";
    for (int i = 0; i < n; i++){
        cout<<'p'<<i+1<<"  |  ";
    }
    cout<<"\n";
    for (int i = 0; i < n*7; i++)
    {
        cout<<'-';
    }
    cout<<'-';
    cout<<endl;
     for (int i = 0; i <= n; i++)
    {
        cout<<cp_time[i]<<"      ";
    }
    cout<<"\n";
}
void avg_tat(int n,int turn_around[],int wt_time[]){
    float avg_sum =0.0;
    float avg_wt = 0.0;
    for (int i = 0; i < n; i++)
    {
            avg_sum += turn_around[i];
            avg_wt +=  wt_time[i];
    }
    cout<< "\nAverage waiting time is "<<(avg_wt/n)<<endl;
    cout<< "\nAverage Turn around time is "<<(avg_sum/n)<<endl;
}
int main(){
    int n; //no of processes
    //take input
    cout<<"Enter no of processes : "<<endl;
    cin>>n;
    int burst_time[n]={0};
    input_burst_time(n,burst_time);
    int arrival_time[n]={0};
    input_arrival_time(n,arrival_time);
    int cp_time[n]={0};
    calculate_cp(n,burst_time,arrival_time,cp_time);
    int wt_time[n]={0};
    calculate_waiting(n,wt_time,cp_time,arrival_time);
    int turn_around[n]={0};
    calulate_turn_around(n,wt_time,burst_time,turn_around);
```

```
    //dispalys wt and tat
    display(n,wt_time,turn_around);
    cout<<"\n";
    //calulate and display average

    avg_tat(n,turn_around,wt_time);
    cout<<"\n";
    //display gantt chart
    gchart(n,burst_time,cp_time);

}
```

Output



Conclusion

The First Come First Served CPU scheduling algorithm is studied and implemented successfully.

Experiment No: 3 **Date: 24/8/23**

NON – PRE-EMPTIVE CPU SCHEDULING ALGORITHM

Aim: b) To implement Shortest Job First CPU scheduling algorithm.

Theory:

Shortest Job First (SJF) is a CPU scheduling algorithm that aims to minimize the average waiting time for processes by selecting the process with the shortest burst time to execute next. The basic idea is to prioritize processes that require the least amount of CPU time, in order to reduce the waiting time for other processes in the queue. SJF can be implemented in both preemptive and non-preemptive variants.

1. Non-Preemptive SJF:

- When a new process arrives or the CPU becomes idle, the scheduler selects the process with the smallest burst time from the ready queue.

- The selected process continues to execute until it completes its burst time.

- After the process finishes, the scheduler selects the next shortest job from the remaining processes.

Advantages of SJF:

- Optimal for Minimizing Waiting Time: SJF scheduling, when non-preemptive, ensures that the process with the shortest burst time is executed first, which leads to the least average waiting time.

- Efficient Utilization: The algorithm tends to use the CPU efficiently by prioritizing short processes that release the CPU quickly.

Disadvantages of SJF:

- Starvation: Longer processes might suffer from starvation (i.e., not getting CPU time) if a continuous stream of short processes arrives.

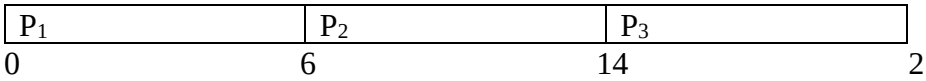
- Prediction Challenge: Determining the exact burst time of a process beforehand can be challenging, making it difficult to implement SJF in real-time systems.

- Preemption Overhead: Preemptive SJF introduces overhead due to frequent context switches when shorter processes arrive.

Example:

Process	AT	BT
P_1	**0**	**6**
P_2	**2**	**8**
P_3	**4**	**7**

Gantt Chart

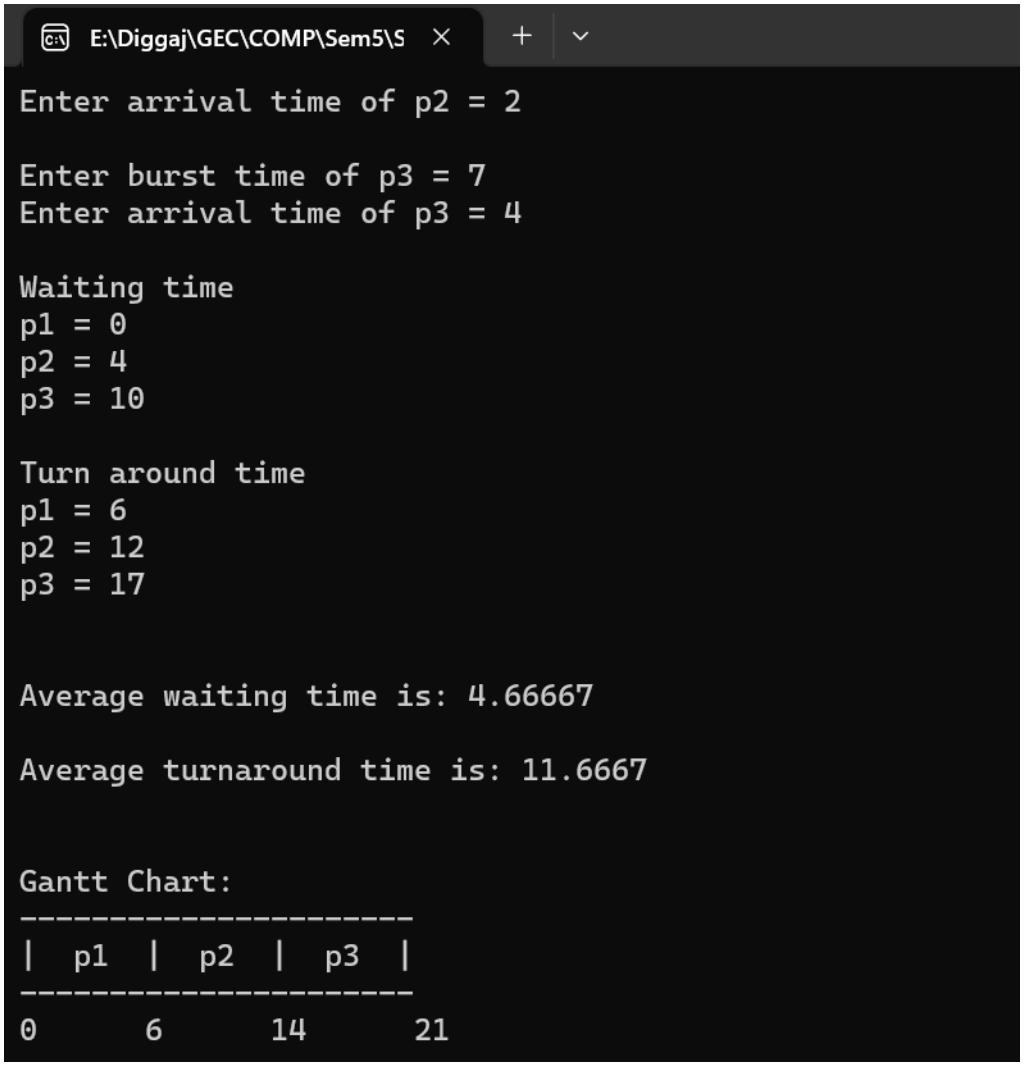


Code:

```
#include<bits/stdc++.h>

using namespace std;

class sjf{

public:

int process;

int burst_time;

int waiting_time;

int arrival_time;

int turnaround_time;

void getdata(int n);

void print_wait();

void print_tat();

sjf(){}

};

bool compareByValue(const sjf & a,const sjf & b){

if(a.arrival_time == b.arrival_time)

return a.burst_time < b.burst_time;

else{

return a.arrival_time < b.arrival_time;

}}

void sjf::print_wait(){

cout<<"p"<<process<<" =
"<<waiting_time<<endl;

}

void sjf::getdata(int n){

cout<<"Enter burst time of"<<" p"<<process<<" =
";

cin>>burst_time;

cout<<"Enter arrival time of"<<" p"<<process<<"
= ";

cin>>arrival_time;

cout<<endl;

}

void sjf::print_tat(){

cout<<"p"<<process<<" =
"<<turnaround_time<<endl;

}

void calculate_tat(int n,sjf p[]){

int completion_time= p[0].arrival_time;

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

completion_time+=p[i].burst_time;

p[i].turnaround_time = completion_time -
p[i].arrival_time;

}

}

void calculate_wt(int n,sjf p[]){

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

p[i].waiting_time= p[i].turnaround_time -
p[i].burst_time;
```

```
}
}
void gchart(int n, sjf process[]){
cout<<"\nGantt Chart:"<<endl;
int current_time = 0;
int size=3;
for (int i = 0; i < n*7; i++){
cout<<'-';
}
cout<<'-';
cout<<endl;
cout<<"|  ";
for (int i = 0; i < n; i++){
cout<<'p'<<process[i].process<<"  |  ";
}
cout<<"\n";
for (int i = 0; i < n*7; i++){
cout<<'-';
}
cout<<'-';
cout<<endl;
for (int i = 0; i <= n; i++) {
cout<<current_time<<"      ";
current_time+= process[i].burst_time;
}
cout<<"\n";
}
void calculate_avg(int n, sjf process[],float
&avg1,float &avg2){
//calulate the average
avg1=0;
avg2=0;
for(int i=0;i<n;i++){
avg1+= process[i].waiting_time;
avg2+=process[i].turnaround_time;
}
cout<<"Average waiting time is: "<<(avg1)/n<<"\n
\n";
cout<<"Average turnaround time is:
"<<(avg2)/n<<"\n \n";
}
int main(){
int n;
float avg_wt,avg_tat;
cout<<"Enter no of processes"<<endl;
cin>>n;
sjf process[n];
for (int i = 0; i < n; i++) {
process[i].process = i+1;
process[i].getdata(n);
}
//sort the array based on the burst_time
sort(process,process+n,compareByValue);
calculate_tat(n,process);
calculate_wt(n,process);
cout<<"Waiting time"<<endl;
for (int i = 0; i < n; i++){
process[i].print_wait();
}
cout<<"\n";
cout<<"Turn around time"<<endl;
for (int i = 0; i < n; i++){
process[i].print_tat();
}
cout<<"\n\n";
calculate_avg(n,process,avg_wt,avg_tat);
//dispaly gannt chart
gchart(n,process);
}
```

OUTPUT:



Conlucsion

The Shortest Job First (SJF) CPU scheduling algorithm is studied and implemented successfully.

NON – PRE-EMPTIVE CPU SCHEDULING ALGORITHM

Experiment No: 3 **Date: 07/9/23**

Aim: c) To implement non pre emptive priority CPU scheduling algorithm.

Theory:

Non-preemptive Priority Scheduling is a CPU scheduling algorithm in which each process is assigned a priority value, and the process with the highest priority is selected to run next. In this algorithm, once a process starts executing, it continues until it completes or blocks for I/O operations, regardless of whether higher-priority processes arrive in the meantime. The basic idea is to give preference to processes with higher priority, ensuring that higher-priority tasks are executed before lower-priority ones.

1. Priority Assignment: Each process is assigned a priority value based on some criteria. This priority value can be predefined or dynamically assigned based on factors like process type, importance, deadlines, etc. Lower numeric values often indicate higher priority (e.g., priority 1 is higher than priority 2).

2. Selection: When the CPU becomes available (either due to a process completion or when the system starts), the process with the highest priority is selected from the ready queue to execute.

3. Execution: The selected process runs until it completes its execution or blocks for I/O operations.

Advantages of Non-preemptive Priority Scheduling:

- Priority Control: This algorithm allows for fine-grained control over process priorities, enabling the system to focus on high-priority tasks.

- Predictable: The scheduling behavior is relatively predictable since the highest-priority process is always chosen next.

Disadvantages of Non-preemptive Priority Scheduling:

- Starvation: Lower-priority processes might suffer from starvation if higher-priority processes are constantly arriving in the system.

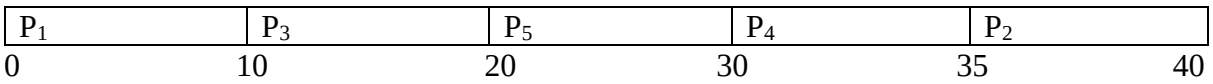
- Indefinite Blocking: If a higher-priority process is blocked (e.g., waiting for I/O), lower-priority processes won't be able to run until the higher-priority process completes.

- Inversion of Priority: The priority assigned to a process might not accurately represent its current needs. For example, a high-priority process waiting for a low-priority process to release a resource can lead to priority inversion.

Example:

Process	AT	BT	Priority
P1	0	10	1
P2	0	5	5
P3	5	10	2
P4	15	5	4
P5	18	10	3

Gantt Chart



Code:

```
#include <iostream>
using namespace std;
class sequencing{
    int burst_time;
    int arrival_time;
    int waiting_time;
    int turnaround_time;
    int status;
    int priority;
    int remaining_time;
public:
    void input(int);
    void FCFS(int, sequencing[]);
    void output(int);
      int minimum(int,sequencing[]);
};
#define pending 1
#define finished 2
void sequencing::input(int i){
    cout << "\n<<  P[" << i + 1 << "]  >>\nBURST TIME : ";
```

```
    cin >> burst_time;
    cout << "ARRIVAL TIME : ";
    cin >> arrival_time;
    cout<<"PRIORITY : ";
    cin>>priority;
    status=pending;
    remaining_time=burst_time;
}
void sequencing::output(int i){
    cout << "P[" << i + 1 << "]    \t" << burst_time << "\t\t" << arrival_time << "  \t" <<
waiting_time << "\t\t" << turnaround_time <<"\t\t"<<priority<< endl;
}
void sequencing::FCFS(int n, sequencing p[]){
    int total_time = minimum(n,p), k = 0;
    float avg_waiting_time = 0;
    float avg_turnaround_time = 0;
    cout << "\nGantt Chart:"<<endl;
      cout<<total_time;
    while(true){
       int pt=9999;
       for (int j = 0; j < n; j++){
          if (p[j].status == pending && p[j].priority < pt && p[j].arrival_time<=total_time){
             if(p[j].priority<pt) {
             k = j;
             pt=p[j].priority;
             }
             else{
                if(p[k].remaining_time>p[j].remaining_time) {
                k=j;
                pt=p[j].priority;
                }
```

```
            else{
               k=j;
               pt=p[j].priority;
            }}}}
      int j,check=0;
      for(j=0;j<p[k].remaining_time;j++){
         for(int l=0;l<n;l++){
            if(l!=k) {
               if(p[l].arrival_time==total_time + j) {
                  if(p[l].priority<=(p[k].priority)) {
                     if(p[l].priority==p[k].priority) {
                        if(p[k].remaining_time>p[l].remaining_time-j) {
                           check=1;
                           break;
                        } }
                     else{
                        check=1;
                        break;
                     }}}}}
         if(check)break;
      }
      total_time+=j;
      if(j==p[k].remaining_time) {
      p[k].status=finished;
      p[k].turnaround_time = total_time - p[k].arrival_time;
      avg_turnaround_time += p[k].turnaround_time;
      p[k].waiting_time= p[k].turnaround_time-p[k].burst_time;
      avg_waiting_time += p[k].waiting_time;
      }
      else{
```

```
            p[k].remaining_time-=j;
        }
        cout << "|__P[" << k + 1 << "]__|" << total_time;
        int if_finished=0;
        for(int i=0;i<n;i++)
        if(p[i].status==pending)
        if_finished=1;
        if(!if_finished)
        break;
    }
cout << "\n";
cout << "\nAVERAGE WAITING TIME : " << avg_waiting_time / n;
cout << "\nAVERAGE TURN AROUND TIME : " << avg_turnaround_time / n << endl;
}
int sequencing::minimum(int n,sequencing p[]){
    int  min=9999;
    for(int i=0;i<n;i++)
    if(p[i].arrival_time<min)
    min=p[i].arrival_time;
    return min;
}
int main(){
    int n;
    cout << "ENTER THE NUMBER OF JOBS : ";
    cin >> n;
    sequencing *p;
    p = new sequencing[n];
    for (int i = 0; i < n; i++)
        p[i].input(i);
    sequencing x;
```

```
    x.FCFS(n, p);
    cout << "\nPROCESS   BURST TIME   ARRIVALTIME  WAITINGTIME
TURNAROUND_TIME  PRIORITY:\n";
    for (int i = 0; i < n; i++)
        p[i].output(i);
    return 0;
}
```

Output:

```
C:\Users\digga\Downloads\pı
ENTER THE NUMBER OF JOBS : 5

<<  P[1]  >>
BURST TIME : 10
ARRIVAL TIME : 0
PRIORITY : 1

<<  P[2]  >>
BURST TIME : 5
ARRIVAL TIME : 0
PRIORITY : 5

<<  P[3]  >>
BURST TIME : 10
ARRIVAL TIME : 5
PRIORITY : 2

<<  P[4]  >>
BURST TIME : 5
ARRIVAL TIME : 15
PRIORITY : 4

<<  P[5]  >>
BURST TIME : 10
ARRIVAL TIME : 18
PRIORITY : 3

Gantt Chart:
0|__P[1]__|10|__P[3]__|20|__P[5]__|30|__P[4]__|35|__P[2]__|40

AVERAGE WAITING TIME : 11.4
AVERAGE TURN AROUND TIME : 19.4
```

PROCESS	BURST TIME	ARRIVALTIME	WAITINGTIME	TURNAROUND_TIME	PRIORITY:
P[1]	10	0	0	10	1
P[2]	5	0	35	40	5
P[3]	10	5	5	15	2
P[4]	5	15	15	20	4
P[5]	10	18	2	12	3

Conclusion:

Non pre emptive priority CPU scheduling algorithm was studied and implemented successfully.