PRE-EMPTIVE CPU SCHEDULING ALGORITHM

Experiment No: 4 **Date: /9/23**

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

Theory:

The Shortest Remaining Time Next (SRTN) CPU scheduling algorithm, also known as Shortest Job Next (SJN) or Preemptive Shortest Job First (PSJF), is a CPU scheduling algorithm used in operating systems. It is a preemptive scheduling algorithm that selects the process with the shortest remaining burst time to execute next. Here's an overview of SRTN, along with its advantages and disadvantages:

Algorithm Description:

When a process arrives in the ready queue, the operating system checks its burst time (the time it requires to complete its execution).The scheduler compares the burst time of the arriving process with the burst time of the currently executing process. If the new process has a shorter burst time than the currently executing process, it preempts the current process and starts executing the new one. The preemption involves saving the state of the currently running process and loading the state of the new process.The process continues to execute until it finishes or is preempted by another process with a shorter remaining burst time.

The scheduler periodically checks for any new arrivals and preempts the running process if a new process with a shorter burst time arrives.

This cycle continues until all processes have completed their execution.

Advantages:

Minimizes Average Waiting Time: SRTN minimizes the average waiting time because it selects the shortest job first, ensuring that processes with shorter burst times get executed before longer ones.

Optimal for Minimizing Turnaround Time: SRTN also minimizes the average turnaround time, as it prioritizes processes that can complete quickly.

Disadvantages:

Preemption Overhead: Frequent preemptions can introduce overhead due to the need to save and restore process states. This can increase the context-switching time and system load.

Starvation: SRTN can potentially lead to starvation of longer jobs. If there is a constant stream of short jobs arriving, longer jobs may never get a chance to execute.

Example:

Process	AT	BT	WT	TAT
P1	0	9	15	24
P2	1	3	0	3
P3	2	7	7	14
P4	3	5	1	6

Average waiting time : 5.75

Average turnaround time : 11.75

Gantt Chart:

P1	P2	P4	P3	P1

0 1 4 9 16 24

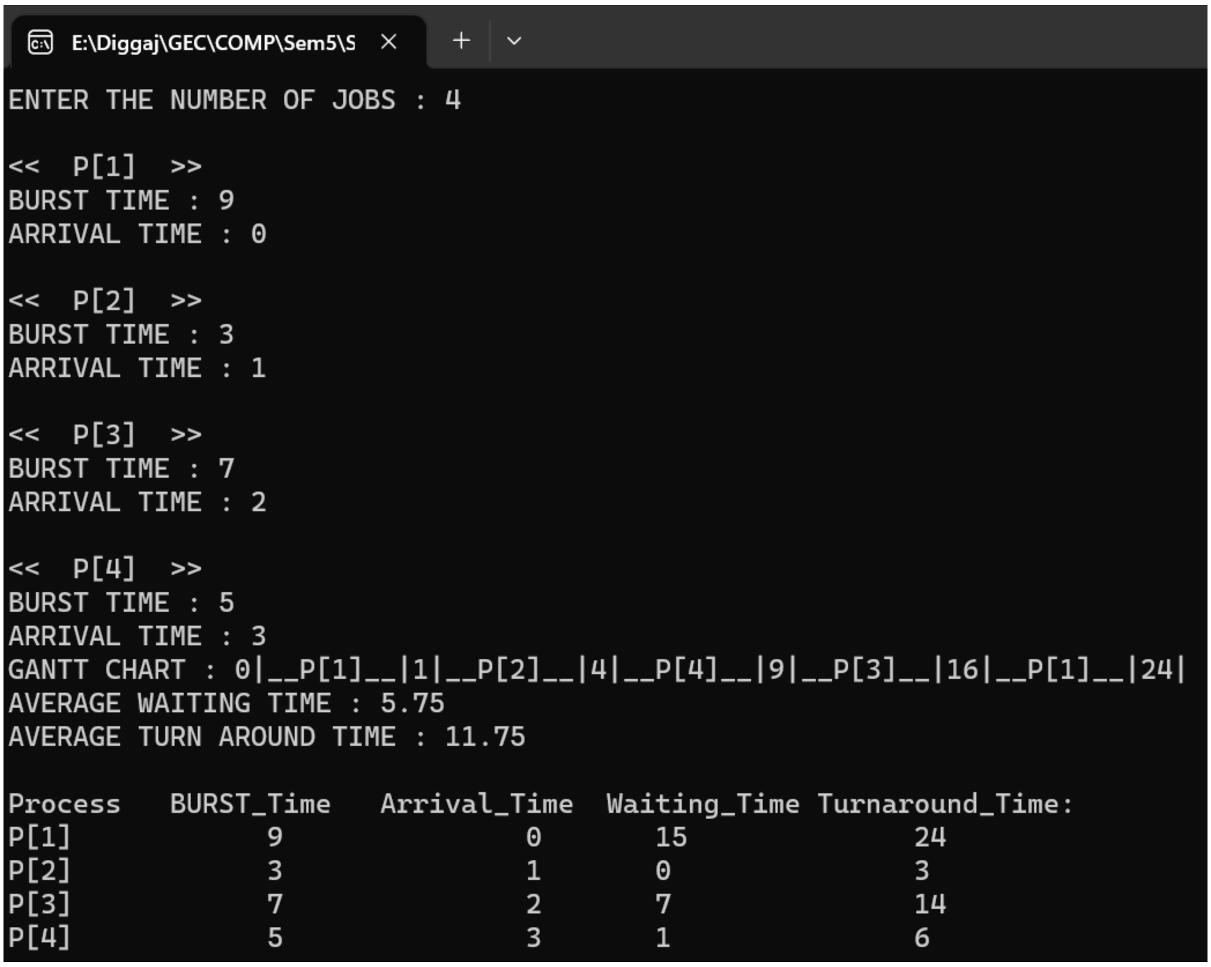
Code

```
#include <iostream>

using namespace std;

class sequencing{

int burst_time;

int arrival_time;

int waiting_time;

int turnaround_time;

int remaining_time;

int status;

public:

void input(int);

void SRTN(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;

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 << endl;

}

void sequencing::SRTN(int n, sequencing p[]){

int total_time = minimum(n,p), k = 0;

float avg_waiting_time = 0;

float avg_turnaround_time = 0;
```

```
cout << "GANTT CHART :
"<<total_time;

while(true){

int rt=9999;

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

if (p[j].status == pending &&
p[j].remaining_time < rt &&
p[j].arrival_time<=total_time){

k = j;

rt=p[j].remaining_time;

}}

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].remaining_time<(p[k].remaining_ti
me-j)){

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 << "|";

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.SRTN(n, p);
cout << "\nProcess   BURST_Time  Arrival_Time  Waiting_Time  Turnaround_Time:\n";
for (int i = 0; i < n; i++)
p[i].output(i);
return 0;
}
```

Output



E:\Diggaj\GEC\COMP\Sem5\S

```
ENTER THE NUMBER OF JOBS : 4

<<  P[1]  >>
BURST TIME : 9
ARRIVAL TIME : 0

<<  P[2]  >>
BURST TIME : 3
ARRIVAL TIME : 1

<<  P[3]  >>
BURST TIME : 7
ARRIVAL TIME : 2

<<  P[4]  >>
BURST TIME : 5
ARRIVAL TIME : 3
GANTT CHART : 0|__P[1]__|1|__P[2]__|4|__P[4]__|9|__P[3]__|16|__P[1]__|24|
AVERAGE WAITING TIME : 5.75
AVERAGE TURN AROUND TIME : 11.75

Process   BURST_Time   Arrival_Time  Waiting_Time Turnaround_Time:
P[1]          9               0       15              24
P[2]          3               1       0               3
P[3]          7               2       7               14
P[4]          5               3       1               6
```

Conclusion

Shortest Remaining Time Next (SRTN) CPU scheduling algorithm was studied and implemented successfully.

PRE-EMPTIVE CPU SCHEDULING ALGORITHM

Experiment No: 4 **Date: /9/23**

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

Theory:

Preemptive Priority CPU Scheduling is an 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 is given access to the CPU. If two or more processes have the same priority, a secondary criterion (such as arrival time or process ID) is used to break the tie.

Here's how the algorithm works:

When a new process arrives or a running process's priority changes, the scheduler checks if the new process has a higher priority than the currently running process. If it does, the CPU is preempted, and the higher-priority process is allowed to run. If a process with a higher priority arrives while another process is running, the running process is interrupted, and the CPU is allocated to the newly arrived process.

The interrupted process is moved to the ready queue and may be scheduled to run later when it has the highest priority among the ready processes.

Advantages:

Response Time: Preemptive Priority scheduling provides better response time for high-priority tasks. Critical tasks can start execution as soon as they arrive, leading to faster system responsiveness.

Priority Adjustment: As priorities can change dynamically, it allows for the adjustment of priorities based on the changing requirements of the system. This can help in managing real-time tasks effectively.

Disadvantages:

Starvation: There is a risk of lower-priority processes experiencing starvation. If high-priority processes keep arriving, lower-priority processes may never get a chance to run.

Complexity: Managing priorities and context switching in a preemptive environment can be complex and may require additional hardware support or sophisticated algorithms.

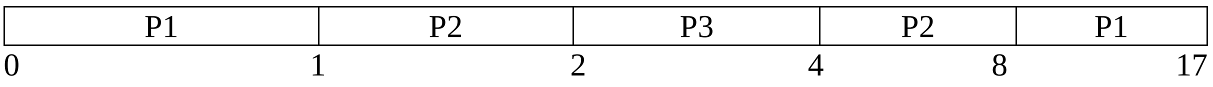
Example:

Process	AT	BT	Priority	WT	TAT
P1	0	10	3	7	17
P2	1	5	2	2	7
P3	2	2	1	0	2

Average waiting time : 3

Average turnaround time : 8.66

Gantt Chart:

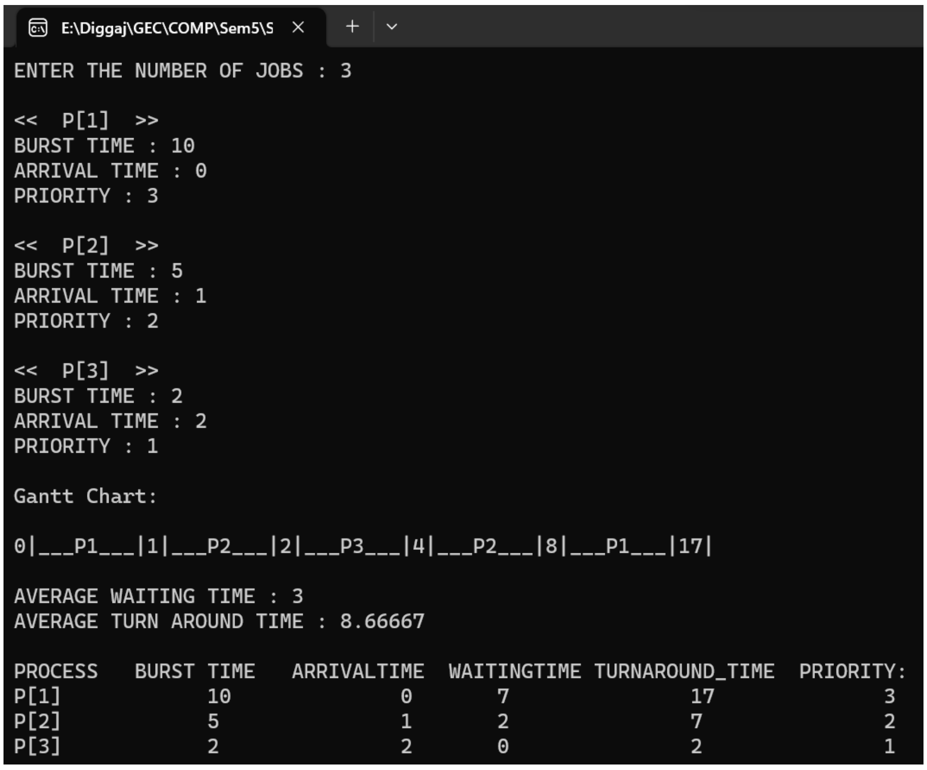


Code

```
#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 PRIPRE(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::PRIPRE(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<<endl;
cout<<"0|";
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_ti
me){
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_ti
me-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.PRIPRE(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


```
ENTER THE NUMBER OF JOBS : 3

<<  P[1]  >>
BURST TIME : 10
ARRIVAL TIME : 0
PRIORITY : 3

<<  P[2]  >>
BURST TIME : 5
ARRIVAL TIME : 1
PRIORITY : 2

<<  P[3]  >>
BURST TIME : 2
ARRIVAL TIME : 2
PRIORITY : 1

Gantt Chart:

0|___P1___|1|___P2___|2|___P3___|4|___P2___|8|___P1___|17|

AVERAGE WAITING TIME : 3
AVERAGE TURN AROUND TIME : 8.66667

PROCESS    BURST TIME   ARRIVALTIME  WAITINGTIME TURNAROUND_TIME  PRIORITY:
P[1]         10              0       7              17               3
P[2]         5               1       2              7                2
P[3]         2               2       0              2                1
```

Conclusion

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

PRE-EMPTIVE CPU SCHEDULING ALGORITHM

Experiment No: 4 **Date: /9/23**

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

Theory:

The Round Robin (RR) CPU scheduling algorithm is a pre-emptive scheduling algorithm widely used in operating systems. It is particularly suited for time-sharing and multitasking systems, where multiple processes compete for CPU time.

In the Round Robin CPU scheduling algorithm, each process is assigned a fixed time slice or quantum. The CPU scheduler allocates the CPU to the processes in a circular queue manner. Here's how it works:

Initialization: Assign each process a time quantum (e.g., 10 milliseconds) and initialize a ready queue with all processes.

Execution: The CPU scheduler selects the first process from the ready queue and runs it for its time quantum. If the process completes its execution within the time quantum, it is removed from the queue. If not, it is placed at the end of the queue, and the next process in the queue is selected for execution.

Preemption: RR is a pre-emptive scheduling algorithm, meaning that a running process is interrupted after its time quantum expires, and the CPU scheduler selects the next process to run.

Completion: The process continues to execute in a cyclic manner until all processes have completed their execution.

Advantages:

Fairness: RR provides fairness as every process gets a fair share of CPU time, preventing any single process from hogging the CPU for an extended period.

Low Response Time: It ensures that each process gets a chance to run quickly because processes are served in a time-sliced manner.

Disadvantages:

Overhead: There is an overhead associated with context switching (saving and restoring process state) each time the time quantum expires. This overhead can become significant if the time quantum is too small.

Inefficient for Long Tasks: RR is not efficient for long-running tasks because frequent context switches can reduce overall system performance.

Example:

Process	AT	BT	WT	TAT
P1	2	11	24	35
P2	3	5	19	24
P3	0	9	16	25
P4	1	4	11	15
P5	6	8	21	29

Average waiting time : 18.2

Average turnaround time : 25.6

Gantt Chart:

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

0 3 6 9 12 15 16 19 22 25 27 30 33 35 37

Code

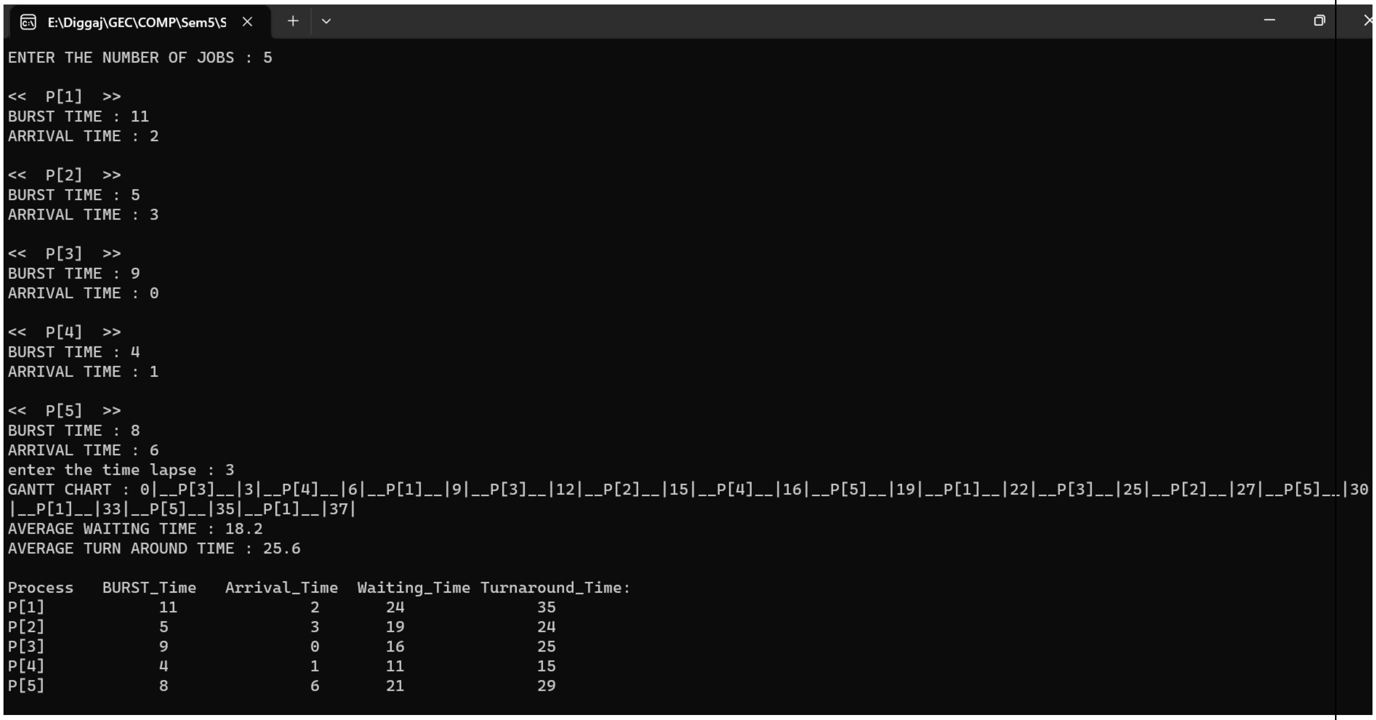
```
#include <iostream>
using namespace std;
class sequencing{
int burst_time;
int arrival_time;
int waiting_time;
int turnaround_time;
int remaining_time;
int status;
public:
void input(int);
void round_robin(int, sequencing[]);
void output(int);
int minimum(int,sequencing[]);
};
#define max 200
int q[max];
int front=-1;
int rear=-1;
int isempty();
int isfull();
int del();
void ins(int);
#define pending 1
#define finished 2
#define queued 3
void sequencing::input(int i){
cout << "\n<<  P[" << i + 1 << "]
>>\nBURST TIME : ";
cin >> burst_time;
cout << "ARRIVAL TIME : ";
cin >> arrival_time;
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 << endl;

}

void sequencing::round_robin(int n,
sequencing p[]){

int total_time =
minimum(n,p),k,time_lapse;

cout<<"enter the time lapse : ";

cin>>time_lapse;

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

if(p[i].arrival_time==total_time){

p[i].status=queued;

ins(i);

}}

float avg_waiting_time = 0;

float avg_turnaround_time = 0;

cout << "GANTT CHART :
"<<total_time;

while(true){

int j;

k=del();

int time;

time=(p[k].remaining_time<=time_lapse?p
[k].remaining_time:time_lapse);

for(j=0;j<time;j++){

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

if(l!=k){

if(p[l].status==pending){

if(p[l].arrival_time==total_time + j){

ins(l);

p[l].status=queued;

}}}}}

total_time+=time;

p[k].remaining_time-=time;

if(p[k].remaining_time==0){

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{

ins(k);

p[k].status=queued;

}

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

if(p[i].status==pending)

if(p[i].arrival_time==total_time){

ins(i);

p[i].status=queued;

}}

cout << "|__P[" << k + 1 << "]__|" <<
total_time;

int if_finished=0;

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

if(p[i].status==pending ||
p[i].status==queued)

if_finished=1;

if(!if_finished)
```

```
break;
}
cout << "|";
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.round_robin(n,p);
cout << "\nProcess   BURST_Time  Arrival_Time  Waiting_Time  Turnaround_Time:\n";
for (int i = 0; i < n; i++)
p[i].output(i);
return 0;
}
int isempty(){
if(front==-1)
return 1;
else
return 0;
}
int isfull(){
if(front==(rear+1)%max)
return 1;
else
return 0;
}
void ins(int a){
if(!isfull()){
q[++rear]=a;
if(front==-1)
front=0;
if(rear==max)
rear=0;
}
else{
printf("queue is full...\n");
}}
int del(){
if(!isempty()){
int x;
x=front;
```

```
if(front==rear){
front=rear=-1;
}
else{front++;}
return q[x];
}}
```

Output



```
ENTER THE NUMBER OF JOBS : 5

<<  P[1]  >>
BURST TIME : 11
ARRIVAL TIME : 2

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

<<  P[3]  >>
BURST TIME : 9
ARRIVAL TIME : 0

<<  P[4]  >>
BURST TIME : 4
ARRIVAL TIME : 1

<<  P[5]  >>
BURST TIME : 8
ARRIVAL TIME : 6
enter the time lapse : 3
GANTT CHART : 0|__P[3]__|3|__P[4]__|6|__P[1]__|9|__P[3]__|12|__P[2]__|15|__P[4]__|16|__P[5]__|19|__P[1]__|22|__P[3]__|25|__P[2]__|27|__P[5]__|30
|__P[1]__|33|__P[5]__|35|__P[1]__|37|
AVERAGE WAITING TIME : 18.2
AVERAGE TURN AROUND TIME : 25.6

Process   BURST_Time   Arrival_Time  Waiting_Time Turnaround_Time:
P[1]          11             2       24             35
P[2]          5              3       19             24
P[3]          9              0       16             25
P[4]          4              1       11             15
P[5]          8              6       21             29
```

Conclusion

Round Robin (RR) CPU scheduling algorithm was studied and implemented successfully.