OPERATING SYSTEM
IT - 204
LAB FILE

SUBMITTED BY: -
RITWIJ KASHYAP
2K21/IT/147

SUBMITTED TO: -
Dr. Ritu Agarwal

DEPARTMENT OF INFORMATION TECHNOLOGY
DELHI TECHNOLOGICAL UNIVERSITY
(FORMERLY DELHI COLLEGE OF ENGINEERING)
BAWANA ROAD, DELHI-110042

S. No	Title	Date	Signature

EXPERIMENT - 1

Aim: WAP to implement First Come First Serve (FCFS) CPU scheduling algorithm.

Description: 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.

Pseudocode:

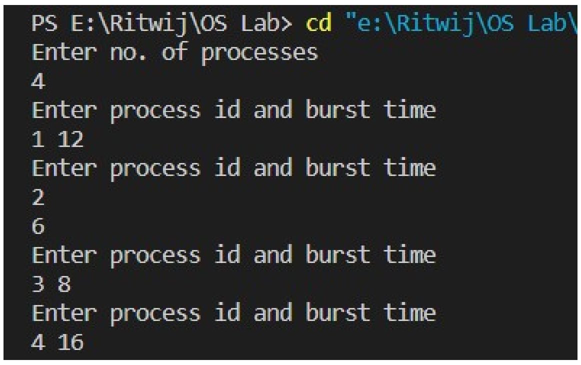
```
1- Input the processes along with their burst time (bt).
2- Find waiting time (wt) for all processes.
 3- As first process that comes need not to wait so
   waiting time for process 1 will be 0 i.e. wt[0] = 0.
4- Find waiting time for all other processes i.e. for
    process i ->
      wt[i] = bt[i-1] + wt[i-1] .
5- Find turnaround time = waiting_time + burst_time
   for all processes.
6- Find average waiting time =
            total_waiting_time / no_of_processes.
7- Similarly, find average turnaround time =
            total_turn_around_time / no_of_processes.
```

Implementation:

```cpp
#include <iostream>
using namespace std;
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
    wt[0] = 0;
    for (int i = 1; i < n; i++)
        wt[i] = bt[i - 1] + wt[i - 1];
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])
{

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

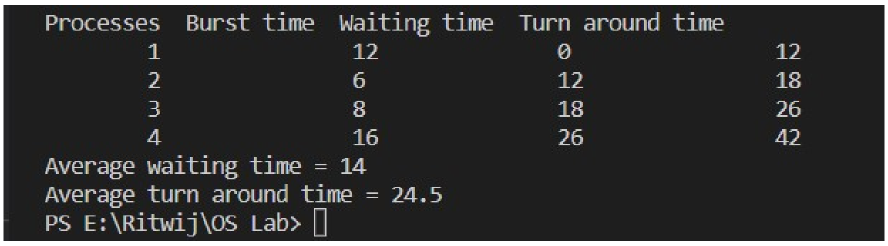
```cpp
        tat[i] = bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);
    cout << "Processes "
         << " Burst time "

         << " Waiting time "
         << " Turn around time\n";

    for (int i = 0; i < n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << "   " << i + 1 << "\t\t" << bt[i] << "\t    "
             << wt[i] << "\t\t " << tat[i] << endl;
    }
    cout << "Average waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}
int main()
{
    int n;
    cout << "Enter no. of processes" << endl;
    cin >> n;
    int processes[n];
    int burst_time[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter process id and burst time" << endl;
        cin >> processes[i];
        cin >> burst_time[i];
    }
    findavgTime(processes, n, burst_time);
}
```

Input :



```
PS E:\Ritwij\OS Lab> cd "e:\Ritwij\OS Lab\
Enter no. of processes
4
Enter process id and burst time
1 12
Enter process id and burst time
2
6
Enter process id and burst time
3 8
Enter process id and burst time
4 16
```

Output:



```
Processes  Burst time  Waiting time  Turn around time
       1              12             0               12
       2              6              12              18
       3              8              18              26
       4              16             26              42
Average waiting time = 14
Average turn around time = 24.5
PS E:\Ritwij\OS Lab> 
```

Conclusion :

From this practical, we learned about First Come and First Serve algorithm and how to implement and find waiting time using the given burst time for the case where arrival time is considered the same.

EXPERIMENT - 2

AIM: Write a Program to implement the Shortest Job First (SJF) CPU Scheduling Algorithm.

Description: The Shortest Job First (SJF) CPU scheduling algorithm is a non-pre-emptive scheduling algorithm that selects the process with the shortest burst time from the ready queue for execution. The idea behind this algorithm is to minimize the average waiting time of processes by giving priority to shorter jobs.

Pseudocode:

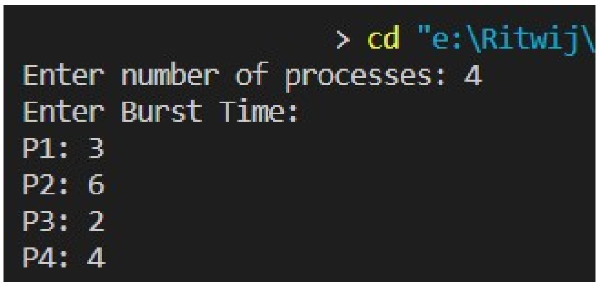
```
1. Firstly, Start process.
2. Declare array size i.e. A[10].
3. Take number of elements to be inserted.
4. Select process which have shortest burst time Among all process will execute first.
5. If process have same burst time length then FCFS ( First come First Serve ) scheduling
   algorithm used.
6. Make average waiting time length of next process.
7. Start with first process, selection as above and other processes are to be in queue.
8. Calculates Burst total number of time.
9. Display the Related values.
10. Now Close / Stop process.
```

Implementation:

```
#include <iostream>
using namespace std;
int main()
{

   int A[100][4];
   int i, j, n, total = 0, index, temp;
   float avg_wt, avg_tat;
   cout << "Enter number of processes: ";
   cin >> n;
   cout << "Enter Burst Time:\n";

   for (i = 0; i < n; i++) {
      cout << "P" << i + 1 << ": ";
      cin >> A[i][1];
      A[i][0] = i + 1;
   }

   for (i = 0; i < n; i++) {
      index = i;
      for (j = i + 1; j < n; j++)
```

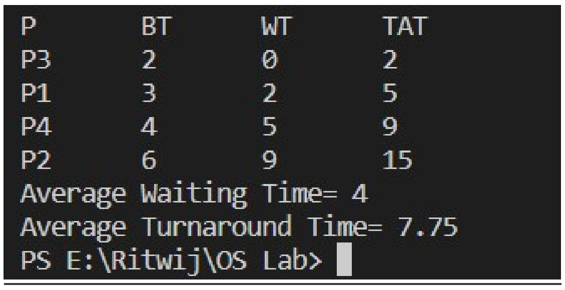
```
            if (A[j][1] < A[index][1])
                index = j;
        temp = A[i][1];
        A[i][1] = A[index][1];
        A[index][1] = temp;

        temp = A[i][0];
        A[i][0] = A[index][0];
        A[index][0] = temp;
    }
    A[0][2] = 0;

    for (i = 1; i < n; i++) {
        A[i][2] = 0;
        for (j = 0; j < i; j++)
            A[i][2] += A[j][1];
        total += A[i][2];
    }
    avg_wt = (float)total / n;
    total = 0;
    cout << "P     BT     WT     TAT\n";


    for (i = 0; i < n; i++) {
        A[i][3] = A[i][1] + A[i][2];
        total += A[i][3];
        cout << "P" << A[i][0] << "     " << A[i][1] << "     " << A[i][2] << "      " << A[i][3] << endl;
    }
    avg_tat = (float)total / n;
    cout << "Average Waiting Time= " << avg_wt;
    cout << "\nAverage Turnaround Time= " << avg_tat;
}
```

Input:



```
                    > cd "e:\Ritwij\
Enter number of processes: 4
Enter Burst Time:
P1: 3
P2: 6
P3: 2
P4: 4
```

Output:



```
P       BT      WT      TAT
P3      2       0       2
P1      3       2       5
P4      4       5       9
P2      6       9       15
Average Waiting Time= 4
Average Turnaround Time= 7.75
PS E:\Ritwij\OS Lab>
```

Conclusion:
From this practical, we learned about Shortest Job First (SJF) CPU Scheduling Algorithm and how to Implement it and fnd waiting time using the given burst time for the case where arrival time is considered the same.

EXPERIMENT - 3

Aim: WAP to implement Shortest Job First with remaining time (pre-emptive)

Description: In Shortest Remaining Time First CPU scheduling algorithm, the process with smallest amount of remaining time until completion is selected to perform frst. This scheduling algorithm can be pre-emptive or non-pre-emptive.

Pseudocode:

```
● Traverse until all process gets completely executed.
    ○ Find process with minimum remaining time at every single time lap.
    ○ Reduce its time by 1.
    ○ Check if its remaining time becomes 0
    ○ Increment the counter of process completion.
    ○ Completion time of current process = current_time + 1;
    ○ Calculate waiting time for each completed process.
        ■ wt[i]= Completion time – arrival_time-burst_time
    ○ Increment time lap by one.
● Find turnaround time (waiting_time + burst_time).
```

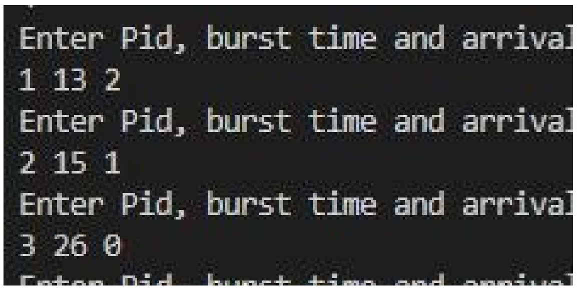
Implementation:

```
#include <bits/stdc++.h>
using namespace std;

struct Process {
   int pid; // Process ID
   int bt; // Burst Time
   int art; // Arrival Time
};

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n,
                                int wt[])
{
   int rt[n];

   // Copy the burst time into
   rt[] for (int i = 0; i < n; i++)
      rt[i] = proc[i].bt;
```

```
int complete = 0, t = 0, minm = INT_MAX;
int shortest = 0, finish_time;
bool check = false;

// Process until all processes gets
// completed
while (complete!= n) {

    // Find process with minimum
    // remaining time among the
    // processes that arrives till the
    // current time`
    for (int j = 0; j < n; j++) {
        if ((proc[j].art <= t) &&
        (rt[j] < minm) && rt[j] > 0) {
            minm = rt[j];
            shortest = j;
            check = true;
        }
    }

    if (check == false) {
        t++;
        continue;
    }

    // Reduce remaining time by one
    rt[shortest]--;

    // Update minimum
    minm = rt[shortest];
    if (minm == 0)
        minm = INT_MAX;

    // If a process gets completely
    // executed
    if (rt[shortest] == 0) {

        // Increment complete
        complete++;
        check = false;

        // Find finish time of current
        // process
        finish_time = t + 1;

        // Calculate waiting time
        wt[shortest] = finish_time -
                    proc[shortest].bt -
                    proc[shortest].art;

        if (wt[shortest] < 0)
            wt[shortest] = 0;
        }
        // Increment time
        t++;
    }
}

// Function to calculate turn around time
 void findTurnAroundTime(Process proc[], int n,
                        int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}

// Function to calculate average time
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt =
                0, total_tat = 0;

    // Function to find waiting time of all
    // processes
    findWaitingTime(proc, n, wt);

    // Function to find turn around time for
    // all processes
    findTurnAroundTime(proc, n, wt, tat);

    // Display processes along with all
    // details
    cout << " P\t\t"
        << "BT\t\t"
        << "WT\t\t"
        << "TAT\t\t\n";

    // Calculate total waiting time and
    // total turnaround time
    for (int i = 0; i < n; i++) {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << proc[i].pid << "\t\t"
            << proc[i].bt << "\t\t " << wt[i]
            << "\t\t " << tat[i] << endl;
    }

    cout << "\nAverage waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
```

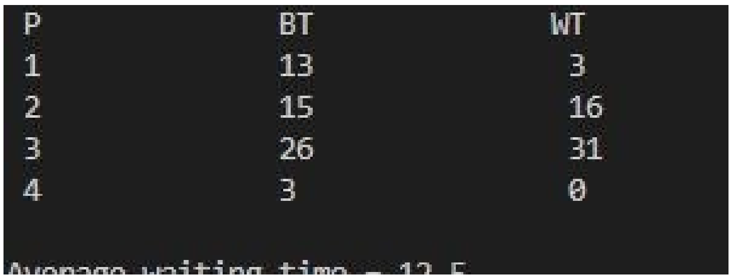
```
}

// Driver code
int main()
{
   int n;
   cout<<"Enter number of
   processes"<<endl; cin>>n;
   Process proc[n];

   for(int i=0; i<n; i++){
      cout<<"Enter Pid, burst time and arrival time"<<endl;
      cin>>proc[i].pid;
      cin>>proc[i].bt;
      cin>>proc[i].art;
   }


   findavgTime(proc, n);
   return 0;
}
```

<u>Input :</u>



```
Enter Pid, burst time and arrival
1 13 2
Enter Pid, burst time and arrival
2 15 1
Enter Pid, burst time and arrival
3 26 0
Enter Pid, burst time and arrival
```

<u>Output:</u>



```
 P              BT              WT
 1              13               3
 2              15               16
 3              26               31
 4              3                0

Average waiting time = 12.5
```

<u>Conclusion:</u>

From this practical, we learned about Shortest Remaining Time First Algorithm with pre-emptive operation and how to implement and find waiting time using the given burst time and arrival Time.

EXPERIMENT - 4

AIM: Write a program to implement the Priority CPU Scheduling Algorithm.

Description: Priority CPU Scheduling Algorithm is a non-pre-emptive scheduling algorithm in which the process with the highest priority is selected for execution. Each process is assigned a priority, and the scheduler selects the process with the highest priority to run next. Priority can be based on various factors, such as the amount of resources the process needs, the time sensitivity of the process, or the importance of the process. The priority can be assigned either internally by the operating system or externally by the user. In Priority CPU Scheduling Algorithm, if two or more processes have the same priority, the algorithm follows a First Come First Serve (FCFS) order to decide which process should be executed first.

Pseudocode:

```
1- First input the processes with their burst time
  and priority.
2- Sort the processes, burst time and priority
  according to the priority.
3- Now simply apply FCFS algorithm.
```

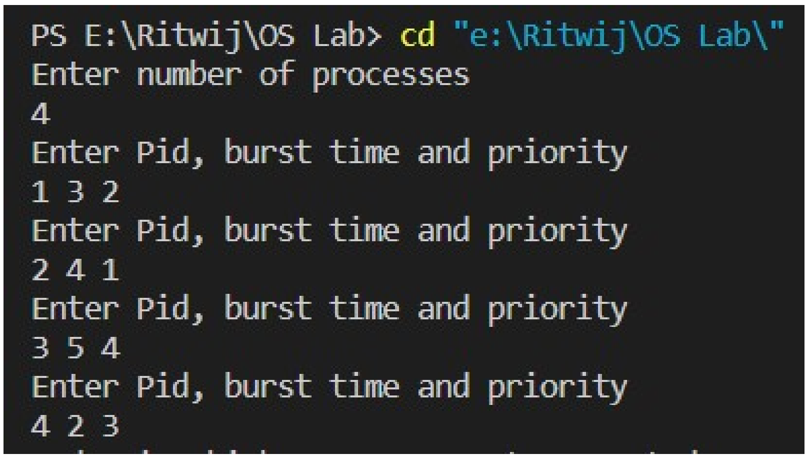
Implementation:

```
#include<bits/stdc++.h>
using namespace std;

struct Process
  {
    int pid;  // Process ID
    int bt;   // CPU Burst time required
    int priority; // Priority of this process
};

// Function to sort the Process acc. to priority
bool comparison(Process a, Process b)
{
    return (a.priority > b.priority);
}

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n,
                    int wt[])
```

```
{
    // waiting time for first process is 0
    wt[0] = 0;

    // calculating waiting time
    for (int i = 1; i < n ; i++ )
        wt[i] = proc[i-1].bt + wt[i-1] ;
}

// Function to calculate turnaround time
void findTurnAroundTime(Process proc[], int n,
                        int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
        tat[i] = proc[i].bt + wt[i];
}

//Function to calculate average time
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    //Function to find waiting time of all processes
    findWaitingTime(proc, n, wt);

    //Function to find turn around time for all processes
    findTurnAroundTime(proc, n, wt, tat);

    //Display processes along with all details
    cout << "\nProcesses "<< " Burst time "
         << " Waiting time " << " Turnaround time\n";

    // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << "   " << proc[i].pid << "\t\t"
             << proc[i].bt << "\t    " << wt[i]
             << "\t\t " << tat[i] <<endl;
    }

    cout << "\nAverage waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turnaround time = "
         << (float)total_tat / (float)n;
}
```

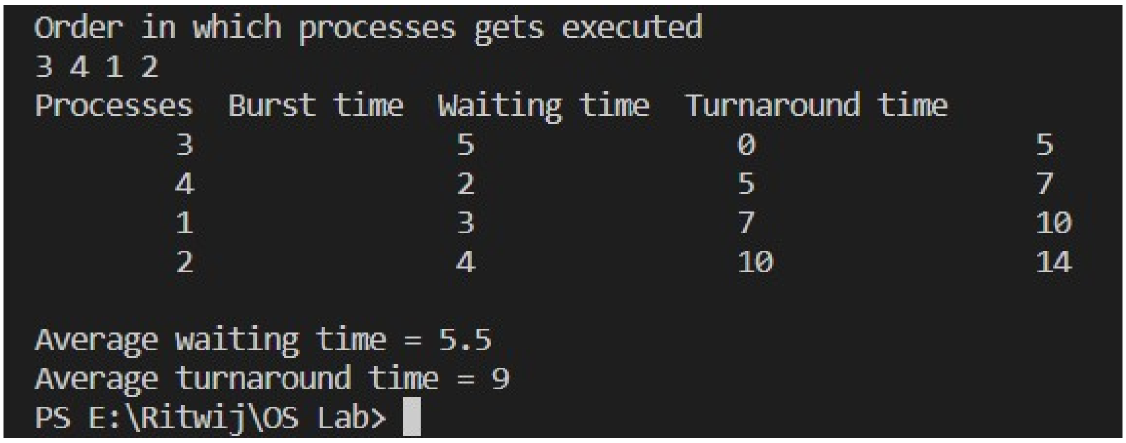
```
void priorityScheduling(Process proc[], int n)
{
    // Sort processes by priority
    sort(proc, proc + n, comparison);

    cout<< "Order in which processes gets executed
    \n"; for (int i = 0 ; i < n; i++)
        cout << proc[i].pid <<" " ;

    findavgTime(proc, n);
}

// Driver code
int main()
{
    int n;
    cout<<"Enter number of
    processes"<<endl; cin>>n;
    Process proc[n];

    for(int i=0; i<n; i++){
        cout<<"Enter Pid, burst time and priority"<<endl;
        cin>>proc[i].pid;
        cin>>proc[i].bt;
        cin>>proc[i].priority;
    }

    priorityScheduling(proc, n);
    return 0;
}
```

Input :



```
PS E:\Ritwij\OS Lab> cd "e:\Ritwij\OS Lab\"
Enter number of processes
4
Enter Pid, burst time and priority
1 3 2
Enter Pid, burst time and priority
2 4 1
Enter Pid, burst time and priority
3 5 4
Enter Pid, burst time and priority
4 2 3
```

Output :



```
Order in which processes gets executed
3 4 1 2
Processes  Burst time  Waiting time  Turnaround time
        3             5               0                5
        4             2               5                7
        1             3               7                10
        2             4               10               14

Average waiting time = 5.5
Average turnaround time = 9
PS E:\Ritwij\OS Lab>
```

Conclusion :
From this practical, we learned about Priority CPU Scheduling Algorithm and how to Implement it and find waiting time using the given burst time.

EXPERIMENT - 5

AIM : Write a program to implement the Priority CPU Scheduling Algorithm(preemptively).

Description:

The Priority CPU Scheduling Algorithm (preemptive) is a scheduling algorithm used in operating systems to determine the order in which processes are executed by the CPU. In this algorithm, each process is assigned a priority, which determines the order in which it is executed. Processes with higher priority are executed before processes with lower priority. In preemptive priority scheduling, if a higher-priority process arrives during the execution of a lower-priority process, the lower-priority process is preempted and the higher-priority process is executed instead. The preempted process is added back to the ready queue, and the algorithm continues selecting processes based on their priority.

Pseudocode:

```
● Initialize the current time to 0, an empty ready queue, and a completed queue.
● While there are still uncompleted processes:
    a. Add all processes that have arrived to the ready queue.
    b. Sort the ready queue in non-decreasing order by priority.
    c. Select the process with the highest priority from the ready queue.
    d. Run the process for one time unit.
    e. If the process has completed, set its completion time and waiting time, and
       move it to the completed queue.
    f. If the process has not completed, add it back to the ready queue.
    g. Increment the current time by one time unit.
● Output the completion time and waiting time for each process in the completed queue
```

Implementation:

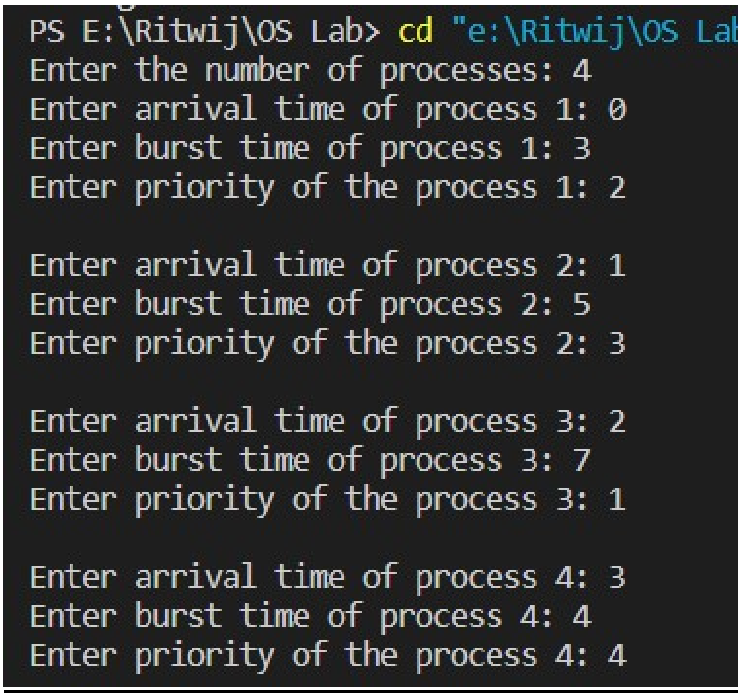
```
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <string.h>
using namespace std;

struct process {
   int pid;
   int arrival_time;
   int burst_time;
```

```
    int priority;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int response_time;
};

int main() {

    int n;
    struct process p[100];
    float avg_turnaround_time;
    float avg_waiting_time;
    float avg_response_time;
    float cpu_utilisation;
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int total_response_time = 0;
    int total_idle_time = 0;
    float throughput;
    int burst_remaining[100];
    int is_completed[100];
    memset(is_completed,0,sizeof(is_completed));

    cout << setprecision(2) << fixed;

    cout<<"Enter the number of processes: ";
    cin>>n;

    for(int i = 0; i < n; i++) {
        cout<<"Enter arrival time of process "<<i+1<<":
        "; cin>>p[i].arrival_time;
        cout<<"Enter burst time of process "<<i+1<<": ";
        cin>>p[i].burst_time;
        cout<<"Enter priority of the process "<<i+1<<": ";
        cin>>p[i].priority;
        p[i].pid = i+1;
        burst_remaining[i] = p[i].burst_time;
        cout<<endl;
    }

    int current_time = 0;
    int completed = 0;
    int prev = 0;

    while(completed != n) {
        int idx = -1;
        int mx = -1;
        for(int i = 0; i < n; i++) {
            if(p[i].arrival_time <= current_time && is_completed[i] == 0) {
```

```
            if(p[i].priority > mx) {
                mx = p[i].priority;
                idx = i;
            }
            if(p[i].priority == mx) {
                if(p[i].arrival_time < p[idx].arrival_time) {
                    mx = p[i].priority;
                    idx = i;
                }
            }
        }
    }

    if(idx != -1) {
        if(burst_remaining[idx] == p[idx].burst_time) {
            p[idx].start_time = current_time;
            total_idle_time += p[idx].start_time - prev;
        }
        burst_remaining[idx] -= 1;
        current_time++;
        prev = current_time;

        if(burst_remaining[idx] == 0) {
            p[idx].completion_time = current_time;
            p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
            p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;
            p[idx].response_time = p[idx].start_time - p[idx].arrival_time;

            total_turnaround_time += p[idx].turnaround_time;
            total_waiting_time += p[idx].waiting_time;
            total_response_time += p[idx].response_time;

            is_completed[idx] = 1;
            completed++;
        }
    }
    else {
         current_time++;
    }
}

int min_arrival_time = 10000000;
int max_completion_time = -1;
for(int i = 0; i < n; i++) {
    min_arrival_time = min(min_arrival_time,p[i].arrival_time);
    max_completion_time = max(max_completion_time,p[i].completion_time);
}

avg_turnaround_time = (float) total_turnaround_time /
n; avg_waiting_time = (float) total_waiting_time / n;
avg_response_time = (float) total_response_time / n;
```

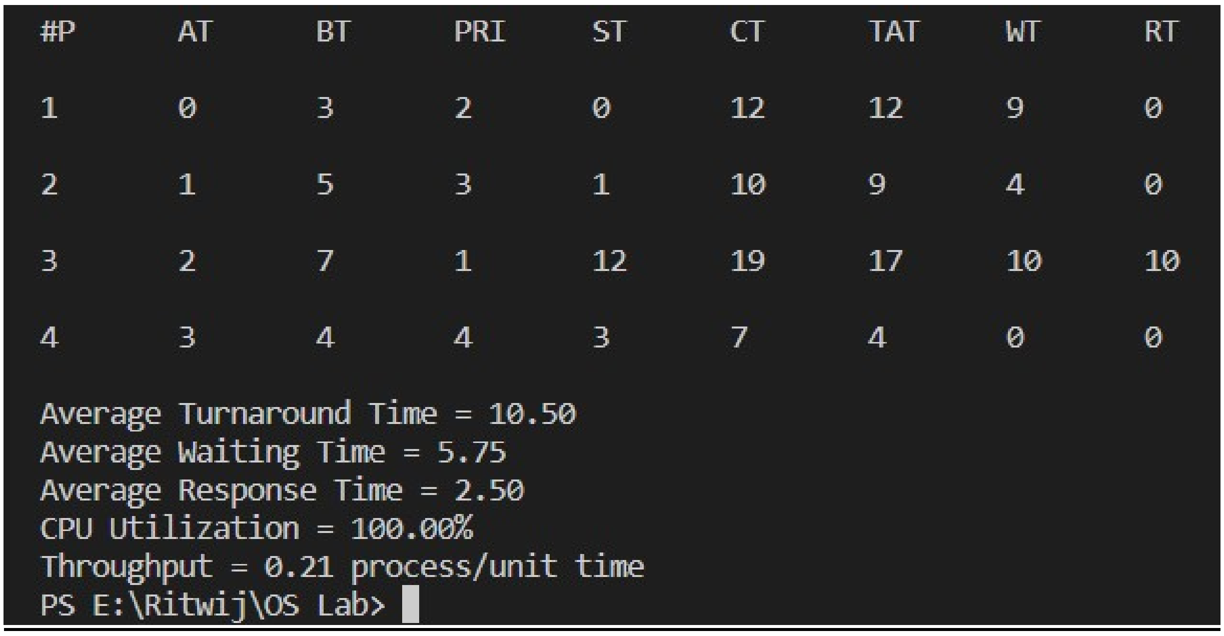
```
    cpu_utilisation = ((max_completion_time - total_idle_time) / (float)
max_completion_time )*100;
    throughput = float(n) / (max_completion_time - min_arrival_time);

    cout<<endl<<endl;


cout<<"#P\t"<<"AT\t"<<"BT\t"<<"PRI\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl;

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

cout<<p[i].pid<<"\t"<<p[i].arrival_time<<"\t"<<p[i].burst_time<<"\t"<<p[i].priority<<"\t"<<p[i].start_t

ime<<"\t"<<p[i].completion_time<<"\t"<<p[i].turnaround_time<<"\t"<<p[i].waiting_time<<"\t"<<p[i] .response_time<<"\t"<<"\n"<<endl;

    }
    cout<<"Average Turnaround Time = "<<avg_turnaround_time<<endl;
    cout<<"Average Waiting Time = "<<avg_waiting_time<<endl;
    cout<<"Average Response Time = "<<avg_response_time<<endl;
    cout<<"CPU Utilization = "<<cpu_utilisation<<"%"<<endl;
    cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;


}
```

Input:



```
PS E:\Ritwij\OS Lab> cd "e:\Ritwij\OS Lab
Enter the number of processes: 4
Enter arrival time of process 1: 0
Enter burst time of process 1: 3
Enter priority of the process 1: 2

Enter arrival time of process 2: 1
Enter burst time of process 2: 5
Enter priority of the process 2: 3

Enter arrival time of process 3: 2
Enter burst time of process 3: 7
Enter priority of the process 3: 1

Enter arrival time of process 4: 3
Enter burst time of process 4: 4
Enter priority of the process 4: 4
```

Output:



```
#P      AT      BT      PRI     ST      CT      TAT     WT      RT

1       0       3       2       0       12      12      9       0

2       1       5       3       1       10      9       4       0

3       2       7       1       12      19      17      10      10

4       3       4       4       3       7       4       0       0

Average Turnaround Time = 10.50
Average Waiting Time = 5.75
Average Response Time = 2.50
CPU Utilization = 100.00%
Throughput = 0.21 process/unit time
PS E:\Ritwij\OS Lab>
```

Conclusion:
From this practical, we learned about Priority CPU Scheduling Algorithm(preemptively) and how to Implement it and find waiting time using the given burst time.

EXPERIMENT - 6

AIM: Write a program to implement the Round Robin Scheduling Algorithm.

Description:

A round-robin scheduling algorithm is used to schedule the process fairly for each job a time slot or quantum and the interrupting the job if it is not completed by then the job come after the other job which is arrived in the quantum time that makes these scheduling fairly.

Pseudocode:

```
1. Declare arrival[], burst[], wait[], turn[] arrays and initialize them. Also declare a timer
variable and initialize it to zero. To sustain the original burst array create another array
(temp_burst[]) and copy all the values of burst array in it.
2. To keep a check we create another array of bool type which keeps the record of whether
a process is completed or not. we also need to maintain a queue array which contains the
process indices (initially the array is filled with 0).
3. Now we increment the timer variable until the first process arrives and when it does, we add
the process index to the queue array
4. Now we execute the first process until the time quanta and during that time quanta, we
check whether any other process has arrived or not and if it has then we add the index in the
queue (by calling the fxn. queueUpdation()).
5. Now, after doing the above steps if a process has finished, we store its exit time and execute
the next process in the queue array. Else, we move the currently executed process at the end
of the queue (by calling another fxn. queueMaintainence()) when the time slice expires.
6. The above steps are then repeated until all the processes have been completely executed.
If a scenario arises where there are some processes left but they have not arrived yet, then we
shall wait and the CPU will remain idle during this interval.
```

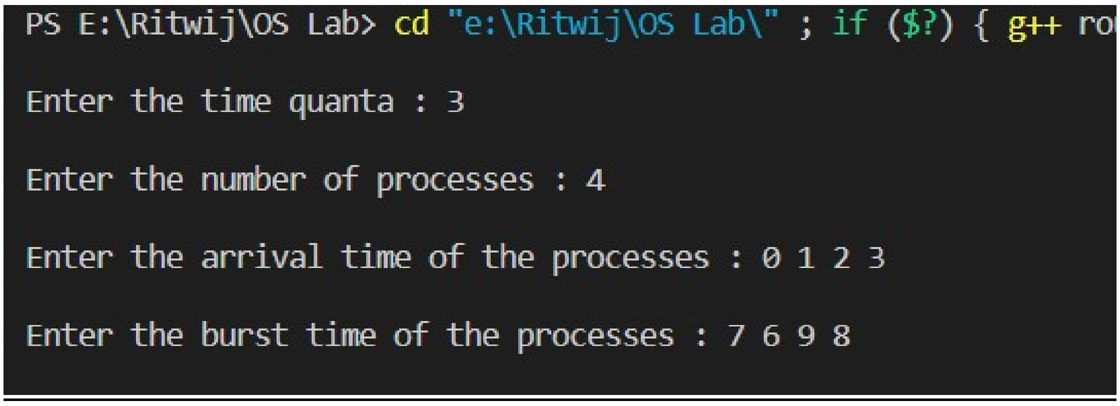
Implementation:

```
#include <iostream>

using namespace std;

void queueUpdation(int queue[],int timer,int arrival[],int n, int
   maxProccessIndex){ int zeroIndex;
   for(int i = 0; i < n; i++){
      if(queue[i] == 0){
         zeroIndex = i;
         break;
      }
   }
```

```
    queue[zeroIndex] = maxProccessIndex + 1;
}

void queueMaintainence(int queue[], int n){ for(int
    i = 0; (i < n-1) && (queue[i+1] != 0) ; i++){
        int temp = queue[i];
        queue[i] = queue[i+1];
        queue[i+1] = temp;
    }
}

void checkNewArrival(int timer, int arrival[], int n, int maxProccessIndex,int
    queue[]){ if(timer <= arrival[n-1]){
       bool newArrival = false;
       for(int j = (maxProccessIndex+1); j < n; j++){
             if(arrival[j] <= timer){
              if(maxProccessIndex < j){
                 maxProccessIndex = j;
                 newArrival = true;
              }
            }
       }
       //adds the incoming process to the ready queue
       //(if any arrives)
       if(newArrival)
          queueUpdation(queue,timer,arrival,n, maxProccessIndex);
    }
}

//Driver Code
int main(){
    int n,tq, timer = 0, maxProccessIndex = 0;
    float avgWait = 0, avgTT = 0;
    cout << "\nEnter the time quanta : ";
    cin>>tq;
    cout << "\nEnter the number of processes : ";
    cin>>n;
    int arrival[n], burst[n], wait[n], turn[n], queue[n],
    temp_burst[n]; bool complete[n];

    cout << "\nEnter the arrival time of the processes :
    "; for(int i = 0; i < n; i++)
        cin>>arrival[i];

    cout << "\nEnter the burst time of the processes : ";
    for(int i = 0; i < n; i++){
        cin>>burst[i];
        temp_burst[i] = burst[i];
    }

    for(int i = 0; i < n; i++){    //Initializing the queue and complete array
        complete[i] = false;
        queue[i] = 0;
    }
  while(timer < arrival[0])          //Incrementing Timer until the first process arrives
        timer++;
    queue[0] = 1;

    while(true){
        bool flag = true;
        for(int i = 0; i < n; i++){
          if(temp_burst[i] != 0){
            flag = false;
            break;
          }
        }
        if(flag)
          break;

        for(int i = 0; (i < n) && (queue[i] != 0); i++){
          int ctr = 0;
          while((ctr < tq) && (temp_burst[queue[0]-1] >
            0)){ temp_burst[queue[0]-1] -= 1;
            timer += 1;
            ctr++;

            //Checking and Updating the ready queue until all the processes
            arrive checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
          }
          //If a process is completed then store its exit
          time //and mark it as completed
          if((temp_burst[queue[0]-1] == 0) && (complete[queue[0]-1] ==
            false)){ //turn array currently stores the completion time
            turn[queue[0]-1] = timer;
            complete[queue[0]-1] = true;
          }

              //checks whether or not CPU is idle
          bool idle = true;
          if(queue[n-1] == 0){
            for(int i = 0; i < n && queue[i] != 0; i++){
              if(complete[queue[i]-1] == false){
                idle = false;
              }
            }
          }
          else
            idle = false;

          if(idle){
            timer++;
            checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
```

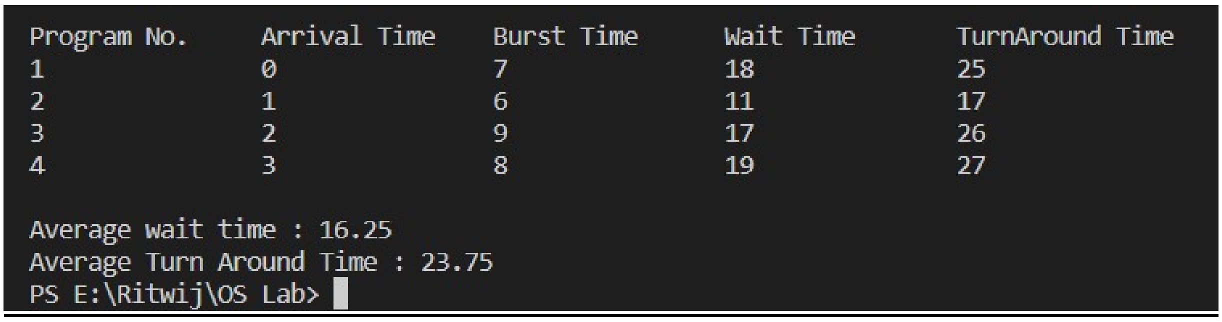
```
            }

            //Maintaining the entries of processes
            //after each premption in the ready Queue
            queueMaintainence(queue,n);
        }
    }

    for(int i = 0; i < n; i++){
        turn[i] = turn[i] - arrival[i];
        wait[i] = turn[i] - burst[i];
    }

    cout << "\nProgram No.\tArrival Time\tBurst Time\tWait Time\tTurnAround
         Time" << endl;
    for(int i = 0; i < n; i++){
        cout<<i+1<<"\t\t"<<arrival[i]<<"\t\t"
          <<burst[i]<<"\t\t"<<wait[i]<<"\t\t"<<turn[i]<<endl;
    }
    for(int i =0; i< n; i++){
        avgWait += wait[i];
        avgTT += turn[i];
    }
    cout<<"\nAverage wait time : "<<(avgWait/n)
      <<"\nAverage Turn Around Time : "<<(avgTT/n);

    return 0;

}
```

Input:



```
PS E:\Ritwij\OS Lab> cd "e:\Ritwij\OS Lab\" ; if ($?) { g++ ro

Enter the time quanta : 3

Enter the number of processes : 4

Enter the arrival time of the processes : 0 1 2 3

Enter the burst time of the processes : 7 6 9 8
```

Output:



```
Program No.     Arrival Time    Burst Time      Wait Time       TurnAround Time
1               0               7               18              25
2               1               6               11              17
3               2               9               17              26
4               3               8               19              27

Average wait time : 16.25
Average Turn Around Time : 23.75
PS E:\Ritwij\OS Lab> 
```

Conclusion:
From this practical, we learned about Round RobinCPU Scheduling Algorithm and how to Implement it and find waiting time using the given burst time