Reg No.: EG/2020/3894

Name : Didulani P.K.S

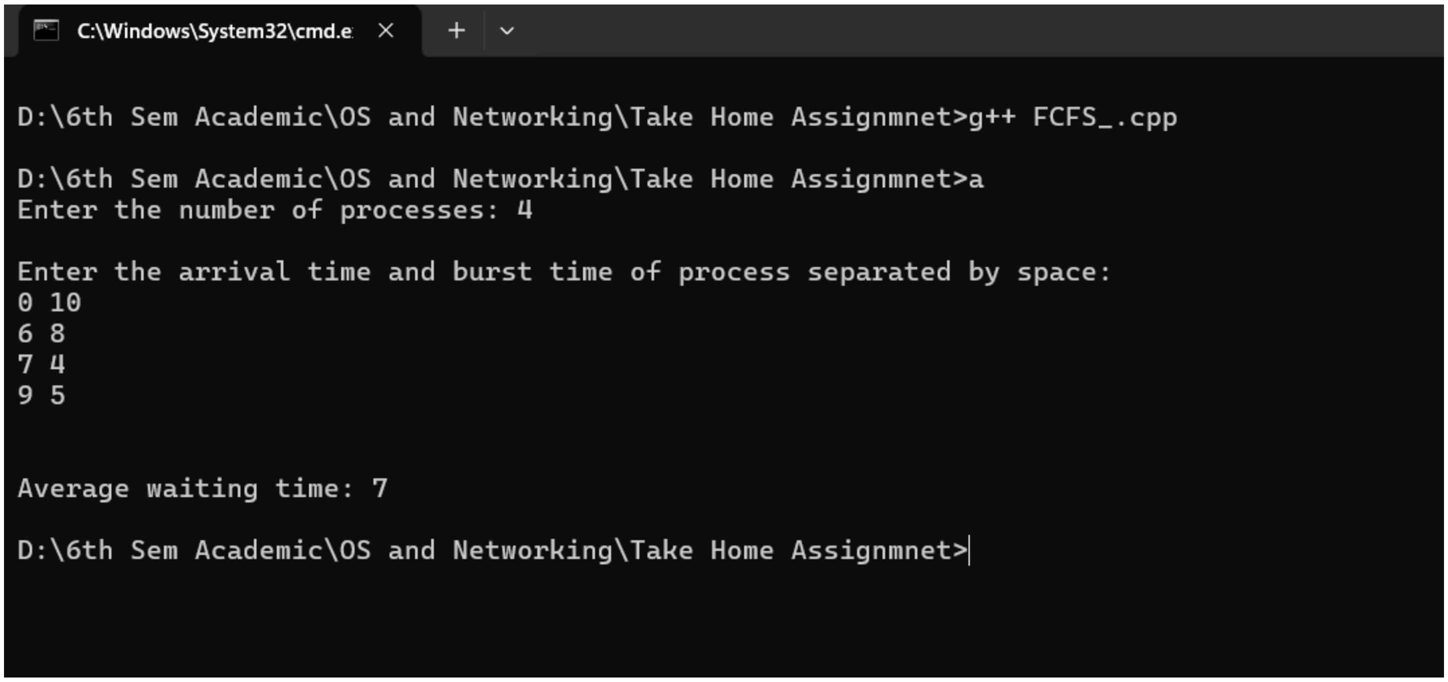
Date   : 01/06/2024

Implement the FCFS scheduling algorithm.

```
#include<iostream>
using namespace std;

#include<vector>
#include<map>

int main(){

    // get the number of processes
    int number_of_processes;
    cout<< "Enter the number of processes: ";
    cin >> number_of_processes;

    cout <<endl;

    // map to store the arrival time and burst time of each process. Here key is arrival
time and value is burst time
    map<int, int> process;

    // Get the arrival time and burst time of each process in a single line
    cout << "Enter the arrival time and burst time of process separated by space: "<<endl;
    for (int i = 0; i < number_of_processes; i++) {
        int arrival_time, burst_time;
        cin >> arrival_time >> burst_time;

        process[arrival_time] = burst_time;
    }
```

```cpp
// array to store the completion time of each process
    vector<int> completion_time(number_of_processes);

    // get the completion time of each process
    int temp=0;
    int index =0;
    for(const auto& i: process){

        if(i.first == 0){
            temp=  i.second;
        }else{
            temp = i.second + temp;
        }

        completion_time[index] = temp;
        index++;
    }
    // array to store the turnaround time of each process
    vector<int> turnaround_time(number_of_processes);

    // get the turnaround time of each process
    int temp1=0;
    for (auto i: process){
        turnaround_time[temp1] = completion_time[temp1] - i.first;
        temp1++;
    }
    // array to store the waiting time of each process
    vector<int> waiting_time(number_of_processes);

    // get the waiting time of each process
    int temp2=0;
    for (auto i: process){
        waiting_time[temp2] = turnaround_time[temp2] - i.second;
        temp2++;
    }


    // print the average waiting times
    int sum =0;
    for(int i=0; i<number_of_processes; i++){
        sum = sum + waiting_time[i];
    }
    double average_waiting_time = sum/number_of_processes;

    cout << "\n\nAverage waiting time: " << average_waiting_time <<endl;
}
```

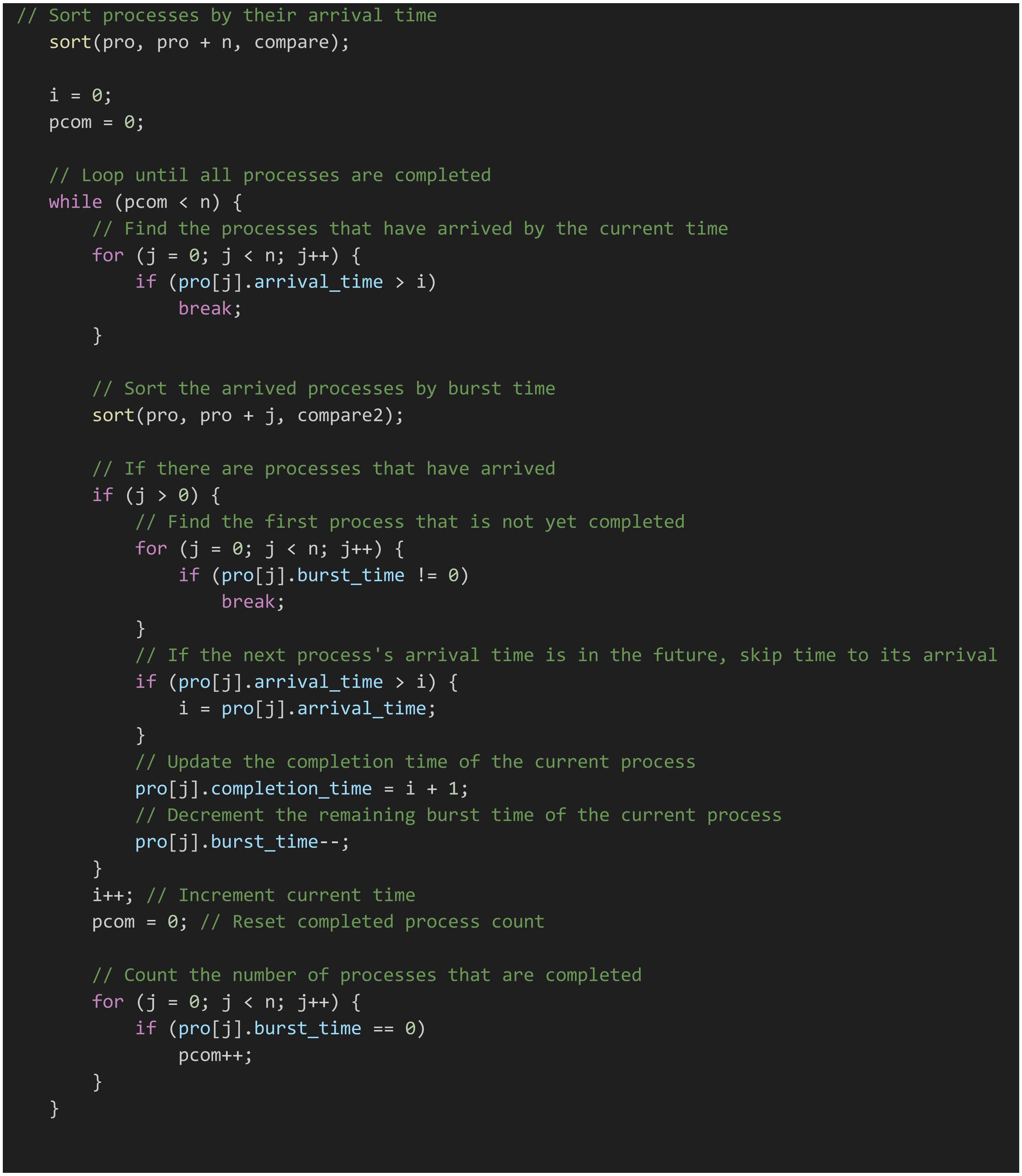
Output of the FCFS algorithm,



```
D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>g++ FCFS_.cpp

D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>a
Enter the number of processes: 4

Enter the arrival time and burst time of process separated by space:
0 10
6 8
7 4
9 5


Average waiting time: 7

D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>
```

Implement the SJF scheduling algorithm.

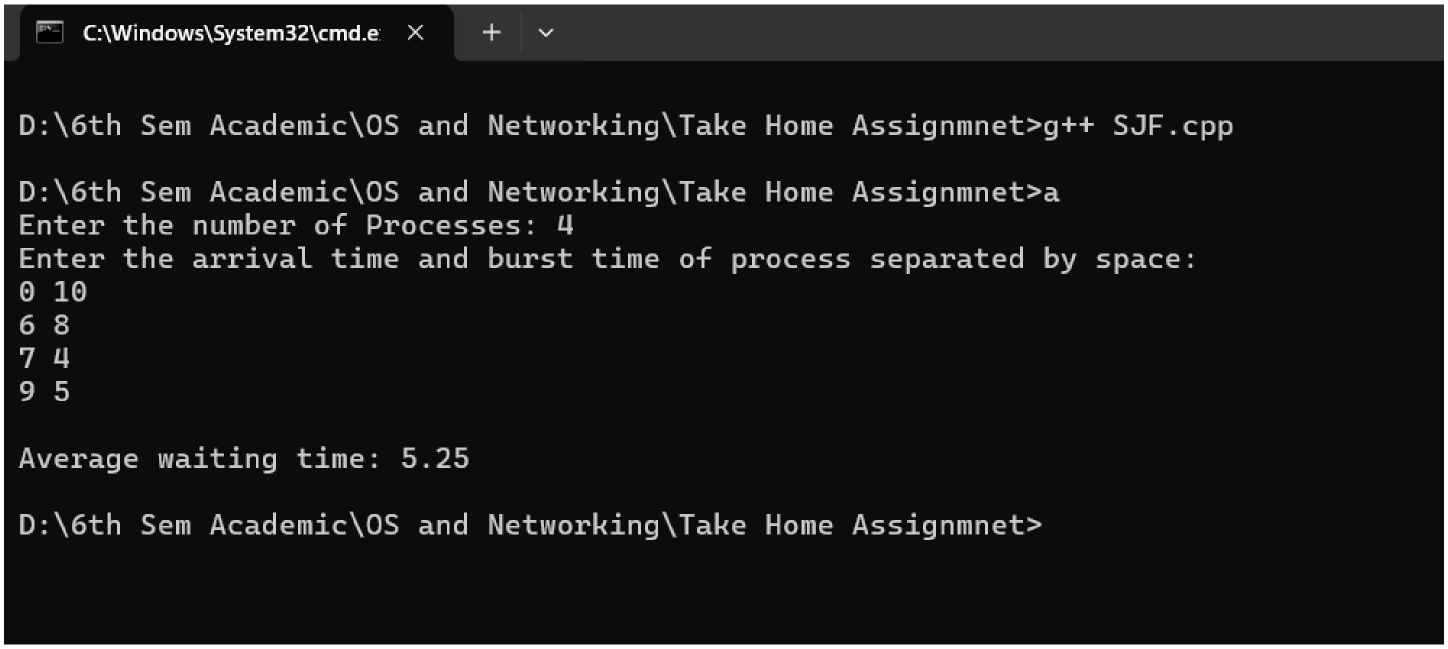
```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

int main() {
    // Define a structure to represent each process
    typedef struct process {
        int arrival_time, burst_time, completion_time, turnaround_time, waiting_time, btt;
        string pro_id; // Process ID
    } Schedule;

    // Comparison function to sort processes by arrival time
    auto compare = [](Schedule a, Schedule b) {
        return a.arrival_time < b.arrival_time;
    };

    // Comparison function to sort processes by burst time
    auto compare2 = [](Schedule a, Schedule b) {
        return a.burst_time < b.burst_time;
    };

    Schedule pro[10]; // Array to store processes
    int n, i, j, pcom; // n: number of processes, pcom: processes completed
      //i & j = iterative variables

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

    cout << "Enter the arrival time and burst time of process separated by space: \n";

    // Input process details
    for (i = 0; i < n; i++) {

        cin >> pro[i].arrival_time;
        cin >> pro[i].burst_time;
        pro[i].btt = pro[i].burst_time;
    }
```



```
// Sort processes by their arrival time
    sort(pro, pro + n, compare);

    i = 0;
    pcom = 0;

    // Loop until all processes are completed
    while (pcom < n) {
        // Find the processes that have arrived by the current time
        for (j = 0; j < n; j++) {
            if (pro[j].arrival_time > i)
                break;
        }

        // Sort the arrived processes by burst time
        sort(pro, pro + j, compare2);

        // If there are processes that have arrived
        if (j > 0) {
            // Find the first process that is not yet completed
            for (j = 0; j < n; j++) {
                if (pro[j].burst_time != 0)
                    break;
            }
            // If the next process's arrival time is in the future, skip time to its arrival
            if (pro[j].arrival_time > i) {
                i = pro[j].arrival_time;
            }
            // Update the completion time of the current process
            pro[j].completion_time = i + 1;
            // Decrement the remaining burst time of the current process
            pro[j].burst_time--;
        }
        i++; // Increment current time
        pcom = 0; // Reset completed process count

        // Count the number of processes that are completed
        for (j = 0; j < n; j++) {
            if (pro[j].burst_time == 0)
                pcom++;
        }
    }
```

```cpp
    double sum = 0;
    double avg_wait_time;

    // Calculate turnaround time and waiting time for each process
    for (i = 0; i < n; i++) {
        pro[i].turnaround_time = pro[i].completion_time - pro[i].arrival_time;
        pro[i].waiting_time = pro[i].turnaround_time - pro[i].btt;

        sum += pro[i].waiting_time; // Add waiting time to sum
    }

    // Calculate average waiting time
    avg_wait_time = sum / n;

    cout << "\nAverage waiting time: " << avg_wait_time << endl;

    return 0;
}
```

Output of the SJF algorithm



```
D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>g++ SJF.cpp

D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>a
Enter the number of Processes: 4
Enter the arrival time and burst time of process separated by space:
0 10
6 8
7 4
9 5

Average waiting time: 5.25

D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>
```

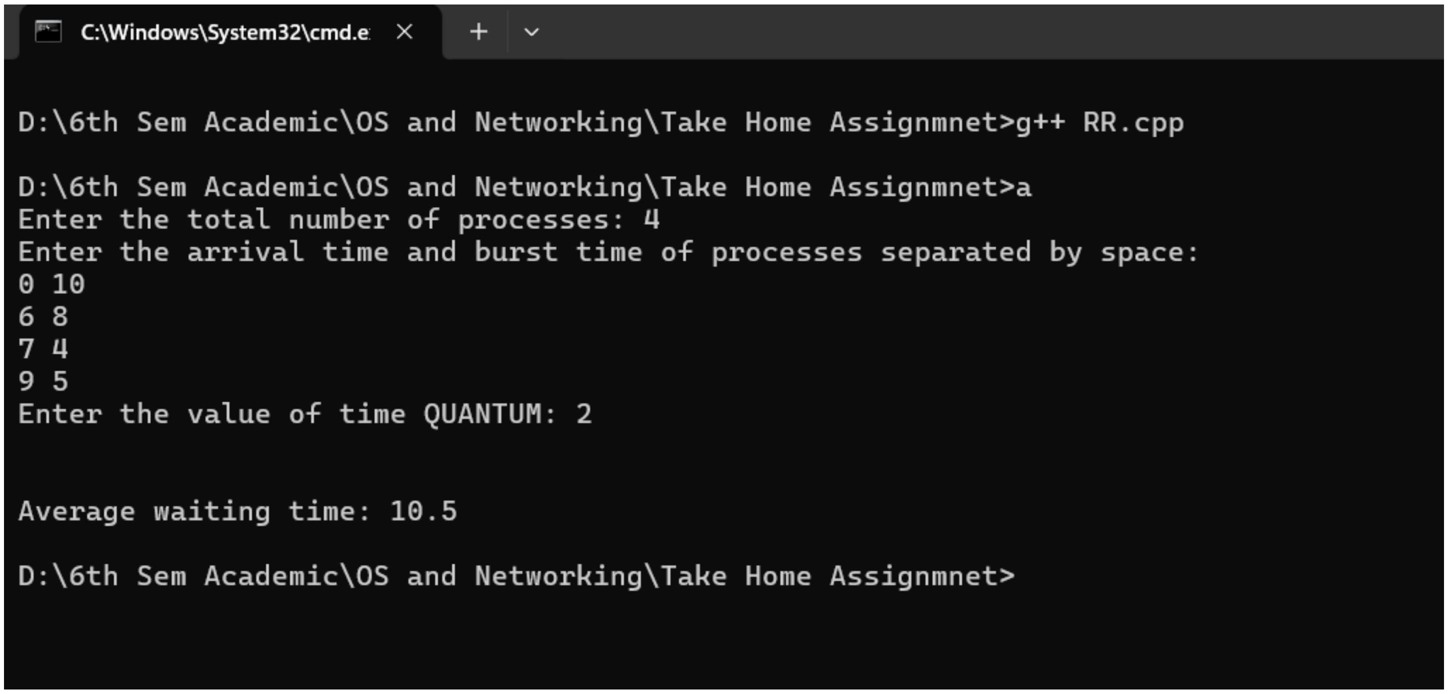
Implement the RR scheduling algorithm.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int i, n, time, remain, temps = 0, time_quantum;
    // wt=Total waiting time, tat=Total turnaround time
    int wt = 0, tat = 0;

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

    // Initialize the remaining processes counter
    remain = n;

    // Define vector arrays for arrival time, burst time, and remaining time for each process
    vector<int> at(n);
    vector<int> bt(n);
    vector<int> rt(n);

    // get the arrival time and burst time of each process
    cout << "Enter the arrival time and burst time of processes separated by space:" << endl;
    for (i = 0; i < n; i++) {
        cin >> at[i] >> bt[i];
        rt[i] = bt[i]; // Initialize remaining time with burst time
    }

    // Get the time quantum
    cout << "Enter the value of time QUANTUM: ";
    cin >> time_quantum;
```

```
 // Start the scheduling algorithm
    for (time = 0, i = 0; remain != 0;) {
        // If remaining time for the current process is less than or equal to time quantum
and greater than 0
        if (rt[i] <= time_quantum && rt[i] > 0) {
            // Update time with the remaining time for the current process
            time += rt[i];
            // Set remaining time for the current process to 0
            rt[i] = 0;
            // Set flag to indicate that the process has finished executing
            temps = 1;
        } else if (rt[i] > 0) {
            // If remaining time for the current process is greater than 0, decrement
remaining time by time quantum
            rt[i] -= time_quantum;
            // Update time with the time quantum
            time += time_quantum;
        }

        // If the current process has finished executing
        if (rt[i] == 0 && temps == 1) {
            // Decrement the remaining processes counter
            remain--;
            // turnaround time for the current process
            int turnaround_time = time - at[i];
            // waiting time for the current process
            int waiting_time = turnaround_time - bt[i];

            // Update total waiting time
            wt += waiting_time;

            // Reset flag indicating that the process has finished executing
            temps = 0;
        }

        // Determine the next process to execute based on arrival time and time quantum
        if (i == n - 1) {
            i = 0; // Wrap around to the beginning of the process queue if reached the end
        }
        // check if the arrival time is less than or equal to the current time
        else if (at[i + 1] <= time) {
            i++; // Move to the next process if so
        }
        else {
            i = 0; // otherwise reset to the beginning of the process queue
        }
    }
```

```
// Output average waiting time
    cout << "\n\nAverage waiting time: " << (double)wt / n << endl;


    return 0;
}
```

Output of the RR algorithm,



```
D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>g++ RR.cpp

D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>a
Enter the total number of processes: 4
Enter the arrival time and burst time of processes separated by space:
0 10
6 8
7 4
9 5
Enter the value of time QUANTUM: 2


Average waiting time: 10.5

D:\6th Sem Academic\OS and Networking\Take Home Assignmnet>
```

Analysis comparing the performance of the algorithms based on average waiting time.

Given Data,

Process	Arrival Time	Burst Time
P1	0	10
P2	6	8
P3	7	4
P4	9	5

Form Results,

CPU Scheduling Algorithm	Average Waiting Time (units)
First Come First Server (FCFS)	7
Shortest Job First	5.25
Round Rodin (RR) for 2 units time quantum	10.5

Here only the FCFS scheduling algorithm is non-preemptive, Other 2 algorithms are preemptive.

From the Results we can see the Shortest Job First has the smallest Average waiting time.

- In the SJF the process which have the shortest burst time is execute first. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. So, we can say SJS prioritizes shorter jobs first. Therefore, it leads to quicker turnaround times for processes overall.

Turn Around Time = Completion Time – Arrival Time

- Since shorter jobs are executed first, they typically complete faster. So, it leads the lower turnaround time for individual processes.

Waiting Time = Turn Around Time – Burst Time

- When the turn around time getting smaller waiting time also getting smaller for each individual processes. Finally, the Overall average waiting time also become small. That is the advantage of the SJF. From this it minimizes the wasted CPU time because it always selected job available, leading to efficient CPU utilization.

- In the FCFS algorithm CPU execute the processes which come first to the CPU. It is not prioritized shorter jobs, it leads longer waiting times. Here in the 7th unit of time the remaining burst time of the P_2 is 7 but the P_3 has 4 units of burst time. But in the FCFS the P_3 have to wait until the P_2 terminate. Therefore, it will lead the higher waiting time (Convoy Effect).
- Here the Round Rodin algorithm has the highest waiting time by compare to the other 2 algorithms. This is because processes are given equal time slices regardless of their burst times, potentially leading to longer waiting times for shorter processes. Also, here we use the time quantum is 2 units. Which is small, that also may impact to the overall system performance because here some processes burst times are large. From the Gantt chart above it prove.

Gantt Chart for FCFS algorithm,

0 10 18 22 27

P_1	P_2	P_3	P_4

Process	Arrival Time	Burst Time	Complete Time	Turnaround Time	Waiting Time
P1	0	10	10	10	0
P2	6	8	18	12	4
P3	7	4	22	15	11
P4	9	5	27	18	13

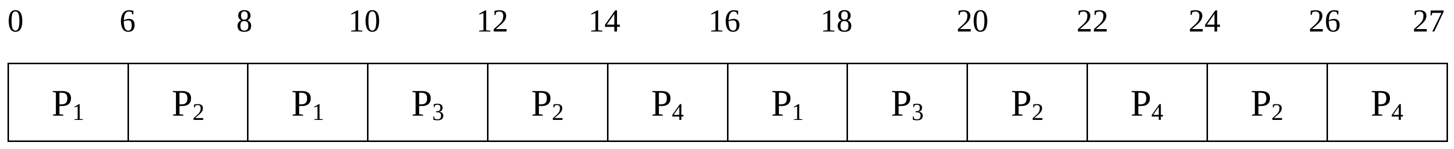
Gantt Chart for SJF algorithm,

0 10 14 19 27

P_1	P_3	P_4	P_2

Process	Arrival Time	Burst Time	Total waiting Time	Process executed before	Waiting Time
P1	0	10	0	0	0
P2	6	8	19	0	13
P3	7	4	10	0	3
P4	9	5	14	0	5

Gantt Chart for RR algorithm,


0 6 8 10 12 14 16 18 20 22 24 26 27
P_1 P_2 P_1 P_3 P_2 P_4 P_1 P_3 P_2 P_4 P_2 P_4

Process	**Arrival Time**	**Burst Time**	**Complete Time**	**Turnaround Time**	**Waiting Time**
P1	0	10	18	18	8
P2	6	8	26	20	12
P3	7	4	20	13	9
P4	9	5	17	18	13