Experiment Name : CPU SCHEDULING ALGORITHM: FIRST COME FIRST SERVE

AIM: To write a program to simulate the CPU scheduling algorithm First Come First Serve (FCFS).

DESCRIPTION: The FCFS scheduling algorithm serves processes in the order of their arrival. The waiting time for the first process is set to zero. The waiting time for subsequent processes is calculated as the sum of the burst times of all previous processes. The average waiting time is computed as the mean of all waiting times. The algorithm emphasizes that the process that arrives first will be executed first.

ALGORITHM:

1. Start the process.
2. Accept the number of processes in the ready queue.
3. For each process in the ready queue, assign the process name and burst time.
4. Set the waiting time of the first process as 0 and its burst time as its turnaround time.
5. For each process in the ready queue, calculate:
    - a) Waiting time (n) = waiting time (n-1) + Burst time (n-1)
    - b) Turnaround time (n) = waiting time (n) + Burst time (n)
6. Calculate:
    - a) Average waiting time = Total waiting time / Number of processes
    - b) Average turnaround time = Total turnaround time / Number of processes
7. Stop the process.

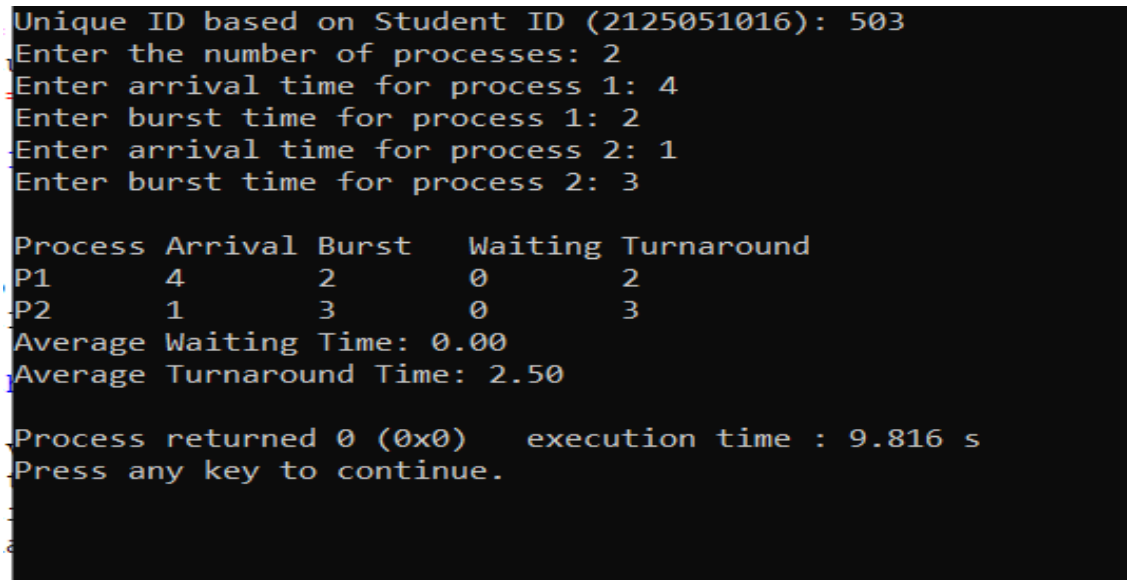
SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;
void UniqueID(string studentID) {
    int uniqueCode = 0;
    for (char c : studentID) {
        uniqueCode += (int)c;
    }
    cout << "Unique ID based on Student ID (" << studentID << "): "
<< uniqueCode << endl;
```

```
}
int main() {
    string studentID = "2125051016";
    UniqueID(studentID);
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;
    vector<int> arrival(n);
    vector<int> burst(n);
    vector<int> waiting(n);
    vector<int> turnaround(n);
    for (int i = 0; i < n; i++) {
        cout << "Enter arrival time for process " << (i + 1) << ": ";
        cin >> arrival[i];
        cout << "Enter burst time for process " << (i + 1) << ": ";
        cin >> burst[i];
    }
    waiting[0] = 0;
    for (int i = 1; i < n; i++) {
        waiting[i] = burst[i - 1] + waiting[i - 1] + (arrival[i] -
(arrival[i - 1] + burst[i - 1]));
        if (waiting[i] < 0) {
            waiting[i] = 0;
        }
    }
    for (int i = 0; i < n; i++) {
        turnaround[i] = waiting[i] + burst[i];
    }
    cout << fixed << setprecision(2);
    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << (i + 1) << "\t" << arrival[i] << "\t" <<
burst[i] << "\t"
             << waiting[i] << "\t" << turnaround[i] << endl;
    }
    double total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        total_waiting_time += waiting[i];
        total_turnaround_time += turnaround[i];
    }
    cout << "Average Waiting Time: " << (total_waiting_time / n) <<
endl;
    cout << "Average Turnaround Time: " << (total_turnaround_time /
n) << endl;

    return 0;}
```

Input and output :



SOURCE CODE:

```
#!/bin/bash
studentID="2125051016"
uniqueCode=0
for (( i=0; i<${#studentID}; i++ )); do
        uniqueCode=$(( uniqueCode + $(printf "%d"
"'${studentID:$i:1}") ))
    done
    echo "Unique ID based on Student ID (${studentID}): $uniqueCode"
fcfs() {
    local n=$1
    local -a arrival=("${!2}")
    local -a burst=("${!3}")
    local -a waiting_time
    local -a turnaround_time
    waiting_time[0]=0
    for ((i=1; i<n; i++)); do
        waiting_time[i]=$((waiting_time[i-1] + burst[i-1]))
        waiting_time[i]=$((waiting_time[i] - arrival[i]))
        if [ ${waiting_time[i]} -lt 0 ]; then
            waiting_time[i]=0
        fi
    done
    for ((i=0; i<n; i++)); do
        turnaround_time[i]=$((waiting_time[i] + burst[i]))
    done
    echo -e "\nProcess\tArrival\tBurst\tWaiting\tTurnaround"
```

```
    for ((i=0; i<n; i++)); do
        echo -e "P$((i +
1))\t${arrival[i]}\t${burst[i]}\t${waiting_time[i]}\t${turnaround_tim
e[i]}"
    done
    local total_waiting_time=0
    local total_turnaround_time=0
    for ((i=0; i<n; i++)); do
        total_waiting_time=$((total_waiting_time + waiting_time[i]))
        total_turnaround_time=$((total_turnaround_time +
turnaround_time[i]))
    done
    echo "Average Waiting Time: $(bc <<< "scale=2;
$total_waiting_time / $n")"
    echo "Average Turnaround Time: $(bc <<< "scale=2;
$total_turnaround_time / $n")"
}
echo "Enter the number of processes: "
read n
declare -a arrival
declare -a burst
for ((i=0; i<n; i++)); do
    echo "Enter arrival time for process $((i + 1)):"
    read arrival[$i]
    echo "Enter burst time for process $((i + 1)):"
    read burst[$i]
done
fcfs $n arrival[@] burst[@]
```

Input and output :

```
Unique ID based on Student ID (2125051016): 503
Enter the number of processes:
2
Enter arrival time for process 1:
3
Enter burst time for process 1:
2
Enter arrival time for process 2:
1
Enter burst time for process 2:
3

Process Arrival Burst   Waiting Turnaround
P1      3       2       0       2
P2      1       3       1       4
Average Waiting Time: .50
Average Turnaround Time: 3.00
```

Discussion:

By running the First Come First Serve (FCFS) algorithm in both Bash and C++ code, we can conclude that the output is correct. FCFS scheduling processes tasks based on their arrival times, meaning that the process that arrives first is executed first. This method is straightforward and easy to implement, but it does not account for the burst times of the processes.

In contrast to non-preemptive Shortest Job First (SJF), where once a process starts executing, it runs to completion without interruption, FCFS can lead to longer wait times, especially if a long process arrives before shorter processes. The SJF algorithm is optimal for minimizing average waiting time compared to FCFS, as shorter processes finish quickly, leading to better overall CPU utilization.

However, SJF requires knowledge of the burst times, which may not always be available in real-time. Moreover, while SJF is efficient in average wait time, it can lead to the "starvation problem," where longer processes may be indefinitely delayed by a continuous influx of shorter jobs.

Conclusion:

In conclusion, while the non-preemptive version of SJF ensures that processes complete without interruption, it can lead to inefficiencies if longer jobs are starved due to continuous shorter job arrivals. This makes SJF a valuable scheduling strategy, particularly in environments where job burst times are predictable, as it significantly reduces average waiting and turnaround times compared to other scheduling algorithms like FCFS.

On the other hand, FCFS is simpler to implement but may result in longer wait times for processes, especially if short processes are blocked behind long ones. Careful consideration must be given to the potential drawbacks of each algorithm, such as process starvation in SJF and inefficiencies in FCFS. Ultimately, the choice of scheduling algorithm depends on the specific requirements and characteristics of the workload being managed.