UNIVERSITY OF INFORMATION TECHNOLOGY & SCIENCES

**Submitted To :** Saima Siddique Tashfia
Lecturer, Dept. of CSE
UITS.

**Submitted By :** Maheru Tabassum Ohana
ID : 2125051015
Dept : CSE, Batch : 50, Section : 7A

**Course Code :** CSE412
**Course Title :** Operating System Lab

**Lab Report :** 01

**Date of Submission :** 7 October, 2024.

University of Information Technology and Sciences

‘

**Experiment No :** 01
**Experiment Name :** CPU SCHEDULING ALGORITHM: First Come First Serve

**AIM :** To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS) in C++ and Bash.

**DESCRIPTION :**
The First-Come, First-Served (FCFS) scheduling algorithm is a simple CPU scheduling method where processes are executed in the order of their arrival in the ready queue. It is non-preemptive, meaning once a process starts executing, it runs to completion without interruption. FCFS operates on a First-In, First-Out (FIFO) basis, ensuring fairness by giving each process an equal opportunity. However, it can lead to high average waiting times, particularly if shorter processes are stuck behind longer ones, a phenomenon known as the "convoy effect." While easy to implement, FCFS may be less efficient than more advanced scheduling algorithms.

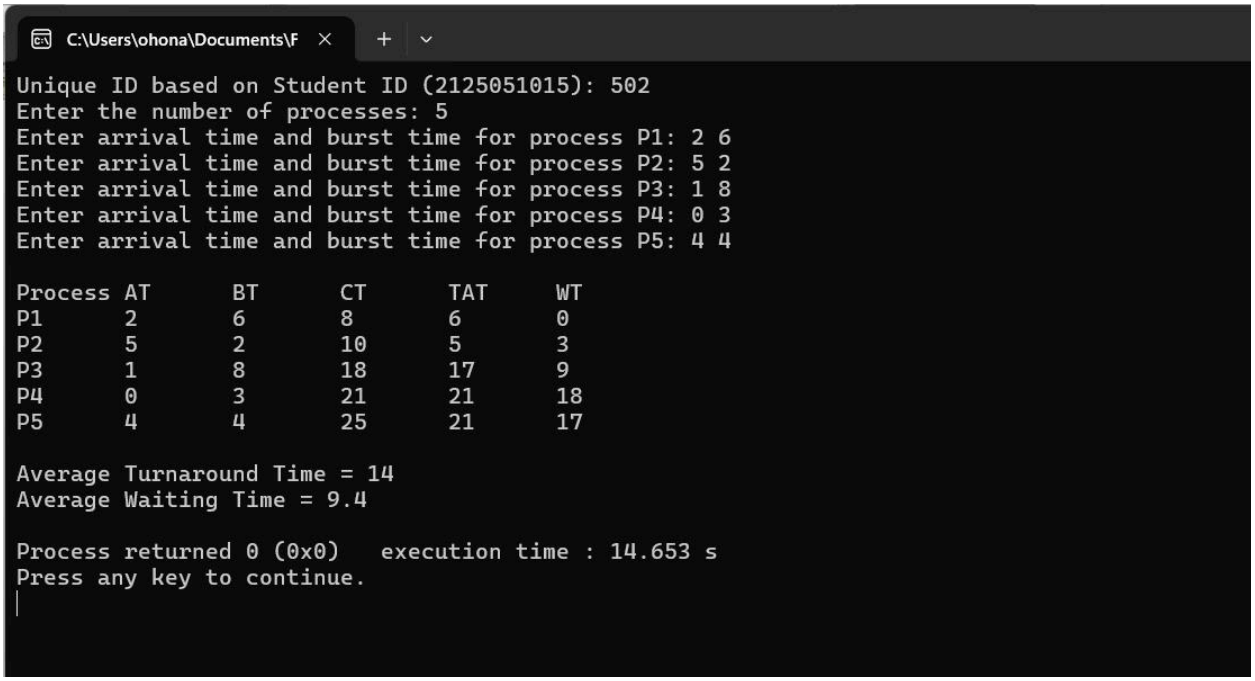
**In C++ :**

**SOURCE CODE:**

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

void generateUniqueID(string studentID) {
int uniqueCode = 0;
for (char c : studentID) {
uniqueCode += (int)c;
}
cout << "Unique ID based on Student ID (" << studentID << "): "<<
uniqueCode << endl;
}
struct Process {
   int id, AT, BT, CT, TAT, WT;
};
void FCFS(vector<Process>& processes, int n) {
   int total_TAT = 0, total_WT = 0;
   processes[0].CT = processes[0].AT + processes[0].BT; // First process
   processes[0].TAT = processes[0].CT - processes[0].AT; // TAT

 processes[0].WT = processes[0].TAT - processes[0].BT; // WT
```

```
    total_TAT += processes[0].TAT;
    total_WT += processes[0].WT;

    for (int i = 1; i < n; i++) {
        processes[i].CT = max(processes[i].AT, processes[i - 1].CT) + processes[i].BT; // CT
        processes[i].TAT = processes[i].CT - processes[i].AT; // TAT
        processes[i].WT = processes[i].TAT - processes[i].BT; // WT

        total_TAT += processes[i].TAT;
        total_WT += processes[i].WT;
    }
    cout << "\nProcess\tAT\tBT\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << processes[i].id << "\t" << processes[i].AT << "\t"
             << processes[i].BT << "\t" << processes[i].CT << "\t"
             << processes[i].TAT << "\t" << processes[i].WT << "\n";
    }
    cout << "\nAverage Turnaround Time = " << (float)total_TAT / n;
    cout << "\nAverage Waiting Time = " << (float)total_WT / n << "\n";
}
int main() {
string studentID = "2125051015";
generateUniqueID(studentID);
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;
    vector<Process> processes(n);

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        cout << "Enter arrival time and burst time for process P" << i + 1 << ": ";
        cin >> processes[i].AT >> processes[i].BT;
    }
    FCFS(processes, n);

    return 0;
}
```

**INPUT & OUTPUT -**




**In BashScript :**

**SOURCE CODE:**

```
#!/bin/bash

generateUniqueID() {
   local studentID=$1
   local uniqueCode=0
   for (( i=0; i<${#studentID}; i++ )); do
      uniqueCode=$(( uniqueCode + $(printf "%d" "'${studentID:$i:1}") ))
   done
   echo "Unique ID based on Student ID ($studentID): $uniqueCode"
}

studentID="2125051015"
generateUniqueID $studentID

fcfsScheduling() {
   local n=$1
   local -a at_arr=("${@:2:$n}")

local -a bt_arr=("${@:$((n+2)):$n}")  # Burst time array
```

```
    local -a ct_arr=()
    local -a tat_arr=()
    local -a wt_arr=()

    ct_arr[0]=$(( at_arr[0] + bt_arr[0] ))
    for (( i=1; i<$n; i++ )); do
        if (( ${at_arr[i]} > ${ct_arr[i-1]} )); then
            ct_arr[i]=$(( at_arr[i] + bt_arr[i] ))
        else
            ct_arr[i]=$(( ct_arr[i-1] + bt_arr[i] ))
        fi
    done

    local total_tat=0
    local total_wt=0

    for (( i=0; i<$n; i++ )); do
        tat_arr[i]=$(( ct_arr[i] - at_arr[i] ))       # TAT = CT - AT
        wt_arr[i]=$(( tat_arr[i] - bt_arr[i] ))       # WT = TAT - BT
        total_tat=$(( total_tat + tat_arr[i] ))
        total_wt=$(( total_wt + wt_arr[i] ))
    Done

    echo -e "\nProcesses\tAT\tBT\tCT\tTAT\tWT"
    echo "----------------------------------------"

    for (( i=0; i<$n; i++ )); do
        echo -e "$i\t\t${at_arr[i]}\t${bt_arr[i]}\t${ct_arr[i]}\t${tat_arr[i]}\t${wt_arr[i]}"
    done

    local avg_tat=$(awk "BEGIN {print $total_tat/$n}")
    local avg_wt=$(awk "BEGIN {print $total_wt/$n}")

    echo -e "\nAverage Turnaround Time (TAT) = $avg_tat"

 echo "Average Waiting Time (WT) = $avg_wt"
}
main() {
    echo -e "Enter number of processes: "
    read n

    at_arr=()
    bt_arr=()
```

```
    for (( i=0; i<$n; i++ )); do
        echo -e "Enter Arrival Time (AT) for process $i: "
        read at
        at_arr+=($at)

        echo -e "Enter Burst Time (BT) for process $i: "
        read bt
        bt_arr+=($bt)
    done

    fcfsScheduling $n "${at_arr[@]}" "${bt_arr[@]}"
}

main
```

**INPUT & OUTPUT -**

```
ohona@OHONA ~
$ bash fcfs.sh
Unique ID based on Student ID (2125051015): 502
Enter number of processes:
5
Enter Arrival Time (AT) for process 0:
2
Enter Burst Time (BT) for process 0:
6
Enter Arrival Time (AT) for process 1:
5
Enter Burst Time (BT) for process 1:
2
Enter Arrival Time (AT) for process 2:
1
Enter Burst Time (BT) for process 2:
8
Enter Arrival Time (AT) for process 3:
0
Enter Burst Time (BT) for process 3:
3
Enter Arrival Time (AT) for process 4:
4
Enter Burst Time (BT) for process 4:
4

Processes       AT      BT      CT      TAT     WT
----------------------------------------
0               2       6       8       6       0
1               5       2       10      5       3
2               1       8       18      17      9
3               0       3       21      21      18
4               4       4       25      21      17

Average Turnaround Time (TAT) = 14
Average Waiting Time (WT) = 9.4
```

**Experiment No :** 02
**Experiment Name :** CPU SCHEDULING ALGORITHM: Shortest Job First (Pre-Emptive)

**AIM :** To write a c program to simulate the CPU scheduling algorithm Pre-Emptive Shortest Job First (SJF) in C++ and Bash.

**DESCRIPTION :**
Preemptive Shortest Job First (SJF) is a CPU scheduling algorithm that allows a currently running process to be interrupted if a new process arrives with a shorter remaining burst time. This approach minimizes average waiting and turnaround times by prioritizing shorter tasks. However, it can lead to starvation, where longer processes are delayed indefinitely as shorter jobs arrive. While effective in optimizing performance, it requires accurate knowledge of burst times, making implementation more complex.

**In C++ :**

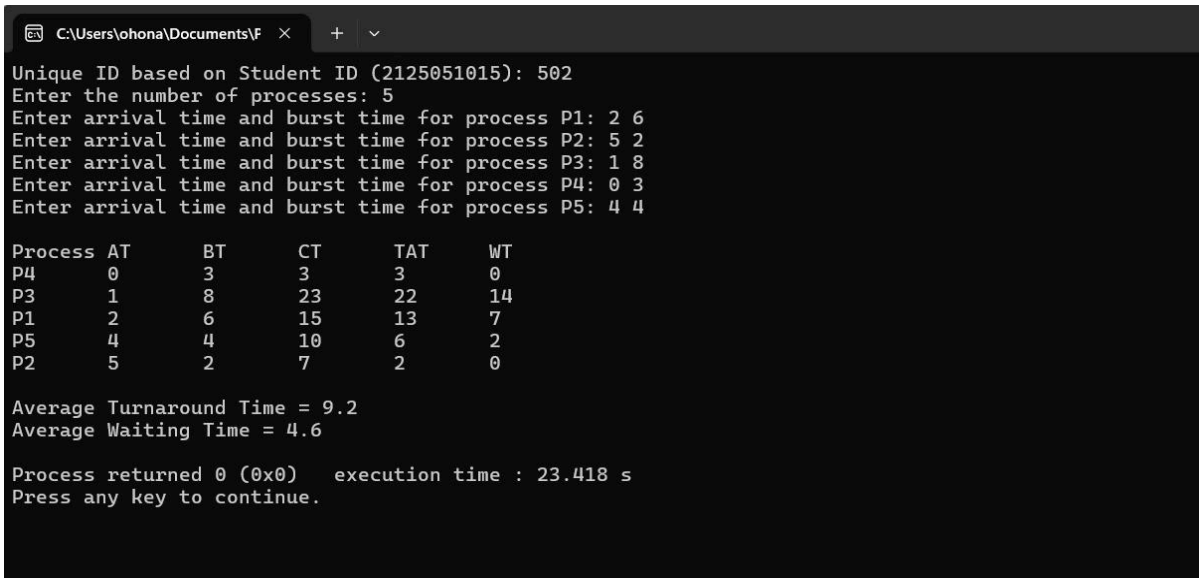
**SOURCE CODE:**

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

void generateUniqueID(string studentID) {
int uniqueCode = 0;
for (char c : studentID) {
uniqueCode += (int)c;
}
cout << "Unique ID based on Student ID (" << studentID << "): "<<
uniqueCode << endl;
}
struct Process {
   int id, AT, BT, CT, TAT, WT, remaining_BT;
};
bool arrivalTimeComparator(const Process &a, const Process &b) {
   return a.AT < b.AT;
}
void SJF_Preemptive(vector<Process>& processes, int n) {
   int total_TAT = 0, total_WT = 0;
   int current_time = 0, completed = 0, min_index = -1;
   bool process_in_execution = false;
```

```
while (completed < n) {

    min_index = -1;
    int min_BT = INT_MAX;
    for (int i = 0; i < n; i++) {
      if (processes[i].AT <= current_time && processes[i].remaining_BT > 0) {
        if (processes[i].remaining_BT < min_BT) {
          min_BT = processes[i].remaining_BT;
          min_index = i;
        }
      }
    }

    if (min_index == -1) {
      current_time++; // No process is ready to execute, so just increment time
    } else {

      processes[min_index].remaining_BT--;
      current_time++;

      if (processes[min_index].remaining_BT == 0) {
        completed++;
        processes[min_index].CT = current_time;
        processes[min_index].TAT = processes[min_index].CT - processes[min_index].AT; // Turn
Around Time
        processes[min_index].WT = processes[min_index].TAT - processes[min_index].BT; // Waiting
Time

        total_TAT += processes[min_index].TAT;
        total_WT += processes[min_index].WT;
      }
    }
  }
  cout << "\nProcess\tAT\tBT\tCT\tTAT\tWT\n";
  for (int i = 0; i < n; i++) {
    cout << "P" << processes[i].id << "\t" << processes[i].AT << "\t"
      << processes[i].BT << "\t" << processes[i].CT << "\t"
      << processes[i].TAT << "\t" << processes[i].WT << "\n";
  }
  cout << "\nAverage Turnaround Time = " << (float)total_TAT / n;
  cout << "\nAverage Waiting Time = " << (float)total_WT / n << "\n";
}
```

```
int main() {

string studentID = "2125051015";
generateUniqueID(studentID);

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

   vector<Process> processes(n);

   for (int i = 0; i < n; i++) {
      processes[i].id = i + 1;
      cout << "Enter arrival time and burst time for process P" << i + 1 << ": ";
      cin >> processes[i].AT >> processes[i].BT;
      processes[i].remaining_BT = processes[i].BT;
   }
   sort(processes.begin(), processes.end(), arrivalTimeComparator);

   SJF_Preemptive(processes, n);

   return 0;
}
```

**INPUT & OUTPUT -**



**In BashScript :**

**SOURCE CODE:**

```
#!/bin/bash

generateUniqueID() {
   local studentID=$1
   local uniqueCode=0
   for (( i=0; i<${#studentID}; i++ )); do
      uniqueCode=$(( uniqueCode + $(printf "%d" "'${studentID:$i:1}") ))
   done
   echo "Unique ID based on Student ID ($studentID): $uniqueCode"
}
sjfPreemptiveScheduling() {
   local n=$1
   local arrival_time=("${!2}")
   local burst_time=("${!3}")

   local remaining_time=("${burst_time[@]}")
   local ct=()   # Completion time
   local tat=()  # Turnaround time
   local wt=()   # Waiting time
   local is_completed=()

   for (( i=0; i<$n; i++ )); do
      is_completed[i]=0
   done

   local current_time=0
   local completed=0
   local total_tat=0
   local total_wt=0
   local min_burst=99999
   local shortest=-1
   local finish_time=0
   local check=0

   while (( completed < n )); do
      for (( i=0; i<$n; i++ )); do
         if (( arrival_time[i] <= current_time && is_completed[i] == 0 )); then
            if (( remaining_time[i] < min_burst && remaining_time[i] > 0 )); then
               min_burst=${remaining_time[i]}
```

```
            shortest=$i
            check=1
        fi
    fi
done

if (( check == 0 )); then
    current_time=$(( current_time + 1 ))
    continue
Fi

remaining_time[shortest]=$(( remaining_time[shortest] - 1 ))

min_burst=${remaining_time[shortest]}
if (( min_burst == 0 )); then
    min_burst=99999
fi

if (( remaining_time[shortest] == 0 )); then
    completed=$(( completed + 1 ))
    finish_time=$(( current_time + 1 ))
    ct[shortest]=$finish_time
    tat[shortest]=$(( ct[shortest] - arrival_time[shortest] ))
    wt[shortest]=$(( tat[shortest] - burst_time[shortest] ))

    total_tat=$(( total_tat + tat[shortest] ))
    total_wt=$(( total_wt + wt[shortest] ))

    is_completed[shortest]=1
fi

current_time=$(( current_time + 1 ))
done

echo -e "\nProcesses\tAT\tBT\tCT\tTAT\tWT"
for (( i=0; i<$n; i++ )); do
    echo -e "$i\t\t${arrival_time[i]}\t${burst_time[i]}\t${ct[i]}\t${tat[i]}\t${wt[i]}"
Done

avg_tat=$(awk "BEGIN {print $total_tat / $n}")
avg_wt=$(awk "BEGIN {print $total_wt / $n}")

echo -e "\nAverage Turnaround Time (TAT) = $avg_tat"
```

```
  echo "Average Waiting Time (WT) = $avg_wt"
}

main() {
   echo -e "Enter number of processes: "
   read n

   arrival_time=()
   burst_time=()

   for (( i=0; i<$n; i++ )); do
      echo -e "Enter Arrival Time (AT) for process $i: "
      read at
      arrival_time+=($at)

      echo -e "Enter Burst Time (BT) for process $i: "
      read bt
      burst_time+=($bt)
   done

   sjfPreemptiveScheduling $n arrival_time[@] burst_time[@]
}
studentID="2125051015"
generateUniqueID $studentID

main
```

**INPUT & OUTPUT -**

```
ohona@OHONA ~
$ bash Psjf.sh
Unique ID based on Student ID (2125051015): 502
Enter number of processes:
5
Enter Arrival Time (AT) for process 0:
2
Enter Burst Time (BT) for process 0:
6
Enter Arrival Time (AT) for process 1:
5
Enter Burst Time (BT) for process 1:
2
Enter Arrival Time (AT) for process 2:
1
Enter Burst Time (BT) for process 2:
8
Enter Arrival Time (AT) for process 3:
0
Enter Burst Time (BT) for process 3:
3
Enter Arrival Time (AT) for process 4:
4
Enter Burst Time (BT) for process 4:
4

Processes       AT      BT      CT      TAT     WT
0               2       6       15      13      7
1               5       2       7       2       0
2               1       8       23      22      14
3               0       3       3       3       0
4               4       4       10      6       2

Average Turnaround Time (TAT) = 9.2
Average Waiting Time (WT) = 4.6
```

**Experiment No :** 03
**Experiment Name :** CPU SCHEDULING ALGORITHM: Shortest Job First (Non-preemptive)

**AIM :** To write a c program to simulate the CPU scheduling algorithm Non-preemptive Shortest Job First (SJF) in C++ and Bash.

**DESCRIPTION :**
Non-Preemptive Shortest Job First (SJF) is a CPU scheduling algorithm where the process with the shortest burst time is selected to run next. Once a process starts, it runs to completion without interruption. This approach minimizes average waiting and turnaround times but can cause longer processes to experience delays if shorter jobs keep arriving. It's simple to implement but requires knowing or estimating process burst times in advance.

**In C++ :**

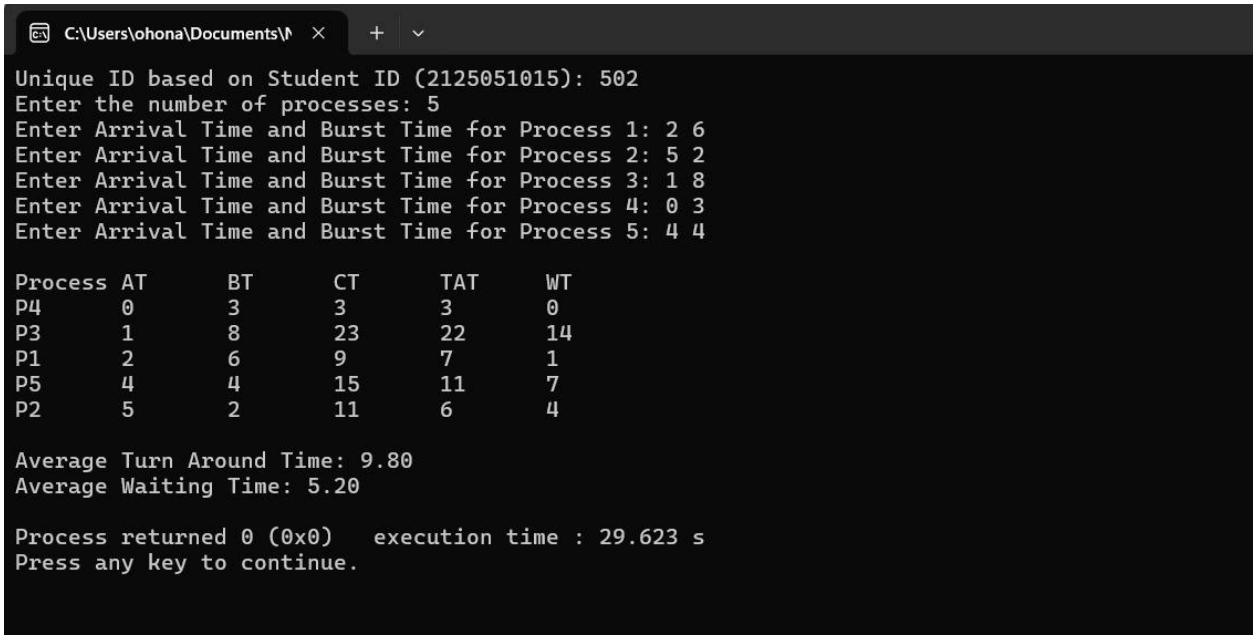
**SOURCE CODE:**

```
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
#include <vector>
#include <limits>

using namespace std;

void generateUniqueID(string studentID) {
int uniqueCode = 0;
for (char c : studentID) {
uniqueCode += (int)c;
}
cout << "Unique ID based on Student ID (" << studentID << "): "<<
uniqueCode << endl;
}

struct Process {
    int id;
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
};
bool arrivalTimeCompare(Process a, Process b) {
    if (a.at == b.at)
        return a.bt < b.bt;
    return a.at < b.at;
}
int findShortestJob(vector<Process> &processes, int currentTime, vector<bool> &completed) {
    int minBT = numeric_limits<int>::max();
    int index = -1;

    for (int i = 0; i < processes.size(); i++) {
        if (!completed[i] && processes[i].at <= currentTime && processes[i].bt < minBT) {
            minBT = processes[i].bt;
            index = i;
        }
    }

    return index;
```

```
}
int main() {

string studentID = "2125051015";
generateUniqueID(studentID);

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

   vector<Process> processes(n);

   for (int i = 0; i < n; i++) {
      processes[i].id = i + 1; // Process IDs start from 1
      cout << "Enter Arrival Time and Burst Time for Process " << processes[i].id << ": ";
      cin >> processes[i].at >> processes[i].bt;
   }
   sort(processes.begin(), processes.end(), arrivalTimeCompare);

   vector<bool> completed(n, false); // Track completed processes
   int currentTime = 0;
   int completedProcesses = 0;
   double totalTAT = 0, totalWT = 0;

   while (completedProcesses < n) {
      int shortestJob = findShortestJob(processes, currentTime, completed);

      if (shortestJob != -1) {
         Process &p = processes[shortestJob];
         currentTime = max(currentTime, p.at) + p.bt;
         p.ct = currentTime;
         p.tat = p.ct - p.at;
         p.wt = p.tat - p.bt;

         totalTAT += p.tat;
         totalWT += p.wt;

         completed[shortestJob] = true; // Mark the job as completed
         completedProcesses++;
      } else {

         currentTime++;
      }
```

```
    }

    cout << "\nProcess\tAT\tBT\tCT\tTAT\tWT\n";
    for (const Process &p : processes) {
        cout << "P" << p.id << "\t" << p.at << "\t" << p.bt << "\t" << p.ct << "\t" << p.tat << "\t" << p.wt <<
"\n";
    }
    cout << fixed << setprecision(2);
    cout << "\nAverage Turn Around Time: " << totalTAT / n;
    cout << "\nAverage Waiting Time: " << totalWT / n << endl;

    return 0;
}
```

**INPUT & OUTPUT -**




**In BashScript :**

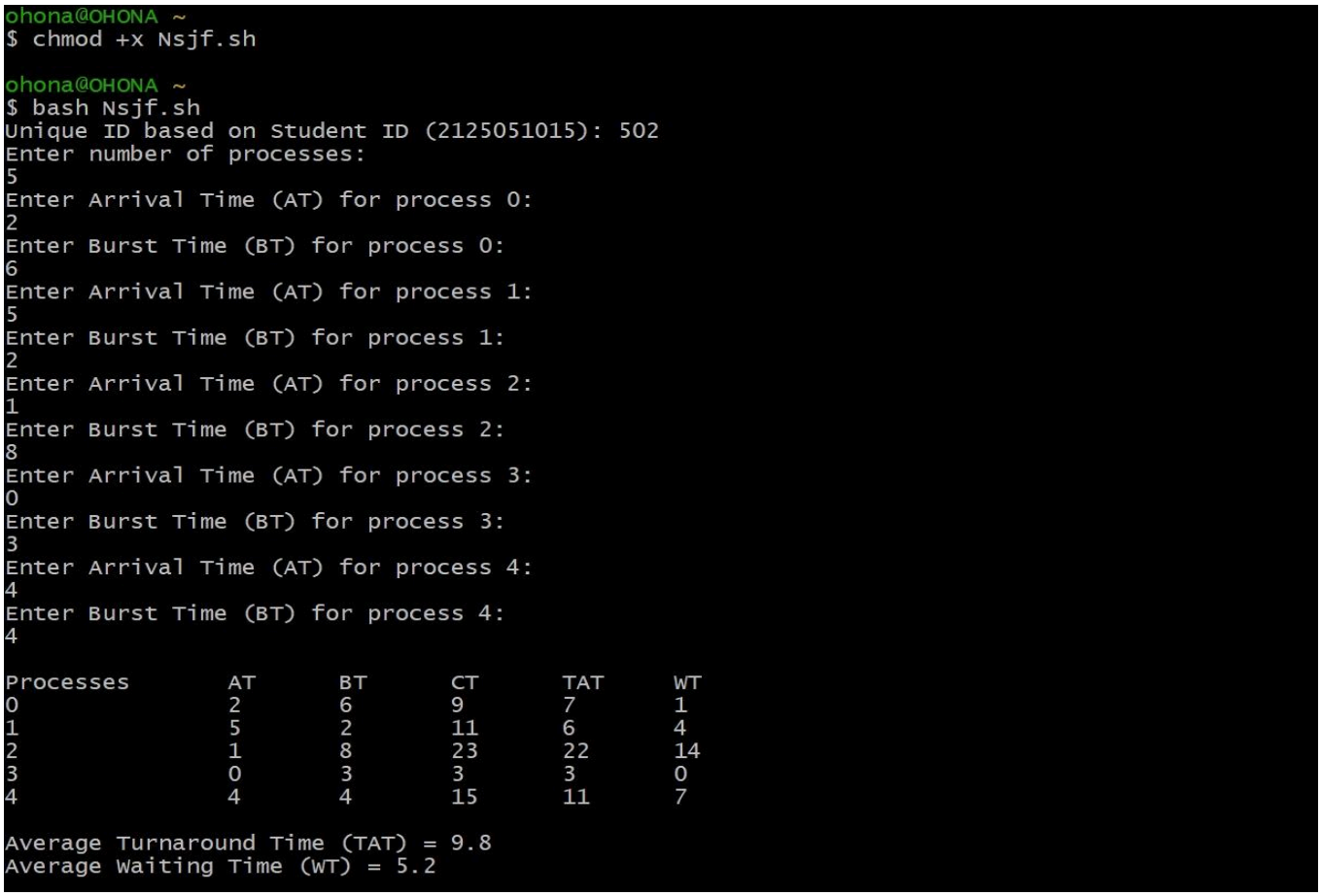
**SOURCE CODE:**

```
#!/bin/bash

generateUniqueID() {
   local studentID=$1
   local uniqueCode=0
   for (( i=0; i<${#studentID}; i++ )); do
      uniqueCode=$(( uniqueCode + $(printf "%d" "'${studentID:$i:1}") ))
   done
   echo "Unique ID based on Student ID ($studentID): $uniqueCode"
}

studentID="2125051015"
generateUniqueID $studentID

sjfScheduling() {
   local n=$1
   local -n arrival_time=$2
   local -n burst_time=$3

   local ct=()
   local tat=()
   local wt=()
   local is_completed=()

   for (( i=0; i<$n; i++ )); do
      is_completed[i]=0
   done

   local current_time=0
   local completed=0
   local total_tat=0
   local total_wt=0

   while (( completed < n )); do
      local min_burst=99999
      local index=-1

      for (( i=0; i<$n; i++ )); do
         if (( arrival_time[i] <= current_time && is_completed[i] == 0 )); then
            if (( burst_time[i] < min_burst )); then
               min_burst=${burst_time[i]}
               index=$i
            fi
```

```
        fi
        done

        if (( index != -1 )); then
            ct[index]=$(( current_time + burst_time[index] ))
            tat[index]=$(( ct[index] - arrival_time[index] ))
            wt[index]=$(( tat[index] - burst_time[index] ))
            total_tat=$(( total_tat + tat[index] ))
            total_wt=$(( total_wt + wt[index] ))
            current_time=${ct[index]}
            is_completed[index]=1
            completed=$(( completed + 1 ))
        else
            current_time=$(( current_time + 1 ))
        fi
    done

    echo -e "\nProcesses\tAT\tBT\tCT\tTAT\tWT"
    for (( i=0; i<$n; i++ )); do
        echo -e "$i\t\t${arrival_time[i]}\t${burst_time[i]}\t${ct[i]}\t${tat[i]}\t${wt[i]}"
    done

    avg_tat=$(bc -l <<< "scale=2; $total_tat/$n")
    avg_wt=$(bc -l <<< "scale=2; $total_wt/$n")
    echo -e "\nAverage Turnaround Time (TAT) = $avg_tat"
    echo "Average Waiting Time (WT) = $avg_wt"
}
main() {
    echo -e "Enter number of processes: "
    read n

    arrival_time=()
    burst_time=()

    for (( i=0; i<$n; i++ )); do
        echo -e "Enter Arrival Time (AT) for process $i: "
        read at
        arrival_time[$i]=$at

        echo -e "Enter Burst Time (BT) for process $i: "
        read bt
        burst_time[$i]=$bt
    done
```

```
    sjfScheduling $n arrival_time burst_time
}

main
```

**INPUT & OUTPUT -**




**Discussion :**

This lab report demonstrates the implementation of CPU scheduling algorithms—**First-Come, First-Served (FCFS)**, **Preemptive Shortest Job First (SJF)**, and **Non-Preemptive Shortest Job First (SJF)**—using C++ and Linux Bash scripting. FCFS schedules processes in the order of their arrival without preemption, which is simple but can cause long waiting times due to the "convoy effect." Preemptive SJF improves efficiency by selecting the process with the

shortest remaining time, though it may lead to starvation for longer tasks. Non-Preemptive SJF, on the other hand, runs the shortest job to completion without interruptions, offering a balance between simplicity and performance but also facing the issue of potential starvation. Performance metrics like waiting time and turnaround time are evaluated for all algorithms in a Linux environment.

**Conclusion :**

In conclusion, the implementation of FCFS, Preemptive SJF, and Non-Preemptive SJF scheduling algorithms in C++ using Linux Bash scripting highlights the strengths and trade-offs of each approach. FCFS offers simplicity but can result in long waiting times, while Preemptive SJF optimizes CPU efficiency at the risk of starvation for longer processes. Non-Preemptive SJF strikes a balance but still faces the challenge of delaying longer tasks. Overall, these algorithms provide valuable insights into CPU scheduling strategies and their impact on process management.

**Limitations :**

**- Cygwin Requirement :** Running the project on Windows needs Cygwin installation, which adds complexity.

**- Command Complexity :** Implementing FCFS and SJF in Linux involves many commands, challenging for users unfamiliar with Bash.

**- Burst Time Uncertainty :** SJF scheduling requires accurate burst time predictions, which can be hard to estimate.

**- Starvation in SJF :** Both Preemptive and Non-Preemptive SJF may cause longer processes to starve if shorter jobs keep arriving.

**References :**


1. **Performance Assessment of CPU Scheduling Algorithms: A Scenario-Based Approach with FCFS, RR, and SJF**. Journal of Computer Science, vol. 20, no. 5, 2024, pp. 972-985. Available: https://thescipub.com/pdf/jcssp.2024.972.985.pdf
2. **Optimizing Task Scheduling in Heterogeneous Computing Environments: A Comparative Analysis of CPU, GPU, and ASIC Platforms Using E2C Simulator**. arXiv preprint arXiv:2405.08187v1, May 2024. Available: https://arxiv.org/html/2405.08187v1
3. **GeeksforGeeks - Shortest Job First (SJF) CPU Scheduling**. GeeksforGeeks. Available: https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive