CSE 406 – Lab Report 1: First Come First Serve (FCFS) Scheduling Algorithm

Submitted By:

Sharif Md. Yousuf
ID: 22101128
Section: C-2
4th Year, 1st Semester
Spring 2025

Submitted To:

Atia Rahman Orthi
Lecturer
Department of Computer Science & Engineering
University of Asia Pacific

Date of Submission:
23 July, 2025 (Wednesday)

1 Problem Statement

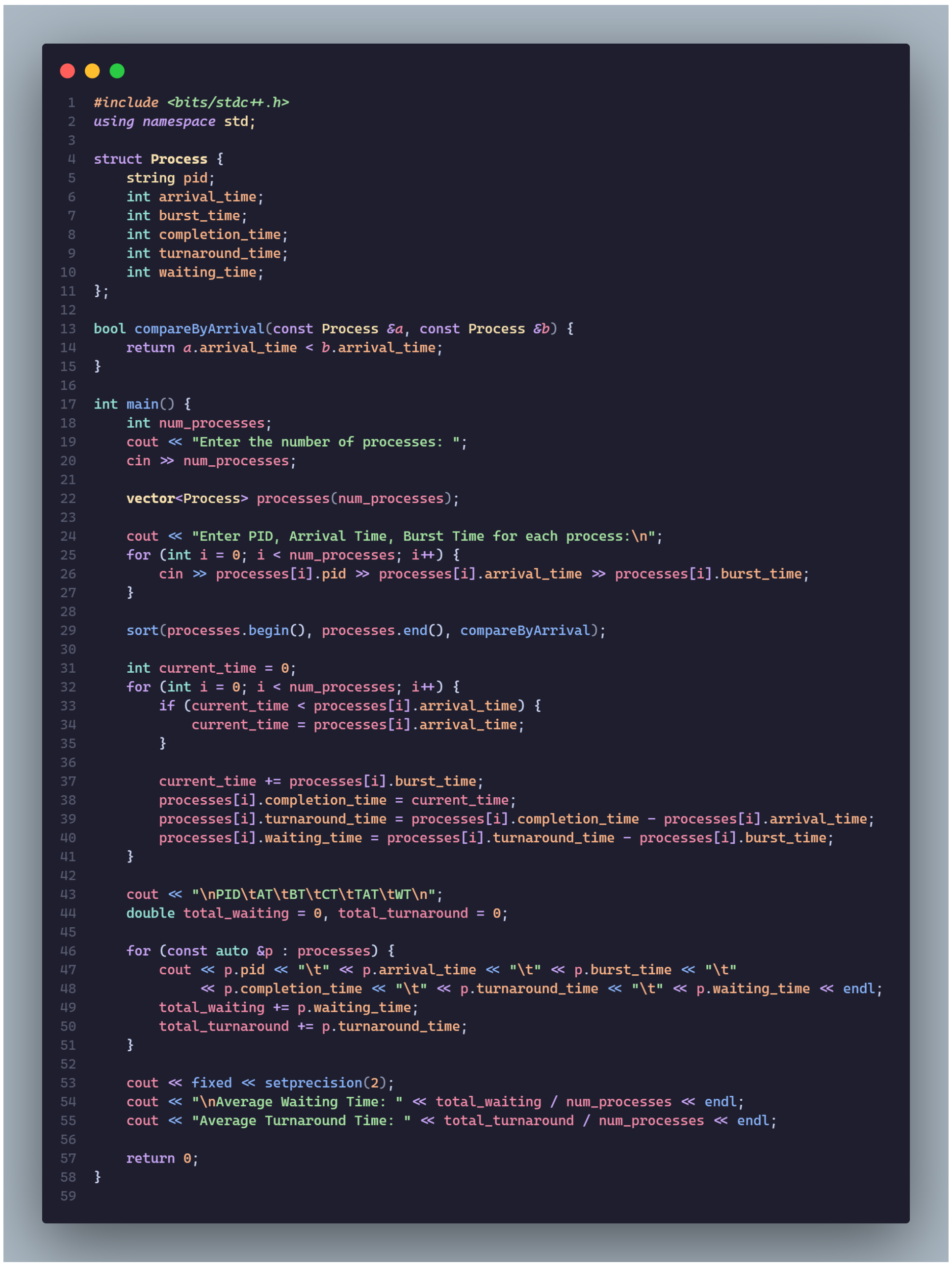
A program is needed to efficiently implement the First Come First Serve (FCFS) CPU scheduling algorithm. The algorithm should schedule processes according to their arrival times, meaning that the first process to arrive gets executed first. The implementation must first handle cases of processes arriving at different times and then compute the major scheduling parameters of completion time, turnaround time, and waiting time for each process.

2 Objective

In this lab we:

- Understood the working mechanism of the FCFS scheduling algorithm
- Implemented an optimized version that sorts processes by arrival time for better efficiency
- Calculated and displayed scheduling metrics for performance analysis
- Analyzed the advantages and limitations of non-preemptive FCFS scheduling

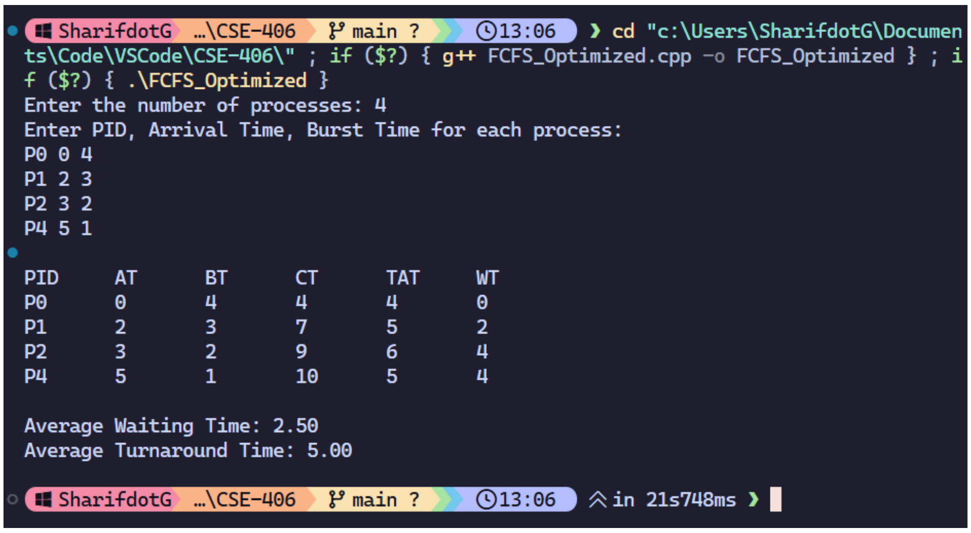
3 Source Code Screenshot



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

struct Process {
    string pid;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

bool compareByArrival(const Process &a, const Process &b) {
    return a.arrival_time < b.arrival_time;
}

int main() {
    int num_processes;
    cout << "Enter the number of processes: ";
    cin >> num_processes;

    vector<Process> processes(num_processes);

    cout << "Enter PID, Arrival Time, Burst Time for each process:\n";
    for (int i = 0; i < num_processes; i++) {
        cin >> processes[i].pid >> processes[i].arrival_time >> processes[i].burst_time;
    }

    sort(processes.begin(), processes.end(), compareByArrival);

    int current_time = 0;
    for (int i = 0; i < num_processes; i++) {
        if (current_time < processes[i].arrival_time) {
            current_time = processes[i].arrival_time;
        }

        current_time += processes[i].burst_time;
        processes[i].completion_time = current_time;
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }

    cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
    double total_waiting = 0, total_turnaround = 0;

    for (const auto &p : processes) {
        cout << p.pid << "\t" << p.arrival_time << "\t" << p.burst_time << "\t"
             << p.completion_time << "\t" << p.turnaround_time << "\t" << p.waiting_time << endl;
        total_waiting += p.waiting_time;
        total_turnaround += p.turnaround_time;
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Waiting Time: " << total_waiting / num_processes << endl;
    cout << "Average Turnaround Time: " << total_turnaround / num_processes << endl;

    return 0;
}
```

Figure 1: FCFS Optimized Algorithm Source Code

4 Output Screenshot



```
 SharifdotG  …\CSE-406  main ?  13:06  ❯ cd "c:\Users\SharifdotG\Documen
ts\Code\VSCode\CSE-406\" ; if ($?) { g++ FCFS_Optimized.cpp -o FCFS_Optimized } ; i
f ($?) { .\FCFS_Optimized }
Enter the number of processes: 4
Enter PID, Arrival Time, Burst Time for each process:
P0 0 4
P1 2 3
P2 3 2
P4 5 1

PID     AT      BT      CT      TAT     WT
P0      0       4       4       4       0
P1      2       3       7       5       2
P2      3       2       9       6       4
P4      5       1       10      5       4

Average Waiting Time: 2.50
Average Turnaround Time: 5.00

 SharifdotG  …\CSE-406  main ?  13:06  in 21s748ms ❯
```

Figure 2: FCFS Algorithm Execution Output

5 Discussion

The implemented FCFS optimized algorithm demonstrates several key improvements over a basic implementation. By sorting processes according to their arrival times at the beginning, the algorithm ensures proper execution order without repeatedly searching for the next process to execute.

The algorithm works by first collecting all process information, then sorting them by arrival time using a custom comparator function. During execution, it maintains a current time counter and processes each job in order. If the CPU is idle (current time is less than the next process arrival time), it advances the time to the next process arrival.

Key observations from the implementation include:

- Time complexity is reduced to O(n log n) due to sorting, compared to O(n^2) in naive implementations
- The algorithm handles CPU idle time effectively by jumping to the next arrival time
- Waiting time calculation is straightforward: turnaround time minus burst time
- The non-preemptive nature means once a process starts, it runs to completion

The results show that FCFS provides predictable scheduling behavior but may not be optimal for minimizing average waiting time, especially when long processes arrive before shorter ones.

6 Conclusion

The FCFS optimized scheduling algorithm successfully demonstrates the fundamental principles of CPU scheduling. While simple to understand and implement, the algorithm has inherent limitations such as the convoy effect, where short processes wait behind long ones. However, its fairness and simplicity make it suitable for systems where predictable execution order is more important than optimal performance metrics. The optimization through sorting provides better algorithmic efficiency while maintaining the core FCFS behavior.