CSE 406 – Lab Report 2: Shortest Job First (SJF) Non-Preemptive 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

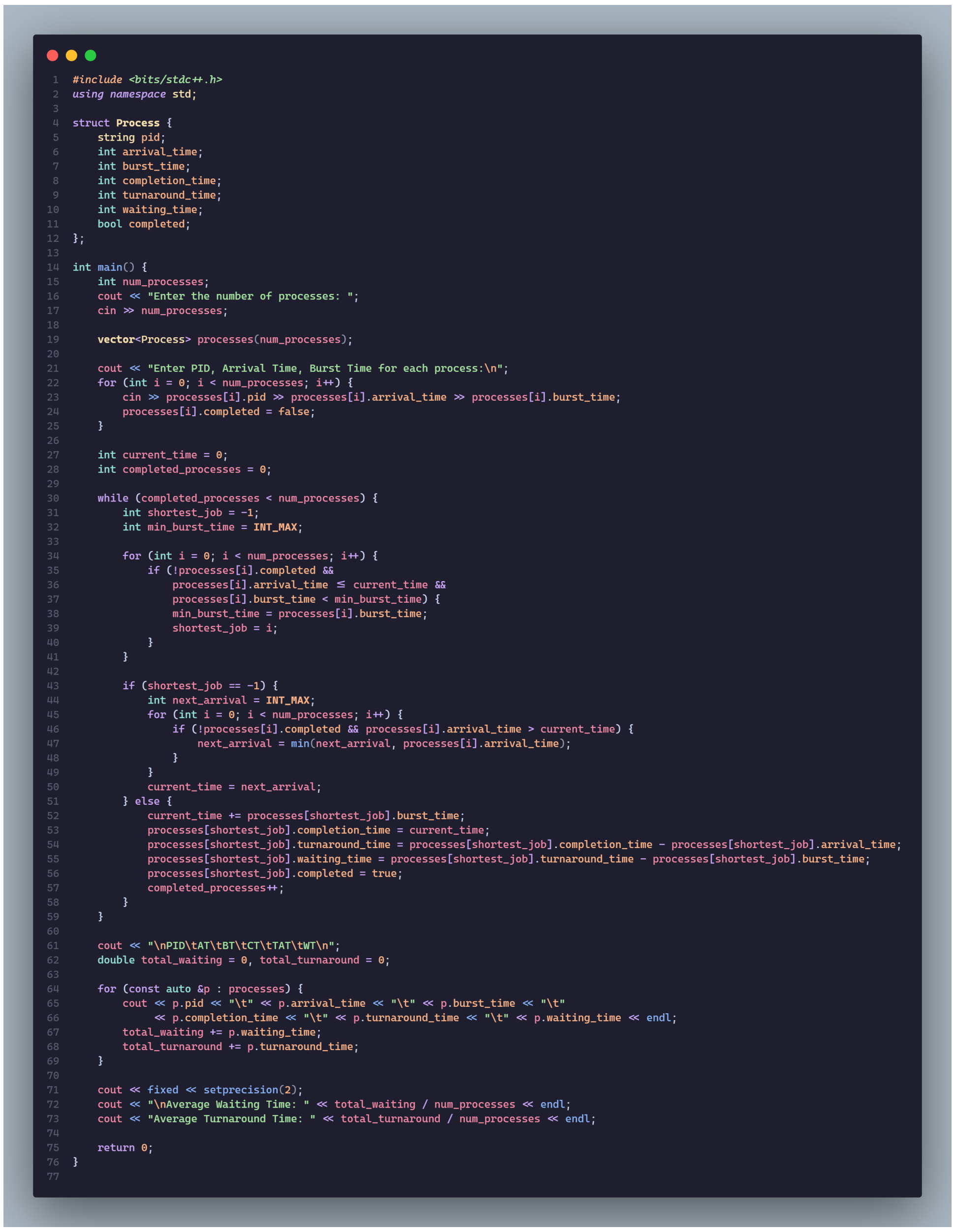
An application is to be developed to perform the Shortest Job First-non-preemptive (SJF-NP) CPU scheduling. The name is a descriptive one in that it chooses a process, which has the smallest burst time amongst all processes that may be ready at a time, for execution. Processes may have diverse arrival times and can be scheduled only if they have arrived. Completion time, turnaround time, and waiting time are the minimum set of performance measures the code must give.

2 Objective

In this lab we:

- Understood the operational principles of the SJF scheduling algorithm
- Implemented a non-preemptive version that selects jobs based on burst time
- Handled dynamic process arrivals and CPU idle time scenarios
- Calculated and analyzed scheduling performance metrics
- Compared the efficiency of SJF with other scheduling algorithms

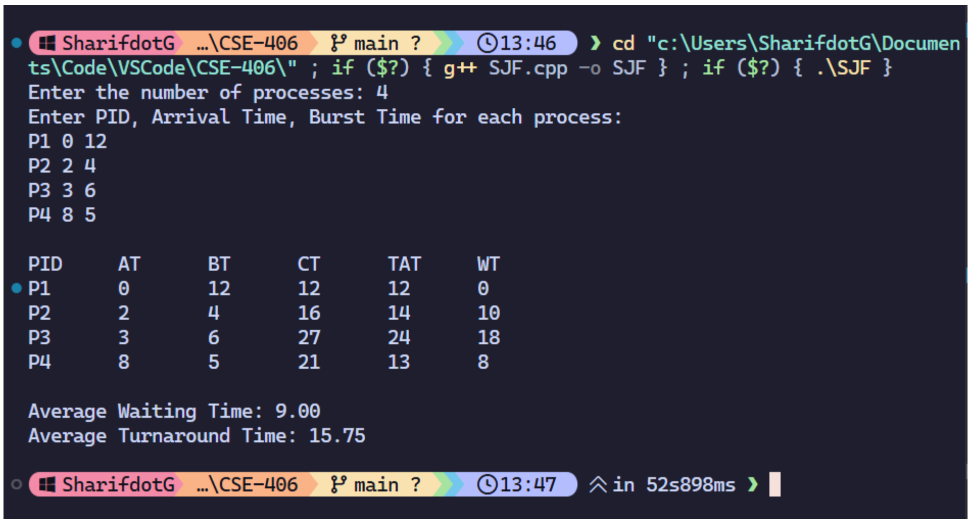
3 Source Code Screenshot



```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 completed;
};

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;
        processes[i].completed = false;
    }

    int current_time = 0;
    int completed_processes = 0;

    while (completed_processes < num_processes) {
        int shortest_job = -1;
        int min_burst_time = INT_MAX;

        for (int i = 0; i < num_processes; i++) {
            if (!processes[i].completed &&
                processes[i].arrival_time <= current_time &&
                processes[i].burst_time < min_burst_time) {
                min_burst_time = processes[i].burst_time;
                shortest_job = i;
            }
        }

        if (shortest_job == -1) {
            int next_arrival = INT_MAX;
            for (int i = 0; i < num_processes; i++) {
                if (!processes[i].completed && processes[i].arrival_time > current_time) {
                    next_arrival = min(next_arrival, processes[i].arrival_time);
                }
            }
            current_time = next_arrival;
        } else {
            current_time += processes[shortest_job].burst_time;
            processes[shortest_job].completion_time = current_time;
            processes[shortest_job].turnaround_time = processes[shortest_job].completion_time - processes[shortest_job].arrival_time;
            processes[shortest_job].waiting_time = processes[shortest_job].turnaround_time - processes[shortest_job].burst_time;
            processes[shortest_job].completed = true;
            completed_processes++;
        }
    }

    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: SJF Non-Preemptive Algorithm Source Code

4 Output Screenshot



```
● SharifdotG  …\CSE-406  main ?  13:46  ❯ cd "c:\Users\SharifdotG\Documen
ts\Code\VSCode\CSE-406\" ; if ($?) { g++ SJF.cpp -o SJF } ; if ($?) { .\SJF }
Enter the number of processes: 4
Enter PID, Arrival Time, Burst Time for each process:
P1 0 12
P2 2 4
P3 3 6
P4 8 5

PID     AT      BT      CT      TAT     WT
P1      0       12      12      12      0
P2      2       4       16      14      10
P3      3       6       27      24      18
P4      8       5       21      13      8

Average Waiting Time: 9.00
Average Turnaround Time: 15.75

○ SharifdotG  …\CSE-406  main ?  13:47  in 52s898ms ❯
```

Figure 2: SJF Algorithm Execution Output

5 Discussion

The SJF non-preemptive algorithm implementation provides an optimal solution for minimizing average waiting time among all non-preemptive scheduling algorithms. The algorithm operates by continuously selecting the process with the shortest burst time from the pool of arrived processes.

The implementation uses a while loop that continues until all processes are completed. At each iteration, it searches through all processes to find the one with the minimum burst time among those that have already arrived and are not yet completed. If no process is available (CPU idle situation), the algorithm advances the current time to the next process arrival time.

Important characteristics observed in this implementation:

- The algorithm guarantees optimal average waiting time for non-preemptive schedulers
- It handles CPU idle time by jumping to the next arrival time when no processes are ready
- The selection process has O(n) complexity for each scheduling decision
- Starvation can occur if shorter processes keep arriving before longer ones complete

The algorithm effectively demonstrates the trade-off between optimality and complexity. While it provides better average performance than FCFS, it requires knowledge of burst times in advance, which is often not practical in real systems.

6 Conclusion

The SJF non-preemptive scheduling algorithm successfully minimizes average waiting time and provides optimal performance for batch processing systems where burst times are known. However, its practical implementation faces challenges due to the difficulty of predicting actual burst times. The algorithm works well in scenarios with predictable workloads but may cause starvation for longer processes. Despite these limitations, SJF serves as an important theoretical benchmark for comparing other scheduling algorithms and understanding the principles of optimal CPU scheduling.