

CSE 406 – Lab Report 4: Non-Preemptive Priority 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:
16 August, 2025 (Saturday)**

1 Problem Statement

We are to implement a non-preemptive Priority CPU scheduling algorithm. Each process is defined by: Process ID, Arrival Time (AT), Burst Time (BT), and Priority (PR) where a lower numerical value indicates a higher scheduling priority. The scheduler must always select, among the processes that have arrived and not yet completed, the process with the highest priority (lowest PR). Ties are broken first by earliest arrival time and then lexicographically by process ID. The program must compute Completion Time (CT), Turnaround Time ($TAT = CT - AT$) and Waiting Time ($WT = TAT - BT$) for every process and report average TAT and WT. Idle CPU periods must be handled when no process has arrived at the current time.

Input

- First line: an integer n ($n > 0$) denoting the number of processes.
- Next n lines: each line contains four tokens: PID AT BT PR
 - PID: a string without spaces (e.g., P1, A, T3)
 - AT: non-negative integer arrival time
 - BT: positive integer CPU burst time
 - PR: non-negative integer priority (smaller value = higher priority)

Example:

```
5
P1 0 4 2
P2 1 3 1
P3 2 2 3
P4 4 4 2
P5 5 3 1
```

Output

- A header line listing the column titles, e.g.:

PID AT BT PR CT TAT WT
- One line per process in the internal (sorted/executed) order with computed metrics.
- Two summary lines giving average Turn Around Time and average Waiting Time to two decimal places.

Example (for the sample above — values depend on scheduling order):

```
PID AT BT PR CT TAT WT
P1 0 4 2 4 4 0
P2 1 3 1 8 7 4
P3 2 2 3 10 8 6
P4 4 4 2 14 10 6
```

P5 5 3 1 17 12 9

Average Turn Around Time: 8.20

Average Waiting Time: 5.00

2 Objective

In this lab we aimed to:

- Understand the working principle of non-preemptive priority-based scheduling
- Implement deterministic tie-breaking for reproducible ordering
- Correctly manage CPU idle intervals by advancing to the next arrival
- Compute per-process and average performance metrics (CT, TAT, WT)
- Analyze fairness, starvation risk, and efficiency characteristics

3 Source Code Screenshot

If included, add a screenshot named `code.png` in this folder for compilation. (Placeholder omitted here.)

```

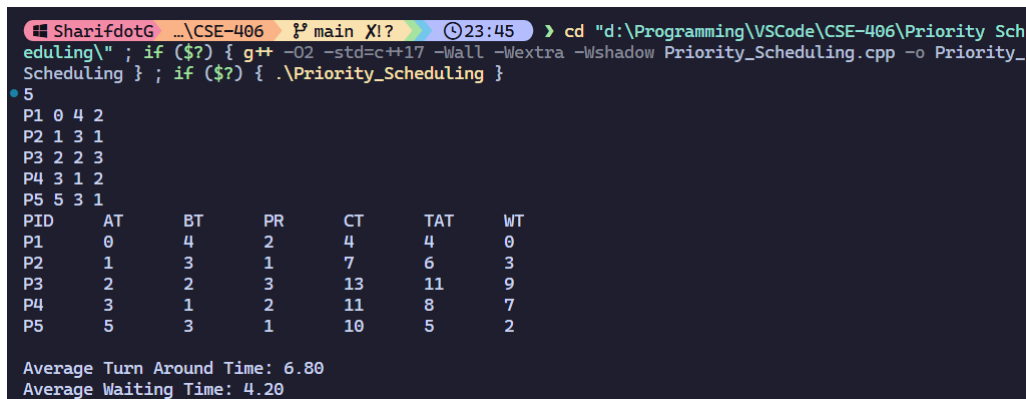
Priority_Scheduling.cpp

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 struct Process {
5     string pid;
6     int at = 0;
7     int bt = 0;
8     int pr = 0;
9     int ct = 0;
10    int tat = 0;
11    int wt = 0;
12    bool done = false;
13 };
14
15 int main() {
16     int n;
17     if (!(cin >> n) || n < 0) {
18         return 0;
19     }
20
21     vector<Process> ps(n);
22     for (int i = 0; i < n; i++) {
23         cin >> ps[i].pid >> ps[i].at >> ps[i].bt >> ps[i].pr;
24     }
25
26     sort(ps.begin(), ps.end(), [](const Process &a, const Process &b) {
27         if (a.at != b.at) return a.at < b.at;
28         if (a.pr != b.pr) return a.pr < b.pr;
29         return a.pid < b.pid;
30     });
31
32     long long time_now = 0;
33     if (!ps.empty()) time_now = ps.front().at;
34
35     int completed = 0;
36     double total_tat = 0.0, total_wt = 0.0;
37
38     while (completed < n) {
39         int pick = -1;
40
41         for (int i = 0; i < n; i++) {
42             if (!ps[i].done && ps[i].at <= time_now) {
43                 if (pick == -1) pick = i;
44                 else if (ps[i].pr < ps[pick].pr) pick = i;
45                 else if (ps[i].pr == ps[pick].pr && ps[i].at < ps[pick].at) pick = i;
46             }
47         }
48
49         if (pick == -1) {
50             int next_at = INT_MAX;
51             for (auto &p : ps) if (!p.done) next_at = min(next_at, p.at);
52             time_now = max<long long>(time_now, next_at);
53             continue;
54         }
55
56         time_now += ps[pick].bt;
57         ps[pick].ct = (int)time_now;
58         ps[pick].tat = ps[pick].ct - ps[pick].at;
59         ps[pick].wt = ps[pick].tat - ps[pick].bt;
60         ps[pick].done = true;
61         completed++;
62
63         total_tat += ps[pick].tat;
64         total_wt += ps[pick].wt;
65     }
66
67     cout << "PID\tAT\tBT\tPR\tCT\tTAT\tWT\n";
68     for (const auto &p : ps) {
69         cout << p.pid << '\t' << p.at << '\t' << p.bt << '\t' << p.pr
70             << '\t' << p.ct << '\t' << p.tat << '\t' << p.wt << '\n';
71     }
72
73     cout << fixed << setprecision(2);
74     cout << "\nAverage Turn Around Time: " << (n ? total_tat / n : 0.0) << "\n";
75     cout << "Average Waiting Time: " << (n ? total_wt / n : 0.0) << "\n";
76
77     return 0;
78 }
79

```

Figure 1: Priority Scheduling Source Code

4 Output Screenshot



```
## SharifdotG ...\CSE-406 }$ main X! ? 23:45 > cd "d:\Programming\VSCode\CSE-406\Priority Sch
eduling\" ; if ($?) { g++ -O2 -std=c++17 -Wall -Wextra -Wshadow Priority_Scheduling.cpp -o Priority_
Scheduling } ; if ($?) { .\Priority_Scheduling }
5
P1 0 4 2
P2 1 3 1
P3 2 2 3
P4 3 1 2
P5 5 3 1
PID    AT    BT    PR    CT    TAT    WT
P1      0     4     2     4     4     0
P2      1     3     1     7     6     3
P3      2     2     3    13    11     9
P4      3     1     2    11     8     7
P5      5     3     1    10     5     2

Average Turn Around Time: 6.80
Average Waiting Time: 4.20
```

Figure 2: Program Output

5 Discussion

Priority scheduling favors higher-priority (numerically smaller) processes, improving response time for critical tasks. However, non-preemptive behavior means once a lower-priority but long job starts, an immediately arriving higher-priority job must wait until completion, increasing potential latency.

Starvation Risk: If high-priority processes continue to arrive, very low-priority jobs could experience indefinite postponement. Aging (gradually decreasing priority values over waiting time) is a common mitigation not implemented here.

Determinism: Explicit tie-breaking ensures reproducibility which aids debugging and grading.

Idle Handling: Jumping the clock directly to the next arrival prevents unnecessary iteration simulating empty time units.

Compared to FCFS, average waiting time can be lower when priorities correlate with short tasks. Compared to SJF, priority may not always reflect burst length, so optimal waiting time minimization is not guaranteed. Unlike Round Robin, this algorithm is not time-sharing and can produce larger response times for late-arriving urgent tasks unless preemption or aging is added.

6 Conclusion

The non-preemptive priority scheduling implementation correctly computes essential metrics and demonstrates trade-offs between responsiveness for important tasks and fairness for low-priority workloads. Extensions such as preemptive priority scheduling and aging would address starvation and improve responsiveness for newly arrived critical tasks.