CSE 406 – Lab Report 3: Round Robin (RR) 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:
29 July, 2025 (Tuesday)

1 Problem Statement

I need to implement the Round Robin CPU scheduling algorithm to fairly distribute CPU time among multiple processes. In this algorithm, each process gets a small unit of CPU time called a time quantum, and if a process doesn't complete in that time, it's put back at the end of the queue to wait for another turn.

For this lab, I'm using the following sample case:

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

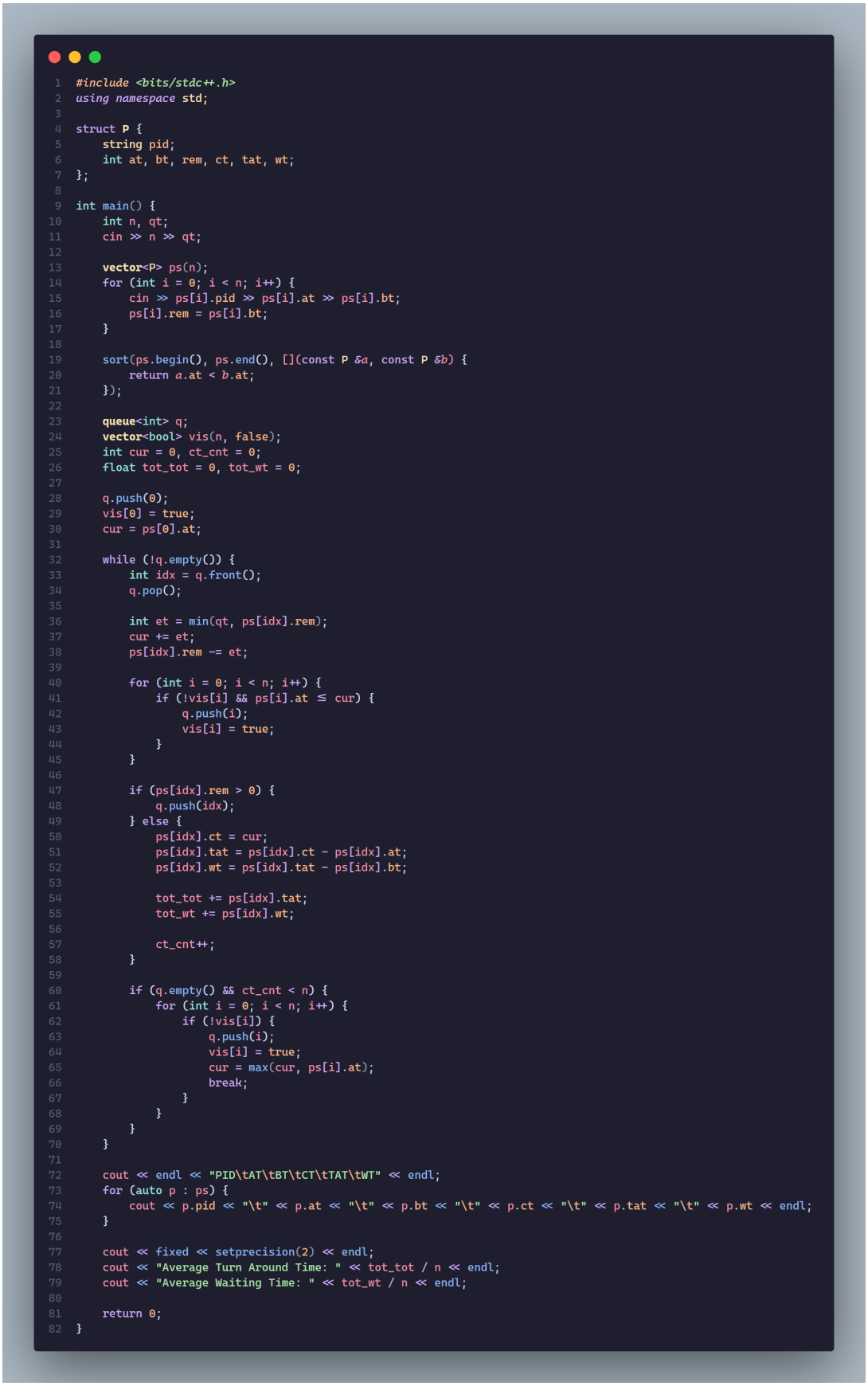
Where the first line contains the number of processes (5) and the time quantum (2), and each subsequent line contains the process ID, arrival time, and burst time.

2 Objective

In this lab, I wanted to:

- Create my own implementation of the Round Robin scheduling algorithm
- See how the time quantum affects process execution
- Calculate important metrics like completion time, turnaround time, and waiting time
- Understand how processes share CPU time fairly
- Get hands-on experience with a preemptive scheduling technique

3 Source Code Screenshot



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

struct P {
    string pid;
    int at, bt, rem, ct, tat, wt;
};

int main() {
    int n, qt;
    cin >> n >> qt;

    vector<P> ps(n);
    for (int i = 0; i < n; i++) {
        cin >> ps[i].pid >> ps[i].at >> ps[i].bt;
        ps[i].rem = ps[i].bt;
    }

    sort(ps.begin(), ps.end(), [](const P &a, const P &b) {
        return a.at < b.at;
    });

    queue<int> q;
    vector<bool> vis(n, false);
    int cur = 0, ct_cnt = 0;
    float tot_tot = 0, tot_wt = 0;

    q.push(0);
    vis[0] = true;
    cur = ps[0].at;

    while (!q.empty()) {
        int idx = q.front();
        q.pop();

        int et = min(qt, ps[idx].rem);
        cur += et;
        ps[idx].rem -= et;

        for (int i = 0; i < n; i++) {
            if (!vis[i] && ps[i].at <= cur) {
                q.push(i);
                vis[i] = true;
            }
        }

        if (ps[idx].rem > 0) {
            q.push(idx);
        } else {
            ps[idx].ct = cur;
            ps[idx].tat = ps[idx].ct - ps[idx].at;
            ps[idx].wt = ps[idx].tat - ps[idx].bt;

            tot_tot += ps[idx].tat;
            tot_wt += ps[idx].wt;

            ct_cnt++;
        }

        if (q.empty() && ct_cnt < n) {
            for (int i = 0; i < n; i++) {
                if (!vis[i]) {
                    q.push(i);
                    vis[i] = true;
                    cur = max(cur, ps[i].at);
                    break;
                }
            }
        }
    }

    cout << endl << "PID\tAT\tBT\tCT\tTAT\tWT" << endl;
    for (auto p : ps) {
        cout << p.pid << "\t" << p.at << "\t" << p.bt << "\t" << p.ct << "\t" << p.tat << "\t" << p.wt << endl;
    }

    cout << fixed << setprecision(2) << endl;
    cout << "Average Turn Around Time: " << tot_tot / n << endl;
    cout << "Average Waiting Time: " << tot_wt / n << endl;

    return 0;
}
```

Figure 1: Round Robin C++ Code Implementation

4 Output Screenshot

Figure 2: Output of Round Robin Algorithm Execution

As you can see from the output above, I ran the program with 5 processes and a time quantum of 2. The results show how each process was handled:

- Process P1 arrived at time 0, needed 5 time units, and finished at time 13
- Process P2 arrived at time 1, needed 3 time units, and finished at time 12
- Process P3 arrived at time 2, needed just 1 time unit, and finished quickly at time 5
- Process P4 arrived at time 3, needed 2 time units, and finished at time 9
- Process P5 arrived at time 4, needed 3 time units, and finished at time 14

The average turnaround time came to 8.60 time units, and the average waiting time was 5.80 time units.

5 Discussion

Working with the Round Robin algorithm was really interesting! Here's what I learned:

The key thing about Round Robin is how it gives each process a fair shot at running. No process can hog the CPU for too long, which means that even if a long process comes in early, shorter processes that arrive later don't have to wait forever.

I found that the time quantum (2 units in my example) really matters:

- If it's too small, the system spends too much time switching between processes
- If it's too large, it starts behaving more like a first-come-first-served system
- Finding the "sweet spot" depends on the types of processes you're running

My implementation used a queue to keep track of which process should run next, along with a visited array so I wouldn't add the same process to the queue multiple times. This made sure everything ran smoothly.

One cool thing about Round Robin is how responsive it feels. Shorter processes get completed relatively quickly since they don't have to wait for long processes to finish completely before getting CPU time.

6 Conclusion

Overall, this Round Robin scheduling lab was really valuable! I got to see firsthand how a time-sharing system works and why it's so important in modern operating systems.

The most important takeaway for me was understanding the tradeoff between fairness and efficiency. Round Robin might not always give the absolute fastest average turnaround time, but it makes sure every process gets its fair share of attention.

I also learned that implementation details matter — using the right data structures (like queues) and handling special cases (like CPU idle time) can make a big difference in how well the algorithm performs.

If I were to expand on this project in the future, I might try implementing different time quantum values to see how they affect performance, or maybe add process priorities so that more important processes get more frequent turns.

This lab helped me understand why Round Robin scheduling is still widely used in modern operating systems and time-sharing systems — it's a simple but powerful approach that balances the needs of multiple processes effectively.