Operating Systems Project1 Report

Process Scheduling Simulation

Course: 4320/6320 – Operating Systems (Fall 2025)

Student: Yangyang Jiang

Github: https://github.com/techyangj/operation-system-project-1

1 Introduction

This project implemented a process scheduling simulation in **C**. The goal was to simulate how an operating system uses the CPU scheduling algorithm to schedule processes. The process scheduler determines which process runs on the CPU at a given time. This project allowed me to learn how real operating systems handle CPU scheduling, furthering my understanding of how operating systems work.

This program reads processes from the input file **processes.txt**, schedules them according to the selected scheduling algorithm, and outputs the wait time (WT), turnaround time (TAT), completion time (C) of each process, as well as the averages and the corresponding Gantt chart.

The implemented two algorithms are:

- **First-Come, First-Served (FCFS)**
- **Shortest Job First (SJF, non-preemptive)**

2 Implementation Details

2.1 Input Handling

Processes are defined in a text file with four fields: **PID, Arrival Time, Burst Time, and Priority.** Lines are parsed into an array of **struct Process**. Empty lines and comments are skipped.

2.2 Command-Line Arguments

The program supports flexible execution using command-line arguments:

- `-h` or `--help` Displays usage information.
- `-a <algo>` Selects the scheduling algorithm:
  - `fcfs` – Run only First-Come, First-Served
  - `sjf` – Run only Shortest Job First
  - `both` – Run both algorithms
- `-f <file>` Specify an input file (default: **processes.txt**).

I think users can quickly test different algorithms and datasets without recompiling or modifying the source code.

2.3 FCFS

- Processes are sorted by arrival time, then by PID.
- Each process runs in order of arrival.
- CPU idle periods are recorded when no process has arrived.
- For each process:
  - Completion Time $C = finish\ time$
  - Waiting Time $WT = start - arrival$
  - Turnaround Time $TAT = C - arrival$

2.4 SJF (Non-Preemptive)

- At each decision point, the ready process with the shortest burst time is chosen.
- Ties are resolved by arrival time, then by PID.
- Idle segments are added if CPU is waiting for arrivals.
- WT, TAT, and C are computed as in FCFS.

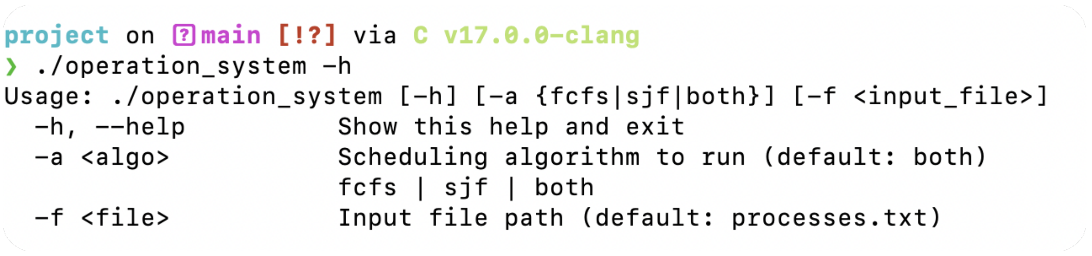
2.5 Gantt Chart

Execution order is visualized as a text-based Gantt chart. For example:

```
| P1 | Idle | P2 | P3 |
0    2      5    7    10
```

3 Results

3.1 Command Argument



```
project on main [!?] via C v17.0.0-clang
❯ ./operation_system -h
Usage: ./operation_system [-h] [-a {fcfs|sjf|both}] [-f <input_file>]
  -h, --help           Show this help and exit
  -a <algo>            Scheduling algorithm to run (default: both)
                       fcfs | sjf | both
  -f <file>            Input file path (default: processes.txt)
```

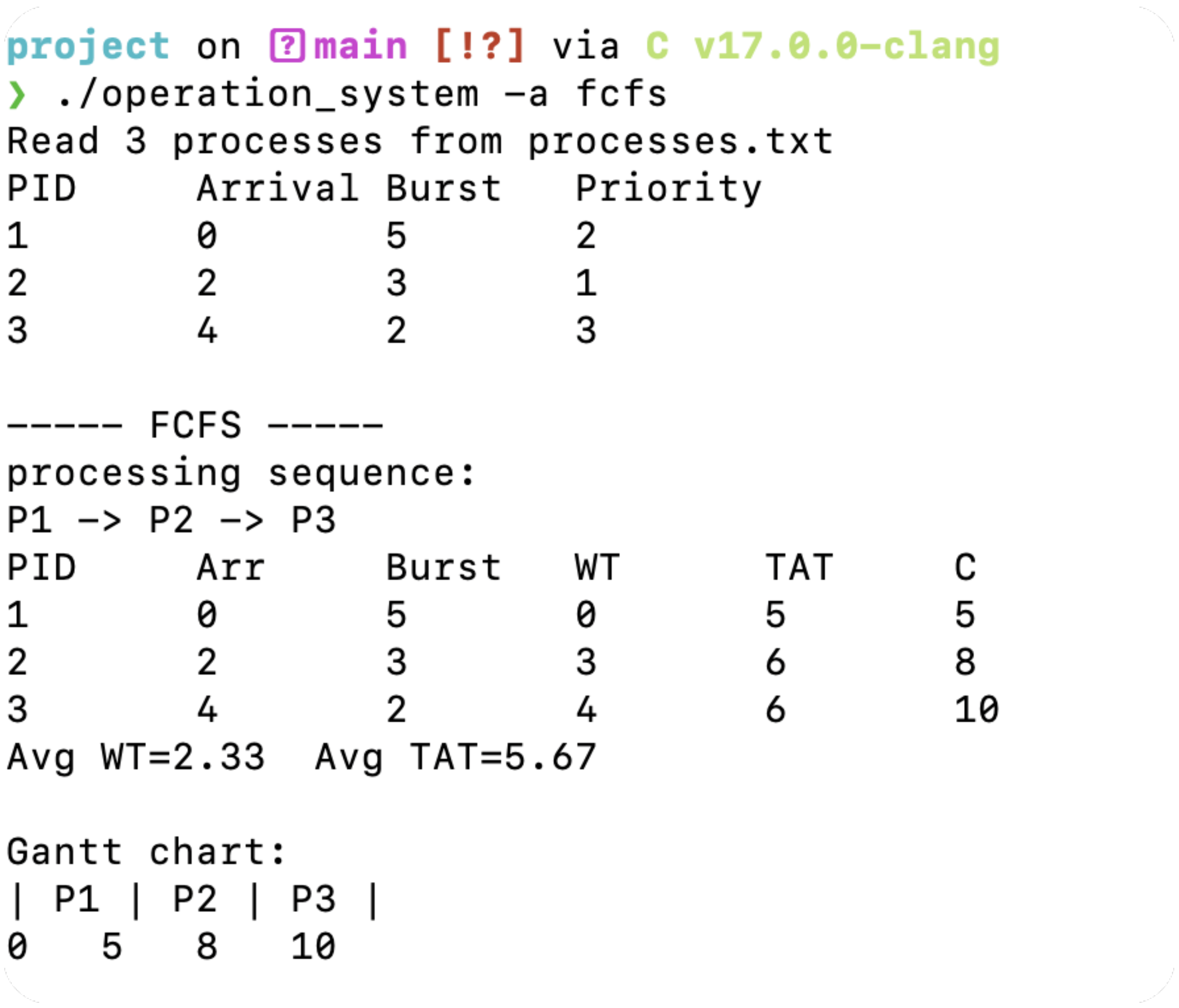
Figure 1: Command Argument.

3.2 Sample Input

```
PID  Arrival_Time  Burst_Time  Priority
1    0             5           2
2    2             3           1
3    4             2           3
```

3.3 FCFS Output

The following figure shows the detailed results of the FCFS scheduling algorithm, including the process execution order and the corresponding Gantt chart.

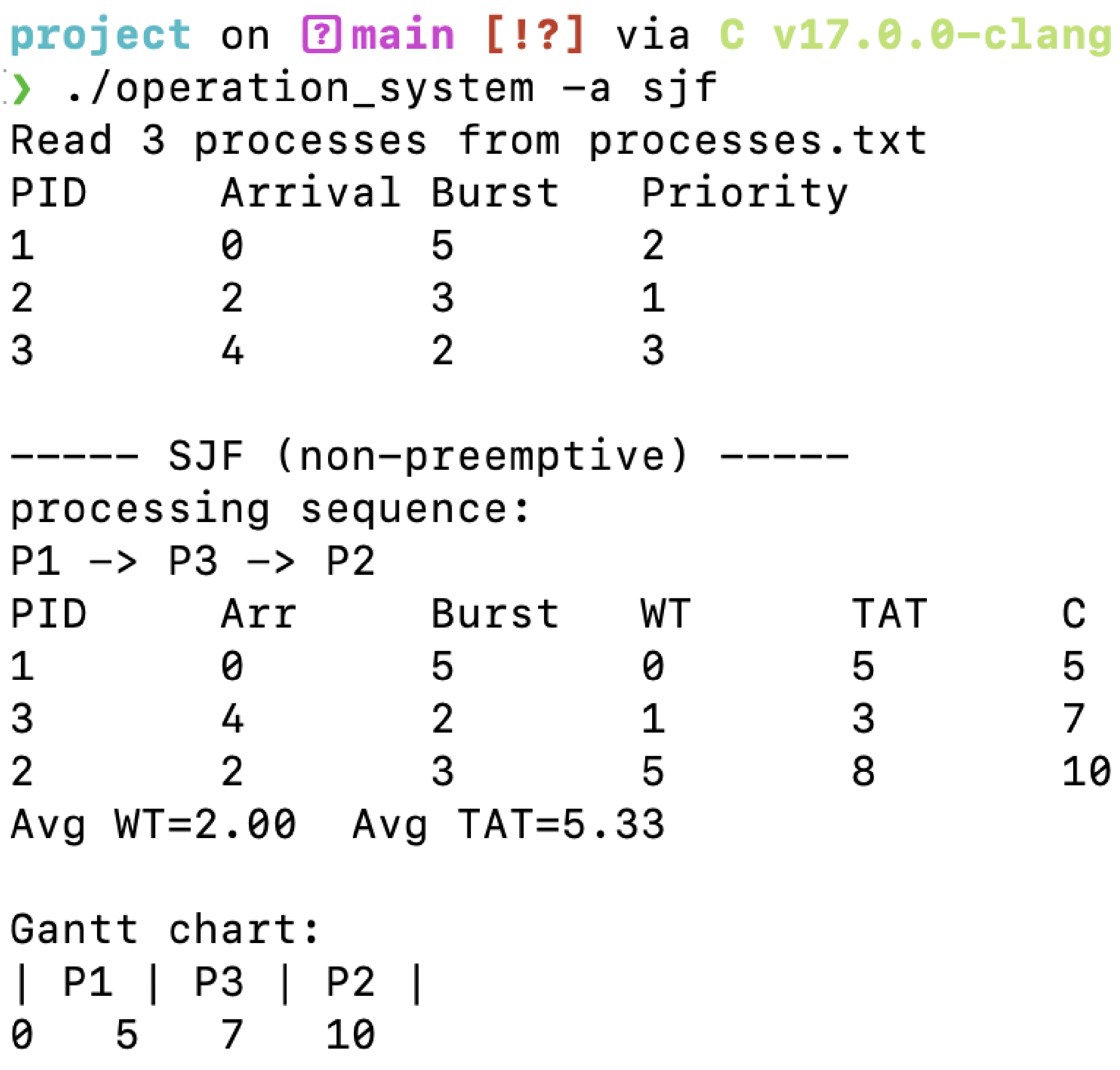


```
project on ?main [!?] via C v17.0.0-clang
❯ ./operation_system -a fcfs
Read 3 processes from processes.txt
PID     Arrival Burst   Priority
1       0       5       2
2       2       3       1
3       4       2       3

----- FCFS -----
processing sequence:
P1 -> P2 -> P3
PID     Arr     Burst   WT      TAT     C
1       0       5       0       5       5
2       2       3       3       6       8
3       4       2       4       6       10
Avg WT=2.33  Avg TAT=5.67

Gantt chart:
| P1 | P2 | P3 |
0    5    8    10
```

Figure 2: FCFS Processes and Gantt Chart for the sample workload.

3.4 SJF Output

The following figure illustrates the SJF (non-preemptive) scheduling results. It shows how shorter jobs are scheduled earlier, reducing the average waiting time.



```
project on ?main [!?] via C v17.0.0-clang
❯ ./operation_system -a sjf
Read 3 processes from processes.txt
PID     Arrival Burst   Priority
1       0       5       2
2       2       3       1
3       4       2       3

----- SJF (non-preemptive) -----
processing sequence:
P1 -> P3 -> P2
PID     Arr     Burst   WT      TAT     C
1       0       5       0       5       5
3       4       2       1       3       7
2       2       3       5       8       10
Avg WT=2.00  Avg TAT=5.33

Gantt chart:
| P1 | P3 | P2 |
0   5   7   10
```

Figure 3: SJF Processes and Gantt Chart for the sample workload.

4 Analysis

The FCFS scheduling algorithm is simple to implement using C code, but it has significant drawbacks. If a process takes a long time to execute, subsequent processes may continue to wait, ultimately leading to slackness. In contrast, the SJF scheduling algorithm prioritizes shorter jobs, effectively reducing average waiting time and turnaround time, resulting in better overall system performance than the FCFS scheduling algorithm.

5 Challenges

Implementing the SJF algorithm is significantly more difficult than the FCFS algorithm. Unlike FCFS, SJF requires checking which processes have reached their current time and then selecting the one with the shortest time. This is primarily due to the difficulty of tracking which processes have completed and which are still waiting. However, this problem is solved by maintaining the `done[]` array and carefully updating the timeline. Optional memory management is difficult to extend. Using memory allocation requires introducing new data types and methods. The amount of code changes required is too large for the current stage of the project, so memory management is left for a later date.

6 Conclusion

This project successfully implemented and tested two CPU scheduling algorithms in C: FCFS and SJF.

- The program reads process data from a file, computes WT/TAT, and displays a Gantt chart.
- FCFS is fair but can penalize shorter processes.
- SJF optimizes average WT and TAT but assumes burst times are known in advance.