Trajuan Smith | Operating Systems | Roya Hosseini
Project 1 | Fall 2025

In this project we implemented a CPU scheduling simulator in JAVA, that reads process data from the “processes.txt” file, containing Process ID (PID), arrival time, burst time, and priority. Processes are executed/scheduled using 2 algorithms: First-Come, First Served (FCFS) and Priority Scheduling (Preemptive, Lower Number = Higher priority). The program creates a timeline of CPU execution, prints a Gantt Chart of the processes, and prints the Waiting Time (WT) and Turnaround Time (TAT) for each process, as well as the average WT and TAT, along with CPU Utilization. Here FCFS is modeled as a queue since processes run to completion in the order they arrive. Priority (Preemptive) checks the ready queue each time unit for higher priority processes, this reduces WT/TAT for high priority short processes at the cost of more context switches.

The following details how the simulator is structured and how FCFS and Priority were implemented:
Proc.java deals with process creation and defines the process data model that is used by the schedulers. Stores the process’ inputs (PID, arrivalTime, burstTime, and priority) as well as its run state variables (remaining, completionTime). Also provides helper methods to compute waitingTime and turnaroundTime, as well as clone()/deepCopy() so each algorithm can simulate and use the process without mutating the original list.

ProcessParser.java reads process.txt file, parses each non-header line into Proc(pid, arrival, burst, priority), and returns a List. It slips empty/header lines, validates the field count, and throws on bad lines or duplicate PIDs. It’s called from Main.java (pointing to src/processes.txt).

FCFS.java, sorts processes by arrival, the pid. For each process: if the CPU is idle before its arrival, insert an IDLE slice; then run to completion (addSlice(pid, start, end)), therefore setting startTime, completionTime, and remaining to 0. Modelling a simple ready queue without preemption.

Priority.java, this maintains a priority queue that is ordered by priority → arrival → PID (lower number = higher priority). Here time units are advanced in 1-unit ticks, it enqueues any arrivals <= t, runs the top process for one tick, decrements ‘remaining’, and either records the completionTime (if finished) or initiates a re-queue (allowing a preemption if a higher priority process has arrived). Inserts IDLE slices when the queue is empty (the CPU is idle) and sets a Process’s startTime to the first time it gets a CPU.

GanttEntry.java, is a simple record for one CPU slice: the pid, startTime, and endTime.

GanttResult.java, this collects the whole run of process calculations and simulations: a timeline of GantEntry Slices (from merging adjacent same-PID slices), a 'byPid' map of the final states of the processes, as well as methods to addSlice(), recordFinal(), print the text Gantt chart with aligned time ticks (printChart()) and compute the specific and average WT & TAT for each process (printMetrics()). Also prints CPU utilization.

IScheduler.java, interface to declare the contract of a single method, GanttResult run(List<Proc> input) so any of the 2 algorithms can be plugged in polymorphically.

Results:

Sample [1] - Process.txt:

PID	Arrival_Time	Burst_Time	Priority
1	0	5	2
2	2	3	1
3	4	2	3
4	5	15	0
5	3	2	0

FCFS:

```
Choose Algorithm:
[1] - First Come First Serve
[2] - Priority Scheduling (Preemptive)

Choose (integer): 1
 === FCFS ===
|    P1    |  P2  | P5 | P3 |             P4             |
0          5      8   10   12                           27

Per-process metrics:
PID WT TAT
  1  0   5
  2  3   6
  3  6   8
  4  7  22
  5  5   7
Avg WT = 4.20, Avg TAT = 9.60
CPU Utilization = 100.00%

Process finished with exit code 0
```

Priority:

```
Choose Algorithm:
[1] - First Come First Serve
[2] - Priority Scheduling (Preemptive)

Choose (integer): 2
 === PRIORITY ===
| P1 |P2| P5 |             P4             | P2 |  P1  | P3 |
0    2  3    5                           20   22     25   27

Per-process metrics:
PID WT TAT
  1 20  25
  2 17  20
  3 21  23
  4  0  15
  5  0   2
Avg WT = 11.60, Avg TAT = 17.00
CPU Utilization = 100.00%

Process finished with exit code 0
```

- Due to a long burst, high priority process (P4, priority 0, burst **15**) arriving at t=5, preemptive priority starves earlier and shorter lower priority processes (P1, P2, P3), thereby inflating the average WT/TAT
- Vs FCFS which largely depends on the size of which process arrives first. If a larger process comes first then later processes can be starved resulting in larger WTs and TATs but here the larger job arrived last lowering WT and TAT.

Sample [1] - Process.txt

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

FCFS:

```
Choose Algorithm:
[1] - First Come First Serve
[2] - Priority Scheduling (Preemptive)

Choose (integer): 1
 === FCFS ===
|    P1    |   P2   | P5 | P3 |P4|
0          5        9   11   13 14

Per-process metrics:
PID WT TAT
  1  0   5
  2  3   7
  3  7   9
  4  8   9
  5  6   8
Avg WT = 4.80, Avg TAT = 7.60
CPU Utilization = 100.00%

Process finished with exit code 0
```

Priority:

```
Choose Algorithm:
[1] - First Come First Serve
[2] - Priority Scheduling (Preemptive)

Choose (integer): 2
 === PRIORITY ===
| P1 |P2| P5 |P4|  P2  |  P1  | P3 |
0    2  3    5  6      9     12   14

Per-process metrics:
PID WT TAT
  1  7  12
  2  3   7
  3  8  10
  4  0   1
  5  0   2
Avg WT = 3.60, Avg TAT = 6.40
CPU Utilization = 100.00%

Process finished with exit code 0
```

- With shorter top-priority processes, preemptive priority finishes high priority work quickly giving lower priority processes a chance and improves average WT and TAT vs FCFS

Challenges & Solutions

- I struggled in Formatting the Gantt chart to align with the time ticks, since I was mostly using guess work and random spaces. In order to fix this I used a fixed scale (SCALE) to size each box (| |), recording the start and end column for each slice (| Pk |). I then drew the time tick line as a char array with the **same length** as the boxes, then wrote each time label beginning at its exact recorded edge column. Also added a helper method, **center()**, so each label stays readable while times stay locked to box edges.
- Struggled in understanding how to access and print the CPU usage, a simple but reluctant google search solved this.