OS-CPU Scheduling Simulator:Comparative Analysis of FCFS, SRTF, and MLFQ

Lindsey Ferguson

College of Computing and Software Engineering, Kennesaw State University

Chris Regan

Abstract—Efficient CPU scheduling is vital for system performance. We present a modular C++ simulator implementing FCFS, SRTF, and a three-level Feedback Queue (MLFQ) with aging and global boosts on CSV-defined workloads. It computes average waiting, turnaround, response times, CPU utilization, and throughput, and uses Python/Plotly to produce histogram, scatter, and bar chart visualizations. Experiments on uniform CPU-bound traces confirm SRTF's minimal latency, FCFS's simplicity with higher wait, and MLFQ's balanced responsiveness and fairness, highlighting the value of workload-driven scheduler tuning.

Keywords—*CPU Scheduling, FCFS, SRTF, MLFQ, Simulation, Performance Metrics*

1. Introduction

This report covers implementation and analysis of CPU scheduling algorithms. Key design choices: workload ingestion from CSV files, abstract scheduler interface, discrete and aggregate metric collection, and an external script for data visualization. Additionally, included analysis of graphical trends and expectations.

2. Implementation Details

2.1. Program Flow and File Structure

2.1.1. File Structure

- `include/`: Header files defining `PCB`, the scheduler interface, workload loader, simulator, and metrics.
- `src/`: Implementation of each component: workload.cpp, scheduler_(fcfs,srtf,mlfq).cpp, metrics.cpp, main.cpp, and simulator.cpp.
- `analyze/`: Python script (analyze.py) plus its output PDF (graphsOutput.pdf).
- `workloads/`: Input CSV files used for simulation
- `results/`: Output CSV files.

2.1.2. Data Flow

1. `main.cpp` parses the path to an input CSV.
2. `load_workload()` (in workload.cpp) reads that CSV into a `std::vector<PCB>`.
3. For each scheduler (FCFS, SRTF, MLFQ), `run_simulation()` (in `simulator.cpp`) dispatches processes, computes metrics (int `metrics.cpp`), and writes both summary and detailed CSV outputs.

2.2. Data Structures and I/O

- `PCB`: id, arr_t, burst_t, rem_t, priority, start_t, exit_t, last_enq_t.
- Scheduler interface (scheduler.h): add_to_ready(PCB*), select_next(uint64_t), name().
- Ready queues by variant: std::deque<PCB*> (FCFS), std::vector<PCB*> (SRTF), std::vector<std::deque<PCB*» (MLFQ).
- CSV formats:
  - Input: id, arrival, burst, [priority].
  - Summary output: scheduler, avg_wait, avg_turnaround, avg_response, throughput, cpu_util.
  - Detailed output: scheduler, id, arrival, burst, start, exit, waiting, turnaround, response.

2.3. Scheduling Algorithms Pseudocode

2.3.1. Core Simulation Loop

```
while completed < procs.size():
    while next < procs.size() and procs[next].arr_t <=
     ↪ time:
        sched.enqueue(procs[next], time); next++
    p = sched.select_next(time)
    if p is null:
        time = procs[next].arr_t; continue
    if p.start_t unset: p.start_t = time
    slice = sched.time_slice(p)
    time += slice; p.rem_t -= slice
    if p.rem_t == 0:
        p.exit_t = time; completed++
    else:
        sched.enqueue(p, time)
```

The simulator advances a global clock, ingests newly arrived jobs, and repeatedly asks the current scheduler which process to run next. When all processes have exited, the loop terminates and per-job metrics and aggregate statistics are written out.

2.3.2. Each Algorithm's 'select_next' Implementation

The First-Come, First-Served (FCFS) algorithm executes processes in the exact order they arrived. The Shortest Remaining Time First (SRTF) algorithm executes the process with the least remaining time out of all processes. The Multi-level Feedback Queue has three different queues, of descending priority, in which processes will execute for a given time at each one before finishing or descending until the next priority level. There are additional additions like an aging threshold and boost interval, which effectively promotes processes' priorities.

- FCFS: pop from a first in first out queue
- SRTF: choose and remove process with minimum remaining time, sorting each iteration.
- MLFQ:
  1. if global boost interval reached, move all back to highest queue
  2. promote processes that have aged beyond threshold
  3. pop from highest-priority nonempty queue

2.4. Visualization Script

A Python script (analyze.py) uses `pandas` and `plotly.express` to read the `results_detailed.csv` output, produce a waiting-time histogram, an arrival-vs-exit scatter plot, and a bar chart of average metrics, and render the graphs to a web browser.

3. Testing and Results

3.1. Process Workloads

Random workloads: CSV files from 500 to 10000 processes with uniform arrival and burst distributions. The burst times ranged from 0 to 100 ticks, and arrived between t = 0 to t = 10000.

3.2. Outputs

Generated two CSV files for each run: results.csv and detailed_results.csv. The results.csv has average turnaround, wait, and response time as well as throughput and CPU utilization. The detailed results list each process and its id, burst, arrival, and exit.

4. Aggregate Metrics Comparison

Figure 1 presents the core performance metrics for FCFS, SRTF, and MLFQ side by side. FCFS suffers from the highest average waiting and turnaround times, since each job must wait for all prior work to complete. SRTF minimizes both waiting and turnaround by always selecting the shortest remaining job, yielding the lowest values in those categories and the highest CPU utilization and throughput. MLFQ strikes a middle ground: its average waiting and turnaround are between the two extremes, but its time-slice demotions and priority boosts deliver the best average response time.

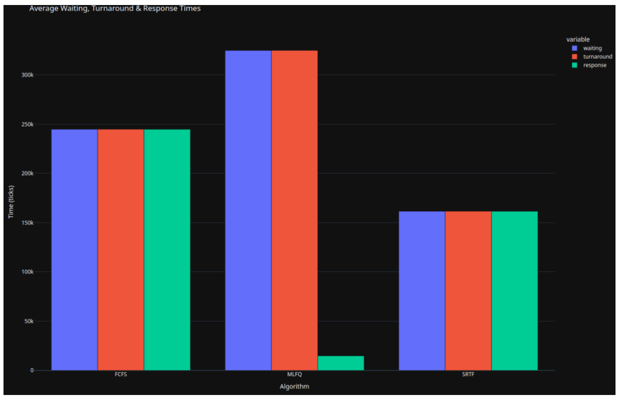

Figure 1. Average Waiting, Turnaround, and Response Times by Algorithm

Conclusions:

- FCFS: simple but high latency and lower utilization.
- SRTF: optimizes wait/turnaround at risk of starvation, highest utilization.
- MLFQ: excels in response time through preemptive quanta and aging, with balanced overall performance.

5. Histogram Analysis

The histogram in Figure 2 shows how each scheduler spreads waiting times across all jobs. FCFS produces a nearly uniform distribution, since every process waits for all earlier arrivals without discrimination. SRTF's left-skewed curve reflects its strategy of running shortest jobs first—most tasks experience very low wait, while a few longer jobs see high delays. MLFQ combines a central cluster of moderate waits, due to demotions and re-enqueues, with a right-hand tail, where lower-priority jobs that fall below the aging threshold accumulate longer waits before a global boost resets them.

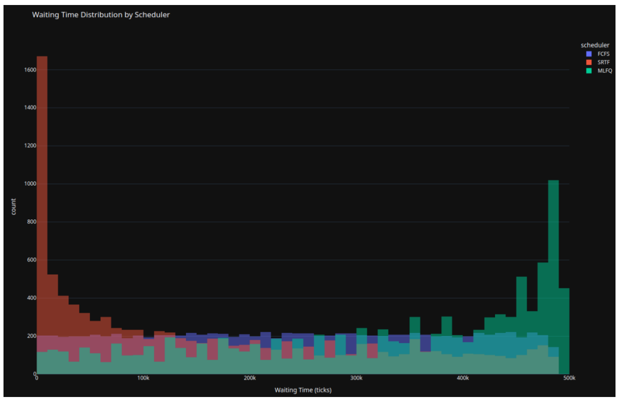

Figure 2. Waiting Time Distribution by Scheduler

Summary points:

- FCFS: flat, uniform wait times tied to arrival order.
- SRTF: steep drop-off at low waits; long-job tail from starvation risk.
- MLFQ: central mass of moderate waits plus a tail from demotions and boosts.

6. Scatter Plot Analysis

Figure 3 maps each job's arrival tick to its completion tick. FCFS yields a straight diagonal line—each process exits exactly after the cumulative runtime of itself and all predecessors. SRTF clusters points close to the diagonal—which represents near-instant completion for short jobs—and disperses outwards for longer tasks as their remaining time increases. MLFQ reveals discrete horizontal bands: each band corresponds to the time-slice quantums (3, 23, 65 ticks) and periodic boosts, illustrating how quanta and global resets shape completion times.

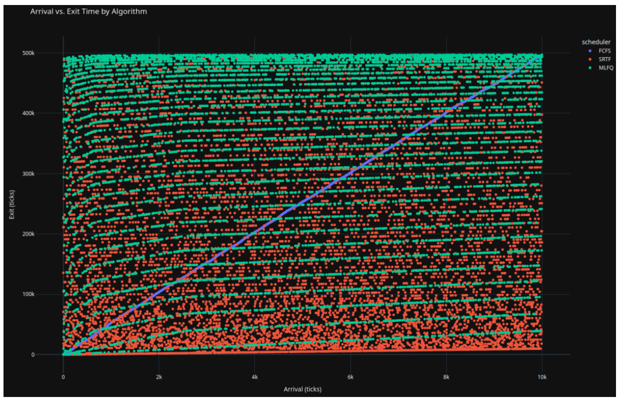

Figure 3. Arrival vs. Exit Time by Algorithm

Recap:

- FCFS: perfect linearity: exit = arrival + full queue runtime.
- SRTF: dense lower region for short tasks, dispersing for longer ones.
- MLFQ: banding patterns from multi-level quanta and boost intervals.

7. Challenges and Lessons Learned

- Interface Abstraction: I started by developing a scheduler/simulator interface which worked quite well for FCFS and SRTF; however, the MLFQ algorithm needed lots of adjustments and required C++'s dynamic_cast<>() for aging promotions and slicing by time instead of running until completion.
- Starvation and Fairness Tuning: MLFQ parameters (aging threshold, boost interval, quantum sizes) had to be tuned experimentally. This was difficult to monitor with average metrics, but using scatter-plots and histograms really helped tune it in.
- Metric Validation: Verifying computed averages, throughput, and CPU utilization matches ensures that the results are more accurate and less likely to fail during edge cases across a variety of data.
- Visualizing Data for Debugging: Creating the Data visualizations optimized troubleshooting with trickier schedulers. It was far easier to see issues/unexpected results after having graphs to reference.
- Scalability Considerations: This handles tens of thousands of jobs smoothly, there are naive data structures and algorithms that will cause issues with larger data. For example, the repeated linear searches with the SRTF implementation will bog the system down quickly.

8. Conclusions and Insights

8.1. Outcome and Real-World Considerations

The experiments demonstrate clear trade-offs among FCFS, SRTF, and MLFQ under a uniformly distributed, CPU-bound workload. FCFS is simple but suffers high latency; SRTF drives down average waiting and turnaround at the risk of starving long jobs; MLFQ balances responsiveness and fairness through its multi-queue quanta, aging promotions, and global boosts.

In real deployments, schedulers must be tuned to the target workload. These tests used highly normalized traces, which reveal baseline behaviors but omit I/O waits, priority bursts, and other real-world patterns. Production systems should profile incoming tasks—distinguishing I/O-bound vs. CPU-bound jobs, burst-length distributions, interactive vs. batch loads—and adapt scheduling parameters accordingly. Real-world systems should adapt their schedulers based on the data they are using, not simply on the ideals that were experimented within uniform data.

8.2. Key insights and potential optimizations

- **MLFQ tuning:** Dynamically adjust quanta and aging thresholds based on observed histograms or queue lengths. Additional feedback loops could be used to shorten/extend quanta depending on load.
- **SRTF enhancements:** Use a priority queue for ready processes to reduce overhead from linear to $O(\log n)$. Add aging boosts, or maximum allowable wait to prevent starvation of longer tasks.
- **Workload-aware scheduling:** For tasks that are I/O intensive, use shorter quanta or separate queues to distinguish between various tasks. For more CPU-heavy tasks favor larger quanta to give more initial processing time.
- **Machine Learning Integration** Consider experimenting with machine learning to recommend adjustments to quantum or shift protocols based on predictions of current/future tasks.
- **Robust testing:** Consider testing with archived real-world production data. Incorporate skewed test data to see how different schedulers will work under nonoptimal/unexpected loads.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Cpu scheduling*, Accessed: 28 April 2025. [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf.

[2] GeeksforGeeks, *Multilevel feedback queue scheduling (mlfq) | cpu scheduling*, Accessed: 28 April 2025. [Online]. Available: https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/.

[3] A. Mistry, *Multi-level feedback queue (mlfq) scheduling*, Accessed: 28 April 2025. [Online]. Available: https://medium.com/@akshat.mistry/multi-level-feedback-queue-mlfq-scheduling-142a6c85e2ad.

[4] OpenAI ChatGPT, *Assistance with report structure and latex document preparation*, https://chat.openai.com/, Model: o4-mini; accessed April 28, 2025.