

# Comparative Evaluation of Classical CPU Scheduling Algorithms in a Multicore Simulation Environment

Mirjana Kocaleva Vitanova<sup>1\*</sup>, Biljana Zlatanovska<sup>2</sup>, José Alejandro Ramón Rocha<sup>3</sup>,  
Aleksandar Krstev<sup>4</sup>, Elena Karamazova Gelova<sup>5</sup>, Marija Miteva<sup>6</sup>, Zoran Zlatev<sup>7\*</sup>

Faculty of Computer Science, Goce Delcev University, Stip, North Macedonia<sup>1, 2, 4, 5, 6, 7</sup>  
E. T. S. de Ingenierías Informática y de Telecomunicación, Universidad de Granada, Granada, Spain<sup>3</sup>

**Abstract**—Classical CPU scheduling algorithms remain useful baselines for understanding the trade-offs among waiting time, turnaround time, responsiveness, processor utilization, and scheduling overhead. This study compares First-Come, First-Served (FCFS), non-preemptive Shortest Job First (SJF), Shortest Remaining Time First (SRTF), and Round-Robin (RR) using a custom discrete-event simulator configured with a global ready queue and four CPU cores. Synthetic CPU-bound workloads are used to compare the policies under a common simulation framework. In the default workload reported here, SJF and SRTF produce lower average waiting and turnaround times than FCFS and RR, while RR incurs more dispatches as its time quantum decreases. The results describe the behavior of this simulator and workload; they should not be interpreted as a direct benchmark of the Linux kernel scheduler. The study also discusses the limitations associated with CPU-only workloads, the global-queue model, and the absence of task migration and kernel-level scheduling overhead. Further validation is required for the interpretation of the utilization results and for metrics not numerically reported in the present study.

**Keywords**—CPU scheduling; FCFS; SJF; SRTF; Round-Robin; multicore simulation; discrete-event simulation; performance evaluation

## I. INTRODUCTION

CPU scheduling determines which ready process receives processor time and therefore influences waiting time, turnaround time, response time, throughput, and processor utilization. FCFS, SJF, SRTF, and RR remain useful baseline policies because their selection rules expose different trade-offs between simplicity, short-job performance, preemption, and time sharing. Their textbook properties, however, are usually stated under specific assumptions, often for a single processor and known CPU-burst lengths. Results obtained from a multicore simulator depend on additional implementation choices, including the ready-queue architecture, event ordering, dispatch rules, and the treatment of simultaneous arrivals. Classical operating-systems treatments provide foundational definitions [1], while the seminal real-time scheduling model of Liu and Layland [2] addresses a task and deadline setting different from that of the finite synthetic workloads examined here.

Previous studies compare classical scheduling policies using analytical examples and simulation, reporting the familiar trade-off between short-job-oriented policies and time-sharing approaches [3]–[7].

Multicore scheduling adds a placement decision: the scheduler must select runnable tasks and assign them to available cores. Processor allocation and migration policies can affect load balance and utilization [8]. Linux uses a production scheduler with per-CPU scheduling structures and balancing mechanisms; the present simulator does not reproduce those kernel mechanisms. Consequently, the Linux discussion is included only as context and not as evidence that the simulated algorithms predict Linux kernel behavior.

This study compares FCFS, non-preemptive SJF, SRTF, and RR in a controlled multicore simulation. The contribution is a comparative account of the measured outcomes under the simulator's stated assumptions, rather than a claim that classical algorithms are universally optimal on multicore hardware. The evaluation considers average waiting time, turnaround time, dispatch count, and CPU utilization where numerical results are available. Response time, throughput, and fairness are also identified as relevant metrics, but their numerical values must be reported before conclusions about those metrics can be substantiated.

## II. RELATED WORK

Research on CPU scheduling includes surveys and comparative studies of classical and newer policies [3], [4], [6]. Comparisons of FCFS, SJF, SRTF, and RR generally show that results depend on workload composition and the performance metric being optimized [5]–[7]. In particular, a policy that minimizes average waiting time need not provide the best response time or fairness. Claims about optimality must also be tied to the assumptions of the relevant scheduling model; single-processor results do not automatically establish optimality for a particular multicore implementation.

Simulation-based comparisons require consistent workload generation, event-ordering semantics, tie-breaking rules, and metric definitions. This study uses paired synthetic CPU-bound workloads with documented deterministic seeds and reports additional trace-based validation. The reconstructed simulator, deterministic test harnesses, fixed seed list, and configuration-level outputs form the reproducibility artifacts for this study.

More complex scheduling settings include container-based real-time interference [9], realistic multiprocessor communication and NUMA costs [10], and memory–processor co-scheduling [11]. Rack-scale CPU workload shaping [12] addresses a different scheduling layer and is cited only to delimit the scope of our single-host global-queue model. A recent sim-

\*Corresponding author.

ulation-based comparison of classical scheduling policies [13] and a 2024 evaluation of CPU scheduling algorithms [14] are methodologically closer to the present study. Current Linux documentation describes EEVDF [15], which is outside the set of policies implemented in our simulator.

For Round-Robin, Khan et al. [16] investigated adaptive quantum selection, providing a contemporary contrast to the fixed 5, 10, and 20 ms quanta used here. Azzam and Siregar [17] provide a recent comparative overview of Linux scheduling, but the present study does not benchmark a Linux kernel. Yoon et al. [18] combined CFS simulation and Linux experiments to study worst-case response time; their distinct validation design underscores why simulated FCFS/SJF/SRTF/RR results cannot be interpreted as measurements of CFS or EEVDF.

Recent directly comparable simulation research includes Dirdal et al. [19], who built a multiprocessor scheduling simulation platform covering classical algorithms and CFS, and Athooyaa et al. [20], who studied classic policies with I/O interruptions. These works support simulation as a useful comparative method while highlighting that workload assumptions and omitted hardware effects constrain external validity.

On multicore systems, scheduling also involves task placement and load balancing. McCann et al. studied dynamic processor allocation for multiprogrammed shared-memory multiprocessors [8]. Modern Linux scheduling is more complex than the global ready-queue model used here. The simulator does not model per-CPU run queues, kernel scheduling classes, task migration costs, I/O blocking, cache effects, or real hardware contention. Therefore, comparisons with Linux are limited to background motivation; no claim is made that these experiments measure or reproduce the performance of the Linux scheduler.

### III. SCHEDULING POLICIES

The simulator evaluates four policies: FCFS, non-preemptive SJF, SRTF (preemptive SJF), and RR. Their selection rules are summarized below. Implementation-specific behavior—particularly event ordering, simultaneous arrivals, tie-breaking, and preemption at equal remaining times—must be documented in the released source code because it can affect the results.

#### A. First-Come, First-Served (FCFS)

FCFS selects the ready process with the earliest arrival time and runs it until completion. Its implementation is straightforward and does not intentionally postpone a ready process in favor of a later arrival. However, when job lengths vary, short jobs may wait behind long jobs, producing the convoy effect and increasing average waiting time. These properties describe the policy under the assumed queueing model; they do not imply that every FCFS workload has poor performance.

The effect of FCFS depends on the arrival process, burst-length distribution, and number of cores. With a global ready queue, an available core can take the next eligible process, but the simulator's exact dispatch and idle-core behavior must be checked against its implementation. The model is CPU-bound and does not include I/O blocking.

FCFS is retained as a simple baseline. Its results are interpreted as outcomes for the specified global-queue simulator, not as a characterization of all FCFS implementations or of Linux scheduling.

#### B. Shortest Job First (SJF) – Non-Preemptive

Non-preemptive SJF selects, whenever a core becomes available, the ready process with the shortest declared CPU-burst length and runs it to completion. This policy assumes that the burst length is known in advance; real systems generally must estimate future CPU demand. When burst lengths differ substantially, SJF can reduce average waiting time relative to FCFS under the classical single-processor assumptions.

SJF's classical optimality result applies to minimizing average waiting time for a set of jobs under specific single-processor assumptions, including known processing times and no release-date complications. It should not be generalized to all multicore scheduling objectives or implementations. In this study, SJF is evaluated empirically under the simulator's multicore global-queue model.

On multiple cores, the simulator dispatches ready processes to available cores according to its implemented queue-selection rule. A globally shortest-job-first assignment is not equivalent to a general proof of multicore optimality. The reported measurements are therefore described as simulation results rather than theoretical guarantees.

The SJF priority queue selects the ready job with the shortest CPU burst; the simulation uses deterministic ordering for ties.

SJF can yield a low average waiting time when CPU-burst lengths are known and vary substantially. Its performance depends on the workload, and long jobs may experience extended waits when shorter jobs continue to arrive. The current draft does not provide a separate starvation or maximum-waiting-time experiment; accordingly, starvation is discussed as a known risk rather than as an experimentally demonstrated outcome.

#### C. Shortest Remaining Time First (Preemptive SJF)

SRTF selects the ready or running process with the shortest remaining CPU time. When a new process arrives, it may preempt a running process if the new process has a shorter remaining time, subject to the simulator's event and tie-breaking rules. This policy requires the simulator to reconsider scheduling decisions on arrivals as well as completions.

SRTF can delay long jobs when shorter jobs continue to arrive. The present experiments use finite synthetic workloads, so they cannot establish unbounded starvation; maximum waiting time and a workload with sustained short-job arrivals would be needed to assess this risk empirically.

The idealized rule of placing the  $N$  shortest remaining jobs on  $N$  cores is a useful conceptual reference, but the simulator's actual behavior is determined by its implementation. The measurements should not be interpreted as a proof that SRTF is optimal for general multicore scheduling.

The policy reconsiders assignments when relevant events occur. Its behavior in this study is an empirical result of the

simulator's queue and event logic; no universal claim is made that SRTF always outperforms SJF for every multicore workload or every objective.

#### D. Round-Robin (RR)

RR gives each ready process up to a fixed time quantum. If a process has not completed by the end of its quantum, it is returned to the ready queue. The quantum controls the trade-off between frequent opportunities to run and the number of scheduling events and context switches. As the quantum increases, RR may resemble FCFS more closely; decreasing the quantum generally increases switching frequency, although the exact effect on waiting time depends on the workload and implementation.

RR is designed to share processor time among runnable processes and can improve first-response behavior relative to non-preemptive policies. However, fairness should be measured explicitly rather than inferred from the algorithm name or from average waiting time alone. A fairness claim requires a defined metric and results for that metric.

The experiments considered quanta of 5, 10, and 20 ms. The default results below use 10 ms unless otherwise stated. A valid comparison must use the same generated workload for each quantum and scheduling policy so that differences are attributable to the policy rather than to changes in the input.

The simulator uses a global ready queue. This differs from the variety of queue structures and scheduling classes used by production operating systems. No claim is made that the RR implementation is identical to a Linux scheduler policy.

For each dispatch, RR executes the selected job for the smaller of the configured quantum and its remaining CPU burst. Jobs with remaining work are requeued after quantum expiration; simultaneous events follow the order specified in Section IV.

In the default workload reported here, RR has a higher average waiting time than FCFS and both SJF variants. This observed ordering differs from a general statement that RR usually lies between FCFS and SJF, and it is retained as a workload-specific result.

A shorter quantum increases the number of scheduling intervals and can increase dispatch overhead. The simulation records dispatch allocations rather than hardware context-switch instructions.

RR offers time-sliced service and its responsiveness is evaluated using measured first-response delay. The present study reports waiting, turnaround, response, productive utilization, dispatch frequency, and tail-delay metrics. It does not claim an empirical CPU-share fairness comparison.

## IV. EXPERIMENTAL SETUP

A custom discrete-event simulator written in C++ was used to compare FCFS, non-preemptive SJF, SRTF, and RR. The simulator represents M CPU cores (four in the default configuration) and a single global ready queue. Up to M processes can execute concurrently. Because the simulator is a user-space model rather than a modification of, or trace-based measurement from, the Linux kernel scheduler, the results are those of

a simulation on a Linux development environment, not direct measurements of Linux scheduling performance.

#### A. Simulation Environment

The reconstructed discrete-event simulator is implemented in C++17 and models a global ready queue and M virtual CPU cores. Results describe simulated core time and are not measurements of the host operating system or Linux kernel scheduling.

The discrete-event simulation uses one global ready queue and M virtual cores. Events at the same simulated timestamp are resolved in this order: process completions, RR quantum expirations, new arrivals, SRTF preemptions, and dispatches. Available cores are filled in ascending core-index order. Selection ties are broken by the timestamp of the process's most recent ready-queue entry and then by process ID. A running SRTF process is preempted only if a ready process has strictly less remaining CPU time; among running processes, the one with the greatest remaining time is selected for replacement. A preempted job returns to the global queue and may subsequently execute on any core. No migration penalty is modeled.

To make the simulator's event semantics auditable, Listing 1 shows the event-priority rule and the corresponding completion and time-slice processing from the reconstructed implementation. At each timestamp, core indices are visited in ascending order; arrivals are admitted in job-vector order (ascending job ID). SRTF preemption and dispatch then follow the priority order described above.

```
// Same-timestamp priority:
// completion > RR expiry > arrival >
// SRTF preemption > dispatch
for (int c=0; c<ncores; ++c) {
    auto& core=cores[c];
    if (core.pid<0 || core.switching) continue;
    auto& p=jobs[core.pid];
    if (p.remaining==0) {
        p.finish=t; core.pid=-1; ++finished;
    } else if (policy=="RR" &&
               core.quantum_left==0) {
        p.ready_since=t;
        ready.push_back(p.id); core.pid=-1;
    }
}
```

Listing 1. Event ordering and core release (C++17 excerpt).

The excerpt omits logging calls only; the full implementation subsequently admits arrivals, applies SRTF preemption, dispatches available cores, and advances to the next event. Thus Listing 1 is an illustration of the implemented order rather than a standalone executable scheduler.

Listing 2 presents the SRTF preemption decision used by the reconstructed multicore simulator. A preemption is considered only when all cores are occupied; the shortest ready job must have strictly less remaining work than the longest currently executing job. Ties among running jobs select the highest core index. The subsequent dispatch pass assigns ready jobs to idle cores in ascending core-index order. This is an implementation-specific rule, not a general multicore optimality guarantee.

```
if (policy=="SRTF") {
    while (!ready.empty()) {
        bool idle=false;
        for (auto& core:cores) if (core.pid<0) idle=true;
        if (idle) break;
        int k=pick(jobs,ready,policy);
        int candidate=ready[k], worst=-1;
        for (int c=0;c<ncores;++c)
            if (cores[c].pid>0 && !cores[c].switching)
                if (worst<0 || jobs[cores[c].pid].remaining >
                    jobs[cores[worst].pid].remaining ||
                    (jobs[cores[c].pid].remaining ==
                     jobs[cores[worst].pid].remaining && c>worst))
                        worst=c;
        if (worst<0 || jobs[candidate].remaining >=
            jobs[cores[worst].pid].remaining) break;
        int id=cores[worst].pid;
        jobs[id].ready_since=t;
        ready.push_back(id);
        cores[worst].pid=-1;
        log("PREEMPT",worst,id);
    }
}
```

Listing 2. SRTF multicore preemption decision (C++17 excerpt).

Each experimental workload consists of 100 independent CPU-bound jobs. Batch arrivals occur at  $t = 0$ ; dynamic arrivals are sampled independently from discrete uniform  $[0,1000]$  ms. CPU bursts are sampled independently from discrete uniform  $[1,100]$  ms, giving an expected total demand of 5,050 core-ms. On four cores, the expected CPU demand divided by the 0–1000 ms arrival-window capacity is  $5050/4000 = 1.2625$ ; this is a workload-intensity indicator, not a measured steady-state utilization.

RR quantum values of 5, 10, and 20 ms were evaluated; the default is 10 ms. The baseline sets per-dispatch overhead  $H = 0$  ms, and sensitivity runs set  $H = 0.1$  or  $0.5$  ms. Every allocation of a process to a core incurs the same modeled  $H$ , including the initial dispatch and a resumption after preemption or quantum expiration. During  $H$ , that core is unavailable for productive execution; the overhead advances simulated time. The model counts dispatch operations rather than measured operating-system context switches.

### B. Performance Metrics

For each CPU-only job  $i$ , total non-execution delay is  $D_i = C_i - A_i - B_i$ , where  $C_i$  is completion time,  $A_i$  is arrival time, and  $B_i$  is required CPU service. The reported `mean_wait_ms` is the mean of  $D_i$ ; it includes modeled dispatch overhead. Separately, `mean_ready_wait_ms` accumulates intervals in the ready queue between admission, preemption, and dispatch, and excludes time spent in dispatch overhead. The two measures coincide when  $H = 0$  but need not coincide for  $H > 0$ .

$$D_i = C_i - A_i - B_i \quad (1)$$

Average turnaround time is the mean completion time minus arrival time. This includes ready-queue waiting and CPU execution, but excludes time before arrival.

Throughput is the number of completed jobs divided by the makespan measured from  $t = 0$  to the final completion; values in jobs/ms can be multiplied by 1000 for jobs/s.

$$\text{Throughput} = N / T \quad (2)$$

Productive utilization is  $100 \times P / (M \times T)$ , where  $P$  is total productive core time,  $M$  is the number of virtual cores, and  $T$  is makespan measured from  $t = 0$  to the last completion. Overhead core time  $O$  and idle core time  $I$  are accounted for separately, and the implementation checks  $P + O + I = M \times T$ . Productive utilization does not include dispatch overhead.

$$U_p (\%) = 100 \times P / (M \times T) \quad (3)$$

Response time is the elapsed time from job arrival to first productive execution (including any modeled first-dispatch overhead). The reconstructed simulator records this metric for every completed job.

The recorded dispatch count increases whenever a job is allocated to a simulated core, including initial allocations and allocations following completion, SRTF preemption, or RR quantum expiration. It is not a count of observed Linux context switches. The modeled overhead is charged per dispatch as defined above.

Waiting-time metrics were audited against the simulator implementation. `mean_wait_ms` is the per-job completion-to-arrival interval minus CPU burst, averaged over jobs; `max_total_delay_ms` and `p95_total_delay_ms` now apply this same definition to the maximum and 95th percentile, respectively. The separately retained `max_wait_ms` and `mean_ready_wait_ms` describe accumulated ready-queue waiting only. Under nonzero dispatch overhead, these definitions differ and must not be interchanged. The archived `jain_wait` index concerns ready-queue delays, NOT normalized CPU service or CPU-share fairness; no CPU-share fairness claim is made. These finite experiments cannot prove starvation freedom.

Every scenario was run with ten deterministic, paired seeds: 20261016–20261025. For each seed, `std::mt19937` generates integer arrival times uniformly on  $[0,1000]$  ms for dynamic workloads (all zero for batch workloads) and integer burst times uniformly on  $[1,100]$  ms. Identical generated jobs are supplied to every policy and parameter configuration. Experiments cover  $M = 1, 2, 4, 8$  virtual cores; dispatch overhead  $H = 0, 0.1, 0.5$  ms; and RR quanta of 5, 10, 20 ms.

Listing 3 reproduces the workload-generation logic. The same generated job vector is passed separately to each scheduling policy, preventing policy-dependent workload differences. All simulation durations are represented as integer ticks of 0.1 ms.

```
mt19937 gen(seed);
uniform_int_distribution<int> a(0,1000);
uniform_int_distribution<int> b(1,100);
for (int i=0; i<n; ++i) {
    long long arrival = batch ? 0 : 10LL*a(gen);
    long long burst = 10LL*b(gen);
    jobs.push_back({i,arrival,burst,
                    burst,arrival});
}
```

Listing 3. Deterministic workload generation (C++17 excerpt).

The snippets are reproduced from the reconstructed C++17 research simulator and are deliberately limited to the reproduc-

ibility-critical portions. They do not imply that Linux kernel scheduling was modified or measured.

The reconstructed C++17 simulator, deterministic test harnesses, fixed seed list, and full configuration-level CSV are maintained as research artifacts for reproducibility. The original implementation was unavailable for independent verification; the reconstructed results must not be presented as replication of its numerical output.

The simulator models CPU-bound jobs only and does not include I/O blocking, memory contention, cache effects, kernel scheduling classes, task migration cost, or real hardware interference. These exclusions restrict the scope of the conclusions and are stated explicitly in the Discussion.

### C. Validation and Reproducibility

The simulator was validated against twelve independently specified small scenarios: eight with zero overhead and four with positive (0.5 ms) per-dispatch overhead. Cases cover FCFS, SJF, SRTF, RR, simultaneous arrivals, equal remaining-time ties, exact quantum boundaries, idle intervals, concurrent multicore execution, and preemption. Expected per-job completion times and dispatch counts were compared with event traces. In addition, productive, overhead, and idle core-time accounting was checked across 1,440 configuration-level records. These tests support internal correctness for the tested cases but do not constitute a formal proof of correctness.

## V. RESULTS

### A. Baseline Performance

Results in this section were generated with an independently reconstructed C++17 discrete-event simulator, not the original source-code implementation. The baseline uses 100 CPU-bound jobs, four simulated cores, uniform arrivals between 0 and 1000 ms, uniform CPU bursts between 1 and 100 ms, and zero dispatch overhead. Round-Robin uses a 10 ms quantum. Ten paired workload seeds (20261016–20261025) were evaluated (Table I). Reported values are means across seed-level runs; they should not be interpreted as Linux kernel measurements (see Fig. 1 to 5).

TABLE I. BASELINE RESULTS OF THE RECONSTRUCTED SIMULATOR  
(TEN PAIRED WORKLOAD SEEDS)

| Policy | Waiting (ms) | Turnaround (ms) | Response (ms) | Useful CPU (%) | Dispatches |
|--------|--------------|-----------------|---------------|----------------|------------|
| FCFS   | 125.65       | 175.78          | 125.65        | 93.01          | 100.0      |
| SJF    | 77.15        | 127.29          | 77.15         | 92.97          | 100.0      |
| SRTF   | 73.80        | 123.94          | 62.89         | 93.09          | 141.1      |
| RR     | 177.56       | 227.69          | 29.55         | 95.17          | 546.8      |

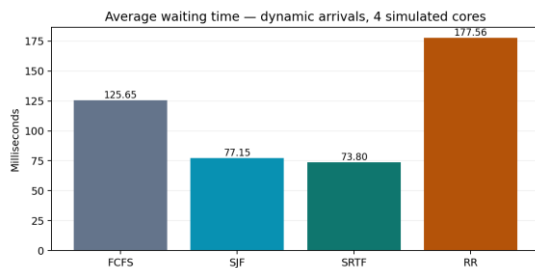


Fig. 1. Average waiting time in the reconstructed multicore simulation.

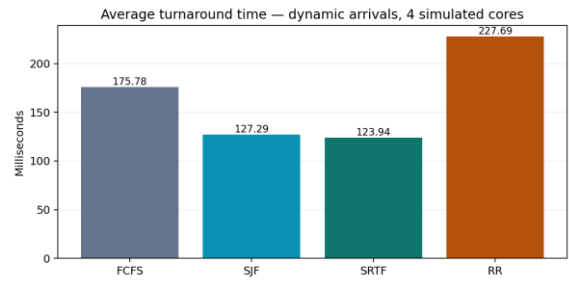


Fig. 2. Average turnaround time in the reconstructed multicore simulation.

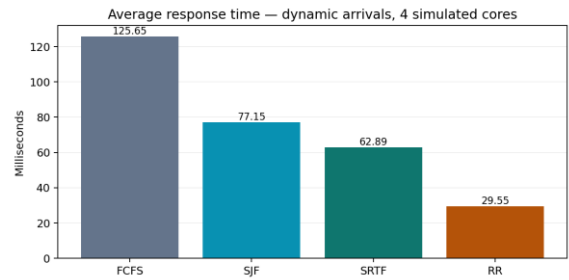


Fig. 3. Average first response time in the reconstructed multicore simulation.

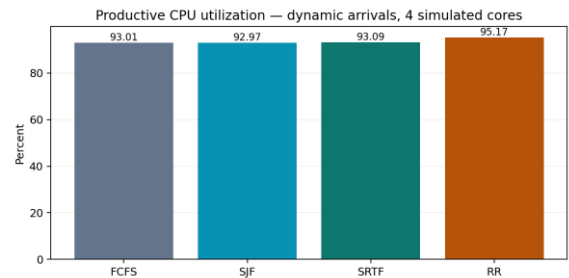


Fig. 4. Productive CPU utilization in the reconstructed multicore simulation.

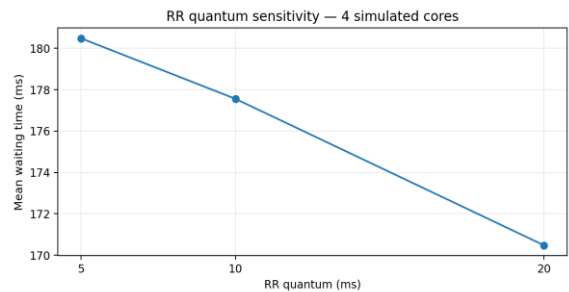


Fig. 5. Round-Robin time-quantum sensitivity in the reconstructed multicore simulation.

SRTF yields the lowest mean waiting (73.80 ms) and turnaround (123.94 ms), while RR provides the shortest mean first response (29.55 ms). The difference between turnaround and waiting is consistent with the mean CPU-burst demand. These comparisons apply to the specified simulation model only.

### B. Dispatch Accounting and Utilization

Dispatch count denotes the number of CPU allocations, not a count of individual save/restore operations. In the baseline, FCFS and SJF each dispatch 100 jobs, SRTF averages 141.1 dispatches, and RR averages 546.8 dispatches. Productive uti-

lization is defined as total CPU-burst execution divided by the available core-time capacity (four times the interval from simulation start to final completion). The reported utilization excludes dispatch overhead, which is zero in this baseline. RR utilization is higher in this configuration, but this does not imply shorter waiting or turnaround times.

### C. Statistical Interpretation

For ten paired workload seeds, sample standard deviations of seed-level mean waiting times were FCFS 34.73, SJF 19.65, SRTF 20.04, and RR 48.86 ms. The paired SJF – SRTF mean waiting-time difference was 3.348 ms (95% Student-t confidence interval 2.240–4.456 ms; two-sided paired t-test  $p = 7.57941e-05$ ;  $n = 10$  seed-level pairs). This inference concerns the specified simulator and synthetic workload, not universal optimality (Table II).

### D. Scope of the Results

The results do not establish starvation freedom or actual Linux scheduler performance. Separate batch/dynamic workload, nonzero-overhead, and core-count sensitivity results are presented below. The archived waiting-time dispersion index is not used to claim CPU-share fairness.

### E. Batch and Dynamic Workload Comparison

Batch arrivals create greater queueing delay because all 100 jobs become ready at time zero. In this simulator, SJF and SRTF coincide for batch arrivals because no later job arrival triggers a preemption; SRTF has a modest advantage for dynamic arrivals. The two regimes should not be pooled as one workload.

TABLE II. MEAN WAITING TIME BY ARRIVAL REGIME (MS; 10 PAIRED SEEDS, 4 CORES)

| Policy | Batch  | Dynamic |
|--------|--------|---------|
| FCFS   | 602.84 | 125.65  |
| SJF    | 404.53 | 77.15   |
| SRTF   | 404.53 | 73.80   |
| RR     | 799.30 | 177.56  |

### F. Dispatch-Overhead Sensitivity

Dispatch overhead is charged for each allocation of a process to a core, including the first allocation, and the assigned core is unavailable for productive execution during this interval. Consequently, waiting time includes dispatch delays while useful utilization excludes overhead. The larger RR waiting-time increase is associated with its more frequent dispatches. These modeled dispatch events should not be described as observed Linux kernel context switches (Table III).

TABLE III. MEAN WAITING TIME BY MODELED DISPATCH OVERHEAD (MS; DYNAMIC ARRIVALS, 4 CORES)

| Policy | 0 ms   | 0.1 ms |
|--------|--------|--------|
| FCFS   | 125.65 | 126.70 |
| SJF    | 77.15  | 78.02  |
| SRTF   | 73.80  | 74.83  |
| RR     | 177.56 | 185.43 |

TABLE IV. MEAN WAITING TIME BY SIMULATED CORE COUNT (MS; DYNAMIC ARRIVALS, TEN PAIRED SEEDS)

| Policy | 1 core  | 2 cores | 4 cores | 8 cores |
|--------|---------|---------|---------|---------|
| FCFS   | 1921.53 | 709.61  | 125.65  | 3.60    |
| SJF    | 1224.64 | 434.36  | 77.15   | 2.88    |
| SRTF   | 1217.74 | 429.01  | 73.80   | 2.49    |
| RR     | 2635.67 | 988.91  | 177.56  | 5.04    |

TABLE V. BASELINE TAIL-DELAY STATISTICS (MS; FOUR SIMULATED CORES, TEN PAIRED SEEDS)

| Policy | Mean delay (ms) | Mean of maximum delay (ms) | Mean p95 delay (ms) |
|--------|-----------------|----------------------------|---------------------|
| FCFS   | 125.65          | 291.60                     | 266.40              |
| SJF    | 77.15           | 793.10                     | 399.10              |
| SRTF   | 73.80           | 795.70                     | 413.20              |
| RR     | 177.56          | 478.10                     | 418.80              |

Increasing simulated core count reduces queueing delay for this fixed workload, but the experiment does not model memory contention, processor affinity, per-core queues, or migration cost. Thus, it does not establish the scalability of the actual Linux scheduler (see Tables IV and V).

## VI. DISCUSSION

For time-sharing workloads, RR offers recurring opportunities for runnable processes to execute. In the reported four-core baseline, RR has the lowest measured mean first-response delay, although it incurs greater mean total delay and more dispatches. This does not establish real-time guarantees or CPU-share fairness.

### A. Fairness, Tail Latency, and Starvation

For a finite workload, a scheduling policy can reduce the mean delay while increasing the delay experienced by the most disadvantaged jobs. The analysis therefore complements the average with the maximum and the 95th percentile of per-job total non-execution delay, defined as completion time minus arrival time minus CPU burst. Each statistic is computed within a run and then averaged across ten paired workload seeds. These tail statistics measure latency dispersion, not an axiomatic fairness guarantee.

The principal limitations are synthetic CPU-only jobs, exact knowledge of CPU bursts for SJF/SRTF, a global ready queue, and the exclusion of I/O, processor affinity, migration cost, cache contention, and direct Linux kernel measurements. CPU-share fairness and burst-prediction sensitivity were not experimentally evaluated. Twelve deterministic trace scenarios and configuration-wide time-accounting checks support, but cannot establish universally, the correctness of the reconstructed implementation.

The exported `jain_wait` statistic is Jain's index applied to nonnegative ready-queue waiting durations. It is not Jain's CPU-share fairness index and should not be interpreted as equal allocation of processor service: every finite job eventually receives its entire requested burst in these runs, and a Jain index over completed CPU service would mainly reflect heterogeneous job sizes. A meaningful CPU-share fairness study

requires time-windowed service allocations, runnable-state eligibility, and a defined entitlement or weight per job; these traces and entitlements have not been recorded. Accordingly, the study reports waiting-time tail measures and explicitly leave CPU-share fairness unmeasured.

In the reconstructed four-core baseline, SRTF produced the lowest mean total delay and turnaround, while RR produced the lowest mean first-response delay. The observed tail-delay differences caution against treating average performance as a universal fairness measure. The experiments are simulation results, not Linux kernel benchmarks.

For batch workloads with known CPU-burst lengths, SJF-like policies may reduce average waiting time, as reflected in this simulation. Their practical value depends on the availability and accuracy of burst-length estimates and on whether long jobs can be delayed unacceptably.

For time-sharing workloads, RR offers recurring execution opportunities. In the measured baseline, RR has the shortest mean first-response delay, but higher mean waiting and turnaround times.

A core-count sensitivity experiment was executed for  $M = 1, 2, 4$ , and  $8$  simulated cores with matched seeds. These results concern idealized virtual cores; they are not evidence of operating-system scalability or task-migration performance.

## VII. CONCLUSION

This study compares FCFS, SJF, SRTF, and RR in a controlled four-core discrete-event simulation. In an independently reconstructed implementation, SRTF achieved the lowest baseline average waiting and turnaround times, whereas RR achieved the fastest first response. The findings illustrate trade-offs under explicit workload and modeling assumptions; they do not constitute Linux kernel benchmarks.

Future work should extend the simulator to I/O-bound and mixed workloads, per-core ready queues, variable core counts, task migration costs, and alternative scheduling policies. A direct comparison with Linux would require a separately designed kernel-level or trace-based experiment. These extensions would strengthen the reproducibility and practical relevance of the comparison.

The central conclusion is that scheduling results depend on the policy, workload, simulator implementation, and selected performance metric. Explicit assumptions, reproducible workloads, and consistent metric accounting are essential for drawing reliable conclusions from multicore scheduling experiments.

The study does not claim that CFS is a descendant of RR, nor that these results validate the Linux scheduler; production Linux scheduling behavior is mentioned only as context.

## ACKNOWLEDGMENT

The Authors express their gratitude to Goce Delcev University, Stip, North Macedonia, for the support through an individual scientific fund.

## REFERENCES

- [1] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Pearson Education, 2009.
- [2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
- [3] N. Harki, A. J. Ahmed, and L. M. Haji, "CPU Scheduling Techniques: A Review on Novel Approaches Strategy and Performance Assessment," *Journal of Applied Science and Technology Trends*, vol. 1, no. 1, pp. 48–55, 2020, doi: 10.38094/jastt1215.
- [4] H. K. Omar, K. H. Jihad, and S. F. Hussein, "Comparative analysis of the essential CPU scheduling algorithms," *Bulletin of Electrical Engineering and Informatics*, vol. 10, no. 5, pp. 2742–2750, 2021, doi: 10.11591/eei.v10i5.2812.
- [5] W. Pan, "Comparison and Analysis of Scheduling Algorithms: Exploring Performance, Time, Fairness, and Applicable Scenarios," *Highlights in Science, Engineering and Technology*, vol. 120, pp. 1–7, 2024, doi: 10.54097/4asdt07.
- [6] S. Pemasinghe and S. Rajapaksha, "Comparison of CPU Scheduling Algorithms: FCFS, SJF, SRTF, Round Robin, Priority Based, and Multilevel Queuing," in *2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC)*, pp. 318–323, 2022, doi: 10.1109/R10-HTC54060.2022.9929533.
- [7] T. D. Putra, "Analysis of Preemptive Shortest Job First (SJF) Algorithm in CPU Scheduling," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 9, no. 4, pp. 41–45, 2020, doi: 10.17148/IJARCC.2020.9408.
- [8] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 146–178, 1993, doi: 10.1145/151244.151246.
- [9] N. Samimi, L. Abeni, D. Casini, M. Marinoni, T. Basten, M. Nasri, M. Geilen, and A. Biondi, "Enabling Containerisation of Distributed Applications with Real-Time Constraints," in *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*, *LIPICs*, vol. 335, art. 3, pp. 3:1–3:29, 2025, doi: 10.4230/LIPICs.ECRTS.2025.3.
- [10] P. A. Papp, G. Anegg, A. Karanasiou, and A.-J. N. Yzelman, "Efficient Multi-Processor Scheduling in Increasingly Realistic Models," in *Proc. 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 463–474, 2024, doi: 10.1145/3626183.3659972.
- [11] I. Senoussaoui, G. Lipari, H.-E. Zahaf, and M. K. Benhaoua, "Memory-processor co-scheduling of AECR-DAG real-time tasks on partitioned multicore platforms with scratchpads," *Journal of Systems Architecture*, vol. 150, art. 103117, 2024, doi: 10.1016/j.sysarc.2024.103117.
- [12] X. Liao, H. Tian, X. Wan, C. Zeng, H. Wang, J. Zhang, M. Ma, G. Liu, and K. Chen, "Towards Optimal Rack-scale  $\mu$ s-level CPU Scheduling through In-Network Workload Shaping," in *Proc. 2025 USENIX Annual Technical Conference (USENIX ATC '25)*, pp. 179–198, 2025.
- [13] M. N. Al Refai, "Scheduling Algorithms Comparisons and Analysis using Simulation Results," *International Journal of Applied Sciences & Development*, vol. 3, pp. 151–160, 2024, doi: 10.37394/232029.2024.3.14.
- [14] M. González-Rodríguez, L. Otero-Cerdeira, E. González-Rufino, and F. J. Rodríguez-Martínez, "Study and evaluation of CPU scheduling algorithms," *Heliyon*, vol. 10, no. 9, art. e29959, 2024, doi: 10.1016/j.heliyon.2024.e29959.
- [15] Linux Kernel Documentation, "EEVDF Scheduler," Linux kernel documentation, accessed Oct. 2026.
- [16] Z. I. Khan, M. Khan, and S. N. M. Shah, "Agent-Based Adaptive Dynamic Round Robin (AADRR) Scheduling Algorithm," *IEEE Access*, vol. 13, pp. 18308–18324, 2025, doi: 10.1109/ACCESS.2025.3534031.
- [17] A. I. Azzam and H. Siregar, "Analisis Perbandingan Algoritma Penjadwalan CPU pada Sistem Operasi Linux," *REMIK*, vol. 9, no. 3, pp. 818–825, 2025, doi: 10.33395/remik.v9i3.14924.

- [18] K. Yoon, E. Jeong, W. Kang, J. Choe, and S. Ha, "Worst case response time analysis for completely fair scheduling in Linux systems," *Real-Time Systems*, vol. 61, no. 1, pp. 118–158, 2025, doi: 10.1007/s11241-025-09435-x.
- [19] D. O. Dirdal, D. Vo, Y. Feng, and R. Davidrajuh, "Developing a Platform Using Petri Nets and GPenSIM for Simulation of Multiprocessor Scheduling Algorithms," *Applied Sciences*, vol. 14, no. 13, art. 5690, 2024, doi: 10.3390/app14135690.
- [20] M. Athooyaa, M. P. F. Firmansyah, Djuniadi, and A. Ardhiansyah, "Comparative Simulation: Four CPU Scheduling Algorithms with I/O Interruption Modeling Using OS-SIM," *ELECTRON Jurnal Ilmiah Teknik Elektro*, vol. 7, no. 1, pp. 11–21, 2026, doi: 10.33019/electron.v7i1.448.