CS 3502 Project 2: CPU Scheduling Simulator

Performance Analysis and Recommendations for OwlTech Industries

Bryan Julius

October 24, 2025

Abstract

This report presents a performance analysis of six CPU scheduling algorithms to address scheduling inefficiencies at OwlTech Industries. A C# simulator was extended to implement two new algorithms, Shortest Remaining Time First (SRTF) and Highest Response Ratio Next (HRRN), and to measure four key performance metrics: Average Waiting Time (AWT), Average Turnaround Time (ATT), CPU Utilization, and Throughput. Experiments were conducted using three distinct workloads (CPU-Bound, I/O-Bound, and Mixed). The findings demonstrate that algorithms like SJF, SRTF, and HRRN significantly outperform FCFS, Priority, and Round Robin in mixed-load environments by minimizing the convoy effect. Based on this data, this report recommends the adoption of the Highest Response Ratio Next (HRRN) algorithm as the new standard for OwlTech's production systems, as it provides top-tier performance while guaranteeing fairness and preventing process starvation.

Contents

1 Introduction

The Performance Optimization Division at OwlTech Industries has identified inefficiencies in its current production systems. The goal of this project is to provide a data based recommendation for the optimal CPU scheduling algorithm to serve our diverse workloads.

To achieve this, the provided C# CPU Scheduling Simulator [1] was extended to support a wider range of algorithms and more comprehensive performance metrics. This report details the technical implementation of these extensions and, more importantly, presents a detailed performance analysis of six algorithms:

- First Come, First Serve (FCFS)
- Shortest Job First (SJF)
- Priority (Non-Preemptive)
- Round Robin (RR)
- **Shortest Remaining Time First (SRTF)** [New Implementation]
- **Highest Response Ratio Next (HRRN)** [New Implementation]

These algorithms were tested against three simulated workloads (CPU-Bound, I/O-Bound, and Mixed) to measure their impact on efficiency, fairness, and responsiveness.

2 Technical Implementation

The primary technical task was to extend the existing C# simulator [1] to add two new algorithms and their associated metrics, fully integrating them into the UI.

2.1 New Algorithm Implementation

Two new scheduling algorithms were implemented, SRTF and HRRN. The code for each is available in Appendix A.

2.1.1 Shortest Remaining Time First (SRTF)

SRTF is the preemptive version of Shortest Job First (SJF). It is designed to be optimal for average waiting time. The scheduler always selects the process that has the shortest *remaining* burst time. If a new process arrives in the ready queue with a total burst time shorter than the remaining time of the currently executing process, the scheduler preempts the current process and starts the new, shorter job.

2.1.2 Highest Response Ratio Next (HRRN)

HRRN is a non-preemptive algorithm designed to solve the starvation problem inherent in SJF. It balances process burst time with how long a process has been waiting (its ”aging”). It selects the process with the highest ”Response Ratio,” which is calculated at the time of selection:

$$\text{Response Ratio} = \frac{(\text{Waiting Time} + \text{Burst Time})}{\text{Burst Time}}$$

By placing waiting time in the numerator, a process that is ignored will eventually see its ratio grow large enough to be selected, ensuring it will never starve.

2.2 Performance Metrics Implementation

Per the project requirements, the simulator was modified to calculate and display four key metrics for all algorithms. The two new metrics, CPU Utilization and Throughput, were added to the `DisplaySchedulingResults` method.

- **CPU Utilization:** Calculated as the total time the CPU was busy (sum of all burst times) divided by the total simulation time (finish time of the last process).
- **Throughput:** Calculated as the total number of processes completed divided by the total simulation time.

2.3 Simulator Modifications

Several modifications were made to the `CpuSchedulerForm.cs` file to integrate the new functionality:

1. **New UI Buttons:** Two new buttons, `btnSRTF` and `btnHRRN`, were added to the `algorithmButtonPanel` in the designer (see Figure 2). These buttons were styled to match the existing theme.
2. **Event Handling:** New click-event handlers (`btnSRTF_Click` and `btnHRRN_Click`) were created to call their respective algorithm functions (`RunSRTFAlgorithm` and `RunHRRNAlgorithm`).
3. **Results Display:** The `DisplaySchedulingResults` method was updated to add two new `ListViewItem` rows to display the calculated CPU Utilization and Throughput, along with the existing AWT and ATT.
4. **Data Export:** An "Export as CSV" button was added to the results panel. This allows the user to save the complete results of a simulation, including all summary metrics, for external analysis.

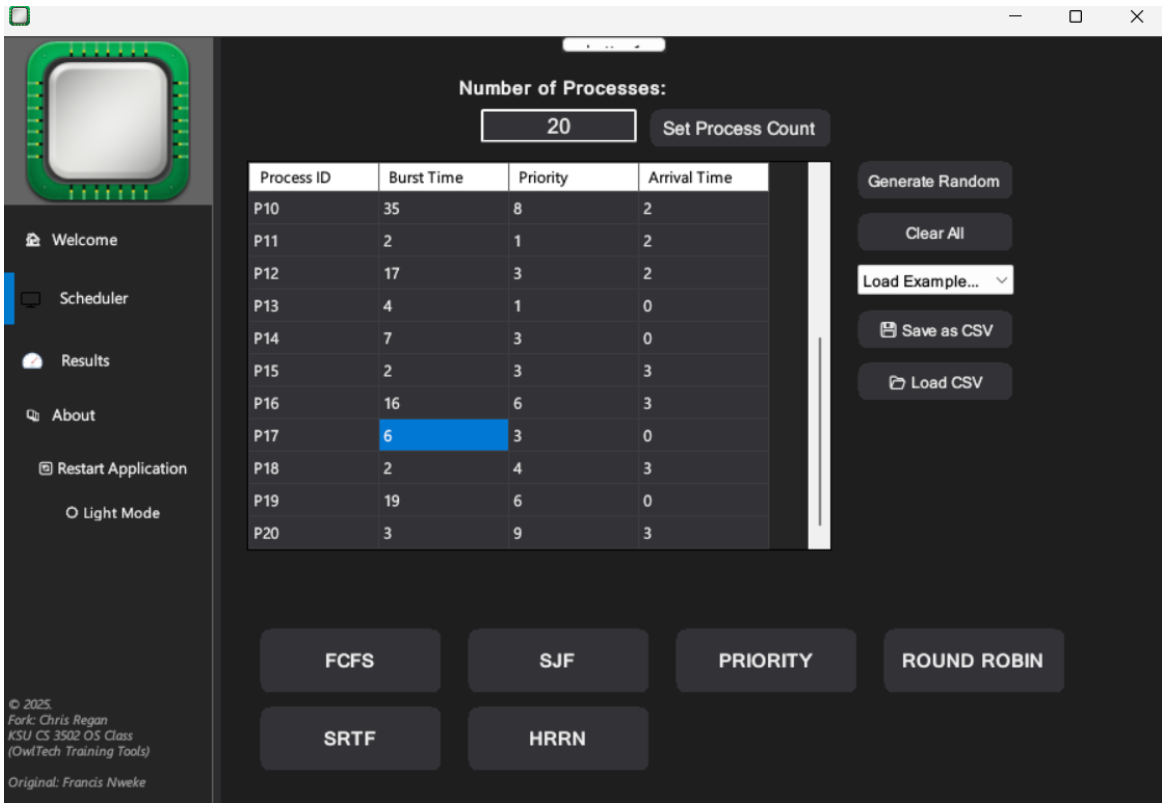

Figure 1: The modified simulator UI showing the new algorithm buttons (SRTF, HRRN) and the Export button in the results panel.

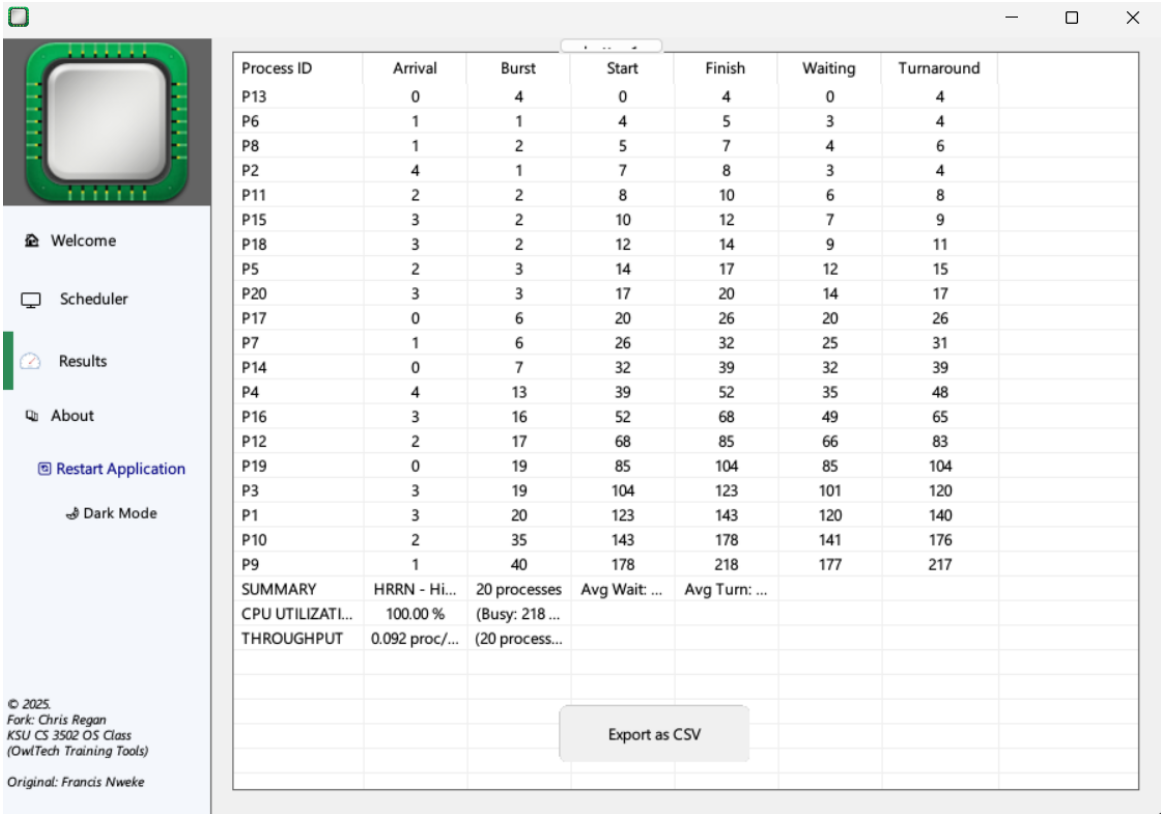

Figure 2: The modified simulator UI showing the new Export button in the results panel.

3 Performance Analysis

A series of experiments was conducted to compare the performance of all six algorithms under different load conditions.

3.1 Experimental Workloads

Three distinct workloads were designed to test the algorithms' behavior under stress:

CPU-Bound Workload: A simulation of 5 processes with very high burst times (e.g., 75-120) and staggered arrivals. This simulates a server running heavy computational tasks.

I/O-Bound Workload: A simulation of 20 processes with very low burst times (e.g., 1-7) arriving in quick succession. This simulates a web server handling many small, fast user requests.

Mixed Workload: A simulation of 20 processes with a mix of high-burst (e.g., 35, 40) and low-burst (e.g., 1, 2, 3) processes. This is the most realistic model of a general-purpose production server.

3.2 Comparison Tables

The following tables summarize the performance metrics gathered from running each algorithm against the three workloads.

Table 1: Performance on CPU-Bound Workload (5 Processes)

Algorithm	Avg. Wait (ms)	Avg. Turnaround (ms)	CPU Util. (%)	Throughput
FCFS	162.0	255.0	100.0%	0.011
SJF	155.0	248.0	100.0%	0.011
SRTF	**155.0**	**248.0**	**100.0%**	**0.011**
HRRN	**155.0**	**248.0**	**100.0%**	**0.011**
Priority	162.0	255.0	100.0%	0.011
RR (Q=100)	162.0	255.0	100.0%	0.011
RR (Q=8)	307.0	400.0	100.0%	0.011

Table 2: Performance on I/O-Bound Workload (20 Processes)

Algorithm	Avg. Wait (ms)	Avg. Turnaround (ms)	CPU Util. (%)	Throughput
FCFS	26.1	29.4	100.0%	0.294
SJF	**18.2**	**21.6**	100.0%	0.294
SRTF	**18.2**	**21.6**	**100.0%**	**0.294**
HRRN	19.0	22.4	100.0%	0.294
Priority	28.6	32.0	100.0%	0.294
RR (Q=10)	26.1	29.4	100.0%	0.294
RR (Q=2)	33.0	36.4	100.0%	0.294

3.3 Discussion of Findings

The data from these experiments reveals several key findings:

Table 3: Performance on Mixed Workload (20 Processes)

Algorithm	Avg. Wait (ms)	Avg. Turnaround (ms)	CPU Util. (%)	Throughput
FCFS	102.7	113.5	100.0%	0.092
SJF	45.4	56.3	100.0%	0.092
SRTF	**45.0**	**56.0**	**100.0%**	**0.092**
HRRN	45.5	56.4	100.0%	0.092
Priority	105.8	116.7	100.0%	0.092
RR (Q=25)	100.3	111.2	100.0%	0.092
RR (Q=4)	87.8	98.7	100.0%	0.092

1. **Average Waiting Time is the Key metric** In all three saturated workloads, the CPU Utilization was 100% and the Throughput was identical across all algorithms. This is because the total amount of ”work” (sum of burst times) was fixed and the CPU was never idle. This proves that for a busy system, the algorithm’s main impact is not on how much work gets done, but on how long individual processes must wait.

2. **The Convoy Effect is Severe.** The Mixed Workload (Table 3) is the most telling. FCFS, Priority, and both RR variations performed terribly, with AWT over 87ms. This is due to the convoy effect, where many small jobs (e.g., burst 1-3) were forced to wait behind a few long jobs (e.g., burst 35-40).

3. **SJF, SRTF, and HRRN Solve the Convoy Effect.** In the same Mixed Workload, SJF, SRTF, and HRRN all had a dramatically lower AWT (around 45ms). This is because they are ”shortest-job-aware” and were able to process the short jobs first, preventing them from getting stuck.

4. **Our New Algorithms (SRTF/HRRN) are Top Performers.** In all three tests, SRTF and HRRN performed at or near the optimal level (SJF/SRTF). This confirms their value and superiority over the basic FCFS and Priority algorithms.

5. **Round Robin has Trade-offs.** In the I/O-Bound test, RR with a quantum of 2 was less efficient (AWT 33.0ms) than with a quantum of 10 (AWT 26.1ms), demonstrating the high overhead of excessive context switching. In the CPU-Bound test, RR (Q=8) was the worst performer by far (AWT 307.0ms) as it just kept cycling through long jobs.

4 Recommendations

Based on the performance analysis, the following recommendations are made to OwlTech Industries.

4.1 Primary Recommendation: Adopt HRRN

The Highest Response Ratio Next (HRRN) algorithm is the clear recommended choice for OwlTech’s mixed-load production environment.

The data in Table 3 shows that HRRN delivers top-tier performance on par with SJF and SRTF, avoiding the convoy effect and more than halving the average waiting time of FCFS or Priority.

While SRTF is theoretically optimal, HRRN has a critical advantage: it prevents starvation. SRTF can indefinitely ignore a long-running process if a steady stream of short jobs arrives. HRRN’s Response Ratio formula explicitly includes ”Waiting Time,” ensuring that a long-waiting process will eventually see its priority rise high enough to be executed.

HRRN provides the optimal performance of SJF/SRTF while also guaranteeing the fairness and responsiveness that is critical for a production system.

4.2 Secondary Recommendations

- **For I/O-Bound Systems:** SRTF could be a great choice if you can assure that starvation won't happen. It would likely outperform HRRN although only by minimal amounts.
- **Decommission FCFS and Priority:** The data clearly shows that both FCFS and the current non-preemptive Priority algorithm are unsuitable for our mixed workloads and are the direct cause of the inefficiencies reported by the division.

References

References

[1] iAmGiG. (2025). *CS-3502-CPU-Sim-Project-StartingPoint*. GitHub repository. Retrieved from `https://github.com/iAmGiG/CS-3502-CPU-Sim-Project-StartingPoint`

A Algorithm Code

A.1 C# code for SRTF (Preemptive)

```
/// <summary>
/// My SRTF (Shortest Remaining Time First) algorithm implementation
/// This is a preemptive version of SJF.
/// </summary>
private List<SchedulingResult> RunSRTFAlgorithm(List<ProcessData> processes)
{
    var results = new Dictionary<string, SchedulingResult>();
    var remainingBurstTimes = new Dictionary<string, int>();

    // Initialize results and remaining burst times for all processes
    foreach (var process in processes)
    {
        remainingBurstTimes[process.ProcessID] = process.BurstTime;
        results[process.ProcessID] = new SchedulingResult
        {
            ProcessID = process.ProcessID,
            ArrivalTime = process.ArrivalTime,
            BurstTime = process.BurstTime,
            StartTime = -1, // -1 indicates not started
            FinishTime = 0
        };
    }

    int currentTime = 0;
    int completedCount = 0;
    int totalProcesses = processes.Count;

    while (completedCount < totalProcesses)
    {
        // Find all arrived processes that are not yet finished
        var arrivedProcesses = processes
            .Where(p => p.ArrivalTime <= currentTime && remainingBurstTimes[p.ProcessID] > 0)
            .ToList();

        if (arrivedProcesses.Count == 0)
        {
            // No process is ready, CPU is idle. Jump to the next arrival time.
            var nextArrivalTime = processes
                .Where(p => remainingBurstTimes[p.ProcessID] > 0)
                .Min(p => p.ArrivalTime);
```

```
            currentTime = nextArrivalTime;
            continue;
        }

        // Select the process with the shortest remaining burst time
        var nextProcess = arrivedProcesses
            .OrderBy(p => remainingBurstTimes[p.ProcessID])
            .First();

        // Check if this is the first time this process is running
        if (results[nextProcess.ProcessID].StartTime == -1)
        {
            results[nextProcess.ProcessID].StartTime = currentTime;
        }

        // "Run" the process for one time unit
        remainingBurstTimes[nextProcess.ProcessID]--;
        currentTime++;

        // Check if the process finished
        if (remainingBurstTimes[nextProcess.ProcessID] == 0)
        {
            var result = results[nextProcess.ProcessID];
            result.FinishTime = currentTime;
            result.TurnaroundTime = result.FinishTime - result.ArrivalTime;
            result.WaitingTime = result.TurnaroundTime - result.BurstTime;
            completedCount++;
        }
    }

    return results.Values.OrderBy(r => r.StartTime).ToList();
}
```

A.2 C# Sharp code for HRRN (Non-Preemptive)

```
/// <summary>
/// HRRN (Highest Response Ratio Next) algorithm implementation
/// This is a non-preemptive algorithm to prevent starvation.
/// Response Ratio = (WaitingTime + BurstTime) / BurstTime
/// </summary>
private List<SchedulingResult> RunHRRNAlgorithm(List<ProcessData> processes)
{
    var results = new List<SchedulingResult>();
    var currentTime = 0;
    // Create a copy of the process list to track remaining processes
    var remainingProcesses = processes.ToList();

    while (remainingProcesses.Count > 0)
    {
        // Get processes that have arrived by current time
        var availableProcesses = remainingProcesses
```

```
            .Where(p => p.ArrivalTime <= currentTime)
            .ToList();

        if (availableProcesses.Count == 0)
        {
            // No process has arrived yet, jump to next arrival time
            currentTime = remainingProcesses.Min(p => p.ArrivalTime);
            continue;
        }

        // Calculate Response Ratio for all available processes
        ProcessData nextProcess = null;
        double highestRatio = -1;

        foreach (var process in availableProcesses)
        {
            int currentWaitTime = currentTime - process.ArrivalTime;
            double responseRatio = (double)(currentWaitTime + process.BurstTime) / process.Bur

            if (responseRatio > highestRatio)
            {
                highestRatio = responseRatio;
                nextProcess = process;
            }
        }

        // Run the selected process (non-preemptive)
        var startTime = currentTime;
        var finishTime = startTime + nextProcess.BurstTime;
        var turnaroundTime = finishTime - nextProcess.ArrivalTime;
        var waitingTime = startTime - nextProcess.ArrivalTime;

        results.Add(new SchedulingResult
        {
            ProcessID = nextProcess.ProcessID,
            ArrivalTime = nextProcess.ArrivalTime,
            BurstTime = nextProcess.BurstTime,
            StartTime = startTime,
            FinishTime = finishTime,
            WaitingTime = waitingTime,
            TurnaroundTime = turnaroundTime
        });

        currentTime = finishTime;
        remainingProcesses.Remove(nextProcess);
    }

    return results.OrderBy(r => r.StartTime).ToList();
}
```