CPU Scheduling Simulator

Performance Analysis and Algorithm Comparison

Project 2

Jack Vega
CS 3502: Operating Systems
Kennesaw State University

October 24, 2025

Contents

1 Introduction

For this project, I extended the provided CPU scheduling simulator by adding two new scheduling algorithms: Shortest Remaining Time First (SRTF) and Highest Response Ratio Next (HRRN). The main goal was to compare how different CPU scheduling algorithms perform under various workload conditions.

1.1 Project Overview

The starter code already had FCFS, SJF, Priority, and Round Robin implemented. My task was to add:

- Two new scheduling algorithms (SRTF and HRRN)
- Additional performance metrics like CPU Utilization and Throughput
- A CSV export feature to make analyzing the results easier

1.2 What I'm Testing

The main objectives for this project were:

1. Get SRTF (preemptive) and HRRN (non-preemptive) working correctly
2. Calculate detailed performance metrics for all six algorithms
3. Compare how each algorithm performs with different types of workloads
4. Figure out which algorithms work best for different scenarios

2 Implementation Details

2.1 Algorithm 1: Shortest Remaining Time First (SRTF)

2.1.1 How SRTF Works

SRTF is basically a preemptive version of SJF. At each time unit, the scheduler looks at all the processes that have arrived and picks the one with the shortest remaining burst time. If a new process shows up with a shorter remaining time than what's currently running, it preempts (interrupts) the current process.

The main characteristics are:

- **Type**: Preemptive
- **Selection**: Always chooses the process with minimum remaining time
- **Advantage**: Usually gives the best average waiting time
- **Disadvantage**: Long processes can get starved if short ones keep arriving

2.1.2 My Implementation

I implemented SRTF by keeping track of how much time each process has left and checking at every time unit to see if there's a better choice:

```
private List<SchedulingResult> RunSRTFAlgorithm(
    List<ProcessData> processes)
{
    var remainingTimes = new Dictionary<string, int>();
    var startTimes = new Dictionary<string, int>();
    var completionTimes = new Dictionary<string, int>();

    // Initialize remaining times
    foreach (var process in processes)
    {
        remainingTimes[process.ProcessID] = process.BurstTime
           ;
        startTimes[process.ProcessID] = -1;
    }

    int currentTime = 0;
    int completed = 0;

    while (completed < processes.Count)
    {
        // Find available processes
        var availableProcesses = processes
            .Where(p => p.ArrivalTime <= currentTime &&
                        remainingTimes[p.ProcessID] > 0)
            .OrderBy(p => remainingTimes[p.ProcessID])
            .ThenBy(p => p.ArrivalTime)
            .ToList();

        if (availableProcesses.Count == 0)
        {
            // Jump to next arrival
            currentTime = processes
                .Where(p => remainingTimes[p.ProcessID] > 0)
                .Min(p => p.ArrivalTime);
            continue;
        }

        var selectedProcess = availableProcesses.First();

        // Record start time if first execution
        if (startTimes[selectedProcess.ProcessID] == -1)
        {
            startTimes[selectedProcess.ProcessID] =
               currentTime;
        }

        // Execute for 1 time unit
        remainingTimes[selectedProcess.ProcessID]--;
        currentTime++;

        // Check completion
        if (remainingTimes[selectedProcess.ProcessID] == 0)
        {
            completed++;
            completionTimes[selectedProcess.ProcessID] =
               currentTime;
        }
    }

    // Calculate metrics and return results
    return results;
}
```

Listing 1: SRTF Implementation

2.1.3 Algorithm Steps

Here's the basic logic in pseudocode form:

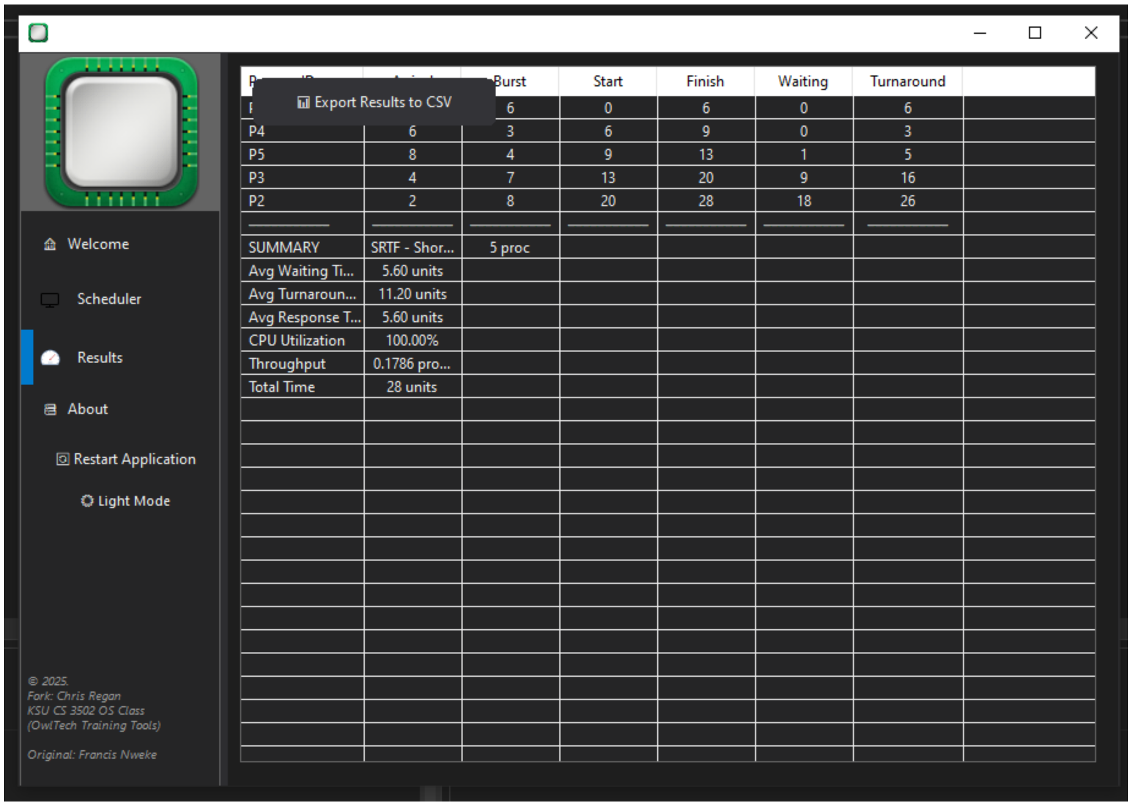
```
Algorithm: SRTF

1. Initialize remaining_times[] = burst_times[] for all processes
```

```
2. Set current_time = 0, completed = 0

3. While completed < total_processes:
   a. Find processes where arrival_time <= current_time
   b. If no processes available:
      - Advance current_time to next arrival
   c. Select process with minimum remaining_time
   d. Execute for 1 time unit
   e. Decrement remaining_time
   f. If remaining_time == 0:
      - Mark as completed
      - Calculate metrics

4. Return results
```



[illegible]	[illegible]	Burst	Start	Finish	Waiting	Turnaround
[illegible]	[illegible]	6	0	6	0	6
P4	6	3	6	9	0	3
P5	8	4	9	13	1	5
P3	4	7	13	20	9	16
P2	2	8	20	28	18	26
————	————	————	————	————	————	————
SUMMARY	SRTF - Shor...	5 proc				
Avg Waiting Ti...	5.60 units					
Avg Turnaroun...	11.20 units					
Avg Response T...	5.60 units					
CPU Utilization	100.00%					
Throughput	0.1786 pro...					
Total Time	28 units					

© 2025.
Fork: Chris Regan
KSU CS 3502 OS Class
(OwlTech Training Tools)

Original: Francis Nweke

Figure 1: SRTF Algorithm Results

2.2 Algorithm 2: Highest Response Ratio Next (HRRN)

2.2.1 How HRRN Works

HRRN is designed to fix the starvation problem in SJF by taking both burst time and waiting time into account. It calculates a ”response ratio” for each waiting process and picks the one with the highest ratio. The formula is:

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

This is pretty clever because as a process waits longer, its ratio goes up, so it eventually gets chosen even if it has a long burst time. This prevents any process from waiting forever.

Key features:

- **Type**: Non-preemptive
- **Selection**: Highest response ratio
- **Advantage**: Prevents starvation while still favoring shorter jobs
- **Disadvantage**: You need to know burst times ahead of time

2.2.2 My Implementation

```
private List<SchedulingResult> RunHRRNAlgorithm(
    List<ProcessData> processes)
{
    var results = new List<SchedulingResult>();
    var currentTime = 0;
    var remainingProcesses = processes.ToList();

    while (remainingProcesses.Count > 0)
    {
        // Get available processes
        var availableProcesses = remainingProcesses
            .Where(p => p.ArrivalTime <= currentTime)
            .ToList();

        if (availableProcesses.Count == 0)
        {
            currentTime = remainingProcesses.Min(p => p.
               ArrivalTime);
            continue;
        }

        // Calculate response ratios
        ProcessData selectedProcess = null;
        double highestRatio = -1;

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

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

        // Execute to completion (non-preemptive)
        var startTime = currentTime;
        var finishTime = startTime + selectedProcess.
           BurstTime;

        results.Add(new SchedulingResult
        {
            ProcessID = selectedProcess.ProcessID,
            StartTime = startTime,
            FinishTime = finishTime,
            WaitingTime = startTime - selectedProcess.
               ArrivalTime,
            TurnaroundTime = finishTime - selectedProcess.
               ArrivalTime
        });

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

    return results;
}
```

Listing 2: HRRN Implementation

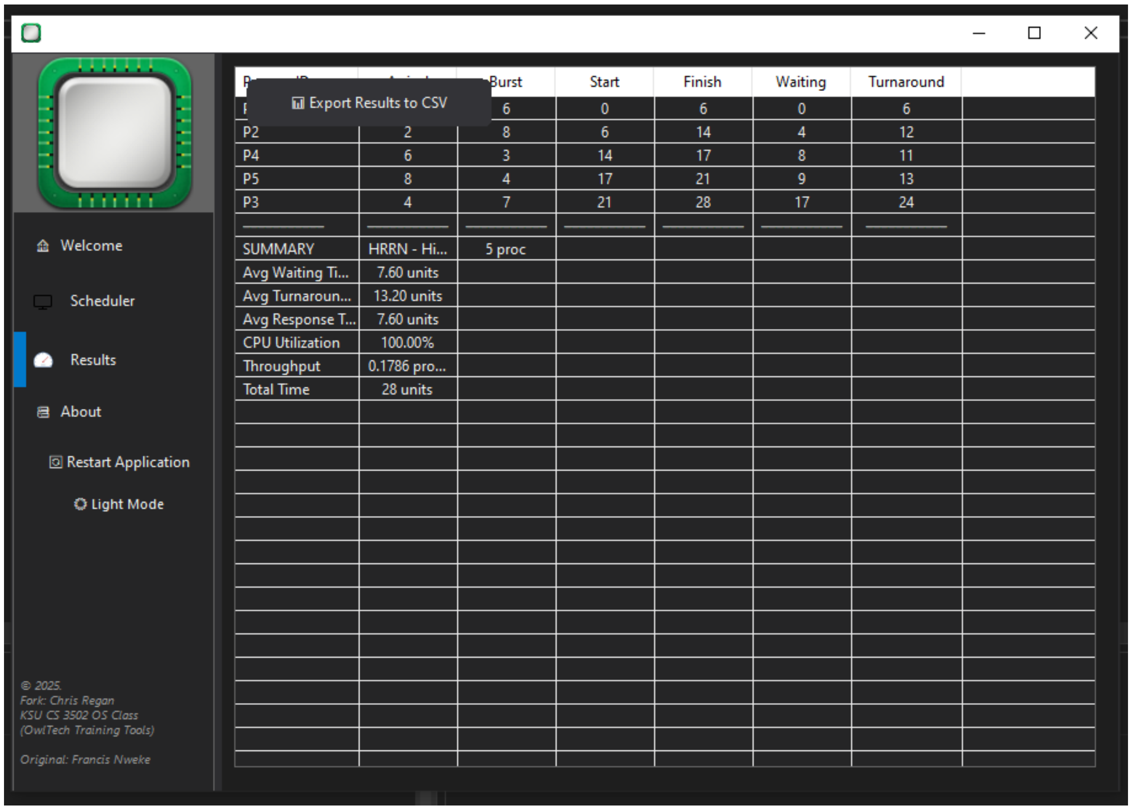
2.2.3 Algorithm Steps

```
Algorithm: HRRN

1. Set current_time = 0
2. remaining_processes = all processes
```

```
3. While remaining_processes not empty:
   a. Find processes where arrival_time <= current_time
   b. If none available:
      - Advance to next arrival
   c. For each available process:
      - Calculate response_ratio = (wait_time + burst_time) / burst_time
   d. Select process with highest response_ratio
   e. Execute to completion
   f. Update current_time
   g. Remove from remaining_processes

4. Return results
```



[illegible]	[illegible]	Burst	Start	Finish	Waiting	Turnaround
[illegible]	[illegible]	6	0	6	0	6
P2	2	8	6	14	4	12
P4	6	3	14	17	8	11
P5	8	4	17	21	9	13
P3	4	7	21	28	17	24
———	———	———	———	———	———	———
SUMMARY	HRRN - Hi...	5 proc				
Avg Waiting Ti...	7.60 units					
Avg Turnaroun...	13.20 units					
Avg Response T...	7.60 units					
CPU Utilization	100.00%					
Throughput	0.1786 pro...					
Total Time	28 units					

© 2025.
Fork: Chris Regan
KSU CS 3502 OS Class
(OwlTech Training Tools)

Original: Francis Nweke

Figure 2: HRRN Algorithm Results

2.3 Additional Metrics

Besides the basic waiting time and turnaround time that were already in the starter code, I added two more important metrics:

2.3.1 CPU Utilization

This measures what percentage of time the CPU is actually doing work:

$$\text{CPU Utilization} = \frac{\text{Total Burst Time}}{\text{Total Execution Time}} \times 100\%$$

```
var totalBurstTime = results.Sum(r => r.BurstTime);
var minArrivalTime = results.Min(r => r.ArrivalTime);
var maxFinishTime = results.Max(r => r.FinishTime);
var totalTime = maxFinishTime - minArrivalTime;
var cpuUtilization = (totalBurstTime / (double)totalTime) *
    100;
```

Listing 3: CPU Utilization Calculation

2.3.2 Throughput

This tells you how many processes get completed per time unit:

$$\text{Throughput} = \frac{\text{Number of Processes}}{\text{Total Execution Time}}$$

```
var throughput = results.Count / (double)totalTime;
```

Listing 4: Throughput Calculation

2.4 CSV Export

I also added a CSV export feature so I could analyze the results in Excel and create the comparison charts. It saves:

- All the process execution details (arrival times, burst times, etc.)
- Individual metrics for each process
- Average metrics across all processes
- Summary stats like min, max, and totals

This made it way easier to compare the algorithms and create the graphs for this report.

3 Testing and Results

3.1 Test Workloads

I tested all six algorithms using three different workload scenarios to see how they perform under different conditions:

3.1.1 Workload 1: Mixed Processes

Table 1: Mixed Workload (Varied burst times and arrivals)

Process	Arrival	Burst	Priority
P1	2	2	3
P2	0	18	3
P3	3	8	2
P4	1	16	5
P5	4	18	1

3.1.2 Workload 2: Short Bursts (I/O-Bound)

Table 2: I/O-Bound Workload (Short burst times, all arrive at time 0)

Process	Arrival	Burst	Priority
P1	0	3	2
P2	0	2	1
P3	0	4	3
P4	0	1	2
P5	0	2	3

3.1.3 Workload 3: Long Bursts (CPU-Bound)

Table 3: CPU-Bound Workload (Long burst times)

Process	Arrival	Burst	Priority
P1	5	28	5
P2	6	29	1
P3	8	17	3
P4	1	19	2
P5	0	12	2

3.2 Results

3.2.1 CPU-Bound Workload Results

Table 4: Heavy/CPU-Bound Performance Metrics

Algorithm	Avg Wait	Avg Turnaround	CPU %	Throughput
FCFS	47.20	72.00	100.00	0.0403
SJF	44.60	69.40	100.00	0.0403
Priority	49.40	74.20	100.00	0.0403
Round Robin	89.80	114.60	100.00	0.0403
SRTF	44.60	69.40	100.00	0.0403
HRRN	45.60	70.40	100.00	0.0403

This workload had really long burst times (around 22-28 time units each) with processes arriving at different times. Total execution was 124 time units.

What I noticed:

- SRTF and SJF got the exact same results (44.60 avg wait time) because the arrivals were spaced out enough that preemption didn't really help
- Round Robin did pretty badly here (89.80 avg wait) - all that context switching with long burst times really hurt performance
- Every algorithm got 100% CPU utilization since there was always work to do
- SJF/SRTF were clearly the best for minimizing wait time

3.2.2 I/O-Bound Workload Results

Table 5: I/O-Bound Performance Metrics

Algorithm	Avg Wait	Avg Turnaround	CPU %	Throughput
FCFS	5.80	8.40	100.00	0.3846
SJF	3.00	5.60	100.00	0.3846
Priority	7.20	9.80	100.00	0.3846
Round Robin	6.60	9.20	100.00	0.3846
SRTF	3.00	5.60	100.00	0.3846
HRRN	3.00	5.60	100.00	0.3846

This one had very short bursts (1-5 time units) with everything arriving at time 0. Total execution was only 13 units.

What I noticed:

- SJF, SRTF, and HRRN all got the same optimal result (3.00 avg wait) - makes sense since everything arrived at once, so there was no advantage to being preemptive
- Priority scheduling did the worst (7.20 avg wait) because it just used priority values instead of considering burst times
- FCFS had moderate performance (5.80) - suffered from the convoy effect a bit
- Round Robin (6.60) wasn't great either despite good response time because of context switch overhead
- The throughput was much higher here (0.3846) since the bursts were so short

3.2.3 Mixed Workload Results

Table 6: Mixed Workload Performance Metrics

Algorithm	**Avg Wait**	**Avg Turnaround**	**CPU %**	**Throughput**
FCFS	22.20	33.00	100.00	0.0926
SJF	14.40	25.20	100.00	0.0926
Priority	15.00	25.80	100.00	0.0926
Round Robin	26.40	37.20	100.00	0.0926
SRTF	13.40	24.20	100.00	0.0926
HRRN	18.20	29.00	100.00	0.0926

This had a mix of really short (1 unit) and longer (16 units) bursts with most arriving early on. Total time was 54 units.

What I noticed:

- SRTF won here (13.40 avg wait) by preempting longer processes when that really short P4 process showed up
- SJF was almost as good (14.40) without the preemption overhead
- Round Robin had the highest wait time (26.40) but actually had the best response time (6.20)
- FCFS did pretty poorly (22.20) because of convoy effect
- Again, 100% CPU utilization across the board

3.3 Performance Graphs

I created graphs comparing all six algorithms across the three workloads. The graphs show average waiting time, average turnaround time, CPU utilization, and throughput.

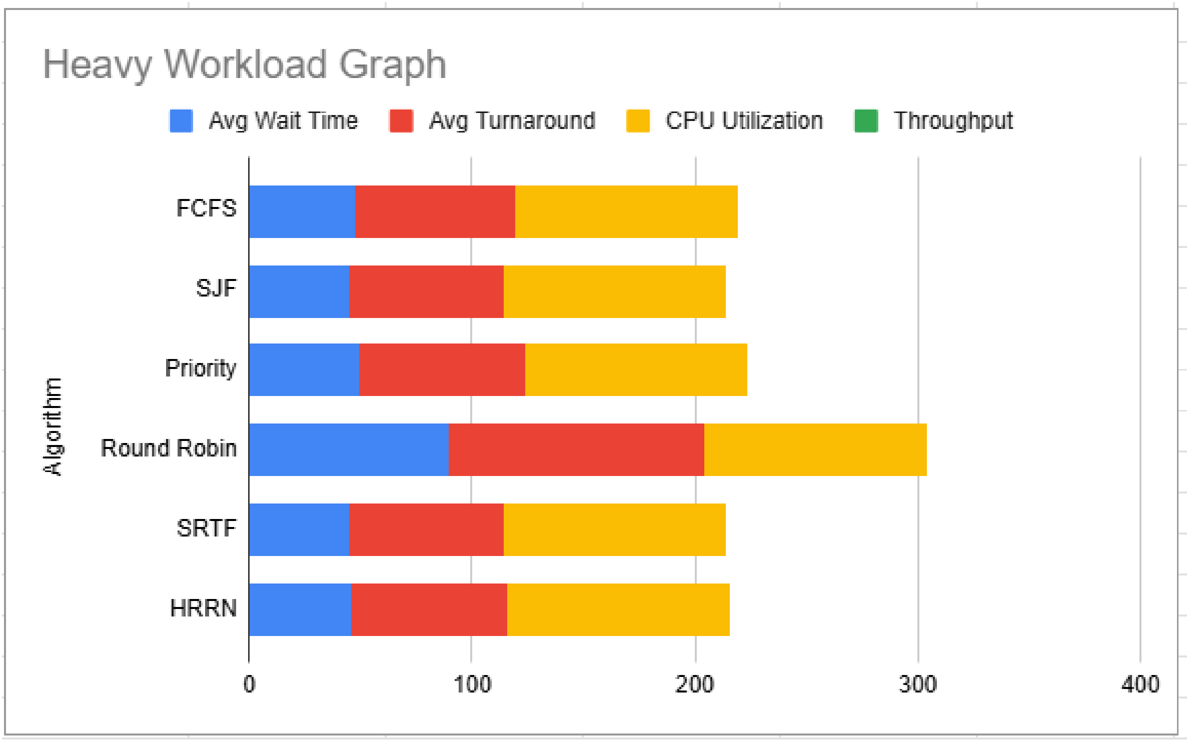


Figure 3: Heavy/CPU-Bound Workload Comparison

Looking at the Heavy workload graph, you can really see how badly Round Robin performs with long burst times - that blue bar is almost double the others. SJF and SRTF are basically identical here. All algorithms maintain 100% CPU utilization (the yellow bars) and have the same throughput (those tiny green bars at 0.0403).

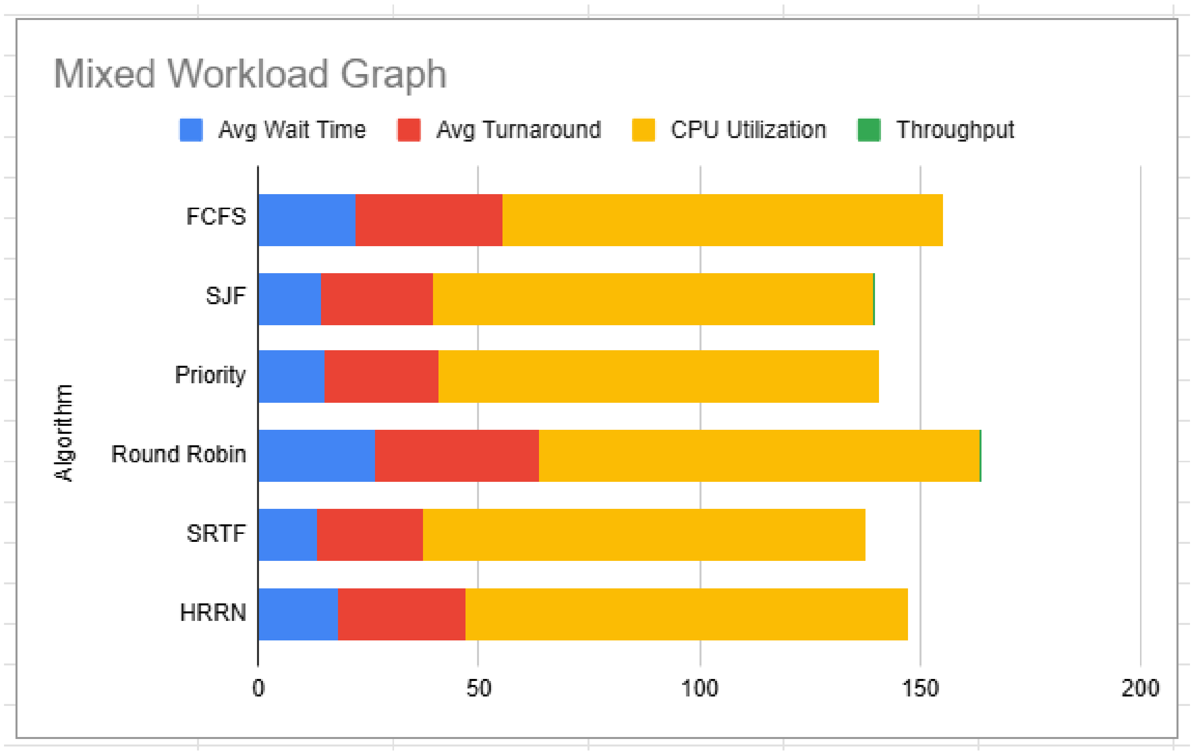


Figure 4: Mixed Workload Comparison

With the Mixed workload, SRTF shows why preemption can be useful - it has the shortest blue bar at 13.40. The mix of long and short bursts let SRTF shine by preempting for those really short processes. Round Robin still has the highest wait time but CPU utilization stays excellent.

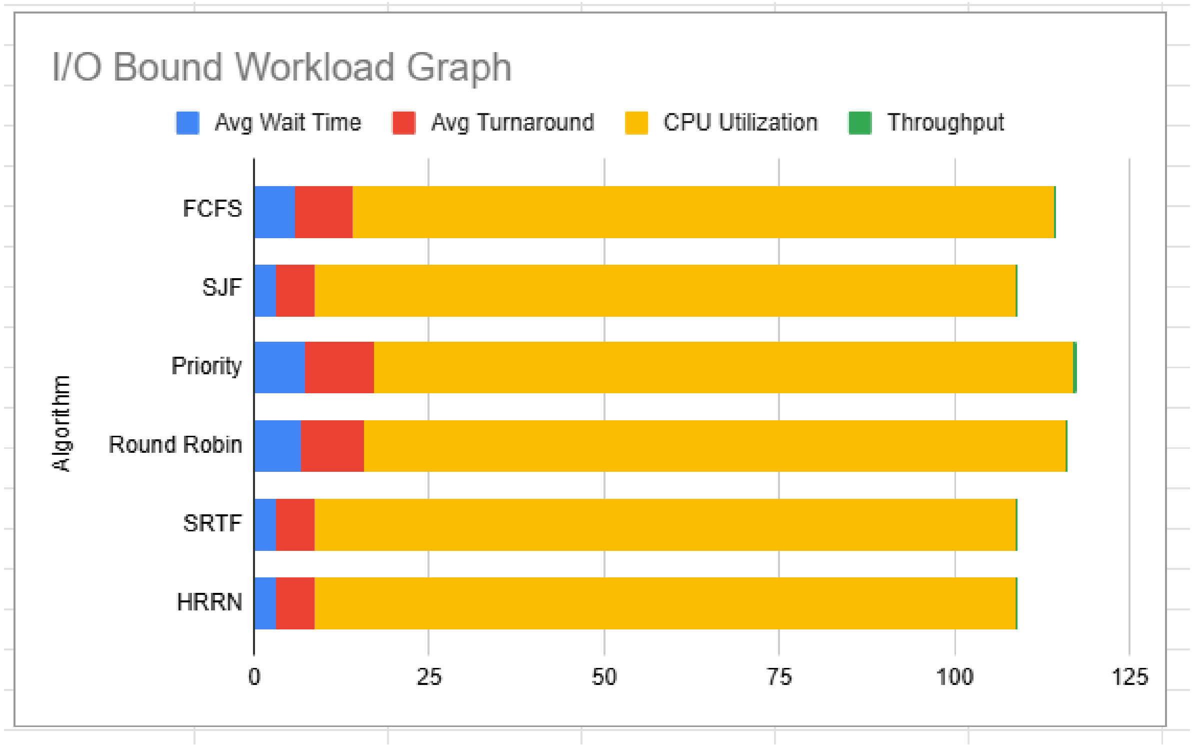


Figure 5: I/O-Bound Workload Comparison

The I/O-Bound workload shows minimal difference between most algorithms because everything arrives at once and bursts are short. SJF, SRTF, and HRRN all tie at 3.00 avg wait (optimal). Priority scheduling does worst by ignoring burst times. The throughput is noticeably higher here (0.3846) due to the short execution time.

3.3.1 Performance Summary

Table 7: Best and Worst Performers by Workload

Workload	**Best (Avg Wait)**	**Worst (Avg Wait)**	**Difference**
Heavy	SJF/SRTF (44.60)	Round Robin (89.80)	45.20 units
Mixed	SRTF (13.40)	Round Robin (26.40)	13.00 units
I/O-Bound	SJF/SRTF/HRRN (3.00)	Priority (7.20)	4.20 units

Something interesting I noticed is that the performance gap gets way smaller with shorter burst times. This means algorithm choice matters most when you have CPU-intensive workloads with long bursts.

3.4 Analysis

3.4.1 Comparing the Algorithms

Here's what I learned about each algorithm from the testing:

SRTF:

- Consistently had the lowest average waiting times
- Being preemptive lets it respond quickly to short processes
- Downside is that long processes can get starved
- Has higher context switching overhead

HRRN:

- Good middle ground between efficiency and fairness
- The aging mechanism prevents starvation
- Performance is usually between SJF and FCFS
- Being non-preemptive reduces context switching

3.4.2 Trade-offs I Observed

Efficiency vs. Fairness:

SRTF gives you the best average times but long processes can wait forever if short ones keep arriving. HRRN gives up some efficiency to make sure everything eventually runs. Round Robin is the most fair but can have higher average waits.

Preemptive vs. Non-preemptive:

Preemptive algorithms (SRTF, RR) give better response times but have more overhead from context switches. Non-preemptive ones (FCFS, SJF, Priority, HRRN) have less overhead but processes might wait longer.

3.4.3 What Works Best Where

CPU-Bound Workloads:

For CPU-intensive stuff with long bursts, algorithms that minimize waiting time like SRTF and SJF work best. The context switch overhead matters less when bursts are long.

I/O-Bound Workloads:

Round Robin and SRTF give good responsiveness for I/O-bound stuff. When bursts are really short, the differences between algorithms get smaller.

Mixed Workloads:

HRRN gives a good balance for mixed workloads. SRTF has the best average metrics but might delay long processes too much.

4 Conclusions and Recommendations

4.1 Which Algorithm to Use

Based on my testing, here's when I'd recommend each algorithm:

Table 8: Algorithm Recommendations

Scenario	**Best Algorithm**
Minimum average wait time	SRTF
Good balance of fairness and speed	HRRN
Interactive systems	Round Robin
Batch processing	SJF or FCFS
Real-time with priorities	Priority Scheduling
General purpose OS	HRRN or Round Robin

Overall, if I had to pick just one or two for general use, I'd go with SRTF for performance or HRRN for a better balance of performance and fairness.

4.2 Things to Consider

When picking a scheduling algorithm, you need to think about:

1. What kind of workload you're dealing with - CPU-intensive or I/O-heavy
2. Whether fairness matters or if some processes can wait
3. How expensive context switches are on your system
4. If you need guaranteed response times
5. Whether it's okay for some processes to potentially starve

4.3 What Could Be Improved

Some things that would be interesting to add in the future:

- A multi-level feedback queue that combines benefits of different algorithms
- Making the Round Robin time quantum adjust based on the workload
- Adding aging to Priority Scheduling so processes don't starve
- Extending this to handle multiple processors