CS 3502 Project 2 - CPU Scheduling

Student Name: Alex Thomas
NETID: Athom596
Course: CS 3502 - 01

Due Date: April 28, 2025

Kennesaw State University

Abstract

This paper presents the implementation of a CPU scheduling simulator designed to evaluate Shortest Remaining Time First (SRTF) and Highest Response Ratio (HRRN). This project was developed in Python, and simulates process scheduling under varying conditions. The project records the average waiting time (AWT), average turnaround time (ATT), CPU utilization, and throughput, and then they are compared to get a better understanding of SRTF and HRRN in the real world.

Throughout the extensive testing with randomized process arrival and burst times, the results demonstrate that SRTF is better than HRRN in minimizing waiting and turnaround times. While this result shows that SRTF is better, the additional research explains how comparing these two algorithms is unreasonable, for they serve two different purposes. SRTF is used to minimize average waiting times, while HRRN is used to prevent starvation.

Contents

Chapter 1

Introduction

This project is an implementation of a CPU scheduling simulator. In the field of operating systems (OS), CPU scheduling determines which process gets access to the CPU based on the algorithm used. The goal is to maximize the CPU utilization, reduce starvation, minimize wasted time, and overall optimization of system performance. This project only covers 2 of the many common CPU scheduling algorithms. These two algorithms are Shortest Remaining Time First (SRTF) and Highest Response Ratio Next (HRRN).

Looking into Shortest Remaining Time First (SRTF), it can also be called Preemptive Shortest Job First (SJF) (GeeksforGeeks, 2025). In the SRTF algorithm, the shortest process is selected to finish. This process will continue until it finishes, or a new process with a shorter remaining time arrives. When a process with a shorter remaining time arrives, it is selected to finish. This is done so that the fastest processes are given priority.

Looking into Highest Response Ratio Next (HRRN), unlike SRTF, it is not preemptive. Though it is still considered one of the most optimal scheduling algorithms (GeeksforGeeks, 2025). Unlike SRTF, it determines which process will run until completion by using the response ratio. This is similar to First-Come-First-Serve (FCFS) and Shortest Job Next (SJN), but it improves by balancing the waiting and burst time (GeeksforGeeks, 2025).

Chapter 2

Implementation Details

To implement this project, there were many steps required. I specifically decided to use Python for this project. To start the project off, I created the Process class. The class itself contains 7 variables. The ID, arrival time, burst time, remaining time, completion time, turnaround time, and waiting time. When the object is created, it takes in an ID, arrival time, and burst time. This is used later for the functions and the application of the SRTF and HRRN algorithms.

The next thing created was the SRTF algorithm. The algorithm for SRTF was implemented as follows:

```
process = min(runnable, key=lambda x: x.remaining_time)
```

This line of code is encapsulated in a while statement to ensure that every iteration checks the available processes for the one with the smallest amount of remaining time. This is what makes the code SRTF instead of SJF. For each of the process objects, the function updates the completion time, turnaround time, and waiting time. Then, at the end of the function, the code returns the CPU utilization and throughput. These values are used later in a different function.

The next function is the HRRN algorithm. The algorithm for HRRN was implemented as follows:

```
for p in runnable:

    p.response_ratio = (current_time - p.arrival_time + p.burst_time) / p.burst_time

process = max(runnable, key=lambda x: x.response_ratio)
```

These lines of code ensure that the process with the highest response ratio is selected to run to completion. The code uses the equation response ratio = (W + S)/S, where W is the waiting time for the process so far and S is the burst time (GeeksforGeeks, 2025). Just as the previous function, the code returns the CPU utilization and throughput for the latter function.

Then I created a function to print the completion times, turnaround times, and wait times for all processes input into the function. These variables were already part of the process objects. The

input for this function was the array of processes, the CPU utilization and throughput from the other functions, and the name of the algorithm used. The function then printed out the average waiting time, average turnaround time, CPU utilization, and throughput. The code for the calculations is as follows:

```
    print(f"\nAverage Waiting Time: {total_wt / len(processes):.2f}")
    print(f"Average Turnaround Time: {total_tat / len(processes):.2f}")
    print(f"CPU Utilization: {cpu_utilization:.2f}%")
    print(f"Throughput: {throughput:.4f} processes per unit time")
```

For the average waiting and turnaround time, the code added together all the waiting times and all of the turnaround times before dividing them by the number of processes. As stated previously, the CPU utilization and throughput were calculated in the other functions. Their code is a follows:

```
cpu_utilization = (busy_time / total_time) * 100 if total_time > 0 else 0
throughput = n / total_time if total_time > 0 else 0
```

As this code shows, the formula for the CPU utilization is the busy time divided by the total time and multiplied by 100. The formula for throughput is the number of processes divided by the total time to determine how many processes are completed during each unit (GeeksforGeeks, 2025).

Finally, I implemented the main. Before implementing the main, I imported the Random library, so the code could generate random numbers for use in testing. In the main, I defined the number of processes randomly, and then I created multiple process objects and stored them in an array using a for loop. The code is as follows.

```
n = random.randint(1,200)

processes = []
for i in range(n):
    arrival = random.randint(1,100)
    burst = random.randint(1,100)
    processes.append(Process(i + 1, arrival, burst))
```

As seen in the code, each process has a randomly generated arrival and burst time, and the ID is based on the iteration in the for loop. I then used both the SRTF and HRRN functions on the array of processes and printed out the results using the print function I created.

Chapter 3

Testing and Results

For testing purposes, most of the numbers have been randomly generated in certain ranges to simulate different scenarios. The first scenario was when the burst and arrival times were from 1 to 10. This test was run 5 times, with the number of processes varying from 36 to 135. The tables created from said test are as follows:

Processes	AWT	ATT	%	Processes per Second
36	70.92	76.83	99.53	0.1682
37	63.11	68.49	99.5	0.185
73	116.63	121.67	99.73	0.1978
94	178.48	184.02	99.81	0.1801
135	246.95	252.35	99.86	0.1849

Table 3.1: SRTF Table Where AT and BT are 1 - 10

Processes	AWT	ATT	%	Processes per Second
36	74.69	80.61	99.53	0.1682
37	63.51	68.89	99.5	0.185
73	118.12	123.16	99.73	0.1978
94	180.21	185.76	99.81	0.1801
135	246.96	252.36	99.86	0.1849

Table 3.2: HRRN Table Where AT and BT are 0 - 10

As seen in these tables, the average waiting time (AWT), average turnaround time (ATT), CPU utilization (%), and throughput (Processes per Second) are similar between SRTF and HRRN. The CPU utilization and throughput are the same for both algorithms, but SRTF has lower AWT and ATT than the HRRN algorithm.

For the second test, the arrival was set to range from 0 to 10, and the burst time was set to range from 500 to 1,000. This test was once again run 5 times. The goal was to see how the

algorithms would handle high burst time scenarios. The tables created from said test are as follows:

Processes	AWT	ATT	%	Processes per Second
39	13438.54	14232.77	100	0.0013
48	15443.04	16184.98	100	0.0013
89	29363.54	30104.55	100	0.0013
103	34924.91	35693.16	100	0.0013
123	41173.89	41938.44	100	0.0013

Table 3.3: SRTF Table Where AT is 0 - 10 and BT is 500 - 1,000

Processes	AWT	ATT	%	Processes per Second
39	13572.31	14366.54	100	0.0013
48	15616.23	16358.17	100	0.0013
89	29510.94	30251.96	100	0.0013
103	34932.63	35700.87	100	0.0013
123	41390.15	42154.70	100	0.0013

Table 3.4: HRRN Table Where AT is 0 - 10 and BT is 500 - 1,000

The results for this test were very similar to the last. The CPU utilization and throughput are the same for both algorithms, but SRTF has lower AWT and ATT than the HRRN algorithm.

This brings us to the final test. In this test, the arrival time was set to 0. The burst time was then set randomly between 1 and 1000. This test, just as the last, was run 5 times. The tables for said test are as follows:

Processes	AWT	ATT	%	Processes per Second
18	3713.78	4313.39	100	0.0017
56	8092.96	8528.41	100	0.0023
67	11228.81	11735.43	100	0.0020
93	15856.65	16376.72	100	0.0019
152	28241.86	28792.26	100	0.0018

Table 3.5: SRTF Table Where AT is 0 and BT is 1 - 1,000

Processes	AWT	ATT	%	Processes per Second
18	3767.33	4366.94	100	0.0017
56	8601.64	9037.09	100	0.0023
67	11711.21	12217.84	100	0.0020
93	15856.65	16376.72	100	0.0019
152	28426.24	28976.63	100	0.0018

Table 3.6: HRRN Table Where AT is 0 and BT is 1 - 1,000

This test ended as with the same results as the last ones. The CPU utilization and throughput are the same for both algorithms, but SRTF has lower AWT and ATT than the HRRN algorithm.

Chapter 4

Analysis and Discussions

When looking at the results from the three tests, it is clear that the SRTF algorithm was better for overall performance based on the metrics measured by the code. For every test, the CPU utilization and throughput were the same between SRTF and HRRN, while SRTF had lower AWTs and ATTs for every test. While the AWTs and ATTs were very similar between the two algorithms, there are some scenarios where slightly larger than average for HRRN in comparison to SRTF. This can be seen in Tables 3.3 and 3.4. In row 4 of these tables, the difference between the AWTs and ATTs of the two algorithms is less than 100, but in row 5, the difference is roughly 200. Is this significant? No, this just means that there were larger wait times for certain processes because the process with the highest response ratio was much longer than the other processes.

Chapter 5

Challenges and Lessons Learned

When creating this code, there were many problems I ran into. I had difficulty implementing the actual algorithms for SRTF and HRRN. With the help of GeeksforGeeks, I was able to understand the implementation of these algorithms. Along with the understanding of the basic algorithms, I also learned more about the metrics used to measure the efficiency of each algorithm. These all helped me come to a better understanding of why the results were as they were in my testing.

Chapter 6

Conclusion and Insights

As stated in the analysis of the test results, SRTF is better on paper when looking at the metrics measured. The question is why. This is due to the nature of SRTF in comparison to HRRN. SRTF chooses the process that would finish fastest and runs it to completion, while HRRN chooses the process with the highest response ratio. What does this mean? This means that there are processes that are starved of resources in SRTF because they would take a long time to finish, while HRRN prevents starvation of long processes.

Therefore, SRTF should be used when trying to minimize waiting time. This would be better since the average waiting time for SRTF was better. On the other hand, HRRN is better when process equality and starvation prevention are important. Thus, when the burst times are short, it is better to use SRTF to minimize waiting times. When there are long and short burst times, its better to use HRRN, for it prevents the starvation that SRTF would have in comparison.

Chapter 7

Citations

"DeepSeek," *Into the Unknown*. https://chat.deepseek.com/

GeeksforGeeks, "GeeksforGeeks," *GeeksforGeeks*. https://www.geeksforgeeks.org/