

# A Comparative Analysis of Scheduling Algorithms and the relationship of their performance with parameters within

Sayan Samanta

CPE-534  
Research Project Summary

© 2025 Sayan Samanta. This report is released under the *Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0)*.  
Commercial use of this material is prohibited without written permission from the author.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Summary of Research Topics</b>	<b>3</b>
2.1	Completely Fair Scheduler . . . . .	3
2.2	Modified Round Robin . . . . .	4
<b>3</b>	<b>Summary of Experiment Conducted</b>	<b>5</b>
<b>4</b>	<b>Results</b>	<b>6</b>
4.1	Comparing the performance of the Algorithms . . . . .	6
4.2	Performance Dependency on their parameters within . . . . .	11
<b>5</b>	<b>Discussion and Conclusion</b>	<b>15</b>
<b>6</b>	<b>Appendix - Python Codes</b>	<b>16</b>

## List of Figures

1	Gantt Chart for CFS . . . . .	7
2	Gantt Chart for Modified RR . . . . .	7
3	Gantt Chart for Classical RR . . . . .	8
4	Gantt Chart for SJF . . . . .	8
5	Context Switches for Testcase-1 . . . . .	9
6	ATA for Testcase-1 . . . . .	9
7	AWT for Testcase-1 . . . . .	10
8	ATA for Testcase-2 . . . . .	10
9	AWT for Testcase-2 . . . . .	11
10	CFS ATA Dependency on Scheduling Latency . . . . .	11
11	CFS AWT Dependency on Scheduling Latency . . . . .	12
12	SJF ATA Dependency on Alpha . . . . .	12
13	SJF AWT Dependency on Alpha . . . . .	13
14	Modified RR ATA Dependency on Initial Time quantum K .	13
15	Modified RR AWT Dependency on Initial Time quantum K .	14
16	Classical RR ATA Dependency on Time quantum K . . . . .	14
17	Classical RR AWT Dependency on Time quantum K . . . . .	15

# 1 Introduction

As CPU scheduling is one of the most important parts of computer performance, it will be beneficial to have a comparative analysis of different algorithms where each of them may be better at one or some performance metric than the other. Overall aim of this project is to implement Completely Fair Scheduler (CFS), an optimized Round Robin (RR) [1], classical Round Robin and Shortest Job First (SJF) algorithms in Python where the input will be a list of processes with their burst time and other important parameters and then look at different performance metric such as number of context switches, average wait time, average turn around time etc. Then vary parameters within the algorithm such as time quantum in RR or alpha in SJF and compare results to find how they impact the performance of the algorithms.

## 2 Summary of Research Topics

### 2.1 Completely Fair Scheduler

Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class.

Thus CFS is used for scheduling non real-time tasks in Linux with an aim to give a fair share of CPU to all tasks in proportion to their priority. As this algorithm is used for non real-time tasks, this does not have a bounded performance which means a lower priority process will definitely get the CPU access but there is no guarantee about the waiting time.

In Linux for non real-time tasks, Highest Priority is 100 and Lowest Priority is 139. Nominal priority is set at 120. A nice value (120 - priority) is then derived from priority and has a range from -20 to +19. Further, weight value is now assigned based on the nice value which is then used to calculate vruntime.

The CFS scheduler records how long each task has run by maintaining the virtual run time of each task using the per-task variable vruntime. The virtual run time is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks. For tasks at normal priority (nice values of 0), virtual run time is

identical to actual physical run time. Thus, if a task with default priority runs for 200 milliseconds, its vruntime will also be 200 milliseconds. However, if a lower-priority task runs for 200 milliseconds, its vruntime will be higher than 200 milliseconds. Similarly, if a higher-priority task runs for 200 milliseconds, its vruntime will be less than 200 milliseconds. To decide which task to run next, the scheduler simply selects the task that has the smallest vruntime value. In addition, a higher-priority task that becomes available to run can preempt a lower-priority task [6].

To achieve this. a Red Black Tree structure, with vruntime as key, is used to keep task list. So automatically Process with lowest vruntime will be scheduled first in CPU for a time slice. Time slice in CFS is dynamically calculated for every iteration. It is given by  $\max(scheduling\_latency/n, minimum\_granularity)$  where n is the number of process in the ready queue at the start of an iteration. The paramemters scheduling latency and minimum granularity varies with system, for example in Blackhawk these values are 24ms and 10ms. Bouron et al. [2] have used 6ms and 0.75ms as these values.

This approach makes CFS a fair scheduler as we can't know the burst time of a process before hand and every process will get CPU access in proportion to their priority.

## 2.2 Modified Round Robin

Singh et al. [1] have modified the classical Round Robin algorithm to improve its performance parameters. It is implemented in two phases:

1. Phase-1: Allocate every process to CPU, a single time by applying RR scheduling with an initial time quantum(say k units).
2. Phase-2: After completing first cycle perform the following steps:
  - a. Double the initial time quantum (2k units)
  - b. Select the shortest process from the waiting queue and assign to CPU and then the next shortest and so on
  - c. Repeat previous step until all processes are completed

### 3 Summary of Experiment Conducted

Python was used implement the 4 algorithms. A separate class called process was defined and the same process class could be used for CFS, Modified RR and Classical RR, so they are in the same python script 1 but SJF had to be implemented in a seprate one 2.

Parameters used for the experiments are listed below:

CFS:

1. Scheduling latency = 5, 10 (Default) and 15ms
2. Minimum granularity = 1ms

Modified RR:

1. Phase-1 Time Quantum, K = 5, 10 (Default) and 15ms
2. Thus Phase-2 Time Quantum, 2K = 10, 20, 30ms

Classical RR:

1. Time Quantum, K = 5, 10 (Default) and 15ms

SJF:

1. Alpha = 0.2, 0.5 (Default) and 0.8
2. Predicted Burst for first process = 5ms

For CFS implementation, I used heap to simulate the RBT behavior. The performance of the algorithms were measured and compared for two different test cases listed in Table 1 with total burst time 255 and Table 2 with total burst time 235. The testcase-1 in Table 1 has an unpredictable CPU burst time with no definitive pattern. Priority values are necessary for CFS where 100 is the highest priority and 139 is the lowest.

Table 1: Testcase-I Input Process List

Process ID	Priority	Burst Time
1	120	8
2	115	5
3	130	30
4	100	9
5	120	27
6	108	115
7	136	7
8	100	2
9	110	18
1	125	34

Table 2: Testcase-2 Input Process List

Process ID	Priority	Burst Time
1	120	8
2	115	12
3	130	18
4	100	19
5	120	27
6	108	38
7	136	50
8	100	31
9	110	18
1	125	14

## 4 Results

### 4.1 Comparing the performance of the Algorithms

Let's first look at the gantt chart of each of the algorithms at their default parameter values for testcase-1 listed in Table 1.

PID 8	0--1 ms	PID 9	29--30 ms	PID 9	58--59 ms	PID 6	110--112 ms	PID 10	168--170 ms
PID 4	1--2 ms	PID 6	30--31 ms	PID 6	59--60 ms	PID 6	112--114 ms	PID 5	178--172 ms
PID 8	2--3 ms	PID 6	31--32 ms	PID 9	60--61 ms	PID 6	114--116 ms	PID 5	172--174 ms
PID 4	3--4 ms	PID 6	32--33 ms	PID 6	61--62 ms	PID 6	116--118 ms	PID 5	174--176 ms
PID 4	4--5 ms	PID 6	33--34 ms	PID 6	62--63 ms	PID 6	118--120 ms	PID 5	176--178 ms
PID 4	5--6 ms	PID 6	34--35 ms	PID 9	63--64 ms	PID 6	128--122 ms	PID 10	178--180 ms
PID 4	6--7 ms	PID 9	35--36 ms	PID 6	64--66 ms	PID 6	122--124 ms	PID 5	186--182 ms
PID 6	7--8 ms	PID 2	36--37 ms	PID 5	66--68 ms	PID 6	124--126 ms	PID 5	182--184 ms
PID 4	8--9 ms	PID 1	37--38 ms	PID 1	68--70 ms	PID 6	126--128 ms	PID 5	184--186 ms
PID 4	9--10 ms	PID 5	38--39 ms	PID 6	70--72 ms	PID 6	130--132 ms	PID 10	186--188 ms
PID 4	10--11 ms	PID 6	39--40 ms	PID 6	72--74 ms	PID 6	132--134 ms	PID 5	188--190 ms
PID 4	11--12 ms	PID 9	40--41 ms	PID 6	74--76 ms	PID 5	134--136 ms	PID 5	192--194 ms
PID 9	12--13 ms	PID 6	41--42 ms	PID 6	76--78 ms	PID 1	136--138 ms	PID 10	194--197 ms
PID 6	13--14 ms	PID 6	42--43 ms	PID 6	78--80 ms	PID 6	138--140 ms	PID 3	197--200 ms
PID 6	14--15 ms	PID 9	43--44 ms	PID 6	80--82 ms	PID 6	148--142 ms	PID 7	208--203 ms
PID 9	15--16 ms	PID 6	44--45 ms	PID 6	82--84 ms	PID 6	142--144 ms	PID 10	203--206 ms
PID 6	16--17 ms	PID 9	45--46 ms	PID 10	84--86 ms	PID 6	144--146 ms	PID 10	206--209 ms
PID 9	17--18 ms	PID 6	46--47 ms	PID 6	86--88 ms	PID 6	148--149 ms	PID 10	209--212 ms
PID 2	18--19 ms	PID 2	47--48 ms	PID 6	88--90 ms	PID 6	150--152 ms	PID 3	212--215 ms
PID 6	19--20 ms	PID 6	48--49 ms	PID 6	90--92 ms	PID 6	152--154 ms	PID 10	215--218 ms
PID 6	20--21 ms	PID 6	49--50 ms	PID 6	92--94 ms	PID 6	154--156 ms	PID 10	218--222 ms
PID 9	21--22 ms	PID 6	50--51 ms	PID 6	94--96 ms	PID 6	156--158 ms	PID 10	220--224 ms
PID 6	22--23 ms	PID 9	51--52 ms	PID 6	96--98 ms	PID 6	158--160 ms	PID 3	224--227 ms
PID 9	23--24 ms	PID 6	52--53 ms	PID 6	98--100 ms	PID 6	160--162 ms	PID 10	227--230 ms
PID 6	24--25 ms	PID 6	53--54 ms	PID 5	100--102 ms	PID 6	162--164 ms	PID 10	234--232 ms
PID 6	25--26 ms	PID 9	54--55 ms	PID 1	102--104 ms	PID 6	164--165 ms	PID 3	232--237 ms
PID 9	26--27 ms	PID 2	55--56 ms	PID 6	104--106 ms	PID 5	165--167 ms	PID 7	237--241 ms
PID 2	27--28 ms	PID 6	56--57 ms	PID 6	106--108 ms	PID 1	167--168 ms	PID 3	241--251 ms
PID 6	28--29 ms	PID 6	57--58 ms	PID 6	108--110 ms	PID 10	168--170 ms	PID 3	251--255 ms

Figure 1: Gantt Chart for CFS

PID 1	0--8 ms
PID 2	8--13 ms
PID 3	13--23 ms
PID 4	23--32 ms
PID 5	32--42 ms
PID 6	42--52 ms
PID 7	52--59 ms
PID 8	59--61 ms
PID 9	61--71 ms
PID 10	71--81 ms
PID 9	81--89 ms
PID 5	89--106 ms
PID 3	106--126 ms
PID 10	126--146 ms
PID 6	146--166 ms
PID 10	166--170 ms
PID 6	170--190 ms
PID 6	190--210 ms
PID 6	210--230 ms
PID 6	230--250 ms
PID 6	250--255 ms

Figure 2: Gantt Chart for Modified RR

PID 1	0--8 ms	PID 6	166--176 ms
PID 2	8--13 ms	PID 10	176--180 ms
PID 3	13--23 ms	PID 6	180--190 ms
PID 4	23--32 ms	PID 6	190--200 ms
PID 5	32--42 ms	PID 6	200--210 ms
PID 6	42--52 ms	PID 6	210--220 ms
PID 7	52--59 ms	PID 6	220--230 ms
PID 8	59--61 ms	PID 6	230--240 ms
PID 9	61--71 ms	PID 6	240--250 ms
PID 10	71--81 ms	PID 6	250--255 ms
PID 3	81--91 ms		
PID 5	91--101 ms		
PID 6	101--111 ms		
PID 9	111--119 ms		
PID 10	119--129 ms		
PID 3	129--139 ms		
PID 5	139--146 ms		
PID 6	146--156 ms		
PID 10	156--166 ms		

Figure 3: Gantt Chart for Classical RR

PID 1	0--8 ms
PID 2	8--13 ms
PID 3	13--43 ms
PID 5	43--70 ms
PID 4	70--79 ms
PID 10	79--113 ms
PID 6	113--228 ms
PID 9	228--246 ms
PID 8	246--248 ms
PID 7	248--255 ms

Figure 4: Gantt Chart for SJF

As we look over this scheduling chart, first thing that comes to mind is the number of context switches for each algorithm for the given test case. So we can see from 5, CFS has a very significant overhead in terms of context switches.

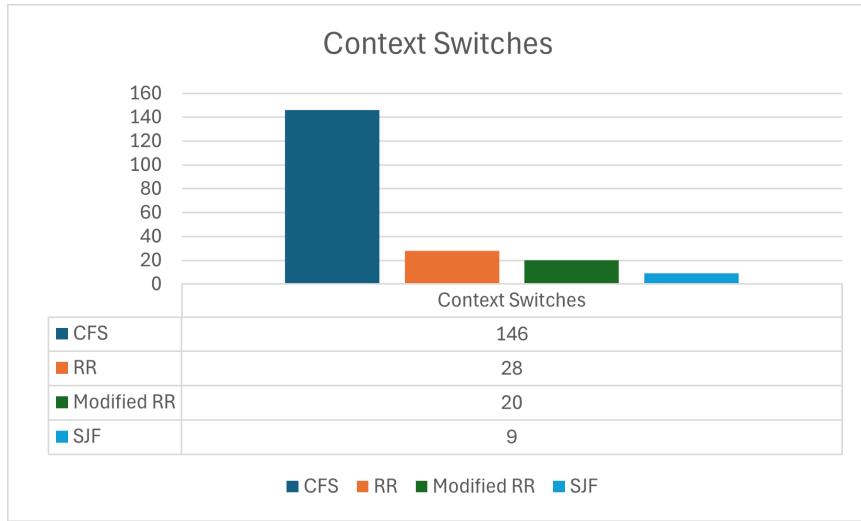


Figure 5: Context Switches for Testcase-1

Next, we compare the performances in terms of their Average Wait Times (AWT) and Average Turnaround Time (ATA) for the testcase-1 from Table 1. From figure 6 and 7, it can be observed that both Round Robins doing much better than the other two and Modified RR is the best performer.



Figure 6: ATA for Testcase-1



Figure 7: AWT for Testcase-1

Next, we compare the performances in terms of their Average Wait Times (AWT) and Average Turnaround Time (ATA) for the testcase-2 from Table 2.

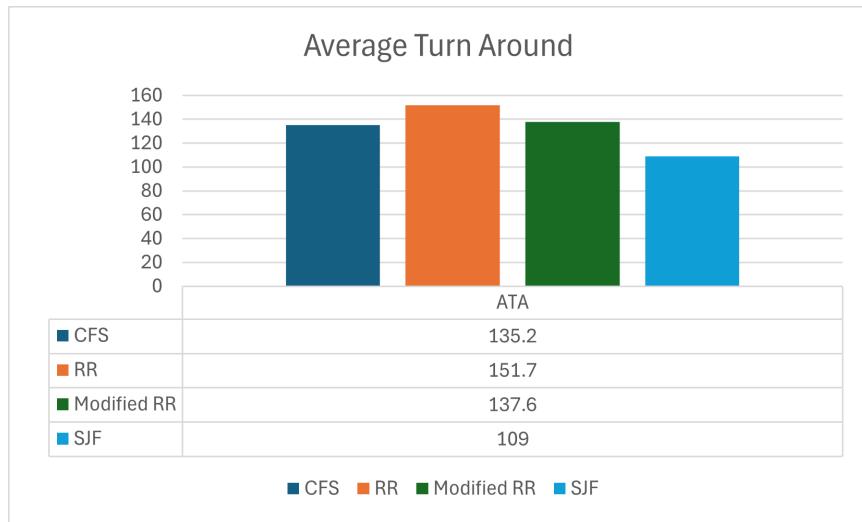


Figure 8: ATA for Testcase-2

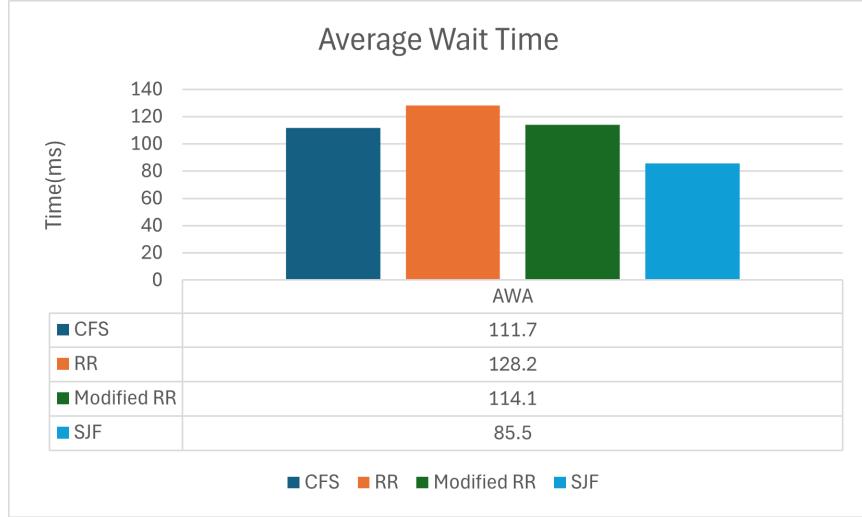


Figure 9: AWT for Testcase-2

We can observe from the figure 8 and 9 that SJF is the best performer here as the process burst times has a pattern.

#### 4.2 Performance Dependency on their parameters within

First, let us observe how ATA and AWT varies for CFS algorithm when the scheduling latency is varied. We can observe that both the parameter is increasing with increasing scheduling latency in figure 10 and 11.

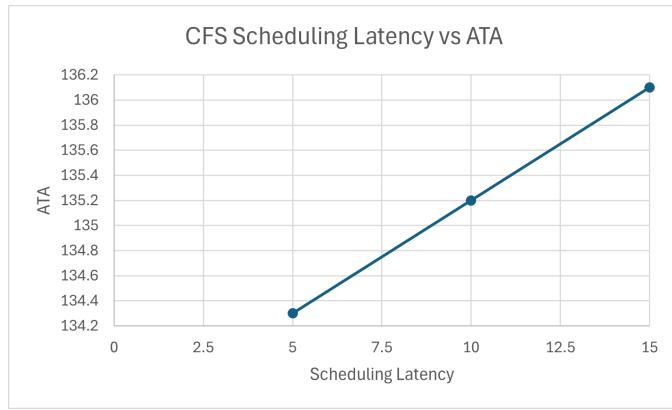


Figure 10: CFS ATA Dependency on Scheduling Latency

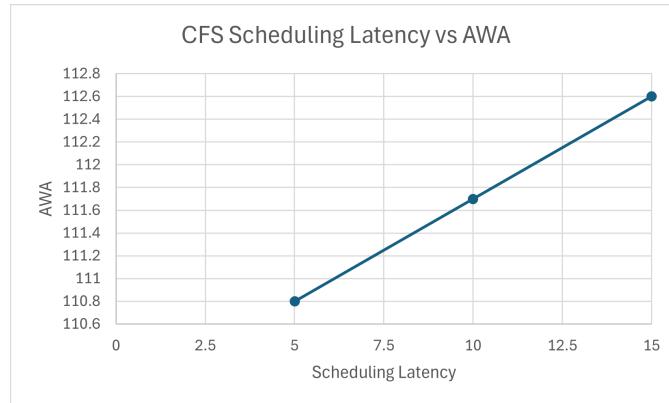


Figure 11: CFS AWT Dependency on Scheduling Latency

Figure 12 and 13 shows how varying alpha value impacts the ATA and AWT for testcase-2. This may differ for other testcases.

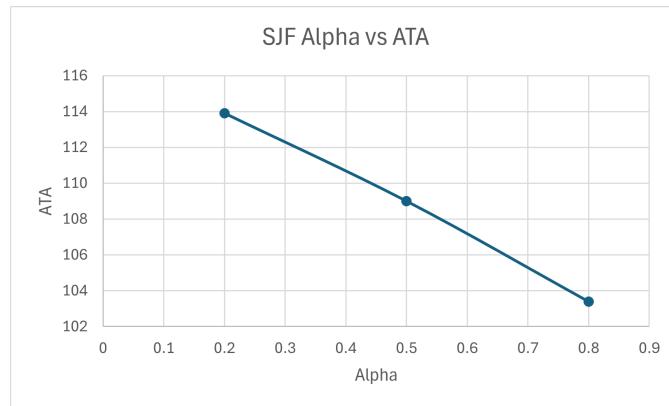


Figure 12: SJF ATA Dependency on Alpha

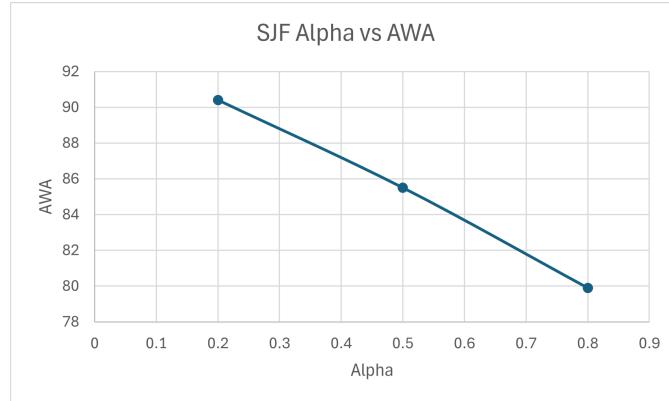


Figure 13: SJF AWT Dependency on Alpha

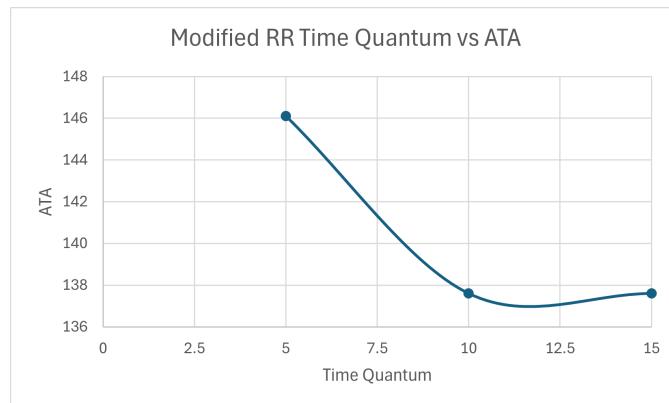


Figure 14: Modified RR ATA Dependency on Initial Time quantum K

The performance of modified RR saturates at initial time quantum  $k = 10$  as it can be observed from figure 14 and 15.

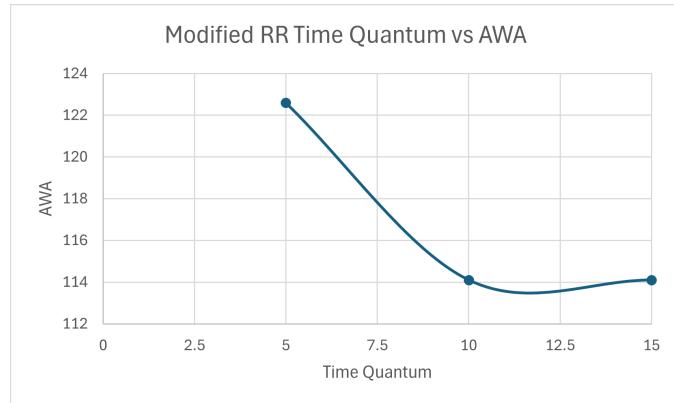


Figure 15: Modified RR AWT Dependency on Initial Time quantum K

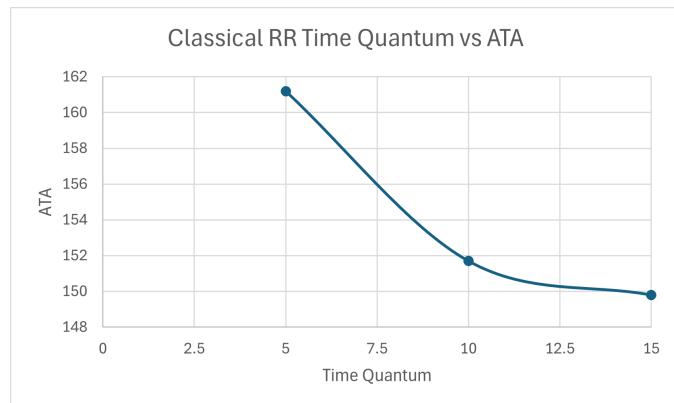


Figure 16: Classical RR ATA Dependency on Time quantum K

But, performance of classical RR improves with increasing the time quantum and can be observed the same from figure 16 and 17.

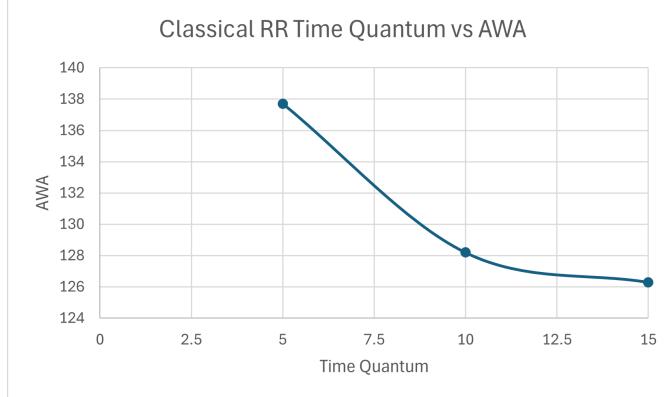


Figure 17: Classical RR AWT Dependency on Time quantum K

## 5 Discussion and Conclusion

As we have discussed the results obtained in the previous section, we can observe that the performance of the CFS algorithm remains similar for the two different testcases. This is because the scheduling solely depends on the vruntime parameter which has a decaying relation with nice value of a process. The relationship is given below:

```

1 Nice_value = 120 - priority
2 Weight = 1024 / (1.25 ^ nice_value)
3 Time_slice = max(round(scheduling_latency/len(ready_queue)),
   ↪ minimum_granularity)
4 exec_time = min(time_slice, process.remaining_time)
5 process.vruntime += exec_time * (1024 / weight)

```

The relations between scheduling latency and its performance is a linear one where performance degrades when latency increases.

The modified RR does not change the performance much if the initial time quantum is already large. We can probably start with a lower time quantum for the same reason. But classical RR performance improves with increasing time quantum.

SJF has a very high dependency on the alpha value. When alpha is 0.5, the predicted burst of the current process is the average of the actual burst and predicted burst of the previous process. When reduced, predicted burst of the previous process gets more weightage and when increased actual burst

of the previous process gets more weightage. Thus depending on the input process list, the result will vary when changing alpha.

Thus we can conclude that the performance of different scheduling algorithms vary highly on the present input list and also on their parameters. We observed that SJF worked better when the burst time of the processes had a trend and for purely random burst time, the modified RR and even classical RR had much better performance. This kind of sums it up on the need of having a hybrid process queue where each queue follows a separate scheduling algorithms.

## 6 Appendix - Python Codes

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Apr 12 15:02:33 2025
4
5 @author: check
6 """
7 from collections import deque
8 import heapq
9 class Process:
10     def __init__(self, pid, priority, burst_time):
11         self.pid = pid
12         self.priority = priority
13         self.burst_time = burst_time
14         self.remaining_time = burst_time
15         self.nice = self.priority - 120
16         self.weight = 1024 / (1.25 ** self.nice)
17         self.vruntime = (1024 / self.weight)
18         self.start_time = None
19         self.end_time = None
20
21     def reset(self):
22         self.remaining_time = self.burst_time
23         self.vruntime = 0
24         self.start_time = None
25         self.end_time = None
26
27     def __lt__(self, other):
28         return self.vruntime < other.vruntime
29
30
31
32 def run_cfs(process_list):
```

```

33     current_time = 0
34     gantt_chart = []
35     ready_queue = []
36     completed = []
37
38     num_process = len(process_list)
39     heapq.heapify(process_list)
40     ready_queue = process_list
41     print("\nInitial Ready Queue Order:")
42     #for p in ready_queue:
43     #    print(f"PID {p.pid} | Nice {p.nice} | Weight {round(p.
44     #        ↪ weight, 2)} | vruntime {p.vruntime}")
45     #ready_queue = sorted(process_list, key=lambda p:
46     #    ↪ calculate_weight(calculate_nice(p.priority)), reverse
47     #    ↪ =True)
48     #ready_queue = heapq.heapify(process_list)
49     print("\n\nstarting CFS...")
50     while ready_queue:
51         time_slice=max(round(10/len(ready_queue)),1)
52         process = heapq.heappop(ready_queue)
53         #nice_value = calculate_nice(process.priority)
54         #weight = calculate_weight(nice_value)
55         #print(f"\n[Time {current_time}] Selected PID {process.
56         #    ↪ pid} with vruntime {round(process.vruntime, 6)}")
57         exec_time = min(time_slice, process.remaining_time)
58         process.remaining_time -= exec_time
59         process.vruntime += exec_time * (1024 / process.weight)
60
61         # Track start and finish time
62         if process.start_time is None:
63             process.start_time = current_time
64             current_time += exec_time
65             gantt_chart.append((str(current_time-exec_time)+'--'+
66             ↪ str(current_time), process.pid))
67
68         if process.remaining_time == 0:
69             process.end_time = current_time
70             completed.append(process)
71         else:
72             heapq.heappush(ready_queue, process)
73
74     # Print Gantt Chart
75     print("\nCFS Gantt Chart :")
76     for time, pid in gantt_chart:
77         print(f"  PID {pid:>2}  |  {time:>3} ms")
78
79     # Metrics
80     total_turnaround = 0;
81     total_waiting = 0;

```

```

77     print("\nPerformance Metrics of CFS:")
78     print(f"{'PID':<5}{'Start':<8}{'End':<8}{'Burst':<8}{'
    ↪ Waiting':<10}{'Turnaround':<12}"))
79     for p in sorted(completed, key=lambda x: x.pid):
80         turnaround = p.end_time
81         waiting = turnaround - p.burst_time
82         total_turnaround += turnaround;
83         total_waiting += waiting;
84         print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
    ↪ burst_time:<8}{waiting:<10}{turnaround:<12}"))
85     Average_TAT = total_turnaround/num_process
86     Average_WT = total_waiting/num_process
87     print(f"Average Turn Around Time: {round(Average_TAT,2)}")
88     print(f"Average Wait Time: {round(Average_WT,2)}")
89     print(f"Context Switches: {len(gantt_chart)-1}")
90
91 def run_round_robin(process_list, time_slice=10):
92     current_time = 0
93     gantt_chart = []
94     queue = deque()
95     completed = []
96
97     ready_queue = sorted(process_list, key=lambda p: p.pid)
98     queue = deque(ready_queue)
99
100    print("\n\nstarting classical RR...")
101    track=1
102    while queue:
103        track += 1
104        process = queue.popleft()
105
106        # Track first start time
107        if process.start_time is None:
108            process.start_time = current_time
109
110        exec_time = min(time_slice, process.remaining_time)
111        process.remaining_time -= exec_time
112        current_time += exec_time
113        gantt_chart.append((str(current_time-exec_time)+'--'+
    ↪ str(current_time), process.pid))
114        if process.remaining_time == 0:
115            process.end_time = current_time
116            completed.append(process)
117        else:
118            queue.append(process)
119
120
121    # Print Gantt Chart
122    print("\n Classical RR Gantt Chart:")

```

```

123     for time, pid in gantt_chart:
124         print(f"  PID {pid:>2} | {time:>3} ms")
125
126     # Metrics
127     total_turnaround = 0;
128     total_waiting = 0;
129     print("\nPerformance Metrics of Classical RR:")
130     print(f'{PID':<5}{Start':<8}{End':<8}{Burst':<8}{'
131         ↪ Waiting':<10}{Turnaround':<12}"')
132     for p in sorted(completed, key=lambda x: x.pid):
133         turnaround = p.end_time
134         waiting = turnaround - p.burst_time
135         total_turnaround += turnaround;
136         total_waiting += waiting;
137         print(f'{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.'
138             ↪ burst_time:<8}{waiting:<10}{turnaround:<12}"')
139     Average_TAT = total_turnaround/len(process_list)
140     Average_WT = total_waiting/len(process_list)
141     print(f"Average Turn Around Time: {round(Average_TAT,2)}")
142     print(f"Average Wait Time: {round(Average_WT,2)}")
143     print(f"Context Switches: {len(gantt_chart)-1}")
144
145 def run_modified_round_robin(process_list, time_slice=10):
146     current_time = 0
147     gantt_chart = []
148     queue = deque()
149     completed = []
150
151     print("\n\nstarting modified RR...")
152     track_loop = 0
153     time_track = 1
154     track=0
155     ready_queue = sorted(process_list, key=lambda p: p.pid)
156     queue = deque(ready_queue)
157     #print(len(queue))
158     n = len(ready_queue)
159     while queue:
160         #print("queue len:"+str(len(queue)))
161         #print(track_loop)
162         if (track_loop == n):
163             updated_queue = sorted(queue, key=lambda p: p.
164                 ↪ remaining_time)
165             queue = deque(updated_queue)
166             n = len(queue)
167             track_loop = 1
168             time_track = 2
169             track=1
170         elif (track==1) & (track_loop == n):

```

```

168         updated_queue = sorted(queue, key=lambda p: p.
169             ↪ remaining_time)
170         queue = deque(updated_queue)
171         n = len(queue)
172         track_loop = 1
173         #print("i am here in 1")
174     else:
175         track_loop += 1
176     process = queue.popleft()
177     #print(process.pid, process.remaining_time, time_track)
178
179     if process.start_time is None:
180         process.start_time = current_time
181
182     exec_time = min(time_track*time_slice, process.
183         ↪ remaining_time)
184     process.remaining_time -= exec_time
185     current_time += exec_time
186     gantt_chart.append((str(current_time-exec_time) + '--' +
187         ↪ str(current_time), process.pid))
188     if process.remaining_time == 0:
189         process.end_time = current_time
190         completed.append(process)
191     else:
192         queue.append(process)
193
194     # Print Gantt Chart
195     print("\n Modified RR Gantt Chart:")
196     for time, pid in gantt_chart:
197         print(f" PID {pid:>2} | {time:>3} ms")
198
199     # Metrics
200     total_turnaround = 0;
201     total_waiting = 0;
202     print("\nPerformance Metrics of Modified RR:")
203     print(f"{'PID':<5}{{'Start':<8}{{'End':<8}{{'Burst':<8}{{
204         ↪ Waiting':<10}{'Turnaround':<12}}}")
205     for p in sorted(completed, key=lambda x: x.pid):
206         turnaround = p.end_time
207         waiting = turnaround - p.burst_time
208         total_turnaround += turnaround;
209         total_waiting += waiting;
210         print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
211             ↪ burst_time:<8}{waiting:<10}{turnaround:<12}}")
212     Average_TAT = total_turnaround/len(process_list)
213     Average_WT = total_waiting/len(process_list)
214     print(f"Average Turn Around Time: {round(Average_TAT,2)}")

```

```

212     print(f"Average Wait Time: {round(Average_WT ,2)}")
213     print(f"Context Switches: {len(gantt_chart)-1}")
214
215 def reset_process(process_list):
216     for item in process_list:
217         item.reset()
218
219
220
221 processes = [
222     Process(1, 120, 8),
223     Process(2, 115, 5),
224     Process(3, 130, 30),
225     Process(4, 100, 9),
226     Process(5, 120, 27),
227     Process(6, 108, 115),
228     Process(7, 136, 7),
229     Process(8, 100, 2),
230     Process(9, 110, 18),
231     Process(10, 125, 34),
232 ]
233 """
234 processes = [
235     Process(1, 120, 8),
236     Process(2, 115, 12),
237     Process(3, 130, 18),
238     Process(4, 100, 19),
239     Process(5, 120, 27),
240     Process(6, 108, 38),
241     Process(7, 136, 50),
242     Process(8, 100, 31),
243     Process(9, 110, 18),
244     Process(10, 125, 14),
245 ]
246 """
247 run_cfs(processes)
248 processes = [
249     Process(1, 120, 8),
250     Process(2, 115, 5),
251     Process(3, 130, 30),
252     Process(4, 100, 9),
253     Process(5, 120, 27),
254     Process(6, 108, 115),
255     Process(7, 136, 7),
256     Process(8, 100, 2),
257     Process(9, 110, 18),
258     Process(10, 125, 34),
259 ]
260 """

```

```

261 processes = [
262     Process(1, 120, 8),
263     Process(2, 115, 12),
264     Process(3, 130, 18),
265     Process(4, 100, 19),
266     Process(5, 120, 27),
267     Process(6, 108, 38),
268     Process(7, 136, 50),
269     Process(8, 100, 31),
270     Process(9, 110, 18),
271     Process(10, 125, 14),
272 ]
273 """
274 run_round_robin(processes)
275 reset_process(processes)
276 run_modified_round_robin(processes)

```

Listing 1: CFS, Classical RR and Modified RR Implementation

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 14 16:39:54 2025
4
5 @author: check
6 """
7 from collections import deque
8 class Process:
9     def __init__(self, pid, burst_time):
10         self.pid = pid
11         self.burst_time = burst_time
12         self.remaining_time = burst_time
13         self.predicted_burst = 0 # For burst prediction
14         self.vruntime = 0
15         self.start_time = None
16         self.end_time = None
17
18     def reset(self):
19         self.remaining_time = self.burst_time
20         self.start_time = None
21         self.end_time = None
22
23     def __lt__(self, other):
24         return self.predicted_burst < other.predicted_burst
25
26 def sjf_with_prediction(process_list, alpha):
27     time = 0
28     completed = []
29     gantt_chart = []
30     ready_queue = deque(process_list)

```

```

31     track=0
32     for process in ready_queue:
33         if track == 0:
34             process.predicted_burst = 5
35         else:
36             process.predicted_burst = round(alpha * ready_queue
37                 ↪ [track-1].burst_time + (1 - alpha) *
38                 ↪ ready_queue[track-1].predicted_burst)
39             track += 1
40     updated_queue = deque(sorted(ready_queue, key=lambda p: p.
41                 ↪ predicted_burst))
42     while updated_queue:
43         current = updated_queue.popleft()
44
45         current.start_time = time
46         current.end_time = time + current.burst_time
47         gantt_chart.append((current.pid, str(current.start_time
48                         ↪ )+'--'+str(current.end_time)))
49
50         time += current.burst_time
51         completed.append(current)
52
53     print("\nSJF Gantt Chart :")
54     for pid, time in gantt_chart:
55         print(f"  PID {pid:>2} | {time:>3} ms")
56
57     # Metrics
58     total_turnaround = 0;
59     total_waiting = 0;
60     print("\nPerformance Metrics of CFS:")
61     print(f"{'PID':<5}{'Start':<8}{'End':<8}{'Burst':<8}{'
62         ↪ Predicted_Burst':<18}{'Waiting':<10}{'Turnaround
63         ↪ ':<12}")
64     for p in sorted(completed, key=lambda x: x.pid):
65         turnaround = p.end_time
66         waiting = turnaround - p.burst_time
67         total_turnaround += turnaround;
68         total_waiting += waiting;
69         print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
70             ↪ burst_time:<8}{p.predicted_burst:<18}{waiting
71             ↪ :<10}{turnaround:<12}")
72
73     Average_TAT = total_turnaround/len(process_list)
74     Average_WT = total_waiting/len(process_list)
75     print(f"Average Turn Around Time: {round(Average_TAT,2)}")
76     print(f"Average Wait Time: {round(Average_WT,2)}")
77     print(f"Context Switches: {len(gantt_chart)-1}")
78
79
80 """

```

```
72 | processes = [
73 |     Process(1, 8),
74 |     Process(2, 5),
75 |     Process(3, 30),
76 |     Process(4, 9),
77 |     Process(5, 27),
78 |     Process(6, 115),
79 |     Process(7, 7),
80 |     Process(8, 2),
81 |     Process(9, 18),
82 |     Process(10, 34),
83 | ]
84 |
85 | """
86 | processes = [
87 |     Process(1, 8),
88 |     Process(2, 12),
89 |     Process(3, 18),
90 |     Process(4, 19),
91 |     Process(5, 27),
92 |     Process(6, 38),
93 |     Process(7, 50),
94 |     Process(8, 31),
95 |     Process(9, 18),
96 |     Process(10, 14),
97 | ]
98 | sjf_with_prediction(processes, alpha=0.8)
```

Listing 2: SJF Implementation

## References

- [1] A. Singh, P. Goyal, and S. Batra. An optimized round robin scheduling algorithm for CPU scheduling. *International Journal on Computer Science and Engineering*, 2(07):2383–2385, 2010. [https://www.researchgate.net/publication/50194216\\_An\\_Optimized\\_Round\\_Robin\\_Scheduling\\_Algorithm\\_for\\_CPU\\_Scheduling](https://www.researchgate.net/publication/50194216_An_Optimized_Round_Robin_Scheduling_Algorithm_for_CPU_Scheduling).
- [2] Justinien Bouron, Sébastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: FreeBSD ULE vs. Linux CFS. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*, pages 85–96. USENIX Association, USA, 2018.
- [3] Neetu Goel and R. B. Garg. A comparative study of CPU scheduling algorithms. *arXiv preprint arXiv:1307.4165*, 2013.
- [4] Shahad M. Ali, Ammar H. Gheni, and Nawar H. Zeki. A review on the CPU scheduling algorithms: Comparative study. *International Journal of Computer Science & Network Security*, 21(1):19–26, 2021.
- [5] Scheduling in Linux by Mary Anitha Rajam <https://ebooks.inflibnet.ac.in/csp3/chapter/scheduling-in-linux/>
- [6] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*, 10th Edition. Chapter 5: CPU Scheduling. Wiley, 2018.