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

Sayan Samanta

CPE-534
Research Project Summary

Contents

List of Figures

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 performace 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.

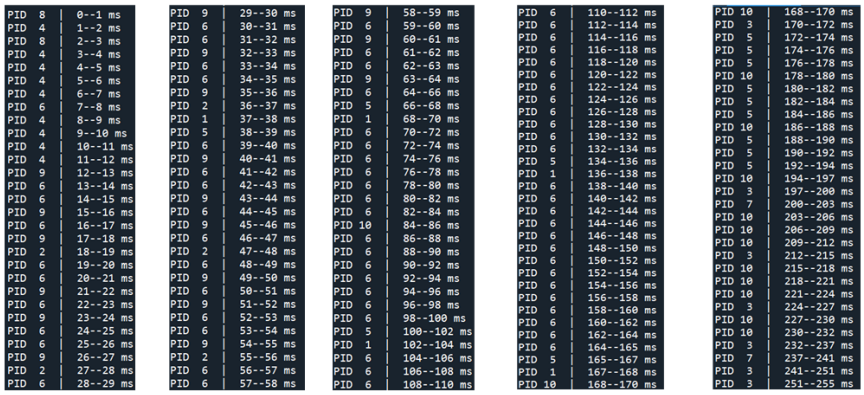

Figure 1: Gantt Chart for CFS

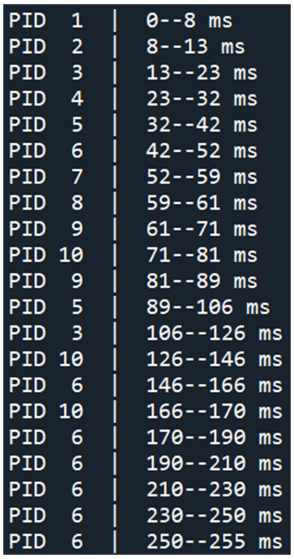

Figure 2: Gantt Chart for Modified RR

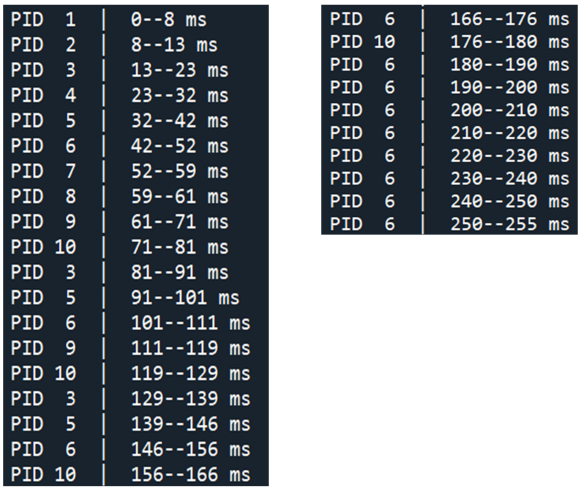

Figure 3: Gantt Chart for Classical RR

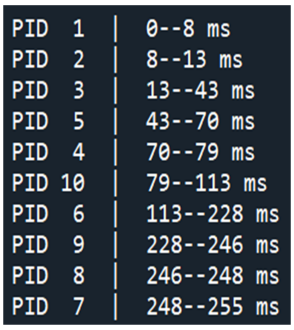

Figure 4: Gantt Chart for SJF

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.

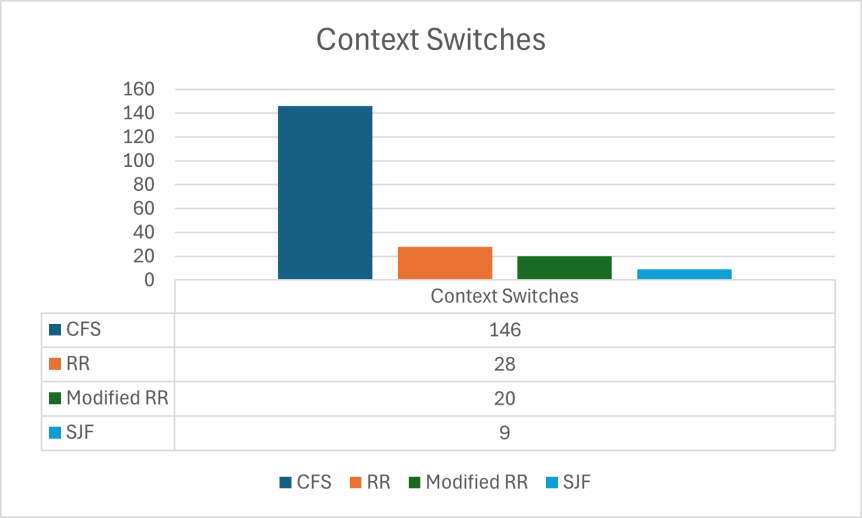

Figure 5: Context Switches for Testcase-1

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.

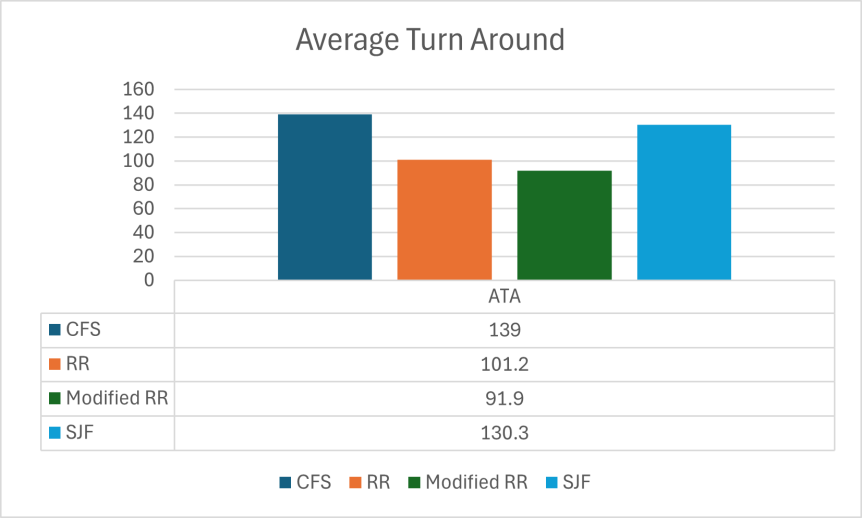

Figure 6: ATA for Testcase-1

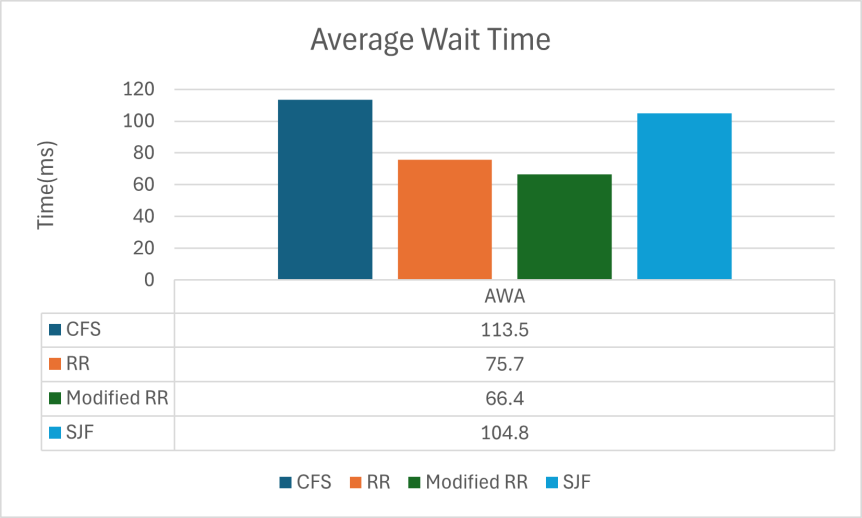

Figure 7: AWT for Testcase-1

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.

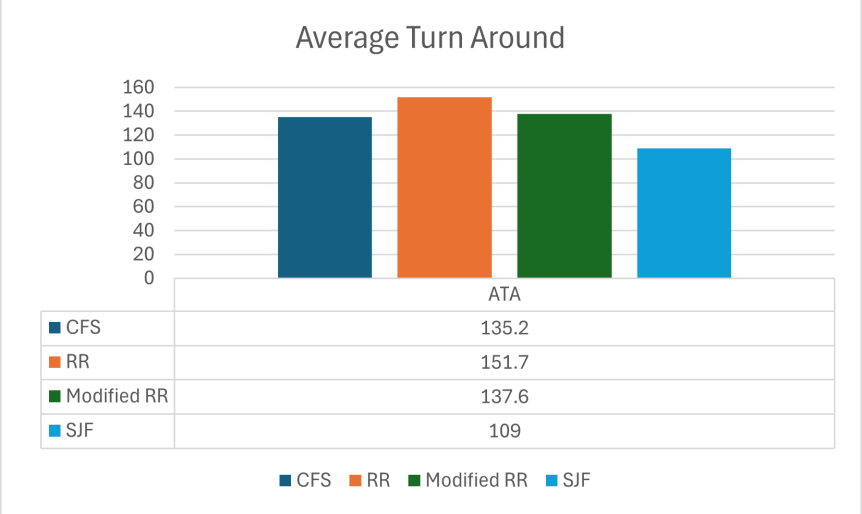

Figure 8: ATA for Testcase-2

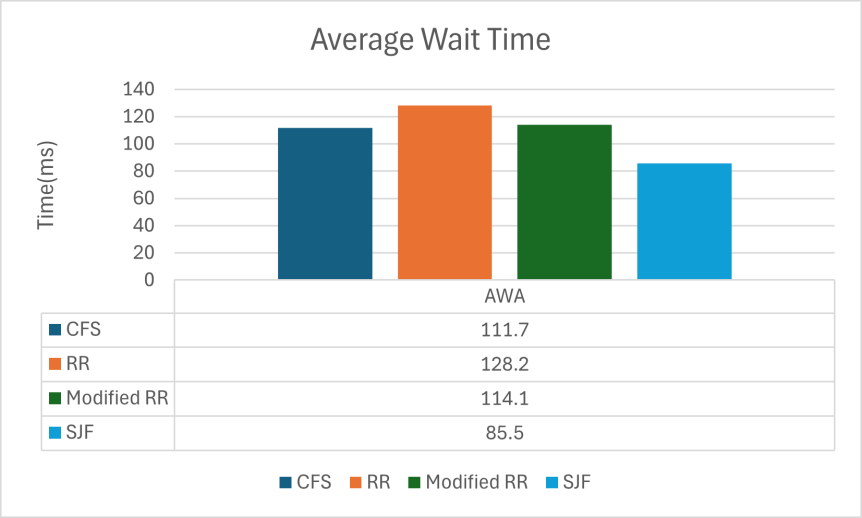

Figure 9: AWT for Testcase-2

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.

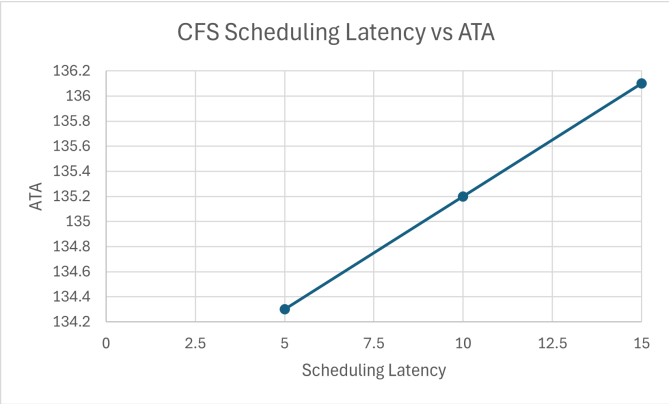

Figure 10: CFS ATA Dependency on Scheduling Latency

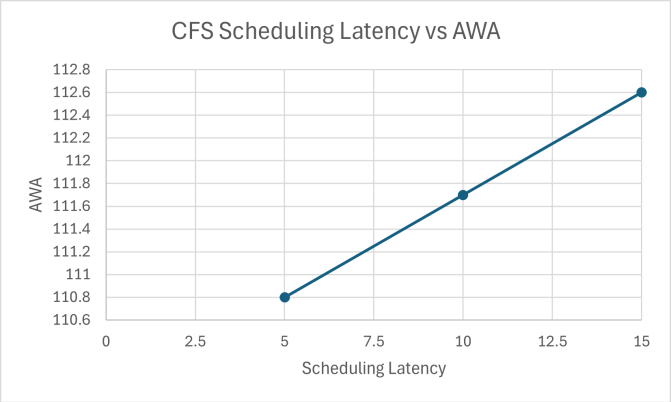

Figure 11: CFS AWT Dependency on Scheduling Latency

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

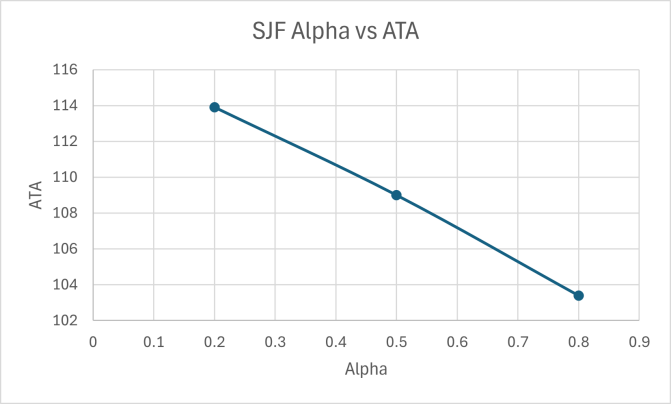

Figure 12: SJF ATA Dependency on Alpha

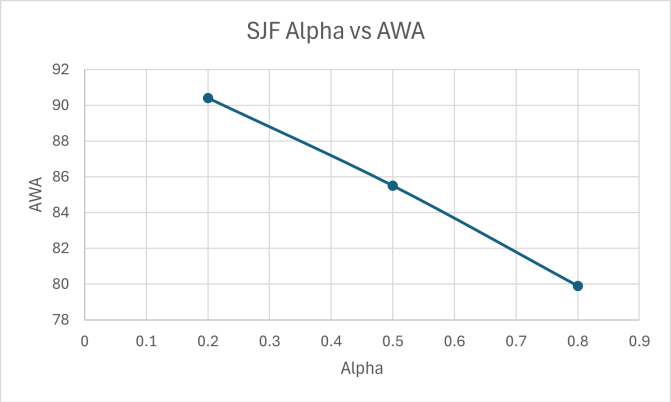

Figure 13: SJF AWT Dependency on Alpha

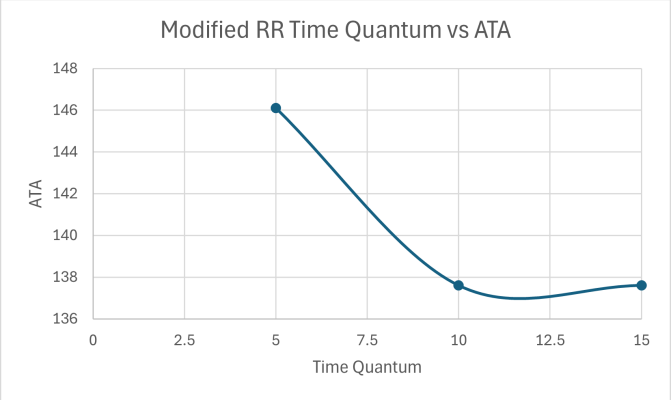

Figure 14: Modified RR ATA Dependency on Initial Time quantum K

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

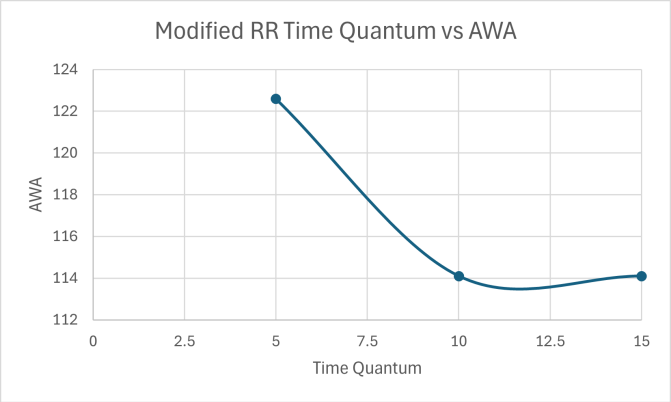

Figure 15: Modified RR AWT Dependency on Initial Time quantum K

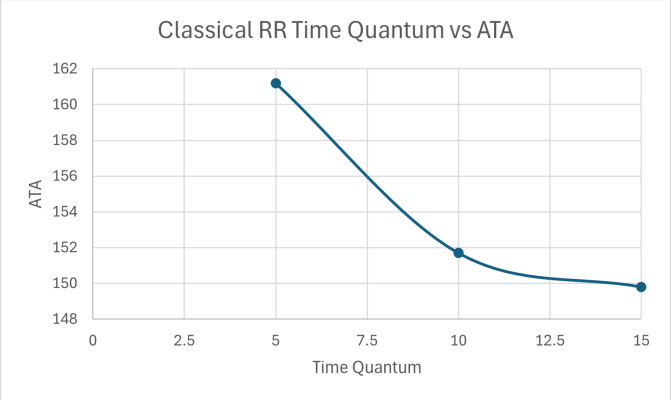

Figure 16: Classical RR ATA Dependency on Time quantum K

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

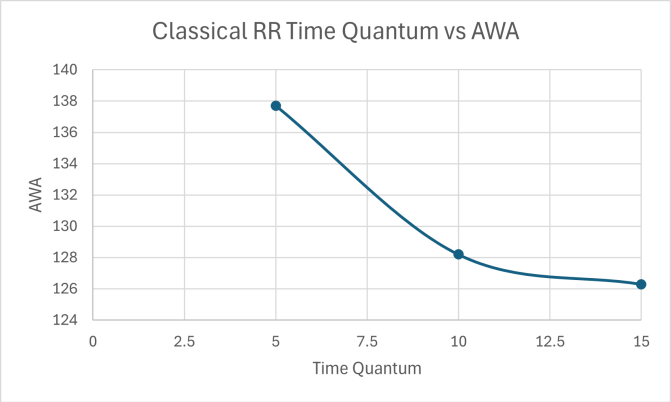

Figure 17: Classical RR AWT Dependency on Time quantum K

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:

```
Nice_value = 120 - priority
Weight = 1024 / (1.25 ^ nice_value)
Time_slice = max(round(scheduling_latency/len(ready_queue)),
    ↪ minimum_granularity)
exec_time = min(time_slice, process.remaining_time)
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

```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 12 15:02:33 2025

@author: check
"""
from collections import deque
import heapq
class Process:
    def __init__(self, pid, priority, burst_time):
        self.pid = pid
        self.priority = priority
        self.burst_time = burst_time
        self.remaining_time = burst_time
        self.nice = self.priority - 120
        self.weight = 1024 / (1.25 ** self.nice)
        self.vruntime =  (1024 / self.weight)
        self.start_time = None
        self.end_time = None

    def reset(self):
        self.remaining_time = self.burst_time
        self.vruntime = 0
        self.start_time = None
        self.end_time = None

    def __lt__(self, other):
        return self.vruntime < other.vruntime


def run_cfs(process_list):
```

```
    current_time = 0
    gantt_chart = []
    ready_queue = []
    completed = []

    num_process = len(process_list)
    heapq.heapify(process_list)
    ready_queue = process_list
    print("\nInitial Ready Queue Order:")
    #for p in ready_queue:
        #print(f"PID {p.pid} | Nice {p.nice} | Weight {round(p.
            ↪ weight, 2)} | vruntime {p.vruntime}")
    #ready_queue = sorted(process_list, key=lambda p:
        ↪ calculate_weight(calculate_nice(p.priority)), reverse
        ↪ =True)
    #ready_queue = heapq.heapify(process_list)
    print("\n\nstarting CFS...")
    while ready_queue:
        time_slice=max(round(10/len(ready_queue)),1)
        process = heapq.heappop(ready_queue)
        #nice_value = calculate_nice(process.priority)
        #weight = calculate_weight(nice_value)
        #print(f"\n[Time {current_time}] Selected PID {process.
            ↪ pid} with vruntime {round(process.vruntime, 6)}")
        exec_time = min(time_slice, process.remaining_time)
        process.remaining_time -= exec_time
        process.vruntime += exec_time * (1024 / process.weight)

            # Track start and finish time
        if process.start_time is None:
            process.start_time = current_time
        current_time += exec_time
        gantt_chart.append((str(current_time-exec_time)+'--'+
            ↪ str(current_time), process.pid))

        if process.remaining_time == 0:
            process.end_time = current_time
            completed.append(process)
        else:
            heapq.heappush(ready_queue, process)

    # Print Gantt Chart
    print("\nCFS Gantt Chart :")
    for time, pid in gantt_chart:
        print(f"  PID {pid:>2}  |  {time:>3} ms")

    # Metrics
    total_turnaround = 0;
    total_waiting = 0;
```

```
    print("\nPerformance Metrics of CFS:")
    print(f"{'PID':<5}{'Start':<8}{'End':<8}{'Burst':<8}{'
        ↪ Waiting':<10}{'Turnaround':<12}")
    for p in sorted(completed, key=lambda x: x.pid):
        turnaround = p.end_time
        waiting = turnaround - p.burst_time
        total_turnaround += turnaround;
        total_waiting += waiting;
        print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
            ↪ burst_time:<8}{waiting:<10}{turnaround:<12}")
    Average_TAT = total_turnaround/num_process
    Average_WT = total_waiting/num_process
    print(f"Average Turn Around Time: {round(Average_TAT,2)}")
    print(f"Average Wait Time: {round(Average_WT,2)}")
    print(f"Context Switches: {len(gantt_chart)-1}")

def run_round_robin(process_list, time_slice=10):
    current_time = 0
    gantt_chart = []
    queue = deque()
    completed = []

    ready_queue = sorted(process_list, key=lambda p: p.pid)
    queue = deque(ready_queue)

    print("\n\nstarting classical RR...")
    track=1
    while queue:
        track += 1
        process = queue.popleft()

        # Track first start time
        if process.start_time is None:
            process.start_time = current_time

        exec_time = min(time_slice, process.remaining_time)
        process.remaining_time -= exec_time
        current_time += exec_time
        gantt_chart.append((str(current_time-exec_time)+'--'+
            ↪ str(current_time), process.pid))
        if process.remaining_time == 0:
            process.end_time = current_time
            completed.append(process)
        else:
            queue.append(process)


    # Print Gantt Chart
    print("\n Classical RR Gantt Chart:")
```

```
    for time, pid in gantt_chart:
        print(f"  PID {pid:>2}  |  {time:>3} ms")

    # Metrics
    total_turnaround = 0;
    total_waiting = 0;
    print("\nPerformance Metrics of Classical RR:")
    print(f"{'PID':<5}{'Start':<8}{'End':<8}{'Burst':<8}{'
        ↪ Waiting':<10}{'Turnaround':<12}")
    for p in sorted(completed, key=lambda x: x.pid):
        turnaround = p.end_time
        waiting = turnaround - p.burst_time
        total_turnaround += turnaround;
        total_waiting += waiting;
        print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
            ↪ burst_time:<8}{waiting:<10}{turnaround:<12}")
    Average_TAT = total_turnaround/len(process_list)
    Average_WT = total_waiting/len(process_list)
    print(f"Average Turn Around Time: {round(Average_TAT,2)}")
    print(f"Average Wait Time: {round(Average_WT,2)}")
    print(f"Context Switches: {len(gantt_chart)-1}")

def run_modified_round_robin(process_list, time_slice=10):
    current_time = 0
    gantt_chart = []
    queue = deque()
    completed = []

    print("\n\nstarting modified RR...")
    track_loop = 0
    time_track = 1
    track=0
    ready_queue = sorted(process_list, key=lambda p: p.pid)
    queue = deque(ready_queue)
    #print(len(queue))
    n = len(ready_queue)
    while queue:
        #print("queue len:"+str(len(queue)))
        #print(track_loop)
        if (track_loop == n):
            updated_queue = sorted(queue, key=lambda p: p.
                ↪ remaining_time)
            queue = deque(updated_queue)
            n = len(queue)
            track_loop = 1
            time_track = 2
            track=1
        elif (track==1) & (track_loop == n):
```

```
            updated_queue = sorted(queue, key=lambda p: p.
                ↪ remaining_time)
            queue = deque(updated_queue)
            n = len(queue)
            track_loop = 1
            #print("i am here in 1")
        else:
            track_loop += 1
        process = queue.popleft()
        #print(process.pid, process.remaining_time, time_track)

        if process.start_time is None:
            process.start_time = current_time

        exec_time = min(time_track*time_slice, process.
            ↪ remaining_time)
        process.remaining_time -= exec_time
        current_time += exec_time
        gantt_chart.append((str(current_time-exec_time)+'--'+
            ↪ str(current_time), process.pid))
        if process.remaining_time == 0:
            process.end_time = current_time
            completed.append(process)
        else:
            queue.append(process)


    # Print Gantt Chart
    print("\n Modified RR Gantt Chart:")
    for time, pid in gantt_chart:
        print(f"  PID {pid:>2}  |  {time:>3} ms")

    # Metrics
    total_turnaround = 0;
    total_waiting = 0;
    print("\nPerformance Metrics of Modified RR:")
    print(f"{'PID':<5}{'Start':<8}{'End':<8}{'Burst':<8}{'
        ↪ Waiting':<10}{'Turnaround':<12}")
    for p in sorted(completed, key=lambda x: x.pid):
        turnaround = p.end_time
        waiting = turnaround - p.burst_time
        total_turnaround += turnaround;
        total_waiting += waiting;
        print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
            ↪ burst_time:<8}{waiting:<10}{turnaround:<12}")
    Average_TAT = total_turnaround/len(process_list)
    Average_WT = total_waiting/len(process_list)
    print(f"Average Turn Around Time: {round(Average_TAT,2)}")
```

```
    print(f"Average Wait Time: {round(Average_WT,2)}")
    print(f"Context Switches: {len(gantt_chart)-1}")

def reset_process(process_list):
    for item in process_list:
        item.reset()



processes = [
    Process(1, 120, 8),
    Process(2, 115, 5),
    Process(3, 130, 30),
    Process(4, 100, 9),
    Process(5, 120, 27),
    Process(6, 108, 115),
    Process(7, 136, 7),
    Process(8, 100, 2),
    Process(9, 110, 18),
    Process(10, 125, 34),
]
"""
processes = [
    Process(1, 120, 8),
    Process(2, 115, 12),
    Process(3, 130, 18),
    Process(4, 100, 19),
    Process(5, 120, 27),
    Process(6, 108, 38),
    Process(7, 136, 50),
    Process(8, 100, 31),
    Process(9, 110, 18),
    Process(10, 125, 14),
]
"""
run_cfs(processes)
processes = [
    Process(1, 120, 8),
    Process(2, 115, 5),
    Process(3, 130, 30),
    Process(4, 100, 9),
    Process(5, 120, 27),
    Process(6, 108, 115),
    Process(7, 136, 7),
    Process(8, 100, 2),
    Process(9, 110, 18),
    Process(10, 125, 34),
]
"""
```

```
processes = [
    Process(1, 120, 8),
    Process(2, 115, 12),
    Process(3, 130, 18),
    Process(4, 100, 19),
    Process(5, 120, 27),
    Process(6, 108, 38),
    Process(7, 136, 50),
    Process(8, 100, 31),
    Process(9, 110, 18),
    Process(10, 125, 14),
]
"""
run_round_robin(processes)
reset_process(processes)
run_modified_round_robin(processes)
```

Listing 1: CFS, Classical RR and Modified RR Implementation

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 14 16:39:54 2025

@author: check
"""
from collections import deque
class Process:
    def __init__(self, pid, burst_time):
        self.pid = pid
        self.burst_time = burst_time
        self.remaining_time = burst_time
        self.predicted_burst = 0  # For burst prediction
        self.vruntime = 0
        self.start_time = None
        self.end_time = None

    def reset(self):
        self.remaining_time = self.burst_time
        self.start_time = None
        self.end_time = None

    def __lt__(self, other):
        return self.predicted_burst < other.predicted_burst

def sjf_with_prediction(process_list, alpha):
    time = 0
    completed = []
    gantt_chart = []
    ready_queue = deque(process_list)
```

```
    track=0
    for process in ready_queue:
        if track == 0:
            process.predicted_burst = 5
        else:
            process.predicted_burst = round(alpha * ready_queue
                ↪ [track-1].burst_time + (1 - alpha) *
                ↪ ready_queue[track-1].predicted_burst)
        track += 1
    updated_queue = deque(sorted(ready_queue, key=lambda p: p.
        ↪ predicted_burst))
    while updated_queue:
        current = updated_queue.popleft()

        current.start_time = time
        current.end_time = time + current.burst_time
        gantt_chart.append((current.pid, str(current.start_time
            ↪ )+'--'+str(current.end_time)))

        time += current.burst_time
        completed.append(current)

    print("\nSJF Gantt Chart :")
    for pid, time in gantt_chart:
        print(f"  PID {pid:>2}  |  {time:>3} ms")

    # Metrics
    total_turnaround = 0;
    total_waiting = 0;
    print("\nPerformance Metrics of CFS:")
    print(f"{'PID':<5}{'Start':<8}{'End':<8}{'Burst':<8}{'
        ↪ Predicted_Burst':<18}{'Waiting':<10}{'Turnaround
        ↪ ':<12}")
    for p in sorted(completed, key=lambda x: x.pid):
        turnaround = p.end_time
        waiting = turnaround - p.burst_time
        total_turnaround += turnaround;
        total_waiting += waiting;
        print(f"{p.pid:<5}{p.start_time:<8}{p.end_time:<8}{p.
            ↪ burst_time:<8}{p.predicted_burst:<18}{waiting
            ↪ :<10}{turnaround:<12}")
    Average_TAT = total_turnaround/len(process_list)
    Average_WT = total_waiting/len(process_list)
    print(f"Average Turn Around Time: {round(Average_TAT,2)}")
    print(f"Average Wait Time: {round(Average_WT,2)}")
    print(f"Context Switches: {len(gantt_chart)-1}")


"""
```

```
processes = [
    Process(1, 8),
    Process(2, 5),
    Process(3, 30),
    Process(4, 9),
    Process(5, 27),
    Process(6, 115),
    Process(7, 7),
    Process(8, 2),
    Process(9, 18),
    Process(10, 34),
]
"""
processes = [
    Process(1, 8),
    Process(2, 12),
    Process(3, 18),
    Process(4, 19),
    Process(5, 27),
    Process(6, 38),
    Process(7, 50),
    Process(8, 31),
    Process(9, 18),
    Process(10, 14),
]
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`.

[2] Justinien Bouron, Sebastien 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.