Queing algorithm Race

Nils Wikström
Computer engineering, engineering
Högskolan väst
Trollhättan, Sweden
nils.wikstrom@student.hv.se

***Abstract*— Queuing algorithms play a crucial role in managing tasks prioritization and resource allocation in systems. This paper presents a comparative analysis of five different queuing algorithms namely Round Robin (RR), Least-Latency-Queuing (LLQ), Class-Based queuing (CBQ), Deficit Round Robin (DRR) and Stochastic Fairness Queuing (SFQ) implemented using the SimPy framework. This study focuses their ability to change depending on workloads size testing their scalability and adaptability. A series of tests called drag races were conducted with ranging workloads from 100,1000000 tasks, revealing distinctive performance trends for each algorithm. The result indicates that DRR exhibited consistent performance across most tests only to faulter with extreme workloads. LLQ and SFQ in terms of processing time showed consistent and reliable performance, with LLQ outperforming SFQ in terms of time. CBQ exhibited unpredictable behavior and struggled immensely with bigger workloads not being able to finish the last tests. RR showed linear time growth with the workload increase. These findings provide a valuable insight for selecting an appropriate algorithm for specific workloads scales.**

Keywords—Queueing algorithms, Scalability, adaptability,

PREFACE

Testing and coding done in part with Marcus Bohm

For more information on the code the pictures in the appendix is commented and the comments should be visible

I. INTRODUCTION

In the rapidly evolving landscape of computer systems, engineering, software development and networks architectures. The effective management of queues has become a necessary aspect of optimizing resource allocations and system performance enhancement. Queueing algorithms, a fundamental component in governing the order and priority of tasks and a seamless flow of data and equitable distribution of computational power. Selecting an appropriate algorithm will be paramount decision in one's project or systems.

This paper will undertake a comparative analysis of several queueing algorithms, aiming to determine which one is the best in the handling objects that will need to be queued and solved and see which one of the chosen algorithms handle scalability the best. And briefly explain how they are tested and how they work.

This comparative exploration is to offer an insight to which queueing algorithm that will be a guide to selecting appropriate algorithm for smaller, medium, and larger queues.

II. BACKGROUND

A comparison of different queueing algorithms is beneficial for several reasons, especially when considering their applications in different systems. Two reasons to compare different queues could be their scalability and how resource friendly they are.

A. *Scalability and adaptability*

Evaluating the scale of a queue is very important and picking an appropriate algorithm for the size is very important, understanding their performance under and increasing workload and change in a system dynamic could be fatal for that system if not an appropriate algorithm is chosen. Comparison will allow an appropriate algorithm for what they are going to be used for and which what scale they are still usable on.

Different queueing algorithms have varying adaptability levels to different workloads and comparing them helps in selecting an appropriate queue system if a change in the system where to happen or a change in the current conditions.

B. *Performance*

Different algorithms have different strengths and weaknesses and identifying these will be crucial depending on what they are used for. Some strengths and weaknesses could be response time, resource utilization and overall system efficiency.

The algorithm that will be compared are Round Robin (RR), Least-Work-Left (LLQ) queueing, Class-Based-queueing (CBQ), Deficit Round Robin (DRR) and Stochastic Fairness Queueing (SFQ). These algorithms are coded in the SimPy framework which is a process-based discrete event simulation and will be times using the python module Time.

- **Round Robin (RR):** RR queueing is a basic schedule that with cycles that allocates an equal time slice to tasks in a pre-determined order. This approach will ensure fairness in the resource distribution amongst the different tasks, this will prevent one single task to monopolize the process time and resources of the system. The simple design of RR promotes equal resource sharing and may lead to longer wait times for longer running tasks. Refer to appendix D for deeper understating of the RR code.

- **Least-Latency-left (LLQ):** LLQ is designed to prioritize certain tasks or workloads over others to reduce latency for critical data. LLQ aims to minimize the delay for high priority traffic to

ensure it gets processed quickly but do not specifically prioritize tasks based on the task's workload or their remaining process time. Refer to appendix C for deeper understating LLQ of the LLQ code.

- **Class-Based Queueing (CBQ):** CBQ is a dynamic queueing algorithm with versatile attributes that categorizes tasks into distinct classes with predefined attributes. This approach enables the allocations of resources based on the specific requirements of each task with that class assigned class, refer to appendix A for deeper understating CBQ of the CBQ code.

- **Deficit Round Robin (DRR):** DRR represents an extension of the classic algorithm Round Robin mentioned earlier and is mostly used in network traffic. DRR makes it so equitable resource allocations among multiple tasks by allowing temporary increase in consumption of resources, when necessary, provided that a compensation for that usage later. This will make it so its bursts of elevated resource demands but with the compensation it still maintaining the fairness. See to appendix B for deeper understating DRR of the DRR code.

- **Stochastic Fairness Queueing (SFQ):** SFQ is a probabilistic queueing algorithm that main concern is making sure a fair bandwidth allocation with different flows in a network. Using probabilistic methods SFQ prevents any single flow of tasks to dominate the ques resources, encouraging equitable distribution and prevent slow flowing congestion. Refer to appendix E for deeper understating SFQ of the SFQ code.

These five queueing algorithms will be drag raced against each other with different workloads to see how they adapt to scalability and adaptation changes.

A. SimPy and Statistics

The SimPy framework [1] is a process-based discrete event simulation that is based on standard Python.
The statistic module provides functionality for calculating mathematical statistics and is used for calculating the median for this project.

III. DRAG RACE

To see what algorithm the best in a race on different scales is they will be run 10 times on different number of workloads, seeing which one is the better on. The different workloads are 100, 1000,10000 ,100000 ,1000000 tasks to see how they will handle such different workloads.
Looking at these 10 runs, the median time will be used as benchmark for these algorithms. The python module Time and Statistics will be used for timing the runs and statistics for calculating the median times.

A. The workload

The algorithms are going to get different workloads as test starting at 100 tasks and then incrementing that by times 10 every iteration. The tasks are randomly created with a python list comprehension as shown in appendix F more that generates a list of task in the format of ""task 1"," task 2"" and a priority value that will cycle between 10 and 1.

```
times = []
    for i in range(1, 11):
        start_time = time.time()   # Record the
start time

        env = simpy.Environment()

        # Generating tasks with priorities based
on the order
        tasks = [(f"Task {j+1}", 10 - j % 10,
j+1) for j in range(task_count)]

        # Run the LLQ scheduler
        llq_scheduler(env, tasks)

        # Run the simulation
        env.run()

        end_time = time.time()  # Record the end
time
        execution_time = end_time - start_time

        times.append(execution_time)

        print(f"Execution time for {task_count}
tasks: {execution_time} seconds")

    print(times)
    median = statistics.median(times)
    print(median, "medians")
```

This code uses the modules Time, simpy and statistics to give the queuing algorithms their tasks, this code will run 10 times and save the run times in a list that will calculate the median time. Just this snippet is for LLQ but this is the same for all the tests and should give an unbiased test of all the algorithms because its random.

A Did not finish (DNF) system is put into place where if a run took too long, 170 seconds is the limit then it will count as a DNF.

B. Future endevor

Further research and testing could be combining these algorithms into hybrid algorithms, combining the strengths of each algorithm and see how it would fare against these workloads.

Even further testing could be to implement these into a real-life scenario to see if they work as well there as in these simulations.

Multithreading or accelerating the system where these algorithms are tested on could improve processing times and improve their adaptability and improve these results.

IV. RESULT

The testes are performed on a computer with an AMD Ryzen 7 7800x3D. The runs are not accelerated or multithreaded as this would be continuation of this comparison and could be a future endeavor. Base clock for the 7 7800X3D 100 MHz and a frequency of 4,2 GHz.

A. *100 task runs.*

The first test will be using 100 tasks as the workload and as figure 1 shows, that quiet evidently that DRR has the fastest execution time and RR having the longest, this is showing that DRR performs the most efficiently among these algorithms demonstrating its effectiveness on smaller workloads and varying resource demands. SFQ and LLQ also exhibit great results, showing a capability to manage tasks and resources efficiently. CBQ still performs well while adequately, shows longer execution times compared to the former and being somewhat less efficient as well. RR being the slowest of these five here showing that RR is less optimal for managing tasks with varying resource demands.

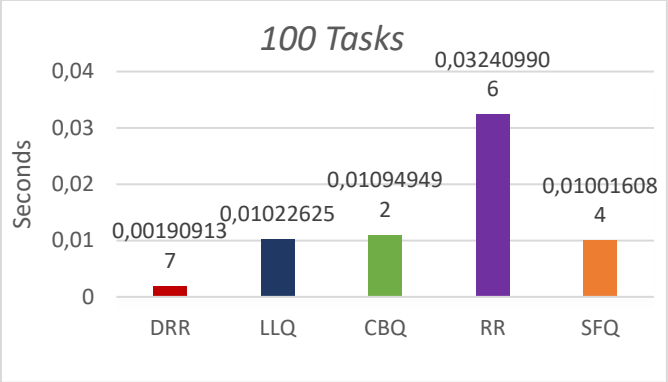

Figure 1. The algorithm performance for a hundred tasks, measured in seconds.

B. *1000 Task runs*

The second race will be using 1000 tasks as the workload and as figure 2 shows that the execution times are higher but not by a lot, RR is still the slowest and DRR has the fastest execution time and can handle this change in workload just fine. LLQ, SFQ are similar from the previous workload. CBQ having the biggest change in execution time.

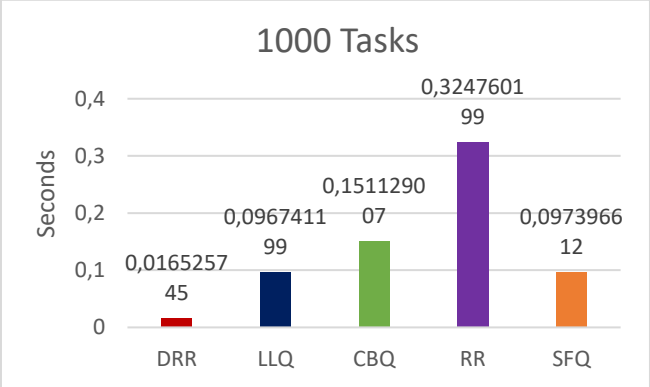

Figure 2. The algorithm performance for 1000 tasks, measured in seconds.

C. *10 000 Task run*

Third race has a significantly higher workload of 10 000 tasks. Displayed in figure 3 a surprising event occurs which is that CBQ 51,44 times slower this time around. This could be happening because how CBQ allocates priorities and as the workload grows this algorithm will struggle as shown.

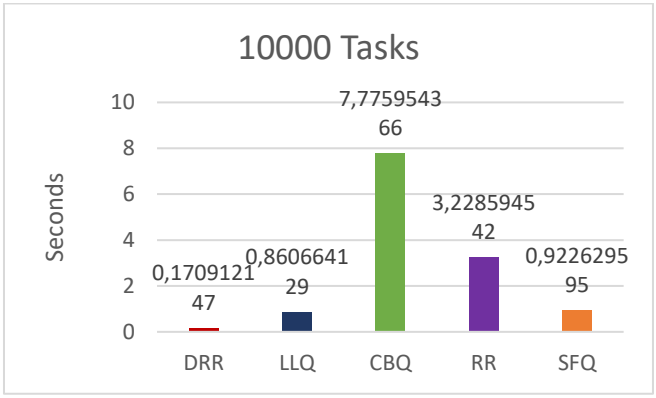

Figure 3. The algorithm performance for 10000 tasks, measured in seconds.

D. *100 000 Task run*

As shown in figure 4 a DNF has occurred, CBQ did not meet the requirement for this round taking more than 170 seconds for one run. DRR I still the best queueing system having a significant lead over the others, but LLQ and SFQ are still resource efficient and adapts to the scalability very well. Studying figure 4 DRR is still the best at resource effectiveness and still the best at handling the scalability. Seeing how CBQ performed in figure 3 this DNF could have been predicted and not suitable for this workload scale.

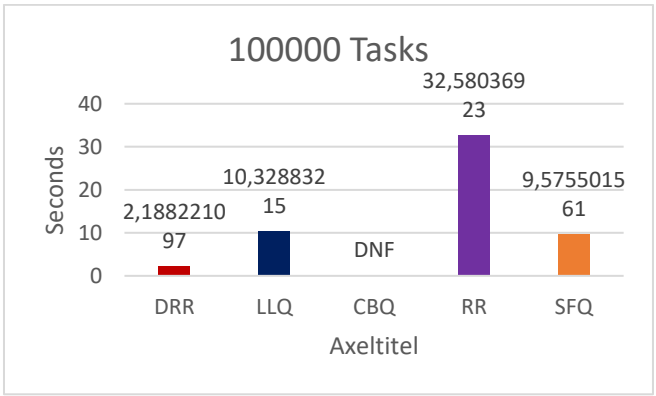

Figure 4. The algorithms performance for 100 000 tasks, measured in seconds.

E. *1000000 Task run*

The final race will be the biggest workload, the resource management and priority will be the hardest for these algorithms to handle. As shown in figure 5 we have two DNF this time and DRR struggle a lot to handle this being 73,76 times slower and this is due to the nature of DRR, when the workload becomes large enough there will be a larger number of processes in the queue and that can lead to a greater amount of time spent switching. With this large of a workload a longer time process time will be a guarantee and LLQ and SFQ handheld this better.

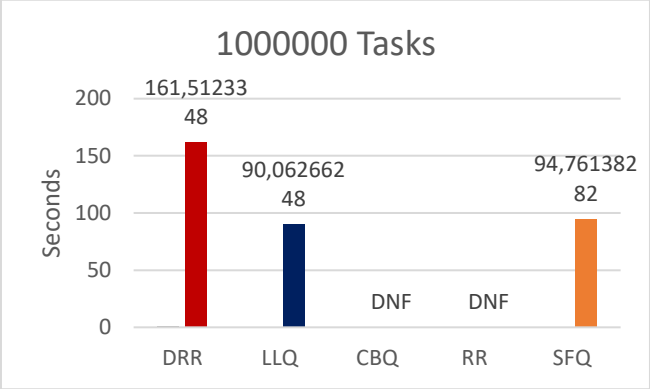

Figure 5. The algorithm performance for 1000000 tasks, measured in seconds.

C. *RR*

RR showcased a linear increase in processing time with an increase in workload having the worst processing time out of all the algorithms overall but was predicted due to the nature of the algorithm struggling with higher workloads, with this in mind RR should only be used in situations where there are smaller workloads, or the tasks are made especially for RR.

D. *LLQ & SFQ*

These two are compared together for their similar results in performance and characteristics. These two consistently handled and adapted to the scale and workload increase without a significant spike in processing time. LLQ in these tests showed lower processing times than SFQ. Both show promising results in these tests and can handle small to even large workloads.

E. *CBQ*

This algorithm displayed erratic behavior with performance varying significantly based on the workload provided and that is in its nature allocating priorities and getting overwhelmed with larger workloads, but in appropriate systems this algorithm will work fine in small to medium size systems.

F. *DRR*

DRR showed consistent performance across all the tests only to faulter at extreme workloads. But this makes sense looking at the nature of this queueing system, with the larger workload the queue will become too long, and the number of process switches will be to lead to greater processing time.

CONCLUSION

By looking at the figure 1-5 we can see how these different queueing algorithms handle different workloads and see how efficient they are during these runs.

LLQ and SFQ consistently demonstrated consistent and reliable performance across the various tests, with LLQ providing lower processing time compared to SFQ. DDR showed both great processing times and horrible processing times, encountering difficulties when workloads become too large. RR shows linear increased processing times with the increase in workload, this would suggest a limited ability to efficiently handle high demand and high workloads, meanwhile CBQ unpredictable behavior and struggle to handle larger workloads effectively.

REFERENCES

[1] SimPy, “SimPy Documentation,” version or publication date, https://simpy.readthedocs.io/en/latest/

APPENDICES

A.

```
def cbq_scheduler(env, task_queue):
    while True:
        # Check if there are tasks in the task queue
        if task_queue:
            # Sort tasks by priority (higher priority first)
            task_queue.sort(key=lambda x: x.priority, reverse=True)

            # Get the task with the highest priority
            task = task_queue.pop(0)

            # Print information about the processing task
            print(f"Processing '{task.name}' with priority {task.priority} at time {env.now}")

            # Simulate processing time based on priority (adjust as needed)
            # For higher priority tasks, simulate a shorter processing time (2 units), otherwise use 4 units.
            yield env.timeout(2 if task.priority >= 5 else 4)
        else:
            # Exit the loop when the task queue is empty
            break
```

B.

```
def deficit_round_robin_scheduler(env, tasks, time_slice):
    # Create a SimPy store to represent the ready queue
    ready_queue = simpy.Store(env)

    # Create a dictionary to track the deficits of tasks
    deficits = {}

    # Populate the ready queue with tasks and their burst times
    for task_name, burst_time in tasks:
        ready_queue.put((task_name, burst_time))

    # Infinite loop for task scheduling
    while True:
        try:
            # Get the task name and burst time from the ready queue
            task_name, burst_time = yield ready_queue.get()

            # If the task has a deficit, process it accordingly
            if task_name in deficits:
                deficit = deficits.pop(task_name)

                # If the deficit is less than or equal to the time slice, process the task with the deficit
                if deficit <= time_slice:
                    env.process(task(env, task_name, burst_time, deficit, ready_queue, deficits))
                # If the deficit is greater than the time slice, process the task with the time slice
                else:
                    env.process(task(env, task_name, burst_time, time_slice, ready_queue, deficits))
        # Handle SimPy Interrupt exceptions
        except simpy.Interrupt:
            pass
```

C.

```
def llq_scheduler(env, tasks):
    # Separate tasks into LLQs based on priority
    llqs = {}

    # Create LLQs for each priority level
    for task in tasks:
        priority = task.priority

        # Check if an LLQ for the priority level already exists
        if priority not in llqs:
            # If not, create a new SimPy store for the LLQ
            llqs[priority] = simpy.Store(env)

        # Put the task into the corresponding LLQ based on priority
        llqs[priority].put(task)

    # Process tasks in LLQ order
    for priority in sorted(llqs.keys(), reverse=True):
        # Iterate through LLQs in descending order of priority
        while not llqs[priority].items == []:
            # Get the next task from the LLQ
            task = llqs[priority].get()

            # Start a SimPy process for the task
            env.process(task(env, task.name, task.burst_time))
```

D.

```python
def round_robin_scheduler(env, tasks, time_slice):
    # Create a SimPy store to represent the ready queue
    ready_queue = simpy.Store(env)

    # Put tasks into the ready queue
    for task_name, burst_time in tasks:
        env.process(task(env, task_name, burst_time, time_slice, ready_

    # Infinite loop for round-robin scheduling
    while True:
        # Check if the ready queue is empty
        if not ready_queue.items:
            break  # Exit the loop if the ready queue is empty

        # Get the next task from the ready queue
        next_task = ready_queue.get()

        # Start a SimPy process for the next task
        env.process(next_task)
```

E.

```python
def task(env, name, burst_time, weight, ready_queue):
    # Simulate task execution for the specified burst time
    yield env.timeout(burst_time)

    # Print a message indicating that the task has finished
    print(f'{name} finished at time {env.now}')

def sfq_scheduler(env, tasks):
    # Create a SimPy store to represent the ready queue
    ready_queue = simpy.Store(env)

    # Assign random weights to tasks and add them to the ready queue
    for task_name, burst_time in tasks:
        weight = random.randint(1, 10)  # You can adjust the weight range as needed
        env.process(task(env, task_name, burst_time, weight, ready_queue))
```

F.

```python
def round_robin_test(task_count):
    times = []
    for i in range(1, 11):
        start_time = time.time()  # Record the start time
        env = simpy.Environment()
        # Generating tasks
        tasks = [(f"Task {j+1}", 10 - j % 10) for j in range(task_count)]
        # Time slice for the Round Robin algorithm
        time_slice = 2
        # Run the Round Robin scheduler
        round_robin_scheduler(env, tasks, time_slice)
        # Run the simulation
        env.run()
        end_time = time.time()  # Record the end time
        execution_time = end_time - start_time
        times.append(execution_time)
        print(f"Execution time for {task_count} tasks: {execution_time} seconds")
    print(times)
    median = statistics.median(times)
    print(median, "medians")
if __name__ == "__main__":
    round_robin_test(1000000)
```