Warren Brown

University of Brighton

Proccess sheduler Report

Contents

[Introduction 2](#_Toc5211133)

[Solution for low and mid priority 3](#_Toc5211134)

[Solution for long burst processes 3](#_Toc5211135)

[Solution design 4](#_Toc5211136)

[Solution demonstration 6](#_Toc5211137)

[Conclusion 9](#_Toc5211138)

# Introduction

A company is having issues with a multi-priority level queue process scheduler. The scheduler uses a Shortest Job First (SJF) strategy within each of its priority queues. The issue is that some low and mid priority process are not being executed when there is a heavy load of high priority processes, furthermore because it implements SJF in each queue, the processes that require longer bursts are being starved of CPU time when there is a heavy load of shorter burst processes.

Therefore, the two issues with this system are, the deterioration of low and mid priority processes, as well as CPU starvation for longer processes. In this report I aim to provide solutions to the above issues as well as providing a working program that implements my solution.

# Solution for low and mid priority

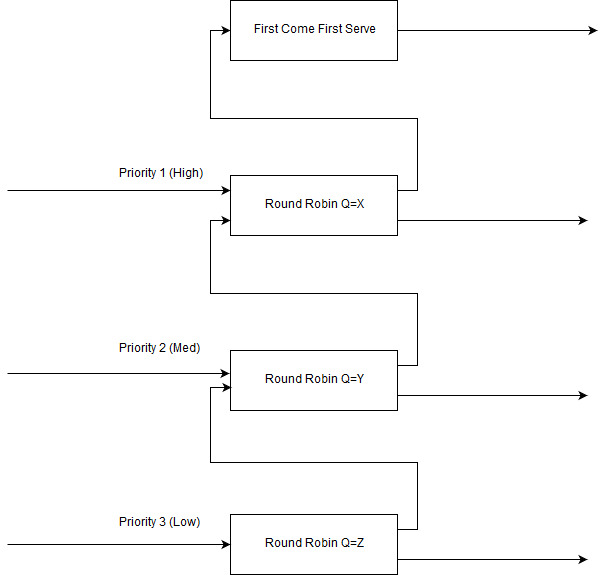
The solution I am proposing to tackle the low and mid priority issue, is an aging function. The aging function will over time increase the priority of a process that has been waiting for X amount of time to run. This would then mean that low and mid priority process would become mid and high priority processes respectfully. Increasing the priority of these processes over time will mean that after X amount of time they are guaranteed to be executed.

# Solution for long burst processes

The solution I want to implement for long processes is a Round Robin algorithm (RR). The RR will have a set Quantum time (Q), it will then run all the processes in this queue for Q time. Therefore, no matter the length of the required process burst, all processes will have an equal amount of CPU time, and so, longer processes will not be starved of CPU time.

# Solution design

The overall solution that I want to implement is a combination of a RR algorithm that will only allow the program to burst Y amount of times. After the Y bursts if the process is still not complete then the priority will be increased. Increasing the priority is the aging function that I will implement. These two methods together will make sure lower priority or processes that require a longer burst time will all be executed not matter what. Figure 1 is an illustration of how the scheduler will work. It is aimed at using four CPU threads to execute the processes in parallel.



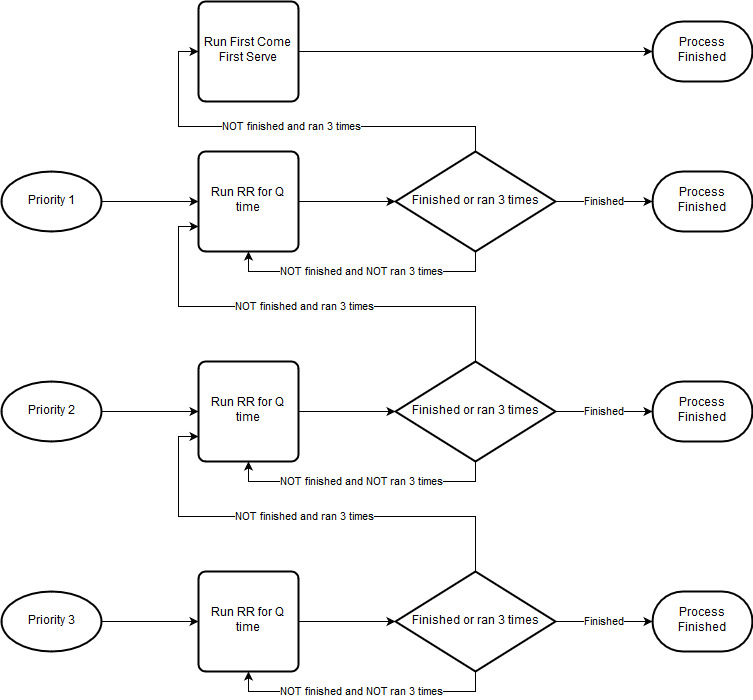
[Figure 1: Scheduler design]

Illustrated bellow in Figure 2 is a high-level flowchart detailing how the scheduler is going to work. My plan is to read in the csv file, then for each row in the csv create a process object that hold all the values needed for the rest of the program such as it’s priority, arrival time and burst time. Finally, sending all the process objects as a list to the scheduler.



[Figure 2: High-level flowchart design]

The flowchart in Figure 3 is the design for the scheduler. All processes come in and then are sent to the corresponding round robin (RR) scheduler. Form here they are ran for Q time, then the scheduler checks to see if the process is finished or if it has been run 3 times, if it has finished then the process is complete. If it has not finished and has not ran for 3 times then the process is added back into the RR and ran again with the other processes that have been loaded in, otherwise the processes priority is increased, and it is sent to the RR that corresponds with its new priority, with the exception of processes that already have a priority of 1. The processes with the priority of one are sent to a First Come First Server scheduler and the remainder of the burst time is executed there.



[Figure 3: Scheduler flowchart design]

# Solution demonstration

The demonstration of the solution is very simple. I ran the project in eclipse, the csv file that it reads from is in the project to make it as simple as possible and reduce the chance of errors with regards to finding the file.

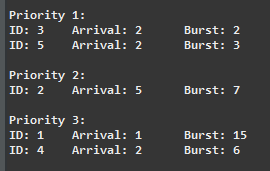
With the three priority queues I have assigned Q to be 5, 4, 3 for priority 1, 2, 3 respectively. This then means that the higher priority tasks get more CPU time and the high priority processes that have longer bursts get more CPU time than those of lower priorities.

Figure 4 shows an incredibly simple screen shot of the CSV file that I have created to demonstrate the program, the columns are ID, arrival time, burst time, priority, left to right. One thing to note is the first process in the list with the ID of 1, It has a longer burst time then rest as well as a low priority of 3. The solution that I have created will not ignore this process and will fix the issue of CPU starvation with low priority and long burst time processes.



[Figure 4: CSV data example]

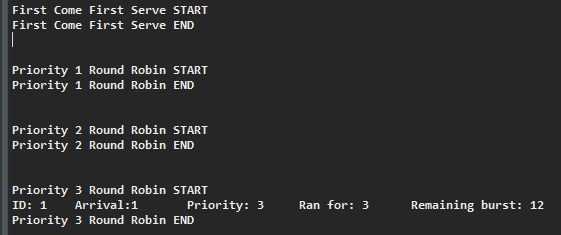
The first thing that happens when the program is ran is it reads in the CSV file and then splits the processes into three Lists as seen in figure 5. All the processes have been read in correctly and this can be double checked by looking at the table in Figure 4 above.



[Figure 5: Initial load]

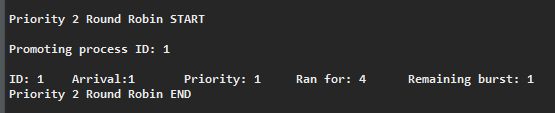
The first cycle of the queue solutions loads in the process with ID, 1. This is correct as this is the only process that has an arrival time of 1. Note how the process is displayed in Figure 6. I have listed out the ‘ID’, ‘Arrival’, ‘Priority’, ‘Ran for’ (The amount of time the process ran this cycle) and lastly, the remaining burst. As we can see in Figure 6, the process ran for 3 and therefore has the expected remaining burst time of 12, as it started with a burst of 15 (Figure 6).

Also take note of the First Come First Serve (FCFS) queue in Figure 6. This is a ‘net’ as to say, that catches all processes that are not able to run within the three Round Robin queues. The FCFS will run a process for its remaining burst time no matter how long it is. This is with the assumption that the FCFS has in theory a higher priority then the priority 1 queue, as it is from this queue that the processes are received for the FCFS. Hence, it is essential that these processes need to be completed as soon as possible. The solution is made in a way that should not allow many processes to get into the FCFS, only in extreme circumstances will a process ever make it to the FCFS. For example, if a process has a high priority and a long burst time then it will make it to the queue or if a process of any priority has an extremely long burst rate.



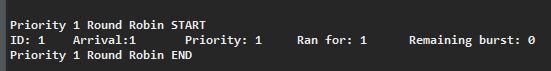
[Figure 6: First cycle]

The after the program has ran a couple times processes that have ran 3 times will start to be promoted as seen in Figure 7. The process with ID, 1 has been promoted to priority 1 from priority 2 illustrated by the printed output.



[Figure 7: Process promotion]

In Figure 8 we can see the process with ID 1, running its final burst. The purpose of the promoting or ‘aging’ of a process is to make sure that longer processes with lower priorities are still able to get the CPU time that they need to run. Through promotion the processes are guaranteed a chance to be ran whereas before they would have been starved of CPU time.



[Figure 8: Promotion proof]

# Conclusion

All in all, I think that I have demonstrated that I have been able to implement a solution to counter the starvation or low priority and long run times. Therefore, allowing these processes to be able to run and perform their functions instead of them being locked out from the CPU. I have demonstrated the aging processes that I wanted to implement along with the multi priority queue.