Mustafa Majeed
CSS 430
P2 Report

MFQS Algorithm Design

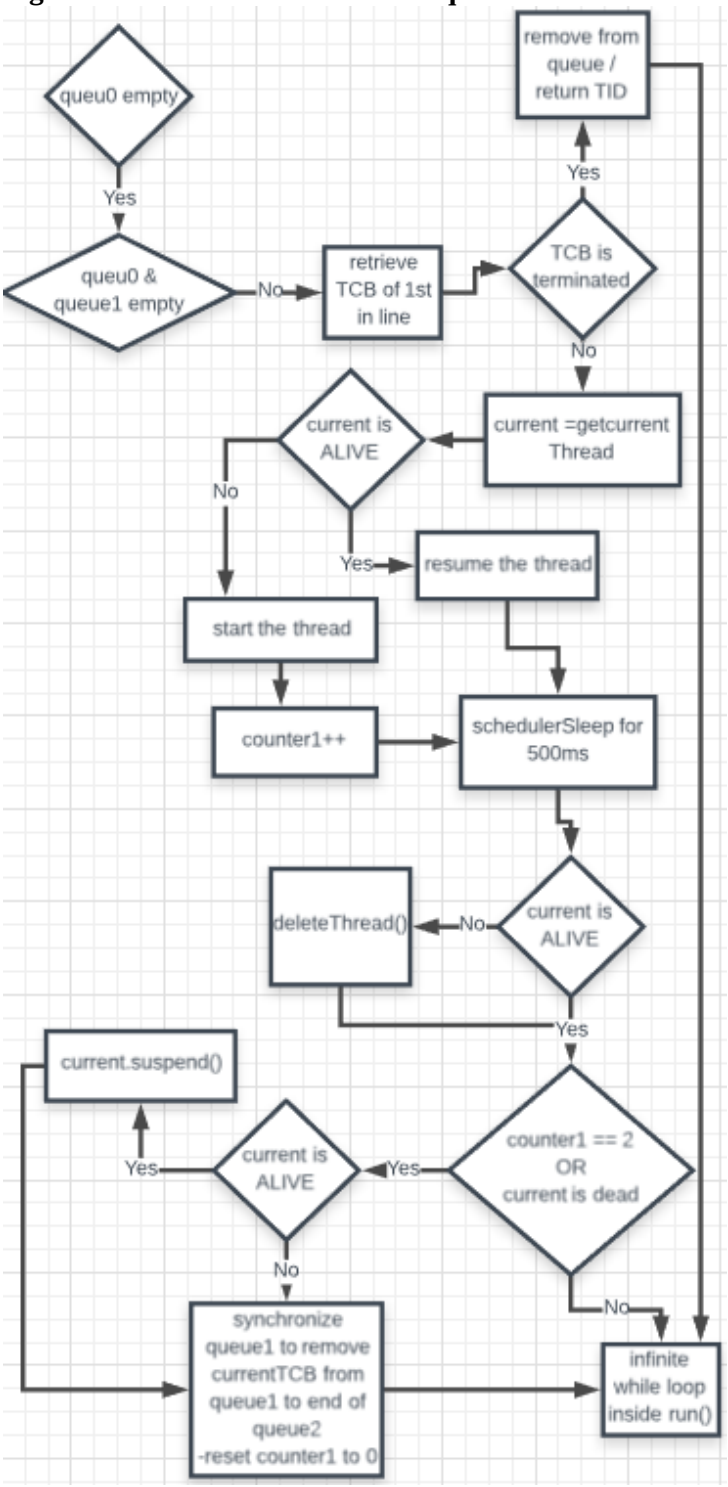
Multilevel feedback queue is designed to move threads through 3 different queues to allocate CPU time for live threads at a given time. It is designed to be pre-emptive which means that if at any moment, a new thread is created in queue0, it must get immediate action on the CPU to begin working.
Newly created threads get added to the end of queue0 which has a time quantum of 500ms. When the thread becomes head of the line, it is started by the OS Scheduler and gets immediate CPU time of 1 quantum. If the thread is still alive after this quantum, it is suspended and moved from the front of queue0 to the back of queue1.

Figure 1: Flowchart for threads in queue0

The time quantum of queue1 is double that of queue0, and in this Scheduler implementation, that is a quantum of 1000ms. Threads that are in queue1 will only get CPU time if queue0 is empty. At that point,

the thread at the head of queue1 will get immediate CPU time of 500ms before the pre-emptive check of queue0 again. If the thread is not finished executing at the end of the first 500ms before the pre-emptive check, it will be suspended until queue0 is empty again. Only then will it get the remaining 500ms of CPU time for queue1's quantum to be complete for that thread. If the thread is still alive after queue1's quantum has finished, it will be dequeued from queue1 and enqueued to the back of queue2.

Figure 2: Flowchart for threads in queue1

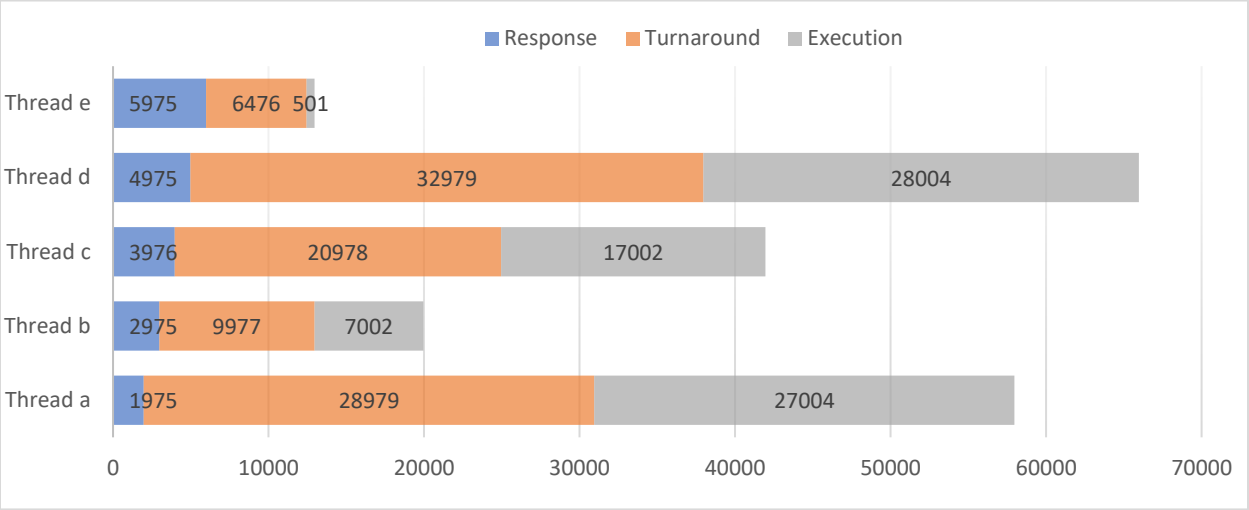


Queue2 has a time quantum of double queue1. This is where a thread that is still alive to this point will spend the remainder of its time running. The thread at the head of queue2 will continue to run in 500ms CPU bursts only if both queue0 and queue1 are empty and until the current thread uses up all of its allotted 2000ms of CPU time. If the thread is still alive at the end of the 2000ms quantum, it will be moved from the front of queue2 to the back of queue2 to awaits its next opportunity to use the CPU.
Queue0

Figure 3: Flowchart for threads in queue2

Analysis of Round Robin Scheduling vs MFQS Scheduling

Figure 4: Response, Turnaround and Execution time for Round Robin Scheduling Gantt Chart
(All threads arrive at the same time)



Project 2 allowed us the opportunity to observe how an OS Scheduler handles threads by scheduling them in a round robin fashion and by using a multilevel feedback queue system. Figure 4 above shows 5 separate threads that were created at the same time for execution using a round robin scheduler. Each process was allowed 1000ms of CPU time before either finishing and being removed from the queue or being removed and placed at the end of the line. This can be seen by the response time of each thread which are all roughly separated by 1000ms. This benefits the thread that gets created first, but its not too bad in terms of sharing the CPU if you do not create a lot of threads at the same time. The real benefit for this type of system is for quick terminating threads. The turnaround time for quick terminating threads can be very small as long as the amount of threads created at the same time is also small. The disadvantage for this system is for any thread that needs to use a lot of CPU and is created near the end of a batch of threads at the same time. This type of thread can see extremely long turnaround time as it will take a long time to get a response and take a long time to execute.

Figure 5: Response, Turnaround and Execution time for MFQS Scheduling Gantt Chart
(All threads arrive at the same time)

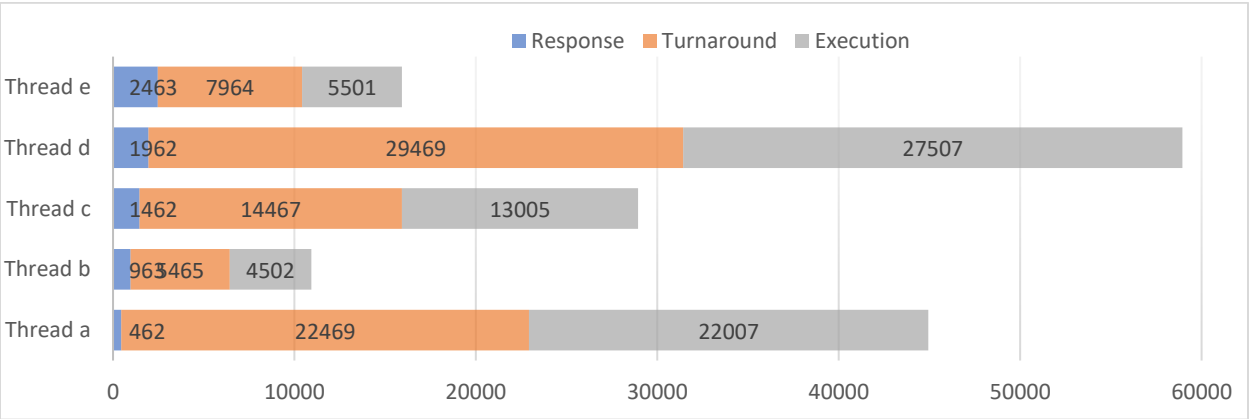


Figure 5 shows the same 5 threads being scheduled by a MFQS scheduler. You can see that the major benefit is the quick response time because newly created threads go into queue0 and get immediate access to the CPU for a quick burst. This is ideal for large amount of threads created at the same time. With that being said, all 5 threads still took a while to finish executing, but the gap between the shortest execution time and the longest was more similar than in Round Robin.

Figure 6: Comparison Table for Round Robin vs MFQS Scheduling

Thread name	CPU burst (ms)	RR Response (ms)	MFQS Response (ms)	RR Turnaround (ms)	MFQS Turnaround (ms)	RR Execution (ms)	MFQS Execution (ms)
Thread[a]	5000	1975	462	28979	22469	27004	22007
Thread[b]	1000	2975	963	9977	5465	7002	4502
Thread[c]	3000	3976	1462	20978	14467	17002	13005
Thread[d]	6000	4975	1962	32979	29469	28004	27507
Thread[e]	500	5975	2463	6476	7964	501	5501
AVG -----		3313.5	1219.5	16565.66	13306.5	13253	12087.83

Figure 6 shows a table that compares response, turnaround and executions times for both RR and MFQS. It is clear from the table that on average, MFQS is a better scheduling algorithm than round robin because all the average times are less for MFQS. Average execution time was the closest of the 3 comparisons, but MFQS still outperformed RR by more than a 1000ms. The biggest difference between the two is in average turnaround time where MFQS outperformed RR by more than 3000ms. That is good for a scheduling algorithm because it will allow the threads to get in and get out as quick as possible to get a quick start and perform all that they need to do.

One strange thing I did notice is that thread e, who's total runtime is supposed to be 500ms according to TestThread2.java, took 5000ms more to execute in MFQS than in RR despite getting a much faster response time. I know that is because the initial time at queue0 is only 500ms and therefore it was unable to finish in its entirety until it got ahold of the CPU again. But that was after it was forced to wait for its turn in queue1 and wait for the threads ahead to use up all their quantum.

What would happen if queue2 in MFQS was implemented in a FCFS without pre-emption?
FCFS stand for First Come First Serve. In general, this means that who ever is first in the queue gets to use the CPU until they finish executing. If that was the case, I don't think it would work for this project, because tid=0 is automatically created by the scheduler to run an infinite loop that never comes to completion and it will be first in line in queue2.
However, if we implement a FCFS to run for a 2000ms quantum before then checking queue0 and queue1 for more threads, the main aspects that will take a hit will be response time and turnaround time. This is because we will be forced to yield the CPU to the head of queue2 for 2 full seconds when they get their turn. This might speed up execution time for threads that make a living in queue2, but they might be doing a lot of waiting from initial creation to total completion.
Furthermore, if we implement a FCFS in queue2 for all of queue2 threads at a 2000ms quantum, it can potentially lock out any new threads from ever using the CPU until all threads in queue2 are complete. This means that all threads in queue0 will be locked out while threads in queue2 take turns 2000ms at a time using the CPU at a FCFS basis.