Explanation of SchedulerMQFS Design and Algorithm

NOTE: This code does not follow the posted coding guidelines. It adheres to the code style already established in the ThreadOS code base. The decision to not follow the posted guidelines was made on the principle that if there is an existing standard being followed then that standard, whatever it may be, should be respected and kept. It is my opinion that it is unreasonable to either edit existing code to make it conform to a new standard or (even worse) to have two different standards existing in the same base. Since “curly bracket on the same line” has been pre-implemented throughout the code we are working with, I am choosing to keep that format.

Scheduler (MQFS) differs from Scheduler (Round Robin) in that it maintains three queues instead of just one. The default time slice in this MQFS implementation is set to 500 ms, and the queues are leveled such that each runs a thread for a different number of time slices before switching to the next thread: queue 0 runs for 1 slice (500 ms); queue 1 runs for 2 slices (1000 ms); and queue 2 runs for 4 slices (2000 ms).

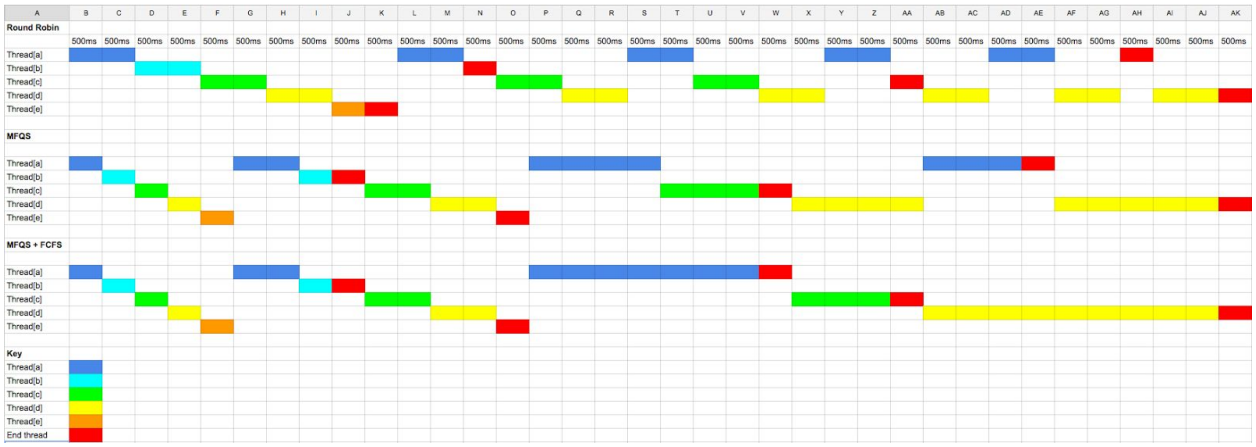
The algorithm works as follows. The constructor initializes an array of three queues to hold threads, and it initializes an array of three integers to keep track of the number of time slices executed by each queue. The run() method calls a helper method getActiveQueue() that begins with queue 0 and checks each queue in order and returns the first queue it finds that contains at least one thread. If no queues are found, the run method continues checking until there is a thread to run. Otherwise, the first thread of the returned queue is executed for one time slice, and the int array of time slices that corresponds to that queue is incremented. The importance of this incrementation is illustrated in what happens immediately next.

Checks are next performed to determine which queue is being processed. If the queue being processed is queue 0, then incidentally the time slice count doesn’t matter because items in queue 0 only execute for one time slice. So if the the current queue is queue 0, the current thread is removed from queue 0 after executing and placed into queue 1. However if the current queue being processed is queue 1, then the time slice count for that queue is inspected, and if the slice count is evenly divisible by 2, then it is known that the current thread in queue 1 has been processed for 2 slices, and it is removed from queue 1 and placed in queue 2. But if the slice count in queue 1 is not divisible by 2, then the current thread still has processing time allocated to it, and it will be allowed to run again before it is bumped to queue 2. Similarly, queue 2 operates in a fashion such as queue 1. However, queue 2’s threads are allowed to run for 4 time slices instead of 2, and when they have reached their limit, they are simply placed at the end of queue 2 again. Every time a thread is executed for one slice, the process of checking for the first queue containing a thread starts over, beginning with queue 0.

Gantt Charts Comparing Algorithms

NOTE: It is difficult to represent the termination of a thread in the Gantt chart. This chart uses a red square to indicate the termination of a thread. Please note that there is no wait time of 500 ms incurred for each termination, as could be interpreted (incorrectly) from this chart.

NOTE: Please see the attached file Gantt_charts.xlsx for a full-sized, human-readable chart.



Now stop squinting. The first group of threads shown has been executed with the Round Robin scheduler. The second group shown has been executed with the MFQS scheduler. The last group of threads shown in the Gantt chart has not been implemented, but is shown as the threads would theoretically execute with queue 2 being implemented with FCFS.

Below are the results from running Test2.

Results from running Test2

Round Robin	Response time	TAT	Execution time
Thread[a]	2008	29109	27101
Thread[b]	3014	10043	7029
Thread[c]	4019	21084	17065
Thread[d]	5023	33126	28103
Thread[e]	6029	6551	522
Average	**4018.6**	**19982.6**	**15964**
MFQS	**Response time**	**TAT**	**Execution time**
Thread[a]	500	24222	23722
Thread[b]	1006	5567	4561
Thread[c]	1511	16156	14645
Thread[d]	2017	31227	29210
Thread[e]	2522	8064	5542
Average	**1511.2**	**17047.2**	**15536**

The average response time was significantly better for MFQS than for Round Robin. You can see by both of the charts that fairness was increased for MFQS because each thread was initially given a shorter CPU burst before allowing all others to have a turn. As such each thread had a faster response time.

Turn around time was marginally faster as well for MFQS. The reason for this is somewhat subtle, but if you look closely at the gantt chart, you can count the number of times each thread was run for each algorithm. With the exception of threads [b] and [c], each MFQS thread ran fewer times than its RR counterpart, and [c] ran the same number of times. In summary this means that MFQS incurred fewer context switches, and thus delivered faster TATs. MFQS also had faster execution times, and the cause for this is the same as for the faster TATs.

Implementing part 2 with FCFS in Queue 2 would not have any effect on response times, as those are determined by Queue 0. However, if you look again at the gantt chart, it is apparent that Thread[a] would have a faster TAT and execution time, albeit at the expense of Thread[c] who would have to wait for [a] to finish (opposed to not waiting beforehand), and Thread[d] would not have been affected (in this scenario) because it would have finished last regardless. In summary, FCFS has a marked effect on fairness, although it could be debated on which algorithm is more “fair”. Of course, this is only one scenario out of nearly infinite possibilities, so it is impossible to say which is better for all situations.