Report Documentation for Program 2

Building files:
To build *Scheduler.java and Scheduler-MFQS.java* simply use the following command:
`javac *.java`

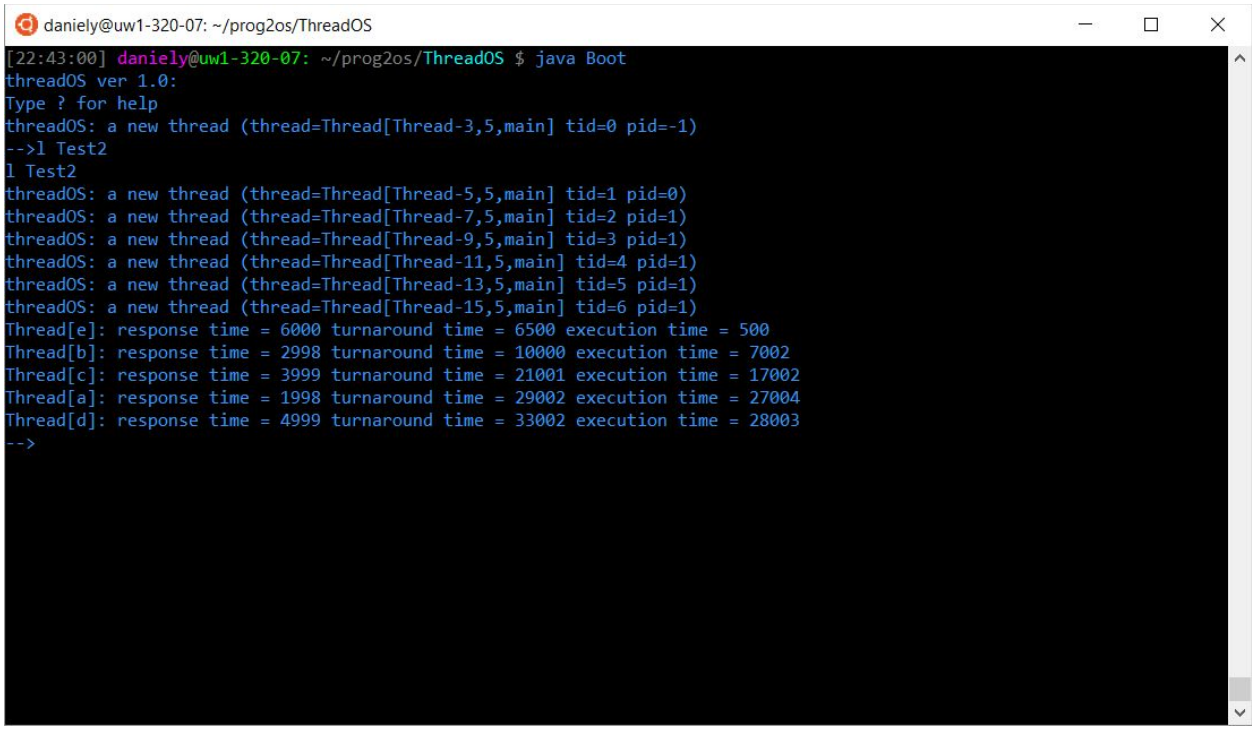
To run *Scheduler.java and Scheduler-MFQS.java* follow the listed steps

1. Import all ThreadOS necessary files into current directory
2. Use `java Boot` to load ThreadOS
3. Use `l Test2` to load Test2 in order to test the Response Time, Turn-around Time, and Execution times of *Scheduler.java* and *Scheduler-MFQS.java*

Program Purpose
Implement various Scheduler algorithms for ThreadOS. Where a scheduler is used to schedule a process for a thread and either preempt it (interrupt) or not for another process.

Round-Robin Scheduler
In the Round-Robin Scheduler, the given Program 2 description already gives the necessary changes in lines of code to change the queue from being disordered and interleaved into structured equal time-quantum slices. After the modifications for Round-Robin, the processes will now run in a Round-Robin sense where they are all given equal quantum time slices of 1000ms and run one after another.

Depiction of running Test 2 for Round-Robin in the Linux Labs (UW1-320-07.uwb.edu):

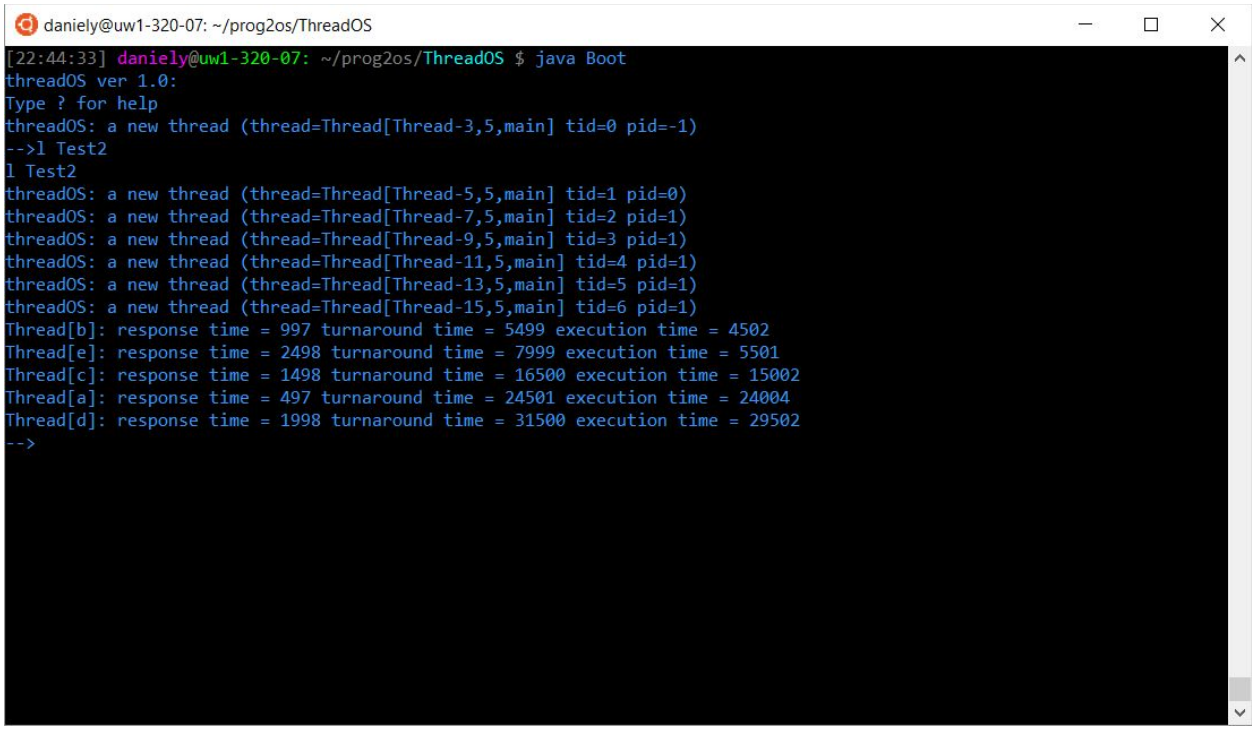


Multi Feedback Queue Scheduler (MFQS)
In the MFQS, we now have 3 different queues with different time quantums. Each queue is implemented in a Round-Robin approach, where every process is given an equal time quantum slice.

However, each queue (numbered 0 to 2) has different time quantums allotted for each process. If a process is unable to finish within queue 0, then it will be bumped down to queue 1, which has a large time quantum slice provided. Queue 0 gives time quantum slices of 500ms, queue 1 of 1000ms, and queue 2 of 2000ms.

The explicit functions that were modified comparative to the Round-Robin Scheduler included the *run(), getMyTCB(), schedulerSleep(), addThread(),* and constructors.

Depiction of running Test 2 for MFQS in the Linux Labs (UW1-320-07.uwb.edu):



Design and Algorithm for MFQS
For the MFQS alterations, I first added 3 new queues which were numbered to hold the different RR time quantum slices allotted. Constructors were made to initialize all 3 Vector queues instead of just a single one.

In the *getMyTCB()* method, the gotten TCB from a single queue was modified to be gotten from 3 queues instead, where the method iterates through all 3 queues and checks if the TCB is in that queue and thread.

In the *schedulerSleep()* method, the sleeping times varied for each queue because the given time quantum slices were different.Whenever queue 0 was not empty, the threads would be sleeped for 500ms because that is the allotted time quantum. If queue 0 was empty, then we check in queue 1 if it's empty. If queue 1 isn't empty we then sleep the threads for 1000ms as that is the equivalent time quantum allotted. If both queue 0 and queue 1 are empty, then we know it must be in queue 2 and sleep the thread for 2000ms, equal to the allotted time quantum slice.

In the *addThread()* method, a small change was made to simply add to queue 0 as that's where all new processes should be initially added.

In the *run()* method, heavy modifications were made to run the proper sleeps and checking for each queue in thread. The following list is performed inside a while loop:

1. If all queues are empty, then keep waiting until a process comes in.
2. If queue 0 is not empty then get the TCB for it and set the current TCB to the first element.
3. If the current TCB is terminated, remove it and set the current Thread to the removed thread ID.
4. As long as the current Thread is alive, let it run, otherwise start a new thread.
5. Sleep the thread by the time as specified in *schedulerSleep()*
6. Suspend the current Thread if it's not null and alive in a synchronized method. This is done as it means the current Thread hasn't finished in queue 0 and needs more time.
7. Remove the current TCB and add it to queue 1.
8. Repeat steps 2-6 for queue 1 instead. If the TCB isn't finished in queue 1, then remove it and add it to queue 2.
9. In queue 2 simply keep running the Thread until it terminates.

As such it's clear to see the algorithm of the MFQS in *run()* as it simply checks if the Thread has had enough time to terminate in its queue, if not then bump it to the next one. On queue 2, if the Thread hasn't terminated simply keep running it until it has.

Comparison of MFQS RR and RR

Here are Average Times for both MFQS and RR in Milliseconds for Response Time, Turnaround Time, and Execution Time for each Thread.

Round-Robin Times

Threads	**Response Time**	**Turnaround Time**	**Execution Time**
B	2999	7500.5	5251.25
E	6000.666667	4875.5	375
C	3999.666667	15751.25	12751.5

A	1998.666667	21752	20253
D	4999.666667	24752	21002.25

MFQS Times

Threads	**Response Time**	**Turnaround Time**	**Execution Time**
B	999	5501	4502
E	2499.666667	8000.666667	5501
C	1500	16502.66667	15002.66667
A	499	24503	24004
D	1999.666667	31502.33333	29502.66667

It's clear by comparing the times for MFQS and RR, that for MFQS, the **Response Time** for each **every single Thread** is **smaller**. The **Turnaround Time** for threads **B & E** are **smaller**, however for threads **C & A & D** the times are **greater**. The **Execution Time** for MFQS is **greater** than RR for **every single Thread** as well.

As such MFQS is best if you want a fast response time, and you want a fast turnaround for threads that are similar in type to B & E.

RR would be best for Execution Time as it supersedes MFQS for all threads, and better for turnaround for threads that are similar to type in C & A & D.

It's clear that neither algorithm is simply flat out better than the other. As such, determining which algorithm to use would depend on the context for the process in which either algorithm will be utilized.

Comparison to FCFS with no Preemption for Queue 2

If you ran *Test2.java* using FCFS with no preemption for Queue 2 rather than Round-Robin, then the response times for each element that needs Queue 2 will go up, as it has to wait for every preceding process to run before the current process can run. The execution time will be reduced to the same time as the burst time as non-preemptive FCFS gives the process the same amount of time to run and the process cannot be interrupted. The turnaround time may or may not be faster, because it will depend which process is run and the size of the process.

For processes that are very large, the turnaround time would then be large for any future processes and the preceding process (assuming arrival time is earlier than execution time). This

can also be a large issue as a large process can effectively ‘hog’ the queue and make all other processes not be able to run because the large process is currently in use.