Khuzaima Mushtaq
CSS 430

Project 2 – Report

Part 2: Multi-level Feedback Queue Scheduler

This section explains the Multi-Level Feedback Queue Scheduler design and algorithm. To start off, my scheduler has 3 queues as specified by the assignment, queue, queue1, and queue2. The assignment specifies that queue has a quantum of 500ms or timeSlice/2, queue1 has a quantum or 1000ms or timeSlice, and queue2 has a quantum of 2000 or timeSlice * 2. The assignment also specifies that queue1 will ONLY execute IF queue is empty, and Queue2 is ONLY execute IF queue and queue1 are empty. Furthermore, the assignment specifies that each 500ms (timeSlice/2) the scheduler must check if the any of the higher priority queues have any threads in them.

My implementation consists of 3 different cases. If queue has threads in it, If queue has no threads and queue1 has threads, and if queue and queue1 have no threads and queue2 has threads.

First Case: queue has threads

In this case I simple check if the thread is still alive and not set to be terminated and then execute it using thread.resume(). I use the schedulerSleep() method to wait 500 ms (timeSlice/2) and then check if the thread is still alive and non null. Then I remove it from queue and add it to queue1. This process repeats until there are no threads left in queue.

Second Case: If queue has no threads and queue1 has threads

Once queue has no threads my algorithm starts working on queue1. The important detail here is that threads in queue1 run for 1000ms (timeSlice) but we have to check every 500ms (timeSlice/2) to see if there are any new threads in queue. Also, if there is a new thread in queue it has to be executed first, and once we go back to executing threads on queue1 we have to resume from where we left off. So, we have 2 cases when we start to execute on queue1, First, queue1 was not interrupted, second, queue1 was interrupted. I check to see if queue1 was interrupted using a new Thread Q1 which is set to null to null if queue 1 was not interrupted, otherwise it hold the thread that needs to be resumed since it was interrupted.

Case 1: Was not interrupted

Here we basically do the same thing as the First Case, we check if the thread is still alive and resume it, then we wait 500ms with schedulerSleep() and check if there are new threads in queue, if there are we save the current thread in Q1 suspend current thread and push it to the back of the que and go back to queue to execute those threads first. If there are no new threads in queue, then we let the thread run for another 500 ms using schedulerSleep(), if the Thread is still alive we suspend it and push it to queue2.

Case2: Was Interrupted

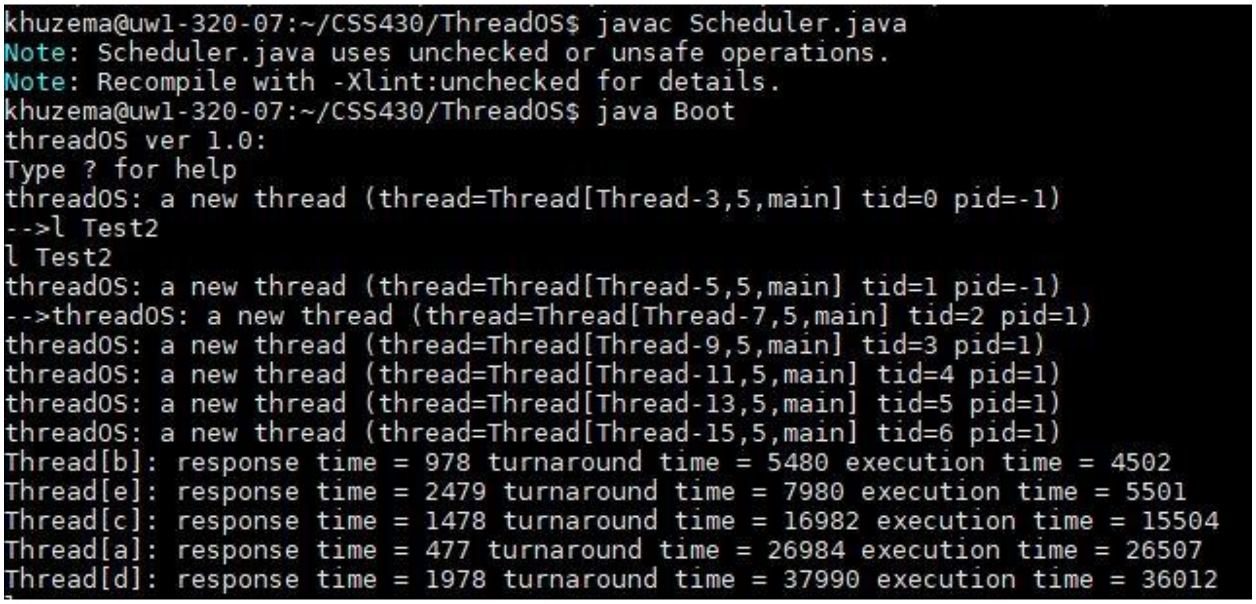
In this case we know that our queue1 was interrupted and so we must resume from where we were. So, rather than starting with the first element of queue1 I use my overloaded version of getMyTcb(queue1, Q1) pass in the Vector queue1 and the Thread Q1 to get back the last Thread that was executing. I run this Thread for another 500ms since it must have already run for 500ms if it was interrupted. This makes sure that each thread in queue1 runs for 1000ms. Then I set Q1 to null and if current thread is still Alive I send it to queue2.

Third Case: queue and queue1 have no threads and queue2 has threads

This case is a bit trickier than the Second Case because in this queue Threads can be interrupted and also they have to run for 2000ms or 500ms 4 times AND if interrupted when we come back we only run for the remaining quantum which can be 500, 1000, or 1500 ms. My solution for this case also splits it into two sub cases like the Second Case. However, to account for the different quantums a thread might need to run for if interrupted I use a new variable int quantumLeft. This int stores how many times the thread that was interrupted (Q2) in queue2 needs to run for 500ms to complete its 2000ms quantum. So, when a thread in queue2 is interrupted I record how many times it still needs to run for 500ms 1, 2, or 3. Since it must have ran once to be interrupted. When an interrupted thread is run again it runs for the required quantum while still checking every 500 ms if theres a new Thread in queue or queue1 and IF it is interrupted again I just decrement quantumLeft and save Q2 as is. Once all the allotted quantum is used Q2 is set to null and the cycle Continues!

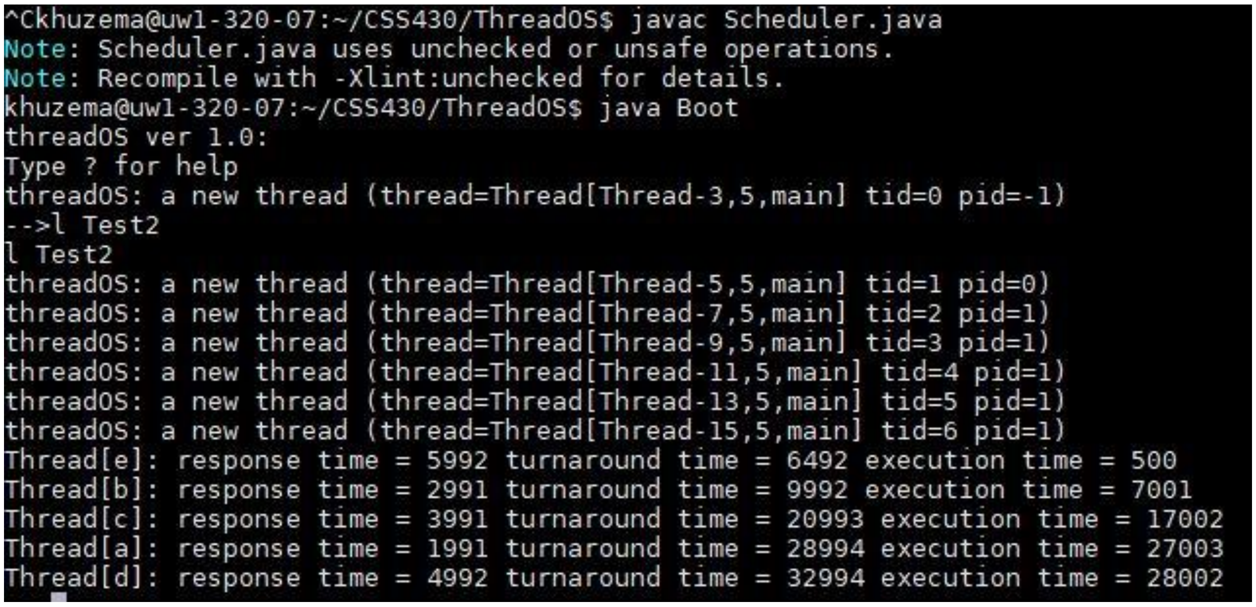
Test Results: RR Vs. MLFBQ

Multilevel Feedback Queue:



```
khuzema@uw1-320-07:~/CSS430/ThreadOS$ javac Scheduler.java
Note: Scheduler.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
khuzema@uw1-320-07:~/CSS430/ThreadOS$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,5,main] tid=0 pid=-1)
-->l Test2
l Test2
threadOS: a new thread (thread=Thread[Thread-5,5,main] tid=1 pid=-1)
-->threadOS: a new thread (thread=Thread[Thread-7,5,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-9,5,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-11,5,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-13,5,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-15,5,main] tid=6 pid=1)
Thread[b]: response time = 978 turnaround time = 5480 execution time = 4502
Thread[e]: response time = 2479 turnaround time = 7980 execution time = 5501
Thread[c]: response time = 1478 turnaround time = 16982 execution time = 15504
Thread[a]: response time = 477 turnaround time = 26984 execution time = 26507
Thread[d]: response time = 1978 turnaround time = 37990 execution time = 36012
```

Round Robin:



```
^Ckhuzema@uw1-320-07:~/CSS430/ThreadOS$ javac Scheduler.java
Note: Scheduler.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
khuzema@uw1-320-07:~/CSS430/ThreadOS$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,5,main] tid=0 pid=-1)
-->l Test2
l Test2
threadOS: a new thread (thread=Thread[Thread-5,5,main] tid=1 pid=0)
threadOS: a new thread (thread=Thread[Thread-7,5,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-9,5,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-11,5,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-13,5,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-15,5,main] tid=6 pid=1)
Thread[e]: response time = 5992 turnaround time = 6492 execution time = 500
Thread[b]: response time = 2991 turnaround time = 9992 execution time = 7001
Thread[c]: response time = 3991 turnaround time = 20993 execution time = 17002
Thread[a]: response time = 1991 turnaround time = 28994 execution time = 27003
Thread[d]: response time = 4992 turnaround time = 32994 execution time = 28002
```

These are the outputs when running Test2.java for my Schedulers. I will talk about the performance of each algorithm in regard to the Response Time, Turnaround Time, and the executing time.

Response Time:

Response Time is the time it takes for the first response of a program to be produced, This is am important metric because some programs can afford to start and then slowly do calculations as they go on, Ie. Video Games.

As you can see from the results the Multilevel feedback Queue (MLFBQ) has much faster response times then the round robin method. This makes sense since in the MLFBQ we run all threads first on a shorter quantum. This kinda of functionality is really important for our general multithreaded computers and user experience since we want as little response time as possible.

Turnaround Time (TAT):

TAT is the time it takes for the thread(process) to finish from time it was submitted until it finished. These are bit a bit mixed in the results as you can see, RR model threads e and d have a lower TAT than the MLFBQ but it still seems to favor in the MLFBQ on average.

Execution Time:

Execution time is the time from the first response (response time) till the end of the thread. This metric also seems to be better for the MLFBQ method aside from the outlier's thread e and thread d. The execution time is important because it shows how fast each thread is processed once it has been started.

What if we implemented queue2 as FCFS?

In this scenario I think the execution times will be lowered for any threads that end up going as into queue2 however this will lead to an increase of TAT for the treads at the end of the queue2 like thread d. the response times will stay the same since we service all the threads in queue(0) first like usual.

Documentation

I am submitting 2 code files and 1 report file. Scheduler.java and Schduler_part2.java as specified by the turn in guidelines will be zipped together and the report will be submitted separately.

How to Compile/Run

Under the assumption that all other require .class files are in the same directory compile Scheduler with javac Scheduler.java -deprecation, and run thread OS using java Boot, test using "l Test2"

To rest part two, make sure to change the name to Scheduler.java and remove the part 1 from the directory.