CSS430  
Martin Metke  
2017/07/23

Program 3 – Dining Philosophers

# Investigating Java Synchronization

Investigate the following options for solving this problem.  For each of the following put write a brief comment about how you could use it to solve the problem, along with a short list of pros & cons for each one.

## synchronized / wait / notify / notifyAll

e = 32977 execution time = 28000

Execution time for each thread was far higher than the total number of bursts allocated (with the exception of Thread[e], which completed in under one time slice). Completion *order* was in line with assigned burst time, however.

## Counting semaphores

e = 32977 execution time = 28000

Execution time for each thread was far higher than the total number of bursts allocated (with the exception of Thread[e], which completed in under one time slice). Completion *order* was in line with assigned burst time, however.

## Reentrant locks

e = 32977 execution time = 28000

Execution time for each thread was far higher than the total number of bursts allocated (with the exception of Thread[e], which completed in under one time slice). Completion *order* was in line with assigned burst time, however.

# Part 2

Part 2 implements a Multilevel Feedback Queue scheduler algorithm with three queues of 1, 2, and 4 quanta, each quantum == 500ms.

## Algorithm

I chose to implement the MLFQ Scheduler using three ConcurrentLinkedQueue (thread-safe) objects and a “for” loop within the main “while” loop to keep track of each thread’s current queue and remaining time quanta (in case of preemption by a higher-queue thread). Additionally, I implemented a shorter time quantum just for threads 0 and 1 (the Scheduler thread and the Test2\* thread) to reduce needless wait time to a minimum, since code inspection and run-time analysis showed that these threads were simply sleeping during their time slices.  
 The MLFQ Scheduler selects a thread from the highest filled queue and assigns that thread a packet of time quanta based on the queue level. It then runs a “for” loop of <quanta> iterations that  
A) run the selected thread for one quantum  
B) decrements the thread’s remaining quanta  
C) checks for thread termination or higher-queue preemption and exits the “for” loop in either case  
D) after the loop is completed and all quanta exhausted, moves the thread to the appropriate queue

## Results

-->l Test2

threadOS: a new thread (thread=Thread[Thread-5,5,main] tid=1 pid=0)  
l Test2  
threadOS: a new thread (thread=Thread[Thread-7,5,main] tid=2 pid=1)  
…  
threadOS: a new thread (thread=Thread[Thread-15,5,main] tid=6 pid=1)  
Thread[b]: response time = 509 turnaround time = 4033 execution time = 3524  
Thread[e]: response time = 2009 turnaround time = 6530 execution time = 4521  
Thread[c]: response time = 1009 turnaround time = 10604 execution time = 9595  
Thread[a]: response time = 9 turnaround time = 14690 execution time = 14681  
Thread[d]: response time = 1509 turnaround time = 17770 execution time = 16261

## Comparisons

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Response Time** | |  | **Turnaround Time** | |  | **Execution Time** | |
|  | **Strict RR** | **MLFQ** |  | **Strict RR** | **MLFQ** |  | **Strict RR** | **MLFQ** |
| **thread[a]** | 1977 | 9 |  | 28977 | 14690 |  | 27000 | 14681 |
| **thread[b]** | 2977 | 509 |  | 9977 | 4033 |  | 7000 | 3524 |
| **thread[c]** | 3977 | 1009 |  | 20977 | 10604 |  | 17000 | 9595 |
| **thread[d]** | 4977 | 1509 |  | 32977 | 17770 |  | 28000 | 16261 |
| **thread[e]** | 5977 | 2009 |  | 6480 | 6530 |  | 503 | 4521 |
| **AVG** | **3977** | **1009** |  | **19877.6** | **10725.4** |  | **15900.6** | **9716.4** |

As seen above, the MLFQ scheduler performed much better than the strict Round-Robin scheduler, between 1.6x (Execution Time) and 3.9x (Response Time) faster. I attribute much of this to the fact that short-running threads were able to complete more quickly than with the strict RR scheduler, but additionally to the fact that new threads were not made to wait for whole time quanta before starting (due to my optimizations for the first and second threads). While *minimum* Execution and TAT were higher than the lowest values for the strict RR scheduler, the mean and variance were much lower.  
 The assignment requires assessing a potential design with a FCFS Queue2, rather than Round-Robin. An FCFS approach would presumably result in even \*worse\* performance overall than the strict Round-Robin scheduler, certainly than my Part 2 implementation: after all threads exit Queues 0 and 1, thread[a] would run until completion before thread[c], which would presumably raise its Execution Time to over 23000ms. There would also be a knock-on effect on thread[d], pushing its Execution Time to 39000ms or more. The end result would be a scheduler with a faster TAT and Execution Time for threads running for < 3 time quanta, but much worse TAT and Execution Time for all other threads.

## Testing

In addition to running the suggested base and testing Test\* classes, I created a Test2a class that only executed one thread for 500 “burst time”, in order to characterize the Scheduler and Test2\* thread overhead.

# Documentation of Design

To compile and run the scheduler for this assignment:  
  
1) Copy Scheduler\_part\_1.java or Scheduler\_part\_2.java to …/ThreadOS/Scheduler.java  
2) Move to the ThreadOS directory, e.g. ‘cd ThreadOS/’  
3) Run ‘javac -deprecation Scheduler.java’ from within the ThreadOS directory  
4) Start ThreadOS by running ‘java Boot’ from within the ThreadOS directory  
5) At the “-->” prompt within ThreadOS, type ‘l Test2’, ‘l Test2b’, etc. to execute a thread test