October 23, 2016

COM SCI 131 HW #3 Report

Instead of using the sample test case "java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3" provided in the course website, I decided to go with test the programs with 100,000 swaps with maximum value of 100 and an array of 5 elements in 2 threads, 4 threads, 8 threads and 16 threads. Different number of threads would illustrate how the multithreading influences the performance of the programs, and different number of swaps would better display the probability of crash and performance statistics. Also, I chose a higher max value to test a greater range of possible values in the dataset.

I used the lnxsrv09 server to run the tests and gather statistical results, using the commands in the course website to check the environment information in the server.

Java version : "1.8.0_102"

OpenJDK 64-Bit Server VM (build 25.102-b14, mixed mode)

OpenJDK Runtime Environment (build 1.8.0_102-b14)

CPU: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz

Memory: 64GB

Synchronized

The Synchronized model given by the course website is both reliable and DRF. The "synchronized" keyword in the swap method guarantees that only a single thread can execute the method. However, since only one thread at a time is permitted to access the code, the performance of Synchronized is really slow in this case. As a matter of fact, it is the slowest of all the models discussed in this assignment.

Unsynchronized

The Unsynchronized model is unreliable as I test the model. Every time I called the Unsynchronized class, there would be some mismatch incurred. For example, "sum mismatch (17 != 208)". The reason that the Unsynchronized model is unreliable is that since there is no Synchronized keyword in the swap method, any threads can simultaneously access, read and write the same element in the array, which results in serious race conditions. Sometimes the race conditions even leads to deadlocks. This happens after I tried using the Unsynchronized model for many times.

GetNSet

The GetNSet model uses AtomicIntegerArray. The AtomicIntegerArray guarantees that no more than one thread can read or write in the array at the same time (in this case, increment or decrement the same element in the array). However, it is still reliable. Similarly, as I tested GetNSet, there would be some mismatch occurring and the probability of deadlock is even much higher than the Unsynchronized model. The problem might be that when multiple threads go into the boundary check in the swap method, there is no way to "stop" the program if all elements are at/over the boundaries, which leads to the crash/deadlocks.

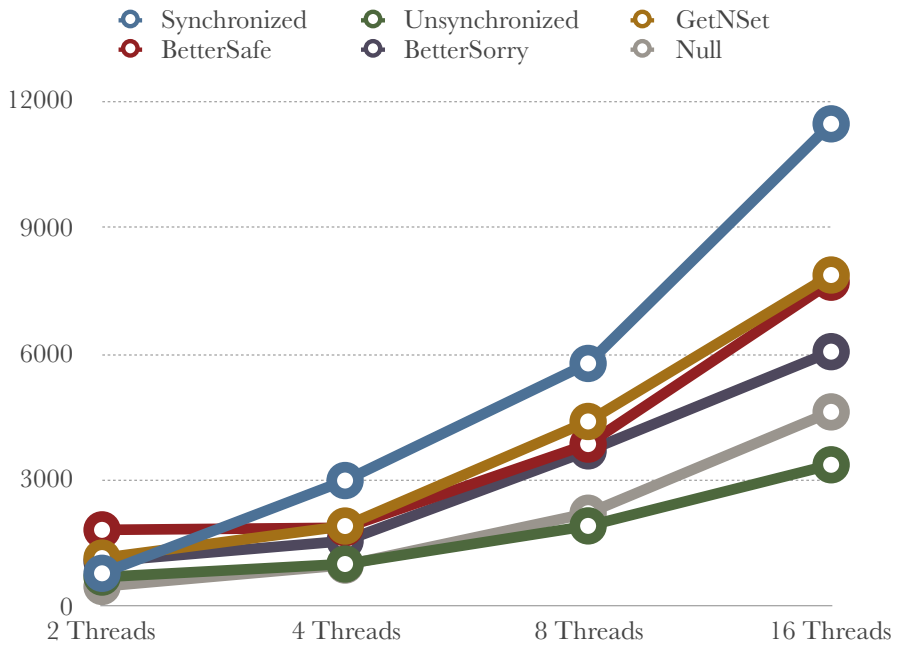
BetterSafe

The BetterSafe model uses ReentrantLock from the hint in the course website. By locking the increment and decrement operations, we can guarantee that no more than one thread can read or write in the array at the same

time. Also, the boundary check in the swap method guarantees that no value would fall below the lower bound or go beyond the upper bound. Thus, in this way the model guarantees that these two serious problems, both possibly leading to deadlocks and race conditions, would not happen in this case. The performance of BetterSafe is also significantly better than Synchronized. The reason of the improvement might be that ReentrantLock is unstructured, so it don't have to construct like what Synchronized does, which might improve the efficiency of the resources.

BetterSorry

The BetterSorry model uses AtomicInteger before the increment and decrement operations in the swap method. Since these operations are now securely atomic.Unlike the AtomicIntegerArray in GetNSet model, now concurrent threads can access the array at the same time as long as the elements in the array are not the same (thus the problem that two threads write into the same element would not happen in this case). It is far better than Unsynchronized since it avoids the situation when multiple threads access the elements in the array with the same value, which leads to a race condition. However, it is not 100% reliable. The values of the elements can be changed during the boundary check, which might result in race conditions. During my tests, the BetterSorry model did not fail frequently compared to Unsynchronized model and GetNSet model. Thus, statistically speaking, BetterSorry achieves better performance than BetterSafe and better reliability than Unsynchronized.



Questions and Difficulties

The results discussed above are only relative results running in the specific environment. That is to say, different environments might have conclusive influence on the testing results. Also, when testing these models on SEASNET linux servers, the number varies a lot among different executions. Thus, I decided to run each model 10 times and take the average value. Also, it takes me some time to figure out how to compare the numbers relatively between different models.