## Homework 3 JMM Report

Zhehao Wang <[zhehao@cs.ucla.edu](mailto:zhehao@cs.ucla.edu)> 404380075

#### Environment information (lnxsrv07.seas.ucla.edu)

Java version "1.8.0\_51"

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

Memory: 64GB

#### Results and discussion

In this assignment we implemented “Unsynchronized”, “BetterSorry”, “GetNSet”, and “BetterSafe” models for testing Java shared memory performance races.

Macintosh HD:Users:zhehaowang:Pictures:data-race-scenario.pdfThe data race comes from a) non-atomic increment and decrement operators. One scenario’s illustrated in Figure 1, in which two increment operations on the shared memory will result in only one increment of the variable X.

Figure 1. Data racing scenario

Another problematic scenario is b) values in the shared memory can fall below the lower bound, or go beyond the upper bound, if multiple threads passed the boundary check before any of them writes the incremented or decremented value. Here we introduced an arbitrary timeout to make sure program exits.

The unsynchronized implementation suffers badly from a), and we can hardly maintain 50% reliability with 4 threads and 100 transitions; b) also occurs frequently, with the chance of successful execution getting higher as larger upper limits are given.

Our GetNSet implementation uses AtomicIntegerArray, which guarantees data race as described in scenario a) would not happen. Scenario b) could still happen, thus reliability is close to, but not 100%. Interestingly, b) happens more often in large number of threads, low amount of transitions cases. (e.g. 16 threads, 30 transitions)

The BestSafe implementation uses ReentrantLock, which is usually considered as a more flexible synchronization method than Synchronized. By locking the increment and decrement operations, we can guarantee that scenario a) wouldn’t appear. And by adding the boundary condition check into this critical section, we can guarantee that b) wouldn’t happen either. Thus we have 100% reliability.

BestSafe outperforms Synchronized model in higher contention rate scenarios, with the possible reason being that the ReentrantLock has a smaller scope, and a lighter-weight internal implementation than the synchronized keyword.

Our BestSorry implementation creates an AtomicInteger before increment and decrement, to guarantee that these operations are atomic.

Similar as GetNSet, scenario a) won’t happen, while b) may still happen; Thus we consider this approach not 100% reliable, though much better than Unsynchronized. (8 threads 20 transitions has ~5% chance of overflowing)

The performance is similar with GetNSet in general, and slightly better in low contention scenarios. It outperforms BetterSafe in a high contention environment, potentially because AtomicInteger utilizes lock-free concurrency facilities from the hardware.

Interestingly, reliability drops as thread number increases and transition number decreases. Also, an array of AtomicInteger performs worse than allocating AtomicInteger when swap is called.

It is worth mentioning that the test results in this assignment are subject to Java version, as compile time code optimization, and JIT’s runtime optimization may be different. In addition to this, the hardware architecture may also influence the test results, as machine code optimization may be different, and different high-level features may be provided.

Here we provide the initial performance measurement of the 5 models under different scenarios in the appendix. A test script is used for automating tests under different scenarios.

Appendix:

Unsynchronized model has a very low reliability under any of the test scenarios below, and is not listed in our results. And tests are carried out with an array of size 5, maximum at 100. The average time per transaction comes from time measured in 20 sequential executions.

Macintosh HD:Users:zhehaowang:Pictures:100000-transaction.pdf

Figure 2. Performance of the five models under 100000 transitions scenario

This result shows that under high contention scenarios with a considerable amount of transitions, the order of performance from worst to best is Synchronized, BetterSafe, GetNSet, and BetterSorry (almost tie).

Macintosh HD:Users:zhehaowang:Pictures:10000-transaction.pdf

Figure 3. Performance of the five models under 10000 transitions scenario

This result shows that with less amount of transitions, BetterSafe is worse than Synchronized, and GetNSet is worse than BetterSorry.

The average time is also much longer than Figure 1, whose reason could be that higher throughput CPU caches, and other hardware optimizations do not have enough amount of operations to fully demonstrate their advantage.

Macintosh HD:Users:zhehaowang:Pictures:8-thread.pdf

Figure 4. Performance of the five models under 8 threads scenario

In this figure, we use a fixed number of threads and observe the performance change as more transitions are performed. Synchronized becomes more costly as the number of transitions grows. One thing that we find hard to explain is that the performance of BetterSafe and GetNSet becomes as good as Null (here it suggests that they are even slightly better than Null).