HW3 jmmplus.jar Report

Xiao Yan, *UCLA*

1.Introduction

Java version "1.8.0_112"
Java(TM) SE Runtime Environment (build 1.8.0_112-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.112-b16, mixed mode)

In this assignment, I implement several new Classes from State: Unsynchronized, BetterSorry, GetNSet and BetterSafe to test the multithread programming of Java. The data race/critical section comes from adding and deleting from a list we pass to the program. Among all the new Classes, Unsynchronized is running without any atomic operation, BetterSorry takes a middle approach such that parts of the program is based on atomic operation. GetNSet and BetterSafe are two other ways to achieve atomic operation besides Synchronized, which is given initially.

2.Results:

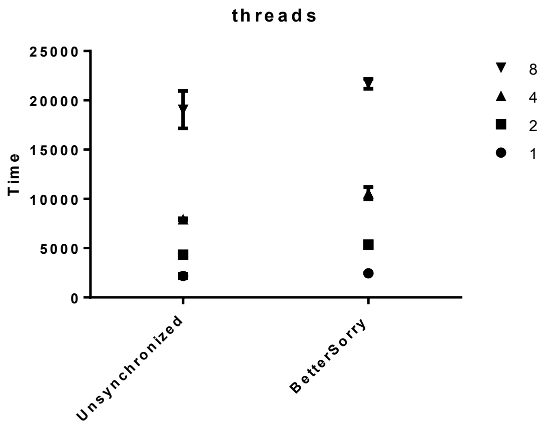

Figure 1: Time/transition using Unsynchronized or BetterSorry method, with a bunch of different number of threads, and transitions =1000. In my system, a transition > 1000 tends to cause infinite loop. Based on the data, BetterSorry takes slightly more time than Unsynchronized.

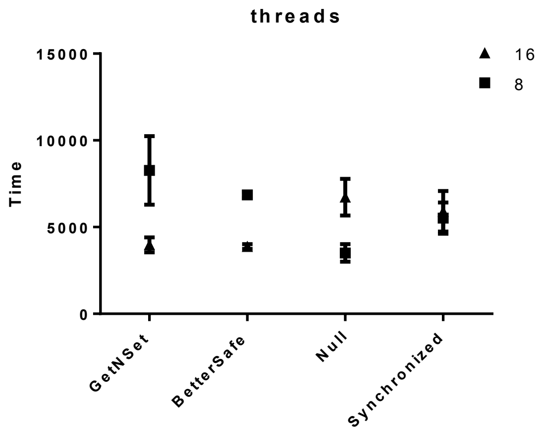

Figure 2: Time/transition using four methods, with threads = 8 or 16, and transition = 100000. Both GetNSet and BetterSafe is faster than Synchronized when there are more threads running. However, the difference is not clear when using a smaller number of threads.

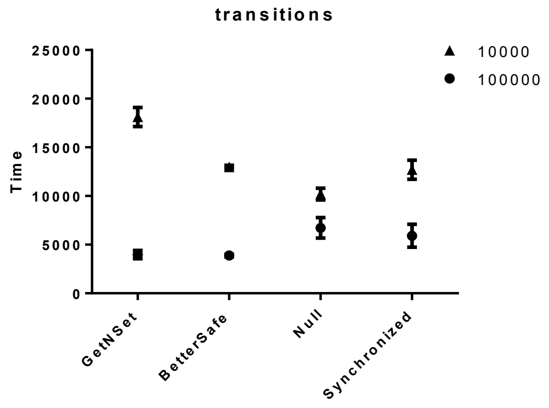

Figure 3: Time/transition using four methods, with threads=16, and transition = 10000 or 100000. Both GetNSet and BetterSafe is faster than Synchronized when there are more transitions. However, the difference is not clear when using a smaller number of transitions.

3. Discussion

BetterSafe is faster than Synchronized because it utilized the ReentrantLock, which is supposed to provide a reentrant mutual exclusion lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities. Since we lock the critical section, we can assure that only one thread can do increment/decrement operation, so it is 100% reliable.

BetterSorry is faster than BetterSafe , because it creates an AtomicInteger to perform increment and decrement operation. However, it didn’t reach 100% protection of the thread, which means in certain situations, the threads will keep running even though they should wait. This increases the overall speed of the program, compared to BetterSafe, and is more reliable than Unsynchronized because part of the operation is done in atomic way. In a race condition, when one thread is about to change the value[i] and value[j], another thread may have already changed these two values, such that the value[i] and value[j] may be beyond the normal range. If we use a higher number of threads, say thread =8 and higher transitions, say transition=100000, in this situation eventually most of the values will out of the range and there will be an infinite loop.

GetNSet and BetterSafe are DRF. For BetterSafe, I locked the critical section that causes the data race such that only one thread can get access to the value. In GetNSet, I used the AtomicIntegerArray to store the value such that all of the operations are atomic and thus DRF. However, in BetterSorry and Unsynchronized, at least part of the critical section is unprotected so it is not DRF. For example, if I do java Unsynchronized/BetterSorry 8 10000 100 50 50 50 50 5 0, it is most likely to fall.

Deadlock
In both BetterSorry and Unsynchronized, if we test a large number of transitions, the program can reach an infinite loop. However, it is not caused by deadlock (unrelated to multithreading). It is caused by the design of the Swaptest program. If most of the values in the value list are out of range, the function run () in Swaptest becomes an infinite loop. **It can be avoided by changing the Swaptest, or by stopping the program whenever one value in the value list goes out of range. The modification hasn’t been implemented because of the requirement in this homework.**