CS131 Homework 3 Report

Jinjing Zhou
Lab Section: 1C

Abstract

This lab investigates the reliability and performance of four classes, Synchronized, Unsynchronized, GetNSet, and BetterSafe, where each implements a different multi-threading approach on a simple prototype that manages a data structure that represents an array of integers,as well as condcuts a state transition.

1 Compare between different classes

I ran the program using different state classes and an input of a byte array of size 100, where each element is in the range of [0,127], with 4, 8, and 16 threads. The result is shown in the table below.

Table 1: Table of average time and DRF state for four classes

Class used	Average real-time/ns			DRF
	4	8	16	
Synchronized	1484.8	3102.8	5995.4	Yes
Unsynchronized	898.8	1823.2	3830.4	No
GetNSet	1023.8	2003.7	4189.3	No
BetterSafe	1249.4	2774.3	4873.9	Yes

1.1 Synchronized

As shown in the table, the synchronized class has the longest average time taken for each transition. It is data race free since the keyword synchronized in java which ensures thread-safety, so this class is reliable but with low performance.

1.2 Unsynchronized

The unsynchronized class achieves the shortest average time per transition, hence the highest speed. However, it is not data race free since we removed all synchronized keyword and did not use any other measure to ensure thread-safety. As a result, since sequential consistency is not guaranteed, between each step of retrieving, modifying and storing data, each thread may interfere with each other and results in race conditions. It also leads to infinite loops in many cases, which makes this class highly unreliable. An example of test code that the class would likely fail would be ”java UnsafeMemory Unsynchronized 4 1000000 100 5 6 3 0 3 1 2 3 4 5”.

1.3 GetNSet

The GetNSet class achieves the second shortest average time per transition since it implements an AtomicIntegerArray which allows each element to be accessed and updated atomically. Still, it is not data race free. Though implementing AtomicIntegerArray ensures the retrieving and updating step for each element in the array are thread-safe, there is an if statement that decides whether the two elements chosen satisfy the criteria we want between the retrieving(get) and updating(set) step in each swap. This leads to possibility that some threads may access the same element of the array at different times and then updating it more than once using the incremented value, resulting in race conditions and therefore infinite loops in many cases. Thus, this class though has good performance, is not reliable as well. In fact, its percentage of mismatch is 100% for all the tests I ran, just like unsynchronized. An example of test code that the class would likely fail would be ”java UnsafeMemory GetNSet 4 1000000 100 5 6 3 0 3 1 2 3 4 5”.

1.4 BetterSafe

The BetterSafe class achieves shorter average time per transition than Synchronized since instead of locking the whole function as Synchronized did, it only implements the Reentrant lock on the accessing and updating part of the function. It is also data race free since the Reentrant lock will ensure that if there are competing threads trying to access or modify the array, only one thread will acquire the Reentrant lock and perform the action while other threads will wait until the action of the first thread is completed and the lock is unlocked.Thus, this class achieves better performance than Synchronized, though has longer average time than the other two methods, retains the reliability of the Synchronized class.

2 Implementing BetterSafe

2.1 Packagejava.util.concurrent

The class that seems to be helpful in this package would be the Semaphore class, and in our case, we will need to implement a binary semaphore. The advantage is that it will also ensure that the first thread will finish its reading and writing tasks before any other threads start to avoid race condition to some extent. However, unlike mutexes, Semaphores are not reentrant since it does not protect shared variables from being accessed by multiple threads at the same time, resulting in some possibility of race condition.

2.2 Package java.util.concurrent.atomic

The advantage of this package, as stated before in the GetNSet method, would be that since it implements thread-safety on a finer-grain - each individual element in the array, it will achieve a higher speed than Synchronized. However, among the functions in class AtomicInteger and AtomicIntegerArray, there is no function that would allow the retrieve, compare and increment/decrement steps to be completed atomically, so the same race condition due to the intermediate compare step between retrieving and updating data will always occur.

2.3 Package java.util.concurrent.locks

The package mainly contains three types of locks that are potentially helpful to our BetterSafe class, namely, ReentrantLock, ReentrantReadWriteLock and StampedLock. The advantage of implementing a lock overall is that each lock will only restrict other threads from accessing the part of the function that are prone to race conditions, so that other threads can still progress to some extent, resulting in overhead that speeds up the program than Synchronized. After a closer look at all three types of locks, I discovered that ReentrantReadWriteLock may not be applicable in our situation since our write depends on the result of our read before the comparison. Even with ReadWriteLock, it is likely that there will be threads that has a previous version of the data after WriteLock is released and therefore updata the wrong value, resulting in race conditions. The StampedLock may not be used in our case as well since they are not reentrant. A locked body will be allowed to call methods that try to reacquire locks, which may result in deadlock in some cases. The reentrant lock on the other hand, will speed up the program as well as ensure data race free as described in section 1.4 above.

2.4 Package java.util.concurrent.VarHandle

The VarHandle package is a generic version of AtomicInteger. It uses reflection to allow atomic access to variables of any type. Therefore, it shares the advantage of the atomic package, that it will achieve a higher speed than Synchronized. However, it also share the downside that there will likely be race conditions due to the separated read and write steps in our program.

2.5 Choosing the Package

To sum up, since the other three packages are all susceptible to race conditions to some extent, and our main target of BetterSafe is to be data race free rather than to speed up the program, I would choose the reentrant lock in the locks package to implement BetterSafe.

3 Conclusion

3.1 Problem overcome

First of all, when I was testing the Unsynchronized class, it always ran into infinite loop. Then I looked up Piazza and found that this is probably due to the race condition, and increasing the maxval to 100 may help to avoid that. Then, when testing my code, I found that it would be a huge amount of work to run the command multiple times for each of the class using different number of threads. Therefore, I wrote a python script to run each case for 50 times and output the average amount of time taken per thread.

3.2 Choice for GDI's applications

Comparing all four class' performance and reliability, I would choose BetterSafe to implement GDI's applications. In addition to being faster than Synchronized, it also maintains 100%reliability, while the other two classes are almost 100% incorrect. Thus, according to the results of this lab, BetterSafe is shown to be the best choice for GDI's applications.