CS 131 Homework 3 Report

Jingyue Shen

1.Abstract

This report compares different concurrent programming methods in Java in aspects of performance and reliability, along with each method's pros and cons.

2.Testing Platform

Java version: 9.0.1

Java(TM) SE Runtime Environment (build 9.0.1+11)

lnxsrv 09: CPU: 4-Core Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz with 64 GB RAM

3. Results

3.1 Different Number of Swaps

*Tested on 8 threads

* java UnsafeMemory xxx 8 xxx 100 10 10 10 10 10

100 successful swap transitions

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	174174	255809	386807	177480
Reliability	3/10	2/10	10/10	10/10

1000 successful swap transitions

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	25885	42862	63016	25848
Reliability	1/10	1/10	10/10	10/10

10000 successful swap transitions

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	6531	14646	13333	8352
Reliability	0/10	0/10	10/10	10/10

100000 successful swap transitions

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	1955	3529	4177	4653
Reliability	0/10(5 frozen)	0/10(4 frozen)	10/10	10/10

1000000 successful swap transitions

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	1895	Infinite loop	1317	2248
Reliability	0/10(4 frozen)	0/10(10 frozen)	10/10	10/10

3.2 Different Number of Threads

*Tested on 100000 successful swap transitions

2 Threads

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	1579	2236	2661	1839
Reliability	1/10	2/10(2 frozen)	10/10	10/10

4 Threads

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	4294	6148	4597	4263
Reliability	1/10	2/10(3 frozen)	10/10	10/10

8 Threads

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	14678	12400	12442	13223
Reliability	5/10	0/10(2 frozen)	10/10	10/10

16 Threads

	Unsynchronized	GetNSet	BetterSafe	Synchronized
Avg. ns per swap	50820	70058	52345	54617
Reliability	6/10	6/10	10/10	10/10

4.Analysis of Results

From the comparison above, we can see that when the number of swap transitions and threads get higher, the BetterSafe implementation performs much better than the Synchronized method while retaining 100% reliability. On the other hand, GetNSet does not show much improvement in both reliability and performance. This is due to the fact that in the swap() function, there will have race condition between if statement and the block of code that updates the state values. Other threads can step on the exact same state and update the value first before the current thread can update its state value. So in general, the BetterSafe method is the best choice for GDI's applications, since it performs better on more threads and when the number of state update gets larger.

5. About BetterSafe

5.1 Why faster and 100% reliable

I used reentrant lock to implement BetterSafe class. Reentrant lock is similar to synchronized keyword, but perform much better when treads and the number of swap transitions gets higher. According to IBM, ReentrantLock "offers far better performance than synchronized under heavy contention. In other words, when many threads attempting to access a shared resource, the JVM will spend less time scheduling threads and more time executing them.[1]

As for reliability, the reentrant lock's implementation is 100% reliable since it locks the access to the states variable at the beginning of swap() function and unlocks it just before exiting the swap() function. So the whole reading and writing process for a variable is protected. No other threads can access and change the value of the same variable while it is locked.

5.2 Pros And Cons of the four packages

1.java.util.concurrent

It provides five classes(synchronizers), namely Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.

Semaphores: though we can use binary semaphore to act like a lock, semaphores are often used to restrict number of threads than can access some resouces. And in BetterSafe class, all I want to do is to protect the read and write block from being accessed via other threads at the same time. So if I want to use a binary semaphore to act like a lock, I can just use a lock instead.

CountDownLatch: CountDownLatch is used for blocking until a given number of signals, events or conditions hold. The point is that it is a one-shot action. After the count reaches zero, all the threads await will be released. And the count will never be reset. So it cannot be used here to implement BetterSafe class. It is not in the right scenario.

CyclicBarriar: The difference between Cyclic Barriar and CountDownLatch is that you can reset the count to open the latch multiple times. However, though it can be reset, it is still not what we want. It is useful in programs involving a fixed sized party of threads that must occasionally wait for each other until they reach the same spot to proceed.

Phaser: A reusable synchronization barrier, similar to Cyclic Barrier, but supporting more flexible usage. So this is also not the right option, since what we want is to protect a block from being accessed by other threads, rather than wait to arrive a same spot and do actions all at once.

Exchanger: It allows two threads to exchange objects at a rendezvous point. Only when both threads invoke exchange method will the exchange be performed. Again, we do not need to exchange things between two threads, only to offer protection during read and write.

2. java.util.concurrent.atomic

The pros of those classes in atomic package is that they support lock-free and thread-free programming on single variables. So each thread does not need to wait for others to access a block. It would be faster. The memory effects of methods within this class is like the volatile keywords as described in Java 9 Documentation.[1] So it will read directly from memory rather than from cache. But the cons are obvious. The swap() function consists of two parts: read and update(write). And between the if statement (read part) and update, race condition can still occur. Other threads can update the value in the same state and one result is that the bound checking if-statement would be useless. So it will lead to unsuccessful swap transition and also lead to different sum with respect to the sum of input states.

3.java.util.concurrent.locks

I use this package and reentrant lock to implement the BetterSafe class since it is easily implementable, can protect the whole read-write process from other threads to avoid race condition, and is faster than synchronized keyword when the number of threads get higher. The cons of this class is that it other threads must wait when one thread is accessing the same object. So the performance would be somewhat limited. But in order to ensure 100% reliability, it is necessary to protect the whole read-write process with a

[1] Goetz, Brian. "Java Theory and Practice: More Flexible, Scalable Locking in JDK 5.0." IBM DeveloperWorks. IBM, 26 Oct. 2004 Web. 21 Dec. 2016

[2]https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/AtomicIntegerArray.html

lock. Besides, at first I used the ReentrantReadWriteLock class. But after testing I found that it is not 100% reliable in this situation. Finally I figured out that race condition still exists between the bounding checking if-statement and the value updating part. What we need is a single lock that cover the whole read-write region to completely avoid race condition.

4.java.lang.invoke.VarHandle

A VarHandle is a dynamically strongly typed reference to a variable. It provides different access mode to the variable referenced, like plain read/write, volatile read/write access and compare-and-swap. So it also provides a lock-free thread-safe programming on single variable, since it can read value of variable directly from memory rather than from cache. But still, between the read and update part in the swap() function, since there is nothing to prevent other threads from accessing the same variable, there will have race condition. And the bound checking if-statement would be useless and might cause unsuccessful swapping and increase the time for a successful swap.

6. Problem Overcame

When testing those methods on the server, the average time spent per successful swap for same conditions varies largely. I think it is due to the varying number of users on the server. So I took the average of 10-20 results per case as the final data put in the report. Also, due to the unreliability of Unsynchronized and GetNSet class, the program always ran into infinite loop when the required number of successful swap and the number of threads get larger. I explicitly recorded how many times this situation occurs in the tables above.

7.DRF

The classes Synchronized and BetterSafe are DRF classes since they use a lock to protect the whole read-write region from other threads so that other threads cannot access and modify any value of the same object when one thread is doing read and write.

The classes GetNSet and Unsynchronized are not DRF. For GetNSet, race condition can still occur between the bound-checking if-statement and the following value updating part. Though methods in GetNSet all read data from memory rather than caches, other threads can still update values that we will use in the set() method, leading to an unreliable result. The reliability test for these two classes are given in the tables above.