Report for CS131 Homework 3

Zixuan Lu
304990072

2019-05-05

Abstract

Homework 3 of CS 131 allows us to have a more thorough understanding of synchronized and unsynchronized java programs through their respective performances on a simple multi-threaded "swap" operation. The purpose of this homework is to learn JMM (Java Memory Model) as well as to write DRF (Data-Race Free) java programs. I will compare the performances of different implementations (both synchronized and unsynchronized) of a class that does a simple multi-threaded swap operation, and explain my implementation of a DRF, more efficient BetterSafe class in this report. All the tests are done on SEASNET linux server 09, which has a openjdk version "11.0.2" by running "java -version".

1 Pros and Cons of four suggested packages

Among the four packages suggested in the homework spec, I chose java.util.concurrent.locks to implement my BetterSafe class. First I will explain the pros and cons of all the four packages suggested, then I will explain why I made my choice.

1.1 java.util.concurrent

This package provides utility classes commonly useful in concurrent programming according to the manual page. Though it is definitely capable of implementing a faster while DRF class, its low-level nature makes it complicated to use (as compared to java.util.concurrent.locks package I choose).

pros: The low-level nature of this package enables the user to fully control the behaviour and ordering of each thread, thus making a DRF and efficient (optimized) implementation of BetterSafe possible.
cons: It is just the pros of this package that makes the implementation over-complicated for a simple multi-threaded "swap" function, thus this package is not suitable and not chosen by me.

1.2 java.util.concurrent.atomic

This package is used to implement the second GetNSet using the AtomicIntegerArray class defined. This package provides atomic ways access and update data, and is easy to use. It would be a perfect choice if we only need to perform one update in the swap method. However, since we need to update both i and j, race conditions can still happen even though each individual action is atomic.

pros: Provides easy way to atomically access data.
cons: the critical section is each individual operation, thus it is not completely DRF, as we can see from the results of GetNSet class.

1.3 java.util.concurrent.locks

This package provides some lock classes to prevent multiple threads entering a critical section, thus leading to race conditions.

pros: Easy to understand and implement for a

java beginner, as the interface of ReentrantLock (which I chose to implement BetterSafe class) is similar to lock in C (which I used in CS111 projects). It allows user to customize, and thus minimize critical section, thus leads to better performance.
cons: Not really. The main issue is since we can customize the critical section, it is the user's responsibility to ensure that the critical section is complete (will not cause potential races).

1.4 java.lang.invoke.VarHandle

This package provides different modes of access to different types of variables.

pros: provides ways like compare-and-set to ensure atomicity.
cons: We only need to deal with byte and integer type, and as stated in java.util.concurrent.atomic section, the critical section is only the operation, thus we can still face race conditions as the swap function includes multiple reads and writes.

1.5 The reason for why I choose java.util.concurrent.locks

As a java beginner, I choose among the four packages suggested based on two criteria: whether the package can be used to write a faster while DRF swap method, and whether the method is easy to use. Thus, I choose java.util.concurrent.locks because the interface of ReentrantLock class is similar to C locks which I have used before in CS111. Also, the lock provides functionality that makes me able to customize my critical section (smaller), thus make my swap function more efficient while remaining DRF.

2 Why BetterSafe is faster while remaining 100% reliable

My implementation of swap method in BetterSafe class is as follow:

```
public boolean swap(int i, int j) {
        lock.lock();
        if (value[i] <= 0
            || value[j] >= maxval) {
            lock.unlock();
            return false;
        }
        value[i]--;
        value[j]++;
        lock.unlock();
        return true;
    }
```

Instead of marking the entire swap function as synchronized and thus a critical section, I only uses locks to wrap sections that include reading and writing to the shared memory. Thus, my swap function has a better performance due to a smaller critical section. However, since my new critical section in BetterSafe still includes all the modification of the critical section, my BetterSafe class is still DRF.

3 Performance and reliability tests of classes

I will test Nullstate, SynchronizedState, Unsynchronized, GetNSet, BetterSafe classes for 10000, 100000, 1000000 swaps with 1, 8, 16, 32 threads. The maxval is set to 6, and the initial values for the five entries in the state array are 5, 6, 3, 0, 3. I will test each case for three times, and record the average result. I used simplified notation of class in the table due to limited space (the double-column format does not allow me to show the full notation.

threads	1	8	16	32
Null/ns	636.019	7548.91	15305.7	31492.7
Sync/ns	699.087	10911.6	22985.7	44975.7
Unsync/ns	683.825	FAIL	FAIL	FAIL
GNS/ns	1065.99	FAIL	FAIL	FAIL
BS/ns	1137.09	16307.7	34206.6	73433.3

Table 1: Test results for 10000 swaps

threads	1	8	16	32
Null/ns	198.061	1689.28	3464.55	10209.1
Sync/ns	232.908	5172.34	12129.8	22805.7
Unsync/ns	268.098	FAIL	FAIL	FAIL
GNS/ns	347.390	FAIL	FAIL	FAIL
BS/ns	300.513	5183.82	10897.0	22087.5

Table 2: Test results for 100000 swaps

threads	1	8	16	32
Null/ns	65.1065	2095.06	4551.45	13815.4
Sync/ns	201.649	2698.32	4150.70	10201.1
Unsync/ns	210.626	FAIL	FAIL	FAIL
GNS/ns	268.651	FAIL	FAIL	FAIL
BS/ns	200.466	1404.17	3159.21	6352.05

Table 3: Test results for 1000000 swaps

4 Analysis of test results

4.1 performance analysis

From the above results, we can see that SynchronizedState has the worst performance among all the classes. This is reasonable because it has the largest critical section as compared to all the other four classes. NullState class has the best overall performance, this is reasonable because essentially nullstate does nothing. BetterSafe has overall better performance than SynchronizedState due to my optimization using ReentrantLock, minimizing the critical section, hence enhancing efficiency. According to Lea's paper, using locks in my BetterSafe class enables me to achieve a partial order.

GetNSet and UnsynchronizedState fail majority of the test cases due to the fact that the two classes are not DRF (will be discussed in the next subsection). However, when there is only 1 thread, we can see unsynchronized has a good performance because it does not have any overhead to ensure synchronization.

4.2 DRF analysis

From the above tables, we can see Unsynchronized and GetNSet are not DRF while the rest of the classes are DRF. This is reasonable because NullState does not access the shared memory at all, BetterSafe and Synchronized define the critical section well. Unsynchronized does not have preventions for race conditions at all. Though GetNSet uses AtomicIntegerArray, only individual get() and set() operations are atomic, the swap method as a whole is not. Thus, race conditions can still happen for this class, but is less likely as compared to Unsynchronized which does not have protection at all. For both of them to fail , the commands "java UnsafeMemory Unsynchronized 32 1000000 6 5 6 3 0 3" and "java UnsafeMemory GetNSet 32 1000000 6 5 6 3 0 3" have very high chances.

4.3 the class that is ideal for GDI

Though Unsynchronized and GetNSet are faster, the results are completely broken when multi-threading is involved. Hence, the best class is still BetterSafe, which is DRF as analysed in the previous subsection, while has a higher performance as compared to SynchronizedState class.

5 Challenges overcomed

The biggest challenge for me is as a java beginner, I spent fairly a large amount of time trying to understand the documentations before I actually write my code. Also, the instructions for GetNSet class are a little confusing to me. I initially used the getAndIecrement() and getAndDecrement() methods instead of set() which are not allowed by the spec.

6 Conclusion

BetterSafe class I implemented generally has a better performance and is DRF when many threads are used to do a large number of operations. Hence, I think BetterSafe is the ideal class for GDI.

7 Bibliography

1. Doug Lea. 2018. Using JDK 9 Memory Order Modes. (November 2018).