CS 131 Homework 3 Report

Abstract

The objective of this lab - Java shared memory performance races Background is to gain understanding of the Java Memory Model (JMM) and to develop high performance programs without race conditions. We create synchronized and unsynchronized classes having race conditions and check their performance, and reliability. Both the classes synchronized and unsynchronized has the same functionality which is to change values in an array. Only the implementation of the classes differs. The test is done by using various values for the number of threads, number of swap transitions, size of the state array, and sum of values in the state array.

1. BetterSafe

BetterSafe does not use the keyword synchronized but uses an alternate method to data race free program. It uses a Reetrant lock from the package java.util.concurrent.locks in the critical section which is inside of the swap section.

1.1. Performance

BetterSafe performs better than synchronized in terms of speed since Reetrant locks are designed to perform better than the synchronized keyword for multi-threading applications. Synchronized is applied to the whole function whereas locks only cover the critical sections which accesses the array. This fine granularity of locks helps to increase the performance.

To check the performance, a test was performed for both the classes using a million transactions and 16 threads. As we can see from the table below, BetterSafe takes significantly lesser time for each transaction.

	BetterSafe	Synchronized
Test 1	2271.16	3535.23
Test 2	2437.86	3704.97
Test 3	2733.18	3180.02
Average	2480 ns	3473 ns

Table 1: Time in ns for each transaction with 16 threads for a million transactions (swaps)

1.2. Reliability

BetterSafe is as reliable as synchronized class. All the critical sections are locked before going into them and are unlocked after we are done with the section. This avoids any race conditions. Figure 1 shows the use of Reetrant locks to protect the critical sections.

1.3. Race Conditions

The race conditions for the program only exists in the swap function. There are two places where the array is accessed or modified.

While obtaining the byte at index i and j, another thread can go and change the values for both of them. Similarly, when the bytes at index i and j and decremented and incremented respectively, another thread running parallelly can jump in and modify the array values for i and j.

```
public boolean swap(int i, int j) {
    relock.lock() ;

    if (value[i] <= 0 || value[j] >= maxval) {
      relock.unlock();
      return false;
    }
    value[i]--;
    value[j]++;

    relock.unlock();
    return true;
  }
```

Figure 1: swap function code displaying race conditions

1.4. Package Analysis

Before coming to the conclusion of using Reetrant lock from the package java.util.concurrent.locks, several other packages were considered for eliminating race conditions.

1.4.1. java.util.concurrent

If java.util.concurrent package is chosen to implement BetterSafe class, low level data structures like Thread Factories and Dequeues can be used for synchronization. This would give programmers authority to order the threads. Threads could be sorted on a Thread dequeue to run the thread with higher priorities first. This advantage however does not outweigh the disadvantage of using this package. This customizability for data structures adds significant complexity to our program. It is hard to coordinate lot of threads that run from an object method. Due to this added complexity of this package, it was not used to implement BetterSafe.

1.4.2. java.util.concurrent.atomic

This package provides numerous classes which do not use locks for atomicity in the critical sections. The data structure would lock itself if any attempts are made to modify multiple data structures inside the object. This implies that we do not have to lock and unlock any object for critical sections. Even though this provides more fine-grained mutual exclusion, there is a substantial disadvantage of using this package. Read and write can be performed individually but not both on the set of data structures provided. Critical section in BetterSafe includes both reading and writing. We first read the value to check if it is in the expected range and then write the value. If another thread run in between these two and changes the value, our results would be incorrect. Hence, data structures in this package cannot be used to implement the class BetterSafe.

1.4.3. java.util.concurrent.locks

This package provides numerous lock classes to protect the critical section. Locks are used to lock the entire critical section like synchronized keyword but there is a significant difference between the two. Synchronized treats the entire method as a critical section and prevents any other thread from accessing it. Locks are used to lock only the shared memory space and prevent other threads from making changes to the shared memory. Lock package is straightforward and easy to use. Since only critical section in the swap method of BetterSafe class is required to lock, this package fits the need. It helps achieve another goal of decreasing the overhead of using the synchronized keyword. So, Reetrant lock is used to implement the critical section in BetterSafe class for reliability and better performance than the Synchronized class.

1.4.4. java.lang.invoke.VarHandle

The VarHandle class is just like AtomicIntegerArray class with the exception that it just doesn't work for integers but other types as well. This class would let us synchronize an array of objects which aren't integers like bytes. We could perform atomic operations on an array of bytes but it still does not help us resolve the issue we faced with AtomicIntegerArray. Between reading and writing to an index an array, another thread can come in and modify the value at that index leading to incorrect results. Since this class does not protect the entire critical section which is needed for BetterSafe class, this class cannot be used as a substitute for synchronized method.

2. Synchronized

This class is implemented from the State interface. It is Data Race Free (DRF) since the keyword synchronized is used to prevent from multiple threads from performing the method having critical section at the same time. This prevents any correctness issue that might arise from running multiple threads for this class. However, it does not utilize multiple threads for increasing performance. On the contrary, it slows down the process since running thread has its own overhead and there is nothing done in parallel to increase the efficiency.

3. Unsynchronized

This class is implemented from the State interface. Unsynchronized class is very similar to the Synchronized class. The only exception is that the swap method in this class does not have the keyword synchronized which prevents multiple threads from performing the method having critical section at the same time. This makes the class Non-DRF since multiple threads can simultaneously access the value at the same index in the array and modify them. Multiple operations would happen at the same time leading to incorrect results. When the Unsynchronized object is used to run UnsafeMemory for 8 threads and a million transactions, it goes into an infinite loop indicating failure.

4. GetNSet

GetNSet class is implemented from the State interface. Package java.util.concurrent.atomic.AtomicIntegerArray was used for the implementation of this class. Get and Set methods of AtomicIntegerArray were used to access the values at a particular index in an array and modify the values at that index. Even though get and set are atomic operations. There is still plenty of room for race conditions. When one thread reads a value at an index and decides to modify them, another thread could come in and change the value at the same index before that thread could change the value. This could make the value go out of bounds and give incorrect results. Thus, this class is Non-DRF. When the GetNSet object is used to run UnsafeMemory for 8 threads and a million transactions, it goes into an infinite loop indicating failure.

5. Null

Null class is implemented from the State interface. This class has no implementation for swap function. It simply returns a value with the correct return type and does not perform any computation at all. This class is only used for comparison of performance of different objects. The average time for each transaction for Null object could tell the programmers the time it takes for the scaffolding of the simulation.

6. Test Results

Objects of different methods were created to test the performance of their class under different circumstances.

Table 2 shows the time it takes for each transaction in nanoseconds (ns) for different classes. The tests were for 1, 8 and 32 threads with 1000000 (million) transactions for each object. Each test was run thrice and then the average was calculated.

	1 Thread	8 Threads	32 Threads	DRF
BetterSafe	93.927	1097.6	4713.5	Yes
Synchronized	43.69	1625.7	6601.5	Yes
Null	30.8	720		Yes
Unsynchronized	35.9	-	-	No
GetNSet	51.3	-	-	No

Table 2: Average Time in ns for each transaction for a million transactions (swaps)

7. Conclusion

The explanation provided in the Package analysis for BetterSafe class gives the comprehension for using Reetrant lock in the class. It provides higher reliability and better performance than other available options.