CS 131 Homework 3 Project Report

Abstract

The purpose of this lab was to exploit different mechanisms of synchronizing a simple Java program. The interface described a class containing an array and a method that allowed a program to add 1 to one element of the array and subtract it from another. I created different mechanisms of synchronizing this method (including one version that did not synchronize them at all, to see which of these different mechanisms would perform faster and still work reliably.

1 Class Implementations

1.1 State Interface

The root interface from which all future classes described here is the State interface that was in the jar file given to us. It specifies three methods: one to return the size of the array, one to return the array itself, and one to "swap" elements. The "swap" method takes in two integers, i and j. If the ith element is positive and the jth element is less than the maximum possible value, the function subtracts one from the ith element of the array and adds one to the jth element. Upon success, swap should return true, and upon failure, swap should return false.

1.2 Null State

The NullState class is an implementation of State meant to be a "dummy" class. Its implementation of the swap method does nothing except return the value true. It is used as a baseline for timing the rest of the program and generally "passes" all the test cases in that there is never a sum mismatch (because the swap method does not do anything).

1.3 Synchronized State

The SynchronizedState class is a correct implementation of State that was given to us. It uses the synchronized keyword for the swap method in order to prevent race conditions. Adding synchronization to this class causes the program to run more slowly when it is multithreaded. The goal of this lab is to see if synchronizing the swap method in this way is necessary.

1.4 Unsynchronized State

The UnsynchronizedState class is an incorrect implementation of State written by me as described in the spec. The code for this class is exactly identical to the code for the SynchronizedState class; the only difference is that the UnsynchronizedState class does not use the keyword `synchronized` for the swap method. This causes the program to have unhandled race conditions when multithreaded, as multiple threads accessing the critical section of the swap method's code may incorrectly alter the array. Although the program runs significantly faster using the Unsynchronized State class, it does not guarantee correct behavior, as at the end of the program the sum is often mismatched. See Summary of Test Results for specific details.

1.5 GetNSet State

The GetNSet class is an implementation of State that is partially synchronized and partially unsynchronized. It does not use the `synchronized` keyword, but instead uses an AtomicIntegerArray. We were told to use the `get` and `set` methods, which update the array atomically. However, because we used `get` and `set` individually, rather than `getAndIncrement` and `getAndDecrement`, the getting and setting happened separately rather than atomically. The processor may still interleave thread instructions such that there are unhandled race conditions occurring between our calls to `get` and `set`, resulting in the mismatched sum errors described in the Summary of Test Results.

1.6 BetterSafe State

The BetterSafe class is yet another implementation of the State interface. In this class, the goal was to write a class that would perform better than the original Synchronized class, but that would still behave correctly when multithreaded and gracefully handle race conditions. I decided to implement this class using Java's `ReentrantLock` class, which is part of the locks package. In my implementation, I apply a lock in the swap method. Just before the critical section, I call the lock() method on my lock, and just before returning from the method, I call the unlock() method on my lock. I found that implementing the BetterSafe State in this way resulted in exactly what we were hoping to achieve: as described in the details of the Summary of Test Results, it not only performed faster than Synchronized State; it also had no

sum mismatches in any of my test cases, providing strong evidence that it correctly prevented any race conditions.

2 Summary of Test Results

2.1 Raw Data

In order to test the different implementations of the State interface, I wrote a brief bash script that I called test.sh. It ran each version of State with 1, 2, 4, 8, and 16 threads, using 1000000 transitions and arrays filled with random numbers.

Below is a summary of the output of my program. The elements of the table signify the average number of nanoseconds per transition. For example, the null model with one thread averaged 37.20 nanoseconds per transition.

Threads	Null	Synch	Unsynch
1	37.20	73.75	41.50
2	84.23	231.1	116.267*
4	283.6	1057.36	575.46*
8	1684.35	1962.75	1748.39*
16	4589.64	3755.47	3965.54*

Threads	GetNSet	BetterSafe
1	57.85	67.99
2	404.29*	484.69
4	643.65*	527.67
8	1287.68*	1119.35
16	2971.33*	2424.93

*Tests marked with an asterisk also had a sum mismatch error due to not synchronizing the swap method (or doing so incorrectly).

I attempted to run this code using 32 threads and compare how each of the States performed in that case, but unfortunately the server complained that it could not run the program with so many threads due to a lack of memory space.

2.2 Observations

Looking at this table containing the average number of nanoseconds per transition depending on the number of threads and which version of the State class we are using, there are a few observations that we can immediately make.

The first observation is that for the Unsynchronized State and the GetNSet State (both of which are described above as having problems with how they handle synchronization), every multithreaded test case results in a sum mismatch error, which indicates that the unhandled race conditions caused the program to behave incorrectly.

Additionally, if you compare The Unsynchronized State data to the Synchronized State data, the Unsynchronized State version of the code runs faster in every test except when there are 16 threads. A similar pattern can be observed when comparing the BetterSafe State data with the Synchronized State data.

However, BetterSafe State not only performed faster than Synchronized State for the most part, it also did not throw any mismatched sum errors like some of the other States did, which provides strong evidence that it correctly handled race conditions.

Another observation is that BetterSafe appears to perform faster than Unsynchronous State (except in the cases that the number of threads is 1 or 2). For larger threads, BetterSafe is faster and more reliable, so there is no reason to use the Unsynchronous State at all. A similar pattern can be seen when comparing the BetterSafe State to the GetNSet State; BetterSafe is faster for a larger number of threads and also more reliable in terms of handling race conditions.

3 Packages

3.1 java.util.concurrent

This package is one possible option that we could have used for the BetterSafe class. An advantage of this package is that we have more control over memory when synchronizing because we are able to use lower level data structures. A disadvantage is that customizing these data structures makes it more difficult to understand and implement the code.

3.2 java.util.concurrent.atomic

This package allows us to perform instructions atomically. An advantage of this is that we do not have to worry about locking and unlocking critical sections of code. However, we need to perform reads and writes. This package does not allow us to read and write atomically; we would still have to do each of these actions individually, leading to race conditions.

3.3 java.util.concurrent.locks

This is the package that I ultimately decided to use because the locks provided in this package are simple and customizable. However, a disadvantage of this package is that the lock locks the entire object when we really only need to lock the two entries of the variable we are accessing and modifying. Despite this performance disadvantage, the ReentrantLock from this package still ended up performing well enough for our purposes (it was still faster than Synchronized State).

3.4 java.lang.invoke.VarHandle

This package is similar to java.util.concurrent.atomic in that it allows us to modify objects atomically. An advantage is that we can perform our reads and writes atomically; however this package allows interrupts which may lead to race conditions.

4 Challenges

The most challenging part of this project was to test the different classes on the server. Writing the code itself was fairly straight forward, but finding the best way to test it was more involved, as the same command with the same number of threads and transitions often had different results. In order to get consistent results, I had to increase my number of transitions to 1,000,000. I also ran the same test multiple times and averaged the results in the hopes of getting data that was more representative of an overall pattern.

5 Conclusions

Based on the classes and their performance in the tests, I would recommend GDI to use the BetterSafe State in order to improve performance. I would recommend that GDI not simply use the Unsynchronous State or the GetNSet State because they result in both having an incorrect program and appear to not perform as quickly as BetterSafe for a large number of threads.