Homework 3 Report - Java Shared Memory Performance Races

Abstract

For this homework assignment, we were in charge of figuring out the tradeoffs between performance and reliability in a Java program.

1. Introduction

We are required to implement the best class for concurrency and reliability for an imaginary company named, Ginormous Data Inc. (GDI). We are given a program that performs an operation many times on an array. This operation, called "Swap", takes two integers as input, and they represent indices of the input array. Then, "Swap" will increment the value at one of the indices and decrement the value at the other index. The program lets us configure the way we run it. For example, we can specify a number of threads to run the program with, or the number of operations to perform, or even the numbers inside the starting array. Additionally, we can specify a particular class in which to run the program. We can use NullState, which essentially does nothing, or Synchronized, which uses the Java keyword, "synchronized" in order to eliminate race conditions in the code. We were then asked to implement an Unsynchronized class, in which race conditions were allowed to occur as many times as possible. We were also asked to implement an "AcmeSafe" class which performed better than the synchronized class, but also maintained the same correctness.

2. Experiment

I conducted the experiment by first modifying the synchronized class file and removing all of the "synchronized" keywords. Then, I verified that this version of the program, named "Unsynchronized", was faster than the synchronized version, but also ran into synchronization issues.

Then, I began to construct the "AcmeSafe" version of the class. My first attempt involved using the data structure, "AtomicIntegerArray", provided by the java.util.concurrent.atomic library. However, when I ran this version of the class, the time per transaction was in fact slower than that of the synchronized class.

As a result, I began to work on a new implementation of the "AcmeSafe" that used lock from the java.util.concurrent.locks.ReentrantLock library. From there, I began to collect data.

3. The Data

The following tables show the data that was collected during the experiments. Each table represents the number of swaps that had to be done. Along the y-axis are the classes. Along the x-axis are the number of threads running. The data shows the approximate time per transition in thousands of nanoseconds. Each experiment was run with the same array and max value.

10 000 Swaps

	1	**2**	**4**	**8**	**16**	**32**
NullState	0.4	1	4	10	17	40
Synchronized class	0.4	1.5	3	12	20	60
Unsynchronized	0.4	1*	5*	13*	21*	45*
AcmeSafe (using Atomic Integer Array)	1	3	7	20	45	90
AcmeSafe (using locks)	1	3	6	11	35	85

100 000 Swaps

	1	**2**	**4**	**8**	**16**	**32**
NullState	0.1	0.3	0.8	2	6	10
Synchronized class	0.1	0.7	2	4	7	13
Unsynchronized	0.2	0.4*	1*	4*	-	-
AcmeSafe (using Atomic Integer Array)	0.2	1	2	10	20	30
AcmeSafe (using locks)	0.2	1	1	2	7	13

Homework 3 Report - Java Shared Memory Performance Races

1 000 000 Swaps

	1	2	4	8	16	32
NullState	0.02	0.1	0.3	1	3	6
Synchronized class	0.05	0.5	0.7	1	3	7
Unsynchronized	0.04	-	-	-	-	-
AcmeSafe (using Atomic Integer Array)	0.06	0.3	1	2	8	15
AcmeSafe (using locks)	0.07	0.3	0.3	0.8	2	4

* = there was a sum mismatch, ie. concurrency issues
- = could not run all the way, program left hanging

4. Analysis - Four Different Implementations

4.1 Synchronized

The Synchronized class was given to us, and it was an implementation that used the Java keyword, "synchronize". This implementation provided a slow but correct way of performing swap operations.

4.2 Unsynchronized

The unsynchronized class was created by removing the "synchronize" keyword in the synchronized version of the class. This implementation disregarded all concurrency issues, allowing race conditions to occur without restriction. As seen in the data, some of its runs did not even complete, as the program hung for over a minute, at which point I stopped it manually. Furthermore, any time it ran with more than 1 thread, there would be concurrency issues, which were outputted by the main program. This implementation is not desirable, since we cannot multithread the application due to concurrency issues.

4.3 AcmeSafe (Atomic Integer Array)

This implementation of "AcmeSafe" used the "AtomicIntegerArray" library from java.util.concurrency. This implementation was not good, because while it maintained correctness and reliability, it performed even slower than the synchronized class.

Another downside to using this approach would be writability. Because I had to keep the interface the same, I had to write a few loops in order to convert the "AtomicIntegerArray" to the byte array. For these two reasons, this implementation is undesirable.

4.4 AcmeSafe (locks)

This implementation of AcmeSafe is probably the most interesting. With a lower number of operations, it almost always performed slower than the synchronized version. However, as the number of operations increased, the speed of this implementation increased drastically. If we had a million operations, this implementation would even be faster than the implementation that used the Atomic Integer Array. This is probably because of the overhead associated with acquiring locks. With a smaller number of operations, this overhead becomes more apparent. The ratio of time taken to grab the lock to the time taken to complete operations is greater than the same ratio with a larger number of operations.

4.5 Analysis - Other Implementations

There were two other options for implementing synchronization, but I picked the above two implementations for AcmeSafe. I did not pick semaphores because I did not feel the need to use such a lock. A semaphore records the number of threads accessing a particular data point. However, since we cannot even have two or more threads at the same data point, there would be no use for a semaphore. A simple lock would suffice.

Another option for synchronization would be the atomic integer, which was also provided by "java.util.concurrency". I did not need this because there already existed the Atomic Integer Array. This is simply an array of atomic integers and has essentially the same function as if I used the atomic integer type for each value in the array. These two implementations would probably perform similarly.

4.6 Analysis - Issues

It was very confusing to record measurements, because each measurement could easily be confused

with another. For example, the number of operations I used differed by a factor of 10, which is a mere ‘0’ to the human eye. Sometimes I would accidentally input data into the incorrect table because of this.

Additionally, results were sometimes inconsistent, so I had to run the test multiple times before I got a definitive answer.

5. Conclusion

In terms of speed, unsynchronized ran the fastest, followed by AcmeSafe and then synchronized. In terms of reliability, AcmeSafe and synchronized were the same, providing very good reliability, while unsynchronized provided poor reliability. Both of my implementations of AcmeSafe are data-race-free (DRF). In theory, they are DRF because one of them uses atomic operations, and the other uses locks. Thus, there is no possibility for concurrency issues to occur. This held true in practice too, because I did not get any sum errors when I ran the experiment with my AcmeSafe classes.

In conclusion, GDI should pick AcmeSafe class, implemented using locks, for their applications. This is because it provides good reliability as well as decently fast speeds.