CS 131 Homework 3:
Java Shared Memory Performance Races

Abstract

In this assignment we examine Java shared memory models and concurrency, especially in the context of performance. We use a simple prototype to test the performance and reliability of various synchronized and unsynchronized Java classes. Overall, we find that

1 Introduction

For this investigation, our prototype implements a Java interface that manages an array of longs called State. Our State interface describes a "swap" method which, given two valid indices i and j, increments the array's value at index i and decrements the value at index j. If the class acts as expected, the sum of all array elements should remain 0, since the array's elements are initialized to 0, and all increment operations are paired with a corresponding decrement operation.

However, we find that if our code is run concurrently by multiple threads, this may not be the case due to data races occurring on the shared memory which represents the array. In order to ensure the reliability of our program under multiple threads, we can use synchronization, which locks certain sections of the code during execution to prevent data races.

Unfortunately, synchronization often results in significant decreases in performance. Thus, we attempt to find an equally reliable solution that is more performant than pure synchronization, which we call "AcmeSafe".

2 Background

Before diving into implementations and testing, we should establish some background on multithreading concepts such as data races, synchronization, and the Java Memory Model (JMM). In the ACM Queue magazine, Hans-J Boehm and Sarita V. Adve wrote an excellent resource on these topics, whose contents are briefly summarized below. [1],

2.1 Multithreading and Data Races

Multithreading refers to the ability of a processor to use multiple paths of execution when executing a program. Threads allow for the concurrent execution of multiple parts of a single application, thereby increasing the application's responsiveness or overall speed.

As alluded to briefly in Section 1, threads operate on a shared memory space, theoretically allowing efficient use of physical memory resources as well as easy and quick communication between threads. However, this also introduces a core problem in concurrency and multithreading: data races.

A data race refers to the situation in which multiple threads attempt to access a shared variable at the same time, often introducing unexpected and undefined behavior if unaccounted for.

For instance, Boehm and Adve present a situation where two threads both increment a value [1]. Normally incrementing a value consists of the following lower-level instructions: a read instruction, the intended operation (increment in this case), and a write instruction. If two threads are both to perform an increment operation on the same variable, the intended result is that the variable's value increases by 2. However, it's possible that the first thread is interrupted after its read operation, allowing the second thread to perform its read operation. In this case, both threads would eventually write the same value to the variable, causing the value to increase by only 1.

Clearly, we need some way to tell the processor about some restrictions on the order in which threads execute instructions. We discuss these solutions in Section 3.

2.2 JMM

With increased need for high-performance computing, multithreading is quite prevalent, causing many popular programming languages to specify certain interfaces for writing multithreaded applications. In this investigation we use Java, which provides two main thread-level interfaces: the Thread class

and the Runnable interface. In our case, we use the Runnable interface in order for our implementation to still extend our basic State class. In order to simplify our various State implementations, all the thread-level programming is taken care of in our test harness.

In order to assess various possible implementations of our AcmeSafeState, we must first understand the JMM, which specifies how shared memory behaves under concurrent accesses. The JMM's abstraction of shared memory consists of the thread stacks and the heap. The thread stacks contain thread-local data such as copies of local primitive variables and the call stack, while the heap contains objects and their member variables. Any object on the heap can be accessed by any thread with a reference to that object.

This abstraction of physical memory diverges from the actual hardware in a couple key ways. First, the hardware doesn't actually distinguish between the thread stack and heap: both are located somewhere in memory.This means that it's possible that updates made by one thread may be stored in CPU registers or caches, invisible to other threads. On the other hand, if variables are stored in shared memory, we run into the data race problem.

Perhaps the most relevant part of the JMM, then, is the data-race freedom (DRF) guarantee, which guarantees the sequential consistency of a program as long as that program properly prevents data races. Specifically, two accesses to the same location may not "conflict." A conflict is defined by Lochbihler as follows:

> Two accesses to the same location conflict if (1) they originate from different threads, (2) at least one is a write, and (3) the location is not explicitly declared as volatile" [3].

The DRF guarantee is critical to creating a reliable application. We will explore these requirements and how we fulfill them in the next section.

3 Implementation

3.1 Volatile Access Mode

One of the conditions of a conflict enumerated in the previous section mentions a location being declared as "volatile." Volatility actually refers to a Java memory mode; Doug Lea's "Using JDK 9 Memory Order Modes" is an excellent resource on this topic [2]. Lea discusses 5 modes: Plain, Opaque, Release/Acquire, and Volatile. For this purpose, the most important is volatile, which is used to indicate that a variable is being shared across threads. Lea writes,

> Volatile mode [...] adds [...] the constraint:
>
> *Volatile mode accesses are totally ordered.*
>
> When all accesses use Volatile mode, program execution is sequentially consistent, in which case, for two Volatile mode accesses A and B, it must be the case that A precedes execution of B, or vice versa. In [another] mode, they might be unordered and concurrent.
>
> The main consequences are seen in a famous example that goes by the names "Dekker", "SB", and "write skew". Using "M" to vary across modes:

```
volatile int x, y; // initially zero

Thread 1                |  Thread 2
X.setM(this, 1);        |  Y.setM(this, 1);
int ry = Y.getM(this); |  int rx = X.getM(this);
```

> If mode M is Volatile, then across all possible sequential orderings of accesses by the two threads, at least one of rx and ry must be 1. But under some of the executions allowed in [other] modes, both may be 0. [2]

One important consequence of the restrictions imposed by volatile is that updates to volatile variables must be written directly to shared memory; they may not be held in thread-local CPU caches or registers. Thus, the *volatile* keyword solves the visibility problem outlined above.

3.2 Synchronization

However, notice that the volatile keyword doesn't solve the other main problem: data races of the read-update-write variety outlined previously. For this problem, the common solution is thread synchronization, which is a mechanism that prevents data races by ensuring that no threads simultaneously access any portion of the code that could cause a data race. These sections are known as critical sections. From the official Java documentation:

> When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

Though simple and reliable, the use of synchronized methods, as mentioned before, presents issues. The one-at-a-time approach somewhat defeats the purpose of multithreading, often decreasing its performance gains.

3.3 AcmeSafeState: A Compromise

The goal in developing AcmeSafeState was to find a medium between volatile variable modes, which don't fully ensure the reliability of our State class, and synchronization, which slows down execution. Researching some of the

packages mentioned in the project specification led to the *java.util.concurrent.atomic* package, described in its documentation as a "small toolkit of classes that support lock-free thread-safe programming on single variables." Indeed, this was exactly what we needed: the ability to perform our increment and decrement operations in our swap method *atomically* without using synchronization or locking. The documentation explains further:

> In essence, the classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update operation of the form:

```
boolean compareAndSet(expectedValue,updateValue);
```

> This method (which varies in argument types across different classes) atomically sets a variable to the updateValue if it currently holds the expectedValue, reporting true on success. [...] compareAndSet and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables. [4].

Thus, we see that the *atomic* package uses the conditional update method *compareAndSet* to give us non-blocking, atomic operations on volatile variables. Additionally, it provides us various classes and methods to support different types and operations: namely, we can use the *AtomicLongArray* since our State has a member array of longs, and its *getAndIncrement* and *getAndDecrement* methods to perform our swap operation. The documentation is abundantly clear that these **operations will be DRF** as long as our "critical updates for an object are confined a single variable," which they are, so I implemented AcmeSafeState's swap as follows:

```
public long[] current() {
    current_value = new long[value.length()];
    for (int i = 0; i < value.length(); i++) {
         current_value[i] = (long)value.get(i);
    }
    return current_value;
}
```

One drawback of this approach is that our abstract State class expects a member field of type long array, not AtomicLongArray, so when we have to "convert" to a long array when we return our current value. Still, it's likely that the benefits of the non-blocking approach more than compensate for this issue.

3.4 Remaining Implementations

In the project specification we were given implementations for NullState, a "dummy" class used to measure scaffolding time and SynchronizedState, a synchronized implementation. Along with AcmeSafeState, we were tasked with implementing UnsynchronizedState, a class without any sort of synchronization. To do so, we just removed the synchronized keyword from the SynchronizedState implementations. Additionally, we added options for these new classes into our testing harness.

4 Testing

4.1 Hardware

We test our implementations on two different servers, lnxsrv06 and lnxsrv10, with the following processor info:

- lnxsrv06 has 32 Intel(R) Xeon(R) CPUs E5620@2.40GHz, each with 8 cores.
- lnxsrv10 has 4 Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz, each with 4 cores.

Both servers are running Java 13.0.2.

4.2 Methodology

In the initial part of the project, the specification recommended running tests on NullState and Synchronized State with varying numbers of threads (1,8,40, etc) and array sizes (5, 100, etc). Thus, for the sake of comparison, I decided to use these same testing parameters when testing each state. Since testing every possible combination of threads and sizes seemed excessive, I decided to only change one variable at a time.

My default configuration was 8 threads and an array size of 5 like the example given in the specification. From there, I kept the array size constant while varying the number of threads between 1, 8, 16, 24, 32, and 40. Likewise, I kept a constant number of 8 threads while varying the state array size between 5, 25, 50, 75, 100, and 500.

The test harness outputted a number of statistics, including real, CPU, sys and average swap time. Since our goal for the project was to optimize performance, especially in terms of time, we use average swap time as the most relevant metric to assess performance.

The tests were run at once using a basic shell script, and the output was parsed using a Python script that kept a running average of the results in a .csv file.

4.3 Challenges

The most significant challenge was the noise level of the data. Due to varying server loads, timing results were quite inconsistent. Hopefully, running many trials helped alleviate this issue.

5 Results

The results for average swap time in nanoseconds of all of our different configurations is given in the tables below:

Table 1: lnxsrv06: Average Swap Time vs. Number of Threads

	1	8	16	24	32	40
AcmeSafe	22.721	760.5	1224	2254	2633	1899
Null	14.20	25.63	38.50	65.41	129.0	159.4
Unsynchronized*	16.03	209.8	420.3	688.9	920.6	1345
Synchronized	23.10	1967	3789	5535	7215	10032

Table 2: lnxsrv06: Average Swap Time vs. State Array Size

	5	25	50	75	100	500
AcmeSafe	802.2	672.1	674.4	577.2	532.6	437.6
Null	27.34	26.76	22.55	22.52	25.86	23.34
Unsynchronized*	323.2	451.2	412.3	391.3	388.0	310.2
Synchronized	2104	2056	2234	2105	1969	2139

Table 3: lnxsrv10: Average Swap Time vs. Number of Threads

	1	8	16	24	32	40
AcmeSafe	16.45	614.1	832.3	1231	1653	2083
Null	11.41	29.5	77.92	98.2	158.5	213.2
Unsynchronized*	12.09	277.7	557.9	519	1191	1448
Synchronized	25.47	917.3	1252	1605	1914	2141

Table 4: lnxsrv10: Average Swap Time vs. State Array Size

	5	25	50	75	100	500
AcmeSafe	812.6	405.5	389.2	406.9	374.6	311.3
Null	32.49	29.63	31.21	31.33	29.60	32.98
Unsynchronized*	245.5	331.5	313.6	293.2	278	204.1
Synchronized	804.2	737.3	528.5	560.7	392.5	313.7

Note: * indicates an output sum mismatch

6 Analysis

Although the data is still somewhat noisy, running over 30 trials of every test really helped provide some consistency, allowing us to see a couple trends.

First, we see that in the multithreaded cases, the Unsynchronized implementation consistently leads to an output sum mismatch, meaning that the sum of the member array is not 0 at the end of the test. As mentioned before, this indicates a data race without proper thread synchronization since every increment operation is paired with a corresponding decrement operation. Thus, despite the high speed of this implementation, we know that it is completely unreliable.

On the other hand, Synchronized, AcmeSafe and Null (it doesn't update any values), had no output sum mismatches as expected, supporting our hypothesis that **AcmeSafeState is DRF**.

Next, in terms of hardware we see that, overall, the implementations performed better on lnxsrv06 than lnxsrv10.

In addition to these relatively unsurprising results, some of the results initially seem counterintuitive.

- For every implementation, the single-threaded implementation is the fastest. Average swap time generally increases with increased thread count
- The small array size is the slowest. Average swap time generally decreases with increased state array size.

These results are explained below:

6.1 Threads

The fact that an increased number of threads decreases performance makes little sense at first. After all, the entire purpose of multithreading is to increase performance. However, we must remember that using multiple threads always induces some overhead. When switching between threads, the processor must perform a *context switch*. These context switches, in addition to thread scheduling, certainly take a toll on performance.

In most multithreaded applications, the assumption is that the performance boost added by the parallel execution outweighs this overhead, but in our case, which simply involves increment and decrement operations, this assumption is likely valid. If we delve a little deeper into our results, we see that the sys time is highly correlated with thread count, supporting this explanation.

6.2 Array Size

From a purely instinctual perspective, it would seem that increased state array size would decrease performance, and from a slightly more educated perspective, we would think that the array size would have no effect because arrays allow for constant-time access. Additionally, smaller array sizes may even lead to slightly increased performance due to an increased likelihood of cache hits. Indeed, this non-dependence on array size is the case for the Null and Unsynchronized implementations, but not for AcmeSafe and Synchronized.

Understanding these results requires some deeper logic involving the nature of multithreaded processing. For the Synchronized and AcmeSafe implementations, the threads are prevented from conflicting (i.e. trying to update the same data). Even in AcmeSafe, which is generally supposed to be non-blocking, only one thread can perform the update at a time.

It follows, then, that for these implementations, conflicts would increase overhead due to this "wait time". Logically, the likelihood of conflicts decreases with increased array size:

the change of two threads picking the same array index between 1 and 5 is much greater than the likelihood of doing so between 1 and 500. This perfectly explains ours results: for the implementations that employ some type of thread synchronization, larger state arrays means less conflicts, and thereby greater performance.

6.3 Hardware

The final variable we tested was the hardware. The specifications stated earlier imply that lnxsrv06 was more powerful than lnxsrv10. Indeed, the average swap time seemed to be generally lower on lnxsrv06 than lnxsrv10, but one curious difference was that there were different results *between* the implementations on each server. For instance, on lnxsrv06, AcmeSafe was quite fast: the average swap time was much lower than that of Synchronized. Meanwhile, on lnxsrv10, AcmeSafe was barely faster than Synchronized; in fact, it was slightly *slower* in some cases.

To make sense of this, we can look at the constraints imposed by each set of hardware. As established previously, lnxsrv06 has 32 processors with 8 cores each, while lnxsrv10 has 4 processors with 4 cores each. First and foremost, this means that lnxsrv06 is better suited to handle the load created by many students using the server. More importantly, though, is that this abundance of processors and cores lends itself much better to multithreaded applications. In order for us to truly see the difference between the non-blocking implementation in AcmeSafe and the pure method synchronization in Synchronized, we need a computing setup that allows for the concurrent execution of many threads. Otherwise, the threads are being run sequentially. This means that we don't even get the expected performance boost which AcmeSafe provides by allowing more efficient concurrent execution.

7 Conclusion

Overall, our implementation of AcmeSafeState was successful. It proved to be DRF both theoretically and empirically and more performant than SychronizedState, especially on lnxsrv06. Outside of this, the results were generally explainable with some background on concurrency.

Unfortunately getting accurate results required the execution of countless trials, as there was such a wide variation between different runs. Somehow limiting this issue by giving students a better environment for this use case with less variable server loads would likely offer a great improvement to this assignment. Without writing shell and Python scripts to automate the testing process, it likely would've taken an unreasonable amount of time to run the number of tests used in this submission.

References

[1] Adve SV Boehm H-J. You don't know jack about shared variables or memory models. *ACM Queue*, 9, 2011. https://queue.acm.org/detail.cfm?id=2088916.

[2] Doug Lea. *Using JDK 9 Memory Order Modes*. SUNY Oswego, November 2018. http://gee.cs.oswego.edu/dl/html/j9mm.html.

[3] ANDREAS LOCHBIHLER. Making the java memory model safe. Technical report, ACM Trans Program, December 2011. https://pp.info.uni-karlsruhe.de/uploads/publikationen/lochbihler14toplas.pdf.

[4] Oracle. *Package java.util.concurrent.atomic*. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html.