CS 131 Homework 3: Java Shared Memory Performance Races

University of California, Los Angeles

Abstract

In this assignment, we are acting as if we are working for a company that handles ginormous data sets and is wondering about the efficiency and reliability of their multithreaded Java programs. We are tasked with exploring and reporting on the tradeoffs between efficiency and reliability in a multithreaded program written in Java that increments an element of an array of longs and decrements an element. As a classic example of potential race conditions, we use this program to test different implementations and their efficiencies while managing the reliability potential data races of multithreaded programs.

1. Introduction

We were given a program that tries different implementations of State objects which each contain an array of longs whose elements add up to zero and a "swap" function. The swap() function increments one value of the array and decrements another. As they can be the same or different elements in the array, the sum of the array *should* continue to be zero after every swap. However, when this task is needed to be performed many times efficiently, multithreaded programming is the answer. To improve speed and efficiency, we split up the swaps into multiple threads that will execute in parallel on the same State object. This greatly reduces our execution time and allows us to do the same amount of swaps in a fraction of the time. However, with multiple threads accessing the same object, problems arise in concurrency and visibility that may cause the array of longs to sum up to something other than zero.

To explore these possibilities, we tried different implementations that use methods of managing memory access and synchronization in multithreaded programs and recorded each program's efficiency and whether it resulted in the correct 0-summed long array output as expected or not. We tested four implementations of State. A NullState, which essentially does nothing, an UnsynchronizedState, which does not utilize any synchronization or multithreaded protection, SynchronizedState which uses the keyword *synchronized* on the swap function, and AcmeSafeState, which implements the array of longs with an AtomicLongArray from the java.utils.concurrent.atomic package.

1.1. Background

According to "Using JDK 9 Memory Order Modes" by Doug Lea, there are three main types of multiprocessing, each with benefits and hazards we must navigate as programmers. These include task parallelism, where multiple ordered tasks run on two different cores/CPUs concurrently without a distinct order. There is also memory parallelism where multiple different parallel processes can read, write, interact with, and access the same objects or variables in a shared memory space. Lastly, there is instruction parallelism where multiple instructions can be carried out at one time by a single system. While these types of multiprocessing improve our efficiency and reduce the amount of time our programs have to run, they come with challenges and pitfalls that require the use of constraint mechanisms that each protect against certain multithreaded errors but reduce the potential parallelism [1].

Problems that programs can run into while performing multithreaded operations include visibility problems, synchronization problems, and__. Visibility problems occur when one thread depends on the value of a variable being changed by another thread in order to run. As threads sometimes store variable information in their own private local cache, some changes made by one thread to a variable could only be seen in their local cache and not reflected in the shared memory, thus not allowing the other thread to run properly. In Java, this can be aided with the keyword "volatile." When this keyword is applied to a variable, it forces changes to that variable to be visible to all other threads running. However, "volatile" doesn't fix all problems as we could still run into a synchronization problem. A synchronization problem occurs when multiple threads update a variable at the same time non-atomically. Even with volatile, each thread needs to read the value of the variable, change it, then write it back. If these steps overlap between threads, we see incorrect values of the variable.

#	Thread-1	Thread-2
1	Read value (=1)	
2		Read value (=1)
3	Add 1 and write (=2)	
4		Add 1 and write (=2)

Figure 1: Example of a synchronization problem [3]

Problems such as these can be fixed by the "synchronized" keyword in Java which ensures one thread can execute a certain method in an object at a time, ensuring that operations on the same variable will not be performed concurrently, but slowing down parallelism as this synchronized code cannot be executed in parallel with itself. Another method of solving this problem is using an atomic variable in which the

adding/subtracting function is instantaneous and there is no separation between the read and write. [3]

Adding constraint to a program aids the programmers in making their code more reliable, and data race safe. However, this comes at a trade off with performance and the amount of parallelism that can be taken advantage of. Doug Lea lays out the different levels of constraint with the memory order modes in Java's JDK 9. Memory order modes, provided by VarHandles, each add a different level of constraint to multithreaded programs. From least to most restrictive, these include: Plain mode, Opaque mode, Release and Acquire mode (R/A), and Volatile Mode.

Plain mode is the weakest constraint mode, being essentially "as-if-serial." This mode allows the compiler to make optimizations within a thread by switching the order of execution of the code to improve single threaded performance. This leaves the possibility for the compiler to skip some reads or writes if it believes the value of a certain variable will not change in one thread. The issues that arise are when we run this with multiple threads, the compiler optimizations could have compromised the integrity of the code meant for multithreaded processing. Plain mode is the fastest and most efficient memory order mode, but comes with the undesired side-effects when used with multithreaded programs. [2]

Opaque mode is the next mode up, adding some constraints to plain mode. Opaque mode is aware that variables can be updated, written, and read in different threads. Because of this, opaque mode guarantees that updates to the same variable are seen in order. Other variables can be optimized by the compiler. [1]

Release and Acquire Mode adds additional constraints on top of opaque mode. In this mode, threads release control and other threads can acquire control. This release and acquire of control ensures that all instructions are executed before the release and that all previous instructions have been executed before acquiring. This adds a "happens-before" causality constraint to a program. The compiler can do optimizations on code between the release and acquire. [2]

In volatile mode, the strongest memory order mode, all access are ordered. This is equivalent to using the volatile keyword in Java. While this is the safest mode and ensures all operations are ordered, it greatly reduces the parallelism potential and therefore causes the program to run much slower. [1]

1.2. AcmeSafeState Implementation

To implement AcmeSafeState, we used the Java package java.util.concurrent.atomic.AtomicLongArray which uses a mixed mode memory order. We implement AcmeSafeState with an AtomicLongArray because we want our multithreaded program to be safe and deliver correct results while being more efficient than the slow SynchronizedState implementation. We utilize an atomic array of longs, so each increment and decrement is atomic and cannot be split, so we do not run into any synchronization problems. The AtomicLongArray implementation ensures that each element of the array is volatile, so updates to an element are seen by other threads before they perform an additional operation on it, keeping the values consistent. The atomic functionality of the increment and decrement methods ensure that threads are not stepping on each other's toes and will perform operations on elements without fear of inconsistencies. Because each thread will always see an updated version of each element when performing an operation on it (and can perform this operation without fear of synchronization problems), the implementation of AcmeSafeState is data race free. This combination of memory order modes allows the implementation to maintain to be safe while being faster than the synchronized version. Multiple threads can be executing swap() at once, only blocking when two threads operate on the same element. With the synchronized keyword, only one thread at a time may be executing the swap() function no matter which element it is operating on, making it quite inefficient and unable to capitalize on potential parallelism.

2. Experiment

The goal of this experiment was to see how the amount of threads and the amount of cores effect each implementation differently. We used the different implementations of the State object and swap() function with the same number of swaps per test to view these effects with different parameters and in different environments.

2.1. Procedure

I performed reliability and efficiency tests on the four implementations of State on the SEASNet Linux Server 09 which has 32 processors with 8 cores each and SEASNet Linux Server 10 which has four processors with four cores each. Both servers have similar amounts of memory and cache, so we observe only the effect on the number of CPUs on performance. On each of these, we ran each implementation on arrays of size 1, 5, 50, and 100 using 1, 8, 20, and 40 threads. For each combination of threads, array size, class, and server, I ran multiple tests and recorded their average Real and CPU times for each combination, all performing 100,000,000 swaps. Since the goal of this experiment was to see the effects of the amount of threads and the amount of cores on each implementation, we report only on the times for an array size of 100. We hold array size, and swap number constant and observe our effects on our response variables of CPU Time and Real Time by our experimental factor of thread amount and machine.

2.2. Results

Real Time/CPU Time on lnxsrv09

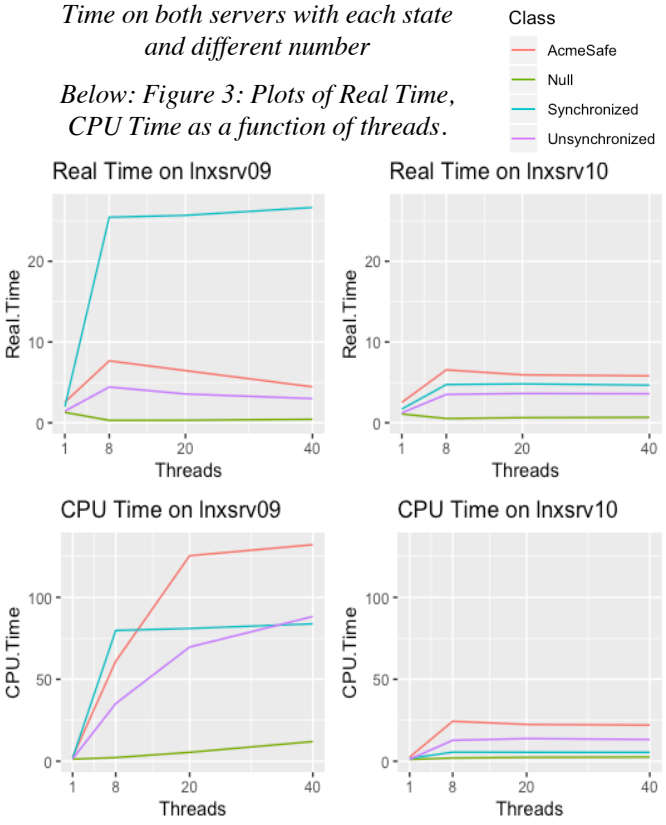
Class	1 thread	8 threads	20 threads	40 threads
Null	1.29/1.29	0.29/79.8	0.30/5.45	0.43/12.0
Unsynchronized	1.43/1.43	4.42/35.2	3.56/69.7	2.99/88.3
Synchronized	1.97/1.97	25.7/79.8	25.7/81.0	26.6/83.8
AcmeSafe	2.55/2.55	7.67/61.0	6.46/125	4.45/132

Real Time/CPU Time on lnxsrv10

Class	1 thread	8 threads	20 threads	40 threads
Null	1.08/1.08	0.52/1.98	0.63/5.58	0.67/2.53
Unsynchronized	1.22/1.22	3.51/12.8	3.63/13.8	3.59/12.3
Synchronized	1.69/1.69	4.73/5.58	4.82/5.52	4.66/5.45
AcmeSafe	2.52/2.52	6.55/24.4	5.93/22.5	5.81/22.1

Above: Figure 2: Tables showing average Real Time/CPU Time on both servers with each state and different number

Below: Figure 3: Plots of Real Time, CPU Time as a function of threads.



2.3 Observations and Conclusions

In general, when using multiple threads, every implementation's Real Time goes down as the number of threads increases and the CPU Time goes up as the number of threads increases due to the multiplication of processing time amongst more threads. We see in ever scenario that NullState consumes the least amount of time because it is not doing anything and therefore has no race conditions and no need to block or spin at any point. Unsynchronized is the next fasted implementation merely because it does not utilize any multithreading constraints but still performs operations. Past one thread for the Unsynchronized implementation produced inaccurate results. The slowest implementations were Synchronized and AcmeSafeState as expected since they utilize constraints to preserve multithreading accuracy.

We observe one interesting case with the scenario of a single thread. When we run any implementation with one thread, the compiler is allowed to assume an "as-if-serial" situation and is able to optimize the operations on the array of longs without fear of multithreading inaccuracies. Because of this, the Real Time and CPU time is very close to zero for all implementations on both machines.

From the data collected, we can observe that both the Real times and CPU times differ from lnxsrv09 to lnxsrv10. lnxsrv09 runs the program with significantly more processors than does lnxsrv10, explaining some of the effects we see here. Looking at Real Time, the real time for the Null, Synchronized, and Unsynchronized implementations is relatively the same. However, when it comes to the Synchronized implementation, the time for lnxsrv09 goes way up. This is because the program is running on so many different processors and the synchronized implementation keeps threads waiting and spinning for much longer. With more processors running more threads, the overhead skyrockets because parallelism is not being taken advantage of. The reason why Synchronized time does not skyrocket for lnxsrv10 is because there are only 4 processors, so there is less blocking and less competition to run the swap method.

With the server with many processors, all threads are able to run concurrently in lnxsrv09, so AcmeSafeState is much faster than Synchronized because it takes advantage of parallelism better as discussed before. With lnxsrv10, there are only 4 processors, so with more threads, even AcmeSafeState is not able to utilize its multiprocessing strengths, so after 8 threads, performance is relatively the same for all implementations.

In terms of CPU time, we look closely at lnxsrv09 because of the abundance of processors, we can view the effects of multithreading more effectively. As AcmeSafeState is the much faster Real Time implementation, it has a much higher CPU time that rises with the amount of threads. This is because as

more threads are used, the CPU time is multiplied. However, AcmeSafeState uses each thread efficiently, which is why we see a high CPU time but a low Real Time. With Synchronized, the Real Time is very high while the CPU time is lower than that of AcmeSafeState. This proves to us that Synchronized does not utilize the potential of multithreading as well as AcmeSafeState.

2.3 Problems

Initially, I recorded only one trial for each combination of thread number, state, and machine. This proved to show inconsistent results because of variation in the number of people logged into SEASNet at a time and the number of processes running on a given server when I was performing the test. There are many other factors to performance time that aren't accounted for in this experiment such as those mentioned previously and variable memory, cache, and RAM usage from one time to another. To try and eliminate these nuisance factors, I ran each test multiple times and took the average of those results. Other problems were ways to record this data efficiently without painstakingly recording each individual time. To aid with this, I wrote a shell script that would run tests and record the results in a csv file that I later ported into R to calculate the averages and create the tables and graphs.

3. Conclusion

Through this exercise, I was able to explore many of the tradeoffs that come with multithreading, and its constraints and memory order models. There are specific situations in which each implementation is most desirable depending on your environment and processing power available. When using a single thread, no synchronization or constraints are necessary. However, when multiple cores are available for use, it is ideal to utilize multithreading to an extent. There is a point where adding more threads and more processors does not increase your efficiency by any means. When utilizing multiprocessing, it is essential to utilize synchronization and/or memory constraints to ensure accurate information and performance. Which method you use, depends on the amount of threads you use and the amount of cores you have at your disposal. Whatever method one chooses should restrict as little as possible while still maintaining accuracy. For this particular scenario, AcmeSafeState is the most ideal implementation because it sets a volatile constraint only on the elements of the array and not the swap() function as a whole.

4. References

[1] Ajila T. Devoxx Belgium [Internet]. 2017 Nov 10. Available from: https://www.youtube.com/watch?v=w2zaqhFczjY

[2] Lea D. Using JDK 9 Memory Order Modes [Internet]. Using JDK 9 Memory Order Modes. 2018. Available from: http://gee.cs.oswego.edu/dl/html/j9mm.html

[3] Defog Tech [Internet]. 2018 Jul 3. Available from: https://www.youtube.com/watch?v=WH5UvQJizH0