CS 131 Homework 3: Java Shared Memory Performance Rates

Ian Conceicao, *UCLA Computer Science and Engineering Undergraduate*

Abstract

In programming, multithreading, the ability of a central processing (CPU) to provide multiple threads of execution concurrently, can drastically speed up the latency of computer programs. While multithreading is not difficult to implement, efficient concurrency can be. Ensuring threads do not interfere with the other threads' data while maintaining an efficiency advantage from using multiple threads is often a challenge. The task in this assignment was as follows: given a Java program designed for multithreaded execution, maintain its functionality and make the program more efficient. The code was made more efficient by eliminating method locks and implementing an atomic data structure, Atomic Long array.

1. SynchronizedState

The provided program that needed fixing was called SynchronizedState.java. The class is initialized with an integer from zero to max int. A member array full of longs is instantiated and all elements are automatically defaulted to zero.

1.1 Swap Method

The member function swap is detailed below:

```
public synchronized void swap(int i, int j) {

value[i]--;

value[j]++;

}
```

Swap takes 2 valid indices and decreases the first indexes' value while increasing the second indexes' value. Note that this function can be called any number of times and the sum of the array should remain at 0, just how it started.

1.2 Multithreaded Design

Swap is functional under multithreaded conditions because of the synchronized keyword. The synchronized keyword tells any thread about the execute the method to lock the entire method. For example, say thread A first calls swap. First, it puts a "lock" on the method. While running the code in the method, thread B attempts to call swap. However, thread A has a lock on the method. Thread B has to wait until thread A finishes the method and releases the lock. Once thread A finishes, it releases the lock and thread B grabs the lock, taking "control" over the method.

1.3 Latency Issues

While the swap method functions correctly for multithreaded execution, it does not execute efficiently. Often, threads are locked out of the swap method for no reason. For example, say thread A is instructed to swap indices 1 and 2. At the same time, thread B is instructed to swap indices 3 and 4. Only one thread will be allowed to work at the same time, even though there is no chance of a race condition. A condition where two threads interfere with the same data and leave an unintended value. Neither thread is accessing the same index, yet one thread will be locked out and will wait for the other. Unnecessary waiting of threads can accumulate and lead to inefficient code, often making multithreading less efficient than it needs to be or even less efficient than single threaded processing.

2. AcmeSafeState

AcmeSafeState.java is the more efficient, smarter, multithread ready program that was created based off of SynchronizedState. In addition to the long array created, an AtomicLongArray was created that will actively manipulate the data in the swap method.

2.1 Atomic Long Array

Imported from the java library:

java.util.concurrent.atomic.AtomicLongArray ,

an atomic array is a long array in which elements may be updated atomically. In other words, no index can be modified by two threads at the same time. The class acts a wrapper and comes with its own methods to manipulate the array.

2.2 New Swap Method

The member function swap in AcmeSafeState is detailed below:

```
public void swap(int i, int j) {

        atomicArray.decrementAndGet(i);

        atomicArray.incrementAndGet(j);

        hasChanged = true;

}
```

The new swap, referred now on as simply swap, now updates an atomic array object called atomicArray. It still decrements index i and increments index j, however, notice the omission of the synchronized keyword. Multiple threads are allowed to access the method, but not the same indices. The AtomicLongArray data type lowers the level of abstraction where our locks live. In other words, instead of locking our entire method, our atomic array locks individual indices for

us. The AcmeSafeState still is required to return a long array when request, so the hasChanged Boolean notifies the class to update its internal long array before returning it because a change was made in the internal atomic long array.

2.3 Data Race Free

The method outlined in AcmeSafeState maintains our accesses as atomic, but just at a low level than SynchronizedState, and as a result, it does so more efficiently. Thread A executing AcmeSafeState only locks thread B out when thread B is attempting to modify the index that thread A is modifying. Therefore, threads can always run the swap method, but may have to wait to change an index depending on other threads. Our code remains *data race free*. A data race in a multithreading context occurs when a thread manipulates data in shared memory that another thread was using— changing the behavior of the program. Because the elements in the long atomic array are maintained atomically, AcmeSafe remains data race free.

2. Measurements and Analysis

To analyze the gained efficiency of AcmeSafe, a series of timed tests were run on two of the Seasnet linux servers. On each server, the Synchronized and AcmeSafe code as described earlier was run with varying numbers of array size and thread. In addition, another version of the swap interface, called Unsynchronized was tested. Unsynchronized is the exact same as Syncrhonized, but without the synchronized keyword before swap, meaning there is no thread race condition protection for data.

2.1 Machine Information

	Linuxsrv06	Linuxsrv09
CPU Model	62	44
CPU Name	Intel Xeon E5 - 2640	Intel Xeon E5620
CPU Clock Speed (Ghz)	2.00	2.40
# of Processors	16	16
Cache Size (KB)	20480	12288
Memory Size (KB)	65755720	65794548

Table 2.1: Information about the processors and memory of the two machines the tests ran on.

Both machines were running Java 13.0.2, mixed mode, sharing.

2.2 Problems Recording Measurements

The data presented in this experiment is to be looked at while understanding the constraints it was recorded in. The latency of each identical execution varied by a second or two when run back to back. For example, it was not unusual to observe that running AcmeSafe with 12 threads and array size of 10 would execute in about 14 seconds then about 16 seconds right after. The discrepancies are most likely due to the variance of hardware availability on the machine as other people use the shared machines. In the future, taking an average or median of each trial and repeating each trial a number of times would yield more accurate data. However, given the time during this project, that was not an option.

Also, linuxsrv10 was giving me contradictory data than what I expected. The AcmeSafe trials on linxsrv10 ran significantly slower than the Syncrhonized trials, and I could not figure out why. AcmeSafe is much faster than Syncrhonized on linuxsrv[06, 09] so that will be the data I am presenting. I still do not know why linuxsrv10 was giving contradictory results, but it is worth looking into. Linuxsrv10 does only have 4 cores compare to the 16 cores that servers 6 and 9 have.

2.3 Unsynchronized vs Synchronized

Once again, the Unsynchronized implementation did not account for multithreading meaning it had no locks or defense against race cases. It's performance is below:

Real Time of Execution of Unsynchronized Swap Implementation on linuxsrv09

Threads/ Array Size	5	10	20
1	1.45778	1.37084	1.38126
4	5.39367	5.0514	5.90794
12	4.6333	4.27478	5.19538
32	3.1968	3.31302	3.46183

Table 2.3.1: The time of execution of the unsynchronized implementation in seconds. All values marked red gave an inaccurate array, meaning the swap methods had race cases.

The boxes marked green gave accurate results after the swapping, but the boxes marked red did not. The results were failures when multiple threads existed because the code did not protect against race conditions.

Real Time of Execution of Synchronized Swap Implementation on linuxsrv09

Threads/ Array Size	5	10	20
1	1.96202	1.90849	1.91903
4	24.4514	27.9264	25.7086
12	27.8991	20.5691	25.1029
32	16.3279	24.5933	24.7185

Table 2.3.2: The time of execution of the synchronized implementation in seconds. All of the arrays were accurate after swapping.

The synchronized implementation did guard against race conditions, so it works in multithreaded conditions. However, the program takes significantly longer than the unsynchronized execution because threads are often locked out

from the swap method. Notice how with 4 threads or more the synchronized implementation often takes 5 times or longer than the unsynchronized implementation. Interestingly, examining the unsynchronized data it can be observed that as you introduce multithreading, latency increases, but gradually that latency recovers when you introduce more threads. A possible reason for this trend is as you introduce more threads, they begin to offset the latency cost of creating threads in the first place at setup and destroying threads at the end of the program.

Also, similar results were observed on linux server 10.

2.4 Synchronized vs AcmeSafe

The results of Synchronized on linuxsrv10 are below:

Real Time of Execution of Synchronized Swap Implementation on linuxsrv10

Threads/ Array Size	**5**	**10**	**20**
1	2.3023	2.31784	2.23651
4	26.5074	30.3817	21.2323
12	22.94	23.7429	21.7696
32	22.9359	20.6619	23.0875

Table 2.3.1: The time of execution of the synchronized implementation in seconds. All of the arrays were accurate after swapping.

The results of AcmeSafe, an implementation that was designed to be more efficient than the synchronized implementation is below:

Real Time of Execution of AcmeSafe Swap Implementation on linuxsrv10

Threads/ Array Size	**5**	**10**	**20**
1	2.26418	2.35202	2.67631
4	20.2082	19.6691	18.5141
12	13.8713	13.3068	13.7376
32	8.28303	9.78422	10.2492

Table 2.4.2: The time of execution of the AcmeSafe implementation in seconds. All of the arrays were accurate after swapping.

The AcmeSafe times were very similar to the synchronized times for a single thread because with a single thread there is no worry of making other threads wait. Therefore how the locks are implemented does not have a great effect on a single threaded program's performance. The differences in times start to appear when the test program generates 4 threads. Then, the AcmeSafe times are noticeably smaller (20.2 vs 26.5 seconds, 19.7 vs 30.4 seconds, and 18.5 vs 21.2 seconds). For the 32 threaded trials the AcmeSafe times are significantly less than half of the Synchronized trial's times. It appears there is a correlation between the number of threads used and the performance gained from minimizing unnecessary locks. The correlation most likely stems from the fact that the more threads present mean the more threads waiting unnecessarily in the synchronized implementation. Meanwhile, in the AcmeSafe implementation threads are only waiting when they have to be, and as a result slow each other down far less. In the future designing multithreaded programs it is important to keep in mind to use locks as sparingly as possible.

3. Sources

http://gee.cs.oswego.edu/dl/html/j9mm.html#summarysec

http://web.cs.ucla.edu/classes/winter20/cs131/hw/hw3.html

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html