Homework 3 (Java) Report
CS 131

Abstract

This report will analyze various methods to solve data race conditions caused by multithreaded applications, looking into both data-race free (DRF) approaches as well as non-DRF approaches.

1 Introduction

As technology advances, the amount of data one has access to only increases. If one were to attempt to process this data or approach a task of a similar magnitude with a single threaded approach, users would find themselves spending far more time than necessary to process this data. In the case of tasks that are "embarrassingly parallel", one can often easily use a multithreaded approach to tackle the problem in an efficient and safe manner. However, "embarrassing parallelism" is not always possible, so when wanting to write multi-threaded solutions to capitalize on parallel processing, one must be able to write these solutions in a DRF manner. This paper analyzes the performance of two DRF multi-threading approaches as well as an approach with race conditions.

2 AcmeSafeState Approach

The *AcmeSafeState* class was written without using the *synchronized* keyword in Java. Instead it makes use of *java.util.concurrent.atomic.AtomicLongArray*, a data structure representing a long array designed such that many common operations performed on elements of the array can be executed atomically. This includes operations such as *AtomicLongArray.getAndDecrement(int index)* and *AtomicLongArray.getAndInrement(int index)*, functions that the implementation of the *AcmeSafeState* use to implement the *swap* function in a DRF manner.

Traditionally, if one were to increment/decrement an element, it would be split into three machine level instructions – a read, an increment/decrement, and a write. The *AtomicLongArray* provides an atomic implementation of this operation, meaning that this sequence of steps is executed in a manner that makes it seem like a single instruction. This atomicity allows the operation to take place without interruption from other threads due to the temporary blocking state the element being modified will enter during the atomic operation.

The only note here is that there is no convenient manner in which a user can access the member variable that represents the long array that *AtomicLongArray* represents. Therefore, the implementation of the *current()* method in *AcmeSafeState* is not thread-safe, or rather, not thread-correct. This is because one has to iterate through the *AtomicLongArray* and *get* each element and store it in a temporary long array. By the time this array has been built, there's a chance that another thread may have modified the contents of the array.

3 Data

In order to collect the data below, 100,000,000 swaps were performed while varying both number of threads and the size of the state array. The test-harness and the various implementations were run on *Java v13.0.2*. The tests were then performed on two servers with differing architecture.

	Size of State Array			
Number of threads		5	50	100
	1	20.8054	20.7141	30.4334
	8	2087.71	1379.88	870.816
	20	4323.96	5159.74	5483.67
	40	11825	10132.1	11065

Figure 1: Average Swap Time (Real) for *Synchronized* (ns) on lnxsrv09

	Size of State Array			
Number of threads		5	50	100
	1	17.5843	16.9344	17.0297
	8	414.395	374.399	355.226
	20	1017.3	947.674	932.96
	40	1953.07	1916.84	1753.39

Figure 2: Average Swap Time (Real) for *Synchronized* (ns) on lnxsrv10

	Size of State Array		
Number of threads	5	50	100
1	14.2709	15.0827	13.9308
8	279.056	398.347	363.544
20	626.838	882.112	680.724
40	1200.62	1359.29	1369.37

Figure 3: Average Swap Time (Real) for *Unsynchronized* (ns) on lnxsrv09

	Size of State Array		
Number of threads	5	50	100
1	12.1338	12.2261	12.1945
8	164.123	306.687	286.647
20	377.331	816.546	695.736
40	1581.71	1581.53	1451.66

Figure 4: Average Swap Time (Real) for *Unsynchronized* (ns) on lnxsrv10

	Size of State Array		
Number of threads	5	50	100
1	27.7979	85.816	60.9149
8	1409.24	568.332	538.082
20	2455.01	1690.75	1315.28
40	2814.5	2671.53	1935.33

Figure 5: Average Swap Time (Real) for *AcmeSafe* (ns) on lnxsrv09

	Size of State Array		
Number of threads	5	50	100
1	25.4179	25.0481	25.0803
8	427.414	352.747	622.178
20	2579.03	2144.66	1474.7
40	5360.45	4217.74	3251.52

Figure 6: Average Swap Time (Real) for *AcmeSafe* (ns) on lnxsrv10

	Program		
# of threads, size of array	Sync.	Acme	Unsync.
1, 5	20.8054	27.7979	14.2709
1, 100	30.4334	60.9149	13.9308
40, 5	11825	2814.5	1200.62
40, 100	11065	1935.33	1369.37

Figure 7: The table compares the Average Swap Time (Real) for *Synchronized*, *Unsynchronized*, and *AcmeSafe* (ns) for varying number of threads and array size on lnxsrv09

	Program		
# of threads	Sync.	Acme	Unsync.
1	0	0	0
8	0	0	21086
20	0	0	43190
40	0	0	39868

Figure 8: The table compares the absolute value of the number of mismatches for *Synchronized*, *Unsynchronized*, and *AcmeSafe* with an array size of 100 for varying number of threads on lnxsrv09

4 Analysis

The collected data can be used to derive insights into the benefits and drawbacks of different approaches to multithreading, both DRF and non-DRF.

4.1 Difference Between Linux Servers

There was clearly a performance difference between lnxsrv09 and lnxsrv10, although not in the manner that one would expect. In spit of lnxsrv09's 8 cores and greater number of processors in comparison to lnxsrv10, lnxsrv10 outperformed lnxsrv09 in many of the conducted tests. In order to determine whether or not there is an actual difference, and what that difference looks like, more trial would need to be conducted. There also other external factors, such as server user traffic, that need to be accounted for to make an accurate comparison of the two servers.

4.2 AcmeSafe vs. Synchronized

Yet if one looks at the data, there is a clear distinction between the *Synchronized* and *AcmeSafe* implemntations. Figure 7 illustrates that, with a large number of threads and a large state array size, *AcmeSafe* performs 3-5 times better than *Synchronized*. This is due to the fact that the *synchronized* keyword placed on the *swap* method declaration effectively declares the entire function a critical section, meaning that no other thread can enter that body while a thread is executing that code. If each increment/decrement is 3 machine instructions, there are a total of 6 instructions for which the section is locked for. This, combined with the overhead of acquiring and releasing locks, results in a far slower implementation compared to the atomic operations afforded by *AtomicLongArray*

which allow the user to not worry about the side-effect of other threads during the execution of a thread.

However, when there are a small number of threads, *Synchronized* performs faster than *AcmeSafe*. This is because lock-contention happens far less frequently with a smaller number of threads.

4.3 Thread Safe vs Unsynchronized

In every test run, as can be seen in Figures 1-7, the *Unsynchronized* (non-DRF) implementation is far more performant. The reason for this is fairly simple – there's no overhead whatsoever, and each thread simply executes instructions with no idea about the existence of others. As a result, swaps are very fast but not DRF.

Figure 8 shows the discrepancies resulting from the *Unsynchronized* approach. The race condition that causes this is fairly trivial – since the incrementing/decrementing of an element is a three step operation, if a second thread begins to perform the increment before the first thread has finished writing its modification to the element back into the array, discrepancies will begin to arise. However, if one compares the degree of discrepancy to the total number of operations being performed, there is at most a .05% error. Row 3 of Figure 7 illustrates that the *Unsynchronized* approach is more than 2 times as fast as *AcmeSafe* and almost 10 times as fast as *Synchronized*. If one considers the importance of accuracy versus performance, the performance boost offered by *Unsynchronized* may often times be crucial to success.

5 Conclusion

Multi-threaded programs can offer the programmer numerous performance boosts in completing a task. There are many ways in which a task can be multi-threaded, both in DRF and non-DRF manners. Depending on the circumstances of the problem, there are different approaches that can be more suitable – each has its own tradeoffs. There is no universal solution, and it is in the hands of the engineer to test a variety of solutions in order to determine the one most appropriate for the task.