Homework 3 (Java) Report
CS 131

Abstract

This paper is a report analyzing different approaches to solving and not solving data race conditions caused by multithreading.

1 Introduction

Race conditions are a prominent problem in the field of computer science today. With most computers now relying on parallel processing and multi-threading to function, the issue of writing data-race free (DRF) programs has become important. This paper analyzes two approaches to DRF programs and also compares them to a program with race conditions.

2 AcmeSafeState Approach

The *AcmeSafeState* class was written without the use of the Java *synchronized* keyword and instead uses the *java.util.concurrent.atomic.AtomicLongArray*. The *AtomicLongArray* is a Java long array that performs operations atomically. Atomic operations combine reads and writes into low-level operations, therefore making them DRF. This operation looks like a single operation to both the user and other threads, thereby blocking other threads from manipulating that value. This removes race conditions because two threads can't modify the same value at the same time.

The code uses *array.getAndDecrement(int index)* and *array.getAndInrement(int index)* to atomically modify the array instead of adding and removing locks before and after operations like *Synchronized* does. Since these are atomic operations, and the only operations the modify the values during multi-threading, this approach is DRF. Additionally, the *current()* method is modified to loop through the *AtomicLongArray* and create a long array from it, maintaining the State interface.

3 Data

The following data (Figures 1-7) was collected by performing 100,000,000 swaps and dividing by the runtime to collect an approximation of average swap time.

	Size of Array		
Number of threads	5	50	100
1	17.6509	17.224	17.1313
8	376.769	384.392	355.944
20	985.789	921.702	939.686
40	1887.08	1862.17	1705.28

Figure 1: Average Swap Time (Real) for *Synchronized* (ns) on lnxsrv10

	Size of Array		
Number of threads	5	50	100
1	12.2495	12.1443	12.2771
8	203.065	326.491	293.623
20	394.952	800.703	717.512
40	783.706	1573.82	1398.37

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

	Size of Array		
Number of threads	5	50	100
1	25.6503	24.4846	24.9999
8	958.818	886.717	498.194
20	2472.16	1251.96	910.436
40	5038.43	4097.37	2119.5

Figure 3: Average Swap Time (Real) for *AcmeSafe* (ns) on lnxsrv10

	Size of Array		
Number of threads	5	50	100
1	19.2152	19.5238	21.3231
8	2476.56	2160.86	2267.03
20	6097.14	5510.11	5669.71
40	11800.9	11244.0	11630.3

Figure 4: Average Swap Time (Real) for *Synchronized* (ns) on lnxsrv09

	Size of Array		
Number of threads	5	50	100
1	13.8972	13.9079	14.0631
8	309.678	421.916	360.143
20	624.200	838.164	714.372
40	997.120	1417.42	1209.02

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

	Size of Array		
Number of threads	5	50	100
1	24.719	24.7749	26.2051
8	1209.78	985.061	318.423
20	1642.96	1867.2	1332.36
40	3549.01	2439.98	1785.37

Figure 6: Average Swap Time (Real) for *AcmeSafe* (ns) on lnxsrv09

	Program			
# of threads, size of array	Sync.	Acme	Unsync.	
1, 5	19.2152	24.719	13.8972	
1, 100	21.3231	26.2051	14.0631	
40, 5	11800.9	3549.01	997.120	
40, 100	11630.3	1785.37	1209.02	

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	18621	
20	0	0	1978	
40	0	0	24735	

Figure 8: The table compares the absolute value of the number of mismatches for *Synchronized*, *Unsynchronized*, and *AcmeSafe* (ns) with an array size of 5 for varying number of threads on lnxsrv09

4 Analysis

The data collected provides valuable insight into the benefits and drawbacks of different DRF approaches as well as how ignoring race conditions can benefit efficiency at the cost of accuracy.

4.1 Difference Between Linux Servers

There was an unusual difference in performances between lnxsrv09 and lnxsrv10. Although lnxsrv09 has 8 cores and more processors, lnxsrv10 performed better in many cases. However, there is not enough data collected and trials run to determine if there is an actual difference. The variances in the data comparing the two servers could possibly be attributed to the number of users on the server causing one to be slower than the other. In order to truly determine the difference between the servers, there would need to be no extra load on them.

4.2 Synchronized vs. AcmeSafe

However, by looking at the data, there is a very clear distinction in the differences between *Synchronized* and *AcmeSafe*. By looking at Figure 7, it is obvious that *AcmeSafe* performs much better when there are a larger number of threads. This is because *AcmeSafe* doesn't use locks on storage. *AcmeSafe* implements an *AtomicLongArray* which uses atomic operations to change the values in the array. This prevents race conditions because two threads can't modify the same value at the same time.

Synchronized uses locks which make it a slower approach than *AcmeSafe*. The approach involves first placing a lock on the value which means no other thread can access or modify that value. Then the thread reads the value and modifies it. After this the lock is removed and value can be accessed by other threads. This process is much slower than an atomic operation as seen in the data provided in Figure 7 when there are a large number of threads. This is because in *AcmeSafe*, other threads have to wait less time before being able to access the value they want.

However, when there are a small number of threads, *Synchronized* performs faster than *AcmeSafe*. This is most likely because the overhead of locking and unlocking doesn't have a significant effect when there aren't as many threads.

4.3 Thread Safe vs Unsynchronized

In every case, by comparing Figures 1-6 and looking at Figure 7, the unsynchronized (not DRF) is much more efficient. The reason why is trivial: threads don't have to wait for each other and just perform

operations as they reach them. This results in very efficient but inaccurate results.

In Figure 8 it is visible how the *Unsynchronized* approach results thousands of mismatches. This is because threads are interweaving reads and writes which causes values not to be updated correctly. However, compared to the 100,000,000 swaps executed over the duration of the program, a couple thousand mismatches are insignificant. With an array size of 5, around than 1 in 5,000 swaps are executed incorrectly. Row 3 of Figure 7, with 40 threads and an array size of 5, shows that the *Unsynchronized* approach is around 3.5 times faster than *AcmeSafe* and almost 10 times faster than *Synchronized*. Depending on the application, the benefits of this performance boost can outweigh the costs of the inaccuracy in some applications.

5 Conclusion

There are many approaches to tackling race conditions and each approach has its benefits and costs. There is no universal best approach to solving the issue because different solutions work well for different applications. In the end, it is up to the programmer to determine through theory or through testing, which approach will work best for them.