Homework 3 - Java Shared Memory Performance Races

Abstract

Exploring the characteristics of a Java program implemented in 3 ways: with synchronized keyword, without synchronized keyword, and with atomic instructions via java.util.concurrent.atomic.AtomicLongArray around the program's critical section. Each was examined to check for data races and efficiency compared to each other.

1. Introduction

To begin this experiment, we were given a .jar file containing the implementation of a program which creates an array initialized to 0 of specified length and number of threads. It implements a swap function which randomly takes two array indices, i and j, and subtracts 1 from i and adds 1 to j. The variation comes the method used to protect the data of the array while each index is in the process of being changed.

The synchronized keyword implements a lock around the function so that no other thread can access the array's data while another thread is in use. The AtomicLongArray version utilizes atomic instructions when calculating swap so that a context switch cannot occur in the middle of an add. The third version implements swap with no precautions and reports the consequences of the data races that occur, mainly that the sum of all array values will no longer be 0. Both the synchronized and AtomicLongArray versions will be data race free (DRF), meaning no two threads will attempt to modify the same piece of data without using some form of synchronization.

2. AcmeSafeState

The AtomicLongArray version of this program was implemented in a class named AcmeSafeState. The basic idea of this class was to improve the efficiency of the synchronized method while remaining DRF. Using atomic instructions accomplishes this because although the atomic instructions take longer to execute than traditional add instructions, it does not require locking to be implemented around the swap function doing the adding.

AcmeSafeState is also theoretically faster than SynchronizedState, the class implementing synchronized swap, because it operates at a lower level of control in terms of Java's memory order modes. AcmeSafeState is able to run in opaque mode, mainly because of its bitwise atomic accesses to the variables that it modifies. It also fulfills all other requirements of Opaque Mode.

2.1 Opaque Mode

Per-variable antecedent acyclicity means that the program will operate in partial order and future events can't affect the past, in respect to any single variable, which is guaranteed by the use of atomic instructions when accessing any single variable. Coherence, which ensures visible (or usable) overwrites of a single variable are ordered, is also ensured from nature of the program. It doesn't matter what order the array entries are being changed as long adds happen by the time the array becomes visible which will happen after the add is finished. The final requirement progress is also ensured because of the lack of spin loops in the program. The program will run and eventually complete or crash, but this is not an issue.

2.2 Release/Acquire Mode

This is different than SynchronizedState which will run in release/acquire (RA) mode. This is because of its functionality of locking the swap function when it is in use by one thread, meaning other threads cannot use the function. This isn't necessary when using atomic adds, however normal adds are implemented using multiple machine instructions. If a context switch occurs in the middle of this group of instructions, then another thread could manipulate a register or piece of memory containing a pertinent value to the overall add and result in an incorrect value being evaluated. This is what happens in UnsynchronizedState, the class implementing the unsynchronized swap, and is why it returns a nonzero array sum.

Because opaque mode has a weaker restriction system placed on it than RA mode, it allows for AcmeSafeState to run quicker than SynchronizedState for multithreaded programs.

AcmeSafeState will be DRF because the use of atomic instructions ensures that no two threads can access the same data location at the same time.

3. Problems with Measurements

The main problems that arose from the fact that testing was being done on remote servers across WIFI and through a VPN connection. To attempt to minimize the effect this had on the results, I sat as close as I could to my router and ran the tests consecutively. Another problem was the fact that

these servers are not private and other people may have been accessing them at the same time, slowing down my program's runtime. These problems are inherent in the nature of the testing and could not be eliminated.

4. Measurements and Analysis

I took measurements of four different implementations of the swap function, run with 100,000,000 iterations of swap to ensure the results were not skewed by startup overhead. I used SynchronizedState, UnsynchronizedState, AcmeSafeState, and NullState which called swap but did not do anything or change the array in any way, acting as a default measurement. I tested each with 1, 4, 8, and 40 threads, and array lengths of 5, 100, and 1000.

I ran all of these tests on two different machines. lnxsrv09 is an Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz with 8 cores and 16 processors. It runs java 13.0.2 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing) and has a total memory of 65755720 kB.
lnxsrv10 is an Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz with 4 cores and 4 processors. It also runs java 13.0.2 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing) and has a total memory of 65799628 kB.

I chose 4 threads as my own value to test to see how lnxsrv10 performed with a max core load compared to lnxsrv09. I chose 1000 as my own array length value to test to see how the tests scaled up to an even larger length.

4.1 State Comparisons

I will be comparing the differences between each State class at each array length and on both machines to see the performance difference between synchronized functions and atomic functions as data race control techniques.

Num Threads-> Machine (StateClass)	1	4	8	40
lnxsrv09 (synch)	2.348	21.797	21.496	21.138
lnxsrv09 (acme)	2.877	8.224	14.947	7.870
lnxsrv09 (unsynch)	1.664	4.452 *	2.612 *	3.142 *
lnxsrv09 (null)	1.479	0.537	0.242	0.520
lnxsrv10 (synch)	5.607	8.292	4.750	5.033
lnxsrv10 (acme)	2.700	12.693	12.979	13.265
lnxsrv10 (unsynch)	1.357	3.454 *	2.213 *	2.114 *
lnxsrv10 (null)	1.216	0.466	0.538	0.793

Figure 1. (Array Length 5) Real time runtime output of bash command 'time timeout 3600 java UnsafeMemory StateClass numThreads iterations arrayLength' with UnsafeMemory being the main class that takes the following variables in as arguments. StateClass corresponds to the data race protection method used, numThreads is the number of threads running, iterations is the number of times swap is called (100,000,000 in all tests), and arrayLength is the length of the array on which the swapping is occurring (5 in all tests).
*Sum of array non-zero due to data races

Num Threads-> Machine (StateClass)	1	4	8	40
lnxsrv09 (synch)	2.226	21.007	27.668	24.607
lnxsrv09 (acme)	2.780	11.754	5.039	3.630
lnxsrv09 (unsynch)	1.529	5.065 *	4.850 *	3.195 *
lnxsrv09 (null)	1.429	0.537	0.403	0.542
lnxsrv10 (synch)	5.735	10.344	4.862	4.988
lnxsrv10 (acme)	2.645	3.904	6.287	9.047
lnxsrv10 (unsynch)	1.342	3.576 *	3.559 *	3.760 *
lnxsrv10 (null)	1.211	0.470	0.496	0.941

Figure 2. (Array Length 100) Real time runtime output of bash command 'time timeout 3600 java UnsafeMemory StateClass numThreads iterations arrayLength' with UnsafeMemory being the main class that takes the following variables in as arguments. StateClass corresponds to the data race

protection method used, numThreads is the number of threads running, iterations is the number of times swap is called (100,000,000 in all tests), and arrayLength is the length of the array on which the swapping is occurring (100 in all tests).
*Sum of array non-zero due to data races

Num Threads-> Machine (StateClass)	1	4	8	40
lnxsrv09 (synch)	2.248	15.249	34.160	33.122
lnxsrv09 (acme)	5.397	4.500	3.201	2.135
lnxsrv09 (unsynch)	1.703	3.463 *	2.404 *	1.760 *
lnxsrv09 (null)	1.391	0.545	0.519	0.617
lnxsrv10 (synch)	1.833	5.872	4.084	4.595
lnxsrv10 (acme)	2.588	3.395	3.299	6.000
lnxsrv10 (unsynch)	1.352	2.400 *	2.353 *	2.533 *
lnxsrv10 (null)	1.217	0.480	0.729	0.595

Figure 3. (Array Length 1000) Real time runtime output of bash command 'time timeout 3600 java UnsafeMemory StateClass numThreads iterations arrayLength' with UnsafeMemory being the main class that takes the following variables in as arguments. StateClass corresponds to the data race protection method used, numThreads is the number of threads running, iterations is the number of times swap is called (100,000,000 in all tests), and arrayLength is the length of the array on which the swapping is occurring (1000 in all tests).
*Sum of array non-zero due to data races

The measured values in each cell are representative of the value returned by the 'time' command which measures the real time taken to execute the entire program measured in seconds. The startup overhead will be very similar in every case, so this is a reliable value to use to compare differences in efficiency.

4.2 AcmeSafeState vs SynchronizedState
This is a very interesting result because the comparisons between the two differ between the two machines used. For lnxsrv09, AcmeSafeState was faster than SynchronizedState in every case. For lnxsrv10 however, it was more mixed, and the runtimes for AcmeSafeState grew as the number of threads increased. This is different than lnxsrv09 where AcmeSafeState runtimes generally fell as number of threads increased because of the difference in number of cores on each machine. lnxsrv09 has 8 cores as opposed to lnxsrv10 which only has 4 cores, making lnxsrv09 better equipped to handle higher thread count programs. lnxsrv10 gets bogged down with more threads making its runtime slower.

It is also interesting to observe how performance varies between single and multi-threaded programs. In most single-threaded variations SynchronizedState runs quicker than AcmeSafeState, the opposite of multi-threaded variations. This is because in a single threaded SynchronizedState there is no worry of other processes trying to access the locked swap function, the main bottleneck of SynchronizedState's multi-threaded implementation, and can complete all swaps efficiently. AcmeSafeState is doing the same thing but is still using atomic add instructions which are much slower than normal adds used by SynchronizedState. In multi-threaded programs SynchronizedState's bottleneck becomes more hindersome, but AcmeSafeState without this bottleneck improves through multithreaded performance.

4.3 lnxsrv09 vs lnxsrv10 SynchronizedState
An odd result is found when comparing SynchronizedState runtimes on lnxsrv09 and lnxsrv10. The runtimes on lnxsrv09 are significantly greater in every case. This is likely because the lock system that the synchronized swap method uses runs more efficiently on less cores because there are less context switches to worry about. lnxsrv10 would let threads run for longer without a context switch with less threads making it more efficient than the higher core lnxsrv09.

4.4 UnsynchronizedState Analysis
UnsynchronizedState was the quickest in every case, excluding null which does nothing, due to its neglection of any form of synchronization and presence of data races. It worked very well in all single-threaded implementations, however in all other tests the array sum ended up being ±10,000 or larger. This is the result of many data races occurring during execution and giving an inaccurate output. In most cases the increased speed is not worth the degree of error it produces.

4.5 Array Length Analysis
In nearly every case, runtime decreased as array length increased. This is likely due to certain memory addresses being less cluttered with updates from swap functions being called. The improvement is better seen in AcmeSafeState implementations, likely because it allows the program to run in parallel easier with the use of atomic functions rather than locking

the method with the synchronized keyword. The increased array size means that the updates are happening at different locations in memory and busses going to and from those locations are less crowded, improving efficiency.

5. Conclusion

In most cases AcmeSafeState will be the better implementation to use for multi-threaded applications of the swap function. However, in machines with fewer cores it is closer and SynchronizedState can perform better in some cases. UnsynchronizedState runs very quick and is accurate in single-threaded programs but is not DNF. AcmeSafeState is DNF would likely be the best implementation to use in most cases.

Bibliography

Eggert, Paul. *Homework 3. Java Shared Memory Performance Races*, 2020, web.cs.ucla.edu/classes/spring20/cs131/hw/hw3.html.

Lea, Doug. *Using JDK 9 Memory Order Modes*, 2018, gee.cs.oswego.edu/dl/html/j9mm.html.