CS 131 Homework 3 Report: Java Shared Memory and Performance Races

1. Introduction

As a background for this homework assignment, we were told we worked for a company that uses multithreading to speed up its applications. Our programs operate on shared-memory representations of the state of a simulation. Currently, using the synchronized keyword in Java is known to be a bottleneck. The goal of this assignment is to understand the Java synchronization model and how an application can safely avoid race conditions when accessing shared memory. This homework assignment tests reliability of data along with efficiency of using synchronized functions, unsynchronized functions, and atomic arrays.

2. Implementation

2.1. State.java

This class defines the interface the tested classes will be following. The state interface has three functions. The first function is size() that returns the size of the array. The second function is current() that returns the current array. The third function is swap() which takes in two integer inputs, i and j.

2.2. Synchronized.java

The is a given class that tests the addition of "synchronized" to the swap function. In the swap() function, the element at index i is decremented and the element at index j is incremented. The keyword "synchronized" protect the private array variable such that when elements are decremented or incremented in different threads, no race conditions occur. It does this by only allowing one thread to invoke the function at a time.

2.3. Unsynchronized.java

This is a new class that is based off of Synchronized.java however does not include the keyword "synchronized" in the swap() function. This means that the private array variable is unprotected against race conditions. The reason to include this class is to test whether having the "synchronized" keyword matters in terms of maintaining correct values of the array and the effects on overall time to complete.

2.4. AcmeSafeState.java

The is a new class that is also based off of Synchronized.java, however instead of having a private long array variable, this class stores an AtomicLongArray object. As for the other functions in this class, there are small changes since the private array is a different object. For the constructor of AcmeSafeState, the function creates a new AtomicLongArray with a given length. For size(), the function calls the length() function of AtomicLongArray. For the current() function, because the State interface calls for a return value of a long array, a new long array must be constructed. To do this, the length of the AtomicLongArray is saved as a temporary variable. Then, a new long array is created with the same length of the AtomicLongArray. After that, each element of the AtomicLongArray is inserted into the long array using the AtomicLongArray function get(). Lastly, the long array is returned. The final function, swap() still decrements and increments the i-th and j-th element, respectively, however this function does not contain the "synchronized" keyword. This is because with an AtomicLongArray, the variable is protected against race conditions. In addition, in the swap() function, the functions decrementAndGet() and incrementAndGet() from AtomicLongArray are used.

2.4.1. Java.util.concurrent.atomic.AtomicLongArray

This is the package that is imported for the AcmeSafeState class. This package defines how the private variable stores the array. With an AtomicLongArray, a long array is stored such that every access to the array is an atomic access. This means that when a thread uses the AtomicLongArray, it is ensured that other thread will not interfere with the shared variable while a thread is using it. The difference of using an atomic variable versus using "synchronized" is that with a synchronized function, locks are used to suspend other threads trying to invoke the function. For atomic functions, each invocation is run either fully or not at all.

2.5. NullState.java

The is a given class that implements the State interface and is similar to the Unsynchronized class with the exception of the swap() function. In NullState's swap() function, i and j are still taken as inputs, however there is no modification to the private array variable. This is to test the efficiency of having no synchronized function and no atomic variables. This is the control of the experimentation.

3. Results

3.1. Testing

To test the mentioned classes, I ran each class with array sizes of 5, 100, and 200. For each of the array sizes, I ran with 1, 8, 40, and 100 threads. For each test I ran with the 100,000,000 swaps. Lastly, I ran each test on both Linux Server 09 and 07.

3.2. Timing

Below is the output for the average swap times, both real and CPU in nanoseconds, based on thread count, class, array size, and server.

	Avg Swap Time (real) by Thread Count				Avg Swap Time (CPU) by Thread Count			
	1	8	40	100	1	8	40	100
Synchronized size 5 server 7	24.1668	3743.97	18997	46133.5	24.1425	1513.77	1550.27	1494.93
Synchronized size 5 server 9	20.3852	2196.08	10083	24803.2	20.3706	942.29	849.411	829.721
Avg Synchronized size 5	**22.276**	**2970.025**	**14540**	**35468.35**	**22.25655**	**1228.03**	**1199.8405**	**1162.3255**
Synchronized size 100 server 7	25.0855	4023.37	19986.4	49156.5	25.0616	1577.35	1554.4	1499.38
Synchronized size 100 server 9	20.3972	2303.51	11333.4	26307.5	20.384	923.16	896.662	821.339
Avg Synchronized size 100	**22.74135**	**3163.44**	**15659.9**	**37732**	**22.7228**	**1250.255**	**1225.531**	**1160.3595**
Synchronized size 200 server 7	24.7905	4355.56	22298.4	52098.9	24.768	1675.98	1717.25	1605.77
Synchronized size 200 server 9	19.6382	2535.74	11835.2	26693.8	19.6244	991.158	913.219	814.339
Avg Synchronized size 200	**22.21435**	**3445.65**	**17066.8**	**39396.35**	**22.1962**	**1333.569**	**1315.2345**	**1210.0545**
Unsynchronized size 5 server 7	17.8063	203.313	1150.95	2864.75	17.7805	190.954	835.245	845.779
Unsynchronized size 5 server 9	14.4491	265.503	1147.21	2531.49	14.4374	244.244	856.818	762.572
Avg Unsynchronized size 5	**16.1277**	**234.408**	**1149.08**	**2698.12**	**16.10895**	**217.599**	**846.0315**	**804.1755**
Unsynchronized size 100 server 7	16.3384	401.334	1311.08	3066.57	16.3241	397.205	927.292	906.428
Unsynchronized size 100 server 9	15.205	364.491	1237.22	2904.44	15.1936	352.688	919.83	856.529
Avg Unsynchronized size 100	**15.7717**	**382.9125**	**1274.15**	**2985.505**	**15.75885**	**374.9465**	**923.561**	**881.4785**
Unsynchronized size 200 server 7	15.2082	314.261	948.968	2546.8	15.1924	313.009	668.865	733.387
Unsynchronized size 200 server 9	15.0476	271.28	906.49	2211.75	15.0335	269.859	673.303	665.285
Avg Unsynchronized size 200	**15.1279**	**292.7705**	**927.729**	**2379.275**	**15.11295**	**291.434**	**671.084**	**699.336**
AcmeSafe size 5 server 7	28.1676	1133.27	4117.99	10170.9	28.1452	1102.76	3059.65	3062.94
AcmeSafe size 5 server 9	27.1146	1313.1	4081.91	5618.27	27.0919	1280.53	2994.34	1713.07
Avg AcmeSafe size 5	**27.6411**	**1223.185**	**4099.95**	**7894.585**	**27.61855**	**1191.645**	**3026.995**	**2388.005**
AcmeSafe size 100 server 7	29.885	579.507	1975.09	4099.45	29.8541	577.049	1391.7	1219.03
AcmeSafe size 100 server 9	26.9557	526.9424	1316.05	4579.72	26.9424	513.502	992.945	1397.35
Avg AcmeSafe size 100	**28.42035**	**553.2247**	**1645.57**	**4339.585**	**28.39825**	**545.2755**	**1192.3225**	**1308.19**
AcmeSafe size 200 server 7	28.4137	468.169	1210.46	3231.29	28.3976	462.102	873.291	946.213
AcmeSafe size 200 server 9	27.1241	450.514	1510.72	2967.56	27.1107	445.73	1138.51	909.921
Avg AcmeSafe size 200	**27.7689**	**459.3415**	**1360.59**	**3099.425**	**27.75415**	**453.916**	**1005.9005**	**928.067**
Null size 5 server 7	14.9112	38.684	238.989	587.443	14.8888	37.6721	163.136	163.281
Null size 5 server 9	13.5097	20.7216	130.209	601.542	13.4961	20.2344	91.6319	173.182
Avg Null size 5	**14.21045**	**29.7028**	**184.599**	**594.4925**	**14.19245**	**28.95325**	**127.38395**	**168.2315**
Null size 100 server 7	16.8447	35.0729	208.435	881.928	16.823	33.8266	139.503	262.498
Null size 100 server 9	13.2098	22.0691	155.015	448.589	13.1978	20.8956	94.4895	129.377
Avg Null size 100	**15.02725**	**28.571**	**181.725**	**665.2585**	**15.0104**	**27.3611**	**116.99625**	**195.9375**
Null size 200 server 7	15.6174	37.9262	221.113	745.512	15.5955	36.2974	144.512	212.327
Null size 200 server 9	13.3606	22.3946	140.883	546.864	13.3472	21.33	89.9506	159.781
Avg Null size 200	**14.489**	**30.1604**	**180.998**	**646.188**	**14.47135**	**28.8137**	**117.2313**	**186.054**

3.3. Accuracy

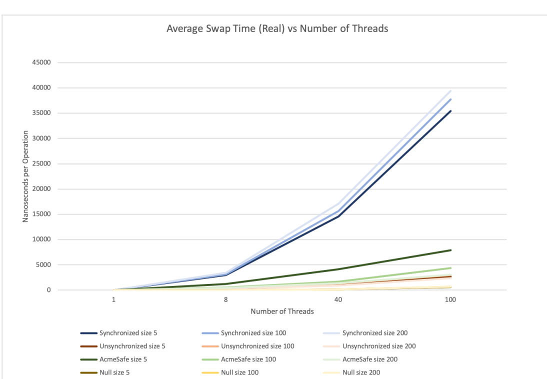
A race condition is detected when the checksum of an array is not equal to zero. This can occur when an invocation of swap does not accurately decrease and increase the i-th and j-th element due to other threads interfering. Below is a table of the amount of incorrect checksum values based on each State class. These are out of the 24 different tests of each class.

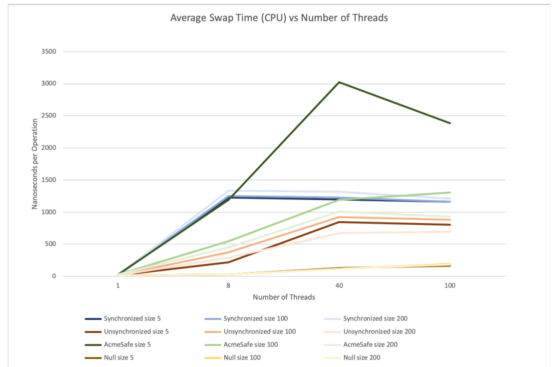
Swap State Class	Errors
Synchronized	0
Unsynchronized	18
AcmeSafe	0
Null	0

4. Analysis

4.1. Timing

The following are graphs plotting the average nanoseconds per operation of the two servers based on the State class and array size.





From this data, we can see that there is a general trend that increasing threads and array size both increase nanoseconds per operation. From both graphs, we can order the classes from slowest to fastest as Synchronized, AcmeSafeState, Unsynchronized, NullState. This confirms the statement that atomic variables are faster because they do not have to spend time locking and unlocking threads as synchronized functions do.

Furthermore, from both graphs, with only one thread, it shows that the average nanoseconds per operation are fairly similar, with a range of about 15 nanoseconds, between NullState and AcmeSafeState. As the thread count increases, the difference between NullState and the safe classes (AcmeSafeState and Synchronized) become more drastic.

Lastly, from the data, we can conclude if there in only one thread, then AcmeSafeState performs worse than Synchronized, however if there are multiple threads, AcmeSafeState performs better in terms of real time but not CPU time.

4.2. Accuracy

In terms of accuracy, the data shows the need for having either an atomic variable or synchronized function. From the data, if there is only one thread, then it does not matter if there is protection against race conditions since there is only one thread, thus there will never be another thread accessing the shared variable. However, when the program uses multiple threads, without any protection against race conditions, it is almost certain that shared variable data will be corrupted, as shown throw incorrect checksum values.

4.3. Analysis Conclusion

If the program requires only one thread, then Unsynchronized.java will work fine because there will be no accesses to the shared memory because no other threads are running with the shared variable. If the program requires to be multithreaded, then AcmeSafeState.java is recommended in terms of real time because it much faster for nanoseconds per operation with higher thread counts. In terms of CPU time, with higher thread counts, AcmeSafeState and Synchronized classes perform similarly.