Abstract
The primary goal of homework three was to examine the tradeoff between thread safety and overall performance of the program. A thread is deemed safe if all threads share the sane data and behave properly by correctly fulfilling their design specifications without unintended consequences, no matter the order in which the threads were executed. We will be using Java to test the performance of compare the performance of safe and unsafe threads. We will be testing three programs, SynchronizedState, UnsynchronizedState, and AcmeSafeState on two different servers, and measure and characterize the class's performance and reliability given the parameters above. For each of the classes, we will use varying array sizes and number of threads.

Testing Hardware and Java Version
Testing occurred on two different UCLA servers, *lnxsrv06.seas.ucla.edu* and *lnxsrv09.seas.ucal.edu*. Using */proc/cpuinfo* to check the CPUs of the server, We get the following specifications for the hardware:

Java Version:
- java version "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)

lnxsrv06.seas.ucla.edu
- Model Name: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
- Model: 44
- CPU MHz: 2395.000
- Cache Size: 12288 KB
- Memory: 65794548 KB

lnxsrv09.seas.ucla.edu
- Model Name: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
- Model: 62
- CPU MHz: 1762.451
- Cache Size: 20480 KB
- Memory: 65755720 KB

Potential Issues with Multiple Threads

New issues arise when running multiple threads at the same time vs. just using a single thread. For example, if we have multiple threads trying to access the same values in an object at the same time, we could potentially get unexpected results. Assume that we have an integer *i* in a class. Assume a method in the class increments the value of *i* and stores it back in *i.* If two threads call the method at the same time, they are both going to load in the same value of *i,* they will both increment its value by one, and then store it back into memory. The problem with this is that calling that method twice should have increased its value by two, but it only increased its value by 1, because both threads took the same value of *i,* before one of them had time to update and store it back to memory. This is one of the issues that occurs when using multiple threads. This can typically be resolved by using the synchronized keyword in Java. Using the "synchronized" keyword when defining a method will ensure that only one thread can use that method for an object, and thus will prevent this issue. Another issue that occurs when using multiple threads is that when a thread updates a value, it will sometimes cache the value and not put it back in memory, to allow for quicker access. This is an issue when multiple threads are relying on this variable staying updated and being able to see any updates to the variable that another thread has performed. If a thread stores it in its cache, the other threads will not be able to see its update. To prevent this issue from occurring, we can use the volatile keyword when defining a variable to prevent that variable from being optimized and allowed to be stored in a cache, it will have to be stored in its location in memory.

Classes

NullState
The implementation of NullState does not do anything to the indexes in the array when the swap function is called on them, it is simply a method that is called and immediately returns. Because there are no values that are being accessed and loaded, changed, or stored, this implementation can be considered Data Race Free (DRF), although it is trivially so because none of the values in the array are even being accessed, so there is no possibility for a data race.

SynchronizedState
Synchronized State class implementation uses the "synchronized" keyword when defining the swap method. The synchronized keyword essentially creates a lock on the instance of the object that the method is called on. This prevents multiple threads from accessing the same object instance at the same time, which would create a data race. Locks are created on the object method, so no other thread can use that method at the same time in the same instance of the object. This will prevent any data races that can occur from loading values that aren't yet updated, and thus makes this program DRF.

UnsynchronizedState
Unsynchronized State class implementation is the same as Synchronized State, except it does not use the synchronized keyword when defining the swap method in the class. Thus, multiple threads can access this method at the same time, and so it is possible that an old value of the array can be loaded in one thread before another thread stores the updated value. Therefore, this class implementation is not DRF.

AcmeSafeState
AcmeSafe State class implementation uses the java.util.concurrent.atomic.AtomicLongArray package. This create a long array such that the elements in the array are updated automatically, in one step. AtomicLongArray uses special CPU instructions that can perform many operations in a single atomic instruction. This prevents the issue of having one thread store an old value in the array before the new value is stored by another thread. Thus, because there is no race in grabbing and storing data, this implementation is considered DRF. When updating the value in the Atomic Array, the methods getAndDecrement and getAndIncrement were used, which are methods in the AtomicLongArray class that atomically subtract and add, respectively, one to an index in the array.

Results

The results below show the average real swap time for each swap in nanoseconds. The program was run with varying parameters including 1 thread, 8 threads, 20 threads, and 40 threads, as well as varying array sizes of 5, 100, and 500. Additionally, figure 7 shows real total time for an array size of 100 on Linux server 6, *lnxsrv06.seas.ucla.edu*.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	34.61	938.848	3832.51	4737.6
Synchronized	40.19	2106.76	5533.38	12311.8
Unsynchronized	39.31	2302.39	5546.31	11672.4
AcmeSafe	44.66	1389.54	8837.52	14761

Figure 1: Results of Tests on Linux Server 6 with varying threads, array size of 5, and 1,000,000 swap operations. Results are average swap time real, in nanoseconds.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	30.12	1211.14	3235.49	4962.17
Synchronized	40.05	2126.52	5473.13	12732.2
Unsynchronized	40.07	2118.63	5126.51	11893.7
AcmeSafe	45.51	1431.08	8393.11	10361.5

Figure 2: Results of Tests on Linux Server 6 with varying threads, array size of 100, and 1,000,000 swap operations. Results are average swap time real, in nanoseconds.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	29.8	1411.32	2834.71	8505.89
Synchronized	43.7	2020.08	6181.92	12386
Unsynchronized	40.03	2211.19	5408.64	12095.4
AcmeSafe	44.23	1341.1	10387.8	19100.2

Figure 3: Results of Tests on Linux Server 6 with varying threads, array size of 500, and 1,000,000 swap operations. Results are average swap time real, in nanoseconds.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	30.2146	1556.92	3444.95	5657.77
Synchronized	39.29	3051.76	7433.58	15735.5
Unsynchronized	37.98	2400.32	7751.81	14203
AcmeSafe	48.67	1626.96	4663.39	10275.8

Figure 4: Results of Tests on Linux Server 9 with varying threads, array size of 5, and 1,000,000 swap operations. Results are average swap time real, in nanoseconds.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	30.06	1444.62	3099.18	4975.66
Synchronized	38.59	2639.54	7896.22	17133.3
Unsynchronized	37.76	2455.75	7208.54	15904.9
AcmeSafe	49.55	1413.14	4487.36	15512.1

Figure 5: Results of Tests on Linux Server 9 with varying threads, array size of 100, and 1,000,000 swap operations. Results are average swap time real, in nanoseconds.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	30.36	1489.95	2988.58	5877.73
Synchronized	38.39	3416.23	7810.22	19214.8
Unsynchronized	37.89	3505.46	8625.6	18015.4
AcmeSafe	48.66	1152.82	3998.92	9245.49

Figure 6: Results of Tests on Linux Server 9 with varying threads, array size of 500, and 1,000,000 swap operations. Results are average swap time real, in nanoseconds.

Class	1 Thread	8 Threads	20 Threads	40 Threads
NULL	0.03	0.15	0.16	0.12
Synchronized	0.04	0.26	0.27	0.31
Unsynchronized	0.04	0.26	0.25	0.29
AcmeSafe	0.046	0.17	0.419	0.26

Figure 7: Results of Tests on Linux Server 6 with varying threads, array size of 100, and 1,000,000 swap operations. Results are total real time, in seconds.

Analysis

Looking at the data, an increase in the number of threads increases the average swap time. This can be expected for the synchronized and DRF classes because there will be a wait time for a thread to access the array if another thread is currently using the swap function. The Null and Unsynchronized classes typically take the least amount of overall time, as can be expected because they have no restrictions to the number of threads that can access the swap function at the same time. However, there are data races in these because there are no

restrictions to the number of threads accessing the data at the same time. AcmeSafe also had a low total time for many of the test cases compared to Synchronized, and sometimes would even beat Null and Unsynnchronized. There is no locking like the Synchronized class which would explain the lower overall time compared to Synchronized.

Overall, the Null and AcmeSafe functions had the lowest average real swap time as the locking for AcmeSafe was optimized and only occurred when was necessary. This decreased the wait time and thus lowered the average time per swap.

In terms of correctness, Synchronized and AcmeSafe are the best classes to use. The class with the best speed and correctness is AcmeSafe, as it is atomic and doesn't require the use of locks, while Synchronized requires the use of locks. Synchronized use locks that take a lot of time to lock and unlock, while AcmeSafe uses atomic instructions that don't require the use of locks, while still ensuring that the program is DRF.

Difficulties

There were little challenges when completing this project, as it was mostly straight forward. The only slightly difficult method was current, because you had to change the data type from AtomicLongArray to long[], but this was easily solved by simply copying each value into a long array. Another tedious part was running all of the test cases. This was easily resolved by creating a shell script to automate the test cases for me and output them into a txt file.

References

AtomicLongArray – https://docs.oracle.com/cd/E17802_01/j2se/j2se/1.5.0/jcp/beta1/apidiffs/java/util/concurrent/atomic/AtomicLongArray.html