Computer Science 131 Homework 3 Report

Abstract

The synchronized keyword in Java allows sections of code to be designated as critical sections, such that those sections are executed sequentially. However, this is not necessarily the best way to designate critical sections in all cases. In this report, we compare and analyze four different synchronization methods in Java.

1 Introduction

The four different synchronization methods are as follows:

AcmeSafe This uses the class AtomicLongArray instead of the long array used in the other classes. This means almost every change to the array is atomic, but they are only atomic on the part of the array being changed.

Null This does not apply any changes to the array after instantiation. This class is for observing the base time of the program.

Synchronized This uses the synchronized keyword around the operation that changes the array. This means that each call to this operation occurs atomically.

Unsynchronized This does not have any forms of synchronization. Multi-threaded operations on this implementation often result in non-deterministic behavior as a result.

I tested these methods based on several benchmarks, which incremented and decremented elements of an array, and they varied in the number of threads, number of operations, number of elements in the array, and type of processor.

2 The AcmeSafe Implementation

The AcmeSafe implementation is more complex than the other implementations, so this report will give a more in-depth analysis of that implementation.

The main operation on the AtomicLongArray is an incrementation of an element followed by a decrementation of an element. This is implemented using the getAndIncrement and getAndDecrement methods of the array. Each of these methods, in turn, are based on the class VarHandle's method getAndAdd [1]. getAndAdd accesses and sets the value of the variable as if it was declared volatile, so the getAndAdd method uses the volatile mode [2]. According to an article by Doug Lea, if a piece of code is using volatile mode, then its instructions are totally ordered [3]. This means that each operation has its instructions occur in a specific order. This also guarantees that no information is delayed or lost due to the delaying or rearranging of instructions. Therefore, using volatile mode makes the code more consistent in a multi-threaded environment.

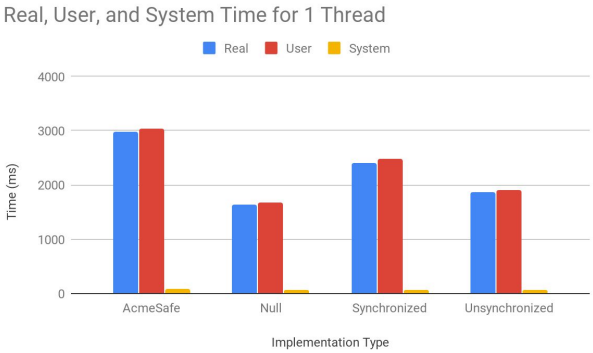
This increase in reliability comes at a cost, however. This mode causes a decrease in performance for the code compared to other modes such as release/acquire and opaque [3], since the rearrangement and delaying of instructions is often used by compilers to increase the speed of programs, so disabling these options means that these speedups are no longer in effect. The AtomicLongArray class attempts to offset this decrease in performance by treating each element of the array as a separate volatile variable, so the instructions for accessing and setting a single element have a determined order, but the instructions accessing and setting different elements does not have a fixed order, meaning that the overall operation is mixed mode [3].

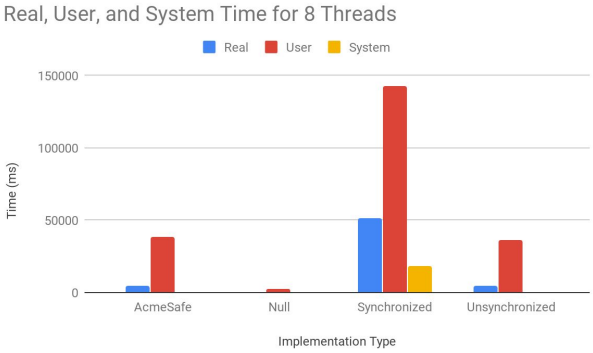
This mixed mode allows the operation to have task, memory, and instruction parallelism, while still being data-race free under the AcmeSafe implementation.[1] The Unsynchronized implementation also has these parallelisms, but at the cost of not being data-race free. The Synchronized implementation is data-race free, but loses memory and task parallelism, because the Synchronized implementation considers the array as one unit of memory, meaning that the operations occur essentially sequentially.

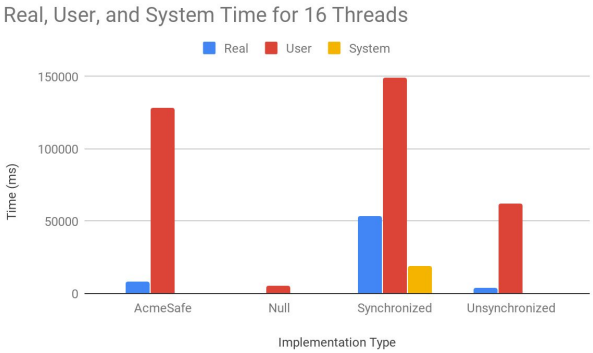
3 Number of Threads

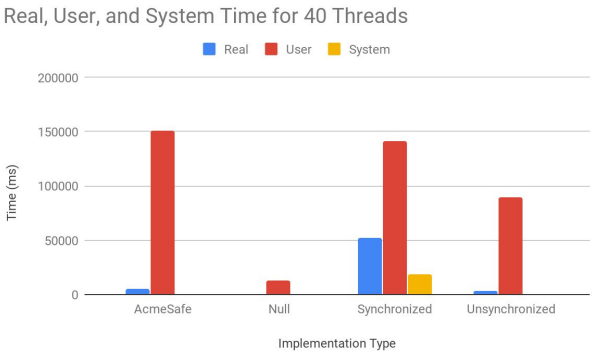
I ran tests on the different implementations with 100000000 operations on 100-element arrays, using variable numbers of threads. The results below were from linux server 7 at UCLA, when there was a relatively light workload on the server:

[1] Although the operation is data-race free, the entire class is not data-race free. The accessor method has almost no synchronization, so if the operation were running at the same time as the accessor, data races would occur.









It is clear from the graphs that the number of threads makes a difference in the relative times of the implementations.

With a single thread, the AcmeSafe implementation is much slower than the Synchronized and Unsynchronized versions. This is due to the AcmeSafe implementation having more synchronization checks than the Synchronized and Unsynchronized versions per operation. The base time from the Null implementation is a significant portion of all three of these implementations, which means the difference between the implementations is greater than it appears on the graph.

With 8 and 16 threads, the AcmeSafe implementation is clearly faster than the Synchronized version, achieving performance similar to the Unsynchronized implementation. The Synchronized implementation makes any operation on the array sequential with every other operation on the array, and that is much slower than the AcmeSafe implementation, because AcmeSafe breaks up the array into multiple pieces, and only operations on the same piece are required to be sequential.

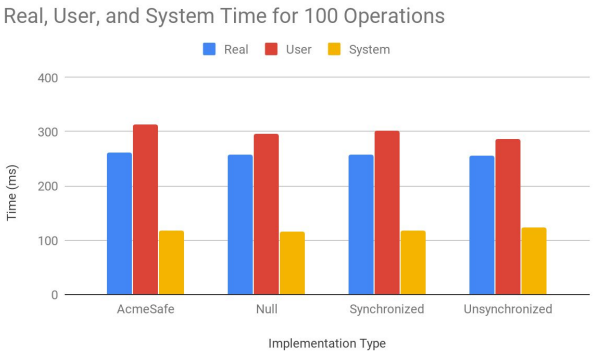
With 40 threads, the AcmeSafe and Unsynchronized implementations have similar real and system times, but the AcmeSafe implementation has a higher user time than the Synchronized and Unsynchronized implementations, and the Synchronized implementation has much higher real and system times compared to the other two implementations. The reason for the AcmeSafe implementation having such a high user time, but such a low real time, is that the getAndAdd method of VarHandle spins while waiting for the element to be available, so the more conflicts there are, the more the thread spins and adds to user time [2]. Greater numbers of threads causes greater numbers of conflicts, so as the number of threads increase, the user time of the AcmeSafe implementation increases. The real time is still short, however, because the getAndAdd method occurs fairly quickly, so the other methods get to execute fairly soon after they are called, which means the throughput of the operations is still higher than the Synchronized implementation.

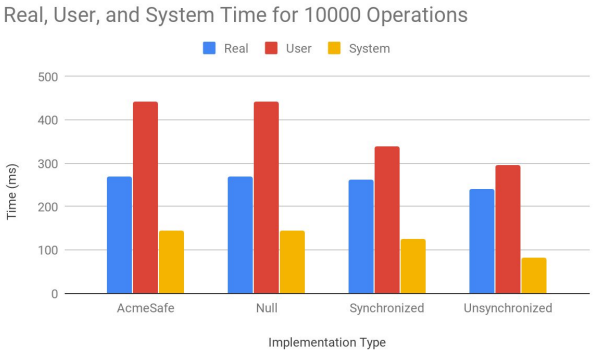
The Synchronized implementation uses system calls to ensure synchronization, so the Synchronized implementation uses more system time than the other implementations. This method of waiting also has more efficient CPU use, because less CPU is being used to check if previous operations are complete. However, the real time is much higher, because the Synchronized implementation still has all operations on the array occur sequentially.

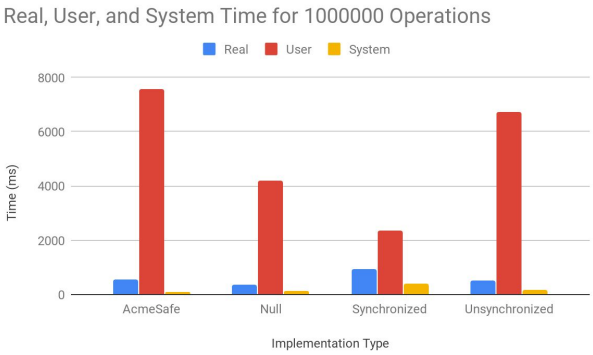
4 Number of Operations

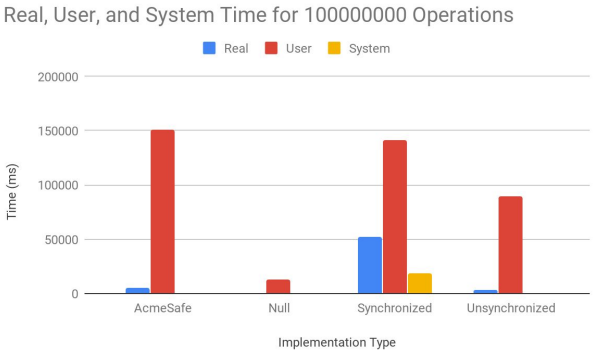
Then, I ran tests with 40 threads and a variable number of operations on 100-element arrays, also on linux server 7

at a time during which there was a relatively light workload. These are the results:







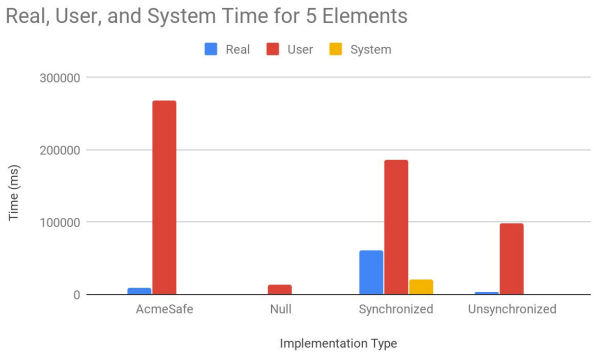


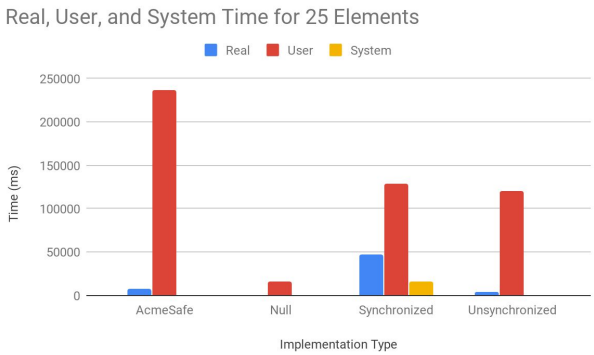
The relative times of the implementations are also affected by the number of operations. With relatively low numbers of operations, such as 100 and 10000 operations, the times for the different implementations are relatively close.

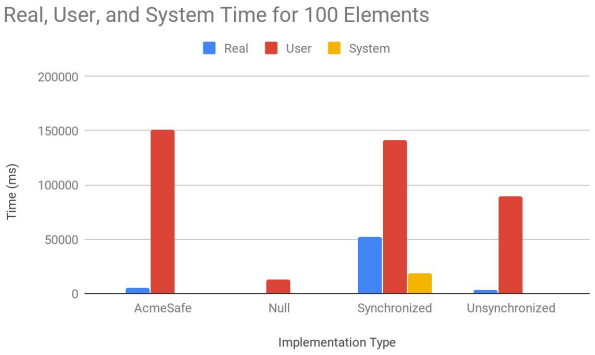
Significant differences emerge with 1000000 operations and 100000000 operations.With these numbers of operations, the AcmeSafe implementation has greater user time compared to the other two implementations, and the Synchronized implementation has greater real and system times compared to the other two implementations. This is the same result as for the tests with high numbers of threads. The same reasons apply to these results, because a greater number of operations increases the time the threads run, and the longer the threads run, the greater difference between the implementations seems.

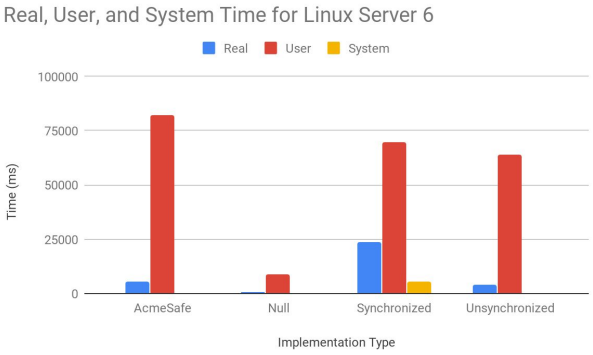
5 Number of Elements

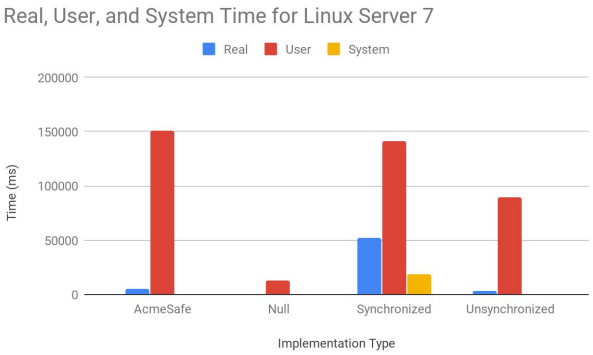
Then, I used the test that varied by the number of elements in the array. I ran the program with 40 threads and 100000000 operations on linux server 7. The graphs below show the results:











As expected, the relative times varied with the number of elements in the array. Although the graphs may appear the same at first glance, there are significant differences between them. As the number of elements increases, the real and user times of the AcmeSafe implementation decrease. Since the times of the other implementations remain approximately the same value, they increase in relative value to the AcmeSafe real and user times.

The increase in number of elements allows the AcmeSafe implementation to multithread the memory accesses, which makes it faster in real time than the Synchronized implementation. In fact, the more elements there are in the array, the faster the AcmeSafe implementation becomes if the other factors are the same, because there are fewer inter-thread conflicts when there are more elements, so the AcmeSafe implementation spends less time spinning when there are more elements.

6 Type of Processors

I also ran the same tests on linux server 6. Linux server 7 has 8 CPU cores, and linux server 6 has 4 CPU cores. Also, the CPU model of linux server 7 has 2.00 GHz and the model of linux server 6 has 2.40 GHz, though both models are Intel Xeon. This means that there are likely differences between the two servers. The two servers had similar loads when I ran the tests. Below are two graphs of the same test (40 threads, 100000000 operations, and 100 elements), but on the different servers:

The same general trends occur on the different servers. However, all the times on linux server 7 are higher than the times on linux server 6. This is likely due to linux server 7 having a lower power of 2.00 GHz compared to linux server 6's 2.40 GHz. Although linux server 6 has fewer cores than linux server 7, and therefore is forced to context switch more than linux server 7, the time lost from more context switches appears to be much less than the time gained from superior processing power. This does not appear to significantly affect the ratios between the times, however.

Conclusion

Besides the Null implementation, the Unsynchronized implementation has the fastest implementation of the operation. It does not experience any performance bottlenecks from synchronization techniques. Unfortunately, this causes the Unsynchronized implementation to perform incorrectly in multi-threaded environments. For single-threaded or single-element applications, the Synchronized implementation is the next fastest. Since there are no competing threads for the synchronization lock, the Synchronized implementation receives little delay in obtaining and releasing its lock. For multi-threaded and multi-element applications, the AcmeSafe implementation is fastest after the Unsynchronized implementation. It uses a

test-and-exchange implementation for each element, which allows different threads to change different elements at the same time while still retaining correct behavior. The AcmeSafe implementation, however, uses more user time than the Synchronized implementation, because the AcmeSafe implementation includes spinning, which uses more user time than locks.

Data

The data referenced in this report can be found at the following links, which were omitted from this report due to size restrictions:

Linux Server 6
https://docs.google.com/spreadsheets/d/1aavdCnDIqU77pAgB74xTpVQnjfnZrzV3oVqoGzPwY0Y/edit?usp=sharing
Linux Server 7
https://docs.google.com/spreadsheets/d/1A-stOg-splmQlhVRV6ICxytuGhAurTmb_4FFRtOb0rE/edit?usp=sharing

References

[1] Class AtomicLongArray. In Java SE 13 & JDK 13, 2019. https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html

[2] Class VarHandle. In Java SE 13 & JDK 13, 2019. https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/invoke/VarHandle.html

[3] Doug Lea. Using JDK 9 Memory Order Modes. 2018. http://gee.cs.oswego.edu/dl/html/j9mm.html