CS131 Homework 3 Report

Abstract

In this lab I aimed to test different methods of synchronization in Java and compare their speed. The critical section in this homework was an increment and decrement of two portions of an array. All testing was done on Linux Server 09.

1. Testing Methods and Results

In order to test each method of synchronization, I ran each method with 2,4,8,16, and 32 threads with a maxval of 127. The array size was 100 bytes.

In implementing BetterSafe, I decided to use the library java.util.concurrent.locks because that allowed to gain the finest granularity of locking. This allowed me to only lock the array specifically when the add operations were being performed and immediately release them after. I did not use java.util.concurrent because java.util.concurrent.locks is simply more specific than it. I did not use java.util.concurrent.atomic because that was already implemented in GetNSet with the atomic array. I did not use the VarHandle module because it also implements atomic operations, and I wanted to use something different than what is in the other files.

Model	8 Threads (ns)	16 Threads (ns)	32 Threads (ns)
Null	230.22	660.12	1552.22
Synchronized	2129.31	4342.67	8982.10
Unsynchronized	402.12	790.21	1788.22
GetNSet	1620.21	2800.22	4765.22
BetterSafe	1761.48	3267.34	5789.22

1.1. Results and Interpretation

What immediately stands out is that synchronized is the slowest, especially for large amounts of threads. The advantage of using this method is that it is in fact Data Race Free or DRF.

Another standout is that Unsynchronized is the fastest, but that is because it does not eliminate any race conditions and in fact just ignores them.

GetNSet is faster because of its use of the AtomicIntegerArray. This is because it implements a finer granularity of locking in comparison to synchronized. The AtomicIntegerArray simply locks the array itself when it is being used, while synchronized locks the entire method or class while it is being used. In my testing, GetNSet is even faster than BetterSafe in performing these operations. The tradeoff however is that GetNSet is not DRF. BetterSafe is completely reliable in contrast.

1.2. DRF Analysis

Synchronized and BetterSafe are the two types that are DRF. They are DRF because they lock the critical sections of the program until they are released. Null is also DRF for obvious reasons. Unsynchronized is not DRF because it does not do anything to make sure that the operation's integrity is conserved. Any test run enough times can be used to show that Unsynchronized is DRF, such as *java UnsafeMemory Unsynchronized 4 8 1000000 6 5 6 3 0 3*.

A less intuitive finding is that GetNSet is not DRF. GetNSet is not DRF because if GetNSet is running a comparison on an array element that is maxval-1, that is it is one below the maximum value, it can be preempted while that array element is being incremented, during which another thread also increments that value. When control is returned to the original thread, the value is incremented again which leads to the element having value maxval + 1. Although this is rare, it does mean that GetNSet is not DRF. One way to test this would be to make the maxval very low, so that the likelihood of this happening increases. However, it is very difficult to replicate. This command did induce the situation I described above *"java UnsafeMemory GetNSet 4 2 1000 6 5 6 3 0"*.

2. Conclusion

In Conclusion, GetNSet is probably the correct choice for GDI, because it both performs quickly and it is "good enough". In this case, good enough means that failure is extremely unlikely and it still performs better than both Synchronized and BetterSafe.