CS 131 Homework 3 Project Report

Abstract

The aim of this lab was to test different variations of concurrent mechanisms when working with swapping of numbers with an increment and decrement. After creating the different mechanisms, we had to test them using the SEASnet server 9 running Red Hat.

1. Test Results

To test our different mechanisms, I decided to run 10,000,000 swaps with 8, 16 and 32 threads and using an array of 650 bytes in the range from 0 to maxval (maximum of 127). The results of the various methods are highlighted below.

Model Used	Time taken for transitions /ns			DRF
	8 threads	16 threads	32 threads	
Null	188.79	565.01	1673.58	Yes
Synchronized	2615.51	4827.44	9202.27	Yes
Unsynchronized	448.90	790.42	1469.01	No
GetNSet	564.72	918.58	2099.59	No
BetterSafe	765.89	1502.96	3107.01	Yes
BetterSorry	510.06	864.54	1392.46	No

1.1. Comparisons

We see that synchronized seems to do the worst out of all the different models, taking the longest to perform the swaps, but is data race free (DRF) since the synchronized keyword in Java makes the swap method blocking, preventing pre-emption by other threads. Unsynchronized is clearly the fastest model because it doesn't have any overhead in performing blocks against competing threads, and just allows whichever thread comes first to perform the swap which in turn makes it prone to race conditions which is why it always gives incorrect results!

We see that GetNSet is much better in performance than Synchronized because it uses an AtomicIntegerArray which is essentially implements finer grained since instead of blocking the entire swap method as Synchronized does, it only makes the access to array elements atomic. It seems to perform even better than the BetterSafe model which uses a ReentrantLock to lock the critical sections of the swap method. However, BetterSafe is 100% reliable as it locks the swapping method in a finer grain than Synchronized does. It is slower than GetNSet because with GetNSet we are using atomic operations which don't require any kind of locking mechanism where other threads have to be blocked.

As mentioned above, BetterSafe is faster than Synchronized because of a finer grained locking mechanism that locks only the sections that are susceptible to race conditions whereas the synchronized keyword locks the entire function.

We also notice that BetterSorry is the second best to Unsynchronized. This is because it uses AtomicIntegers which perform the updates atomically and thus don't need to block other threads over a coarser definition. This is why it is faster than BetterSafe (for reasons analogous to that for GetNSet). However, the reason it is faster than GetNSet is because with AtomicIntegerArray the any operation on the entire array is atomic, which requires some internal locking mechanism which causes more overhead than when we have an array of atomic objects (as is the case with BetterSorry) which would make updates to the object itself atomic – note how this is finer than the case with GetNSet. This is why it is even faster than GetNSet. The reason it still loses out to Unsynchronized is because there is still some internal locking mechanism to make the operations atomic which will have some blocking/wait times associated while unsynchronized has no wait time.

1.2. Issues with DRF

Synchronized and BetterSafe are DRF because they effectively lock the critical sections of the swapping mechanism and thus if there are any competing threads that attempt to perform the update, they are set to a waiting state because of the currently held lock. Unsycnhronized is clearly not DRF because there is no locking mechanism and so threads can come and update whenever they feel like, thus creating several points of race conditions, but this is also what gives it the fastest times. Any test can show the non DRF property of Unsnchronized such as "*java UnsafeMemory Unsynchronized 8 1000000 127 120 12 3 4 19 0 10*"

The reason GetNSet and BetterSorry are not DRF is the same. We can take the following example. Let us as-

sume the MaxVal is set to 127. If thread A is running a comparison on value[j] where value[j] == 126, it performs the check and sees that the condition passes and so it moves to perform the update on value[j] (that is it is about to increment the value) when it can get preempted – because there is no lock preventing an interruption at this point – when Thread B performs it's swap function, seeing value[j] == 126 which also passes the condition, and Thread B continues to update the jth value to 127. When Thread B yields control back to Thread A, it will continue its execution of incrementing the jth value which is now at 128 leading to a non DRF condition. This is very difficult to test with certainty however because the Operating System or in this case the JVM will be responsible for the control of threads and interruption, and so this condition is very non-deterministic and thus hard to replicate.

This is one of the issues faced in trying to perform the measurements as it is extremely difficult to reproduce hypothesis that are based on non-determinism. The other issue faced when trying to take measurements is the number of swaps and the size of the input array to produce results that fit in line with expectation. Only after noticing the TestSwap and the way it performed, was I able to deduce that we needed a large input array. Server load also caused issues as the more people used the CPU, the slower the times I got since there are fewer resources to run the threads. To overcome this problem, I just had to wait until the server load decreased.

2. Conclusion

To finally answer the question as to which model to use for GDI's implementation, I would recommend BetterSorry, since it performs the best second to Unsynchronized and only has a few cases where it would cause issues (unlike Unsynchronized which constantly fails) which is deemed acceptable by the requirements for the implementation.