CS131 Homework Three Report

Shannon Hoang, *UCLA*

Abstract

The focus of project three was to examine and evaluate the different options and tools for concurrency in Java. Another purpose of project three was to gain greater insight on the Java Memory Model, as why which concurrency methods work and which concurrency methods fail depend on it. In regards to the code specifically, our job was to run the code concurrently while eliminating race conditions. It was found that while synchronized Java classes lessened or eliminated race conditions, the overhead would cause the program to run slower. On the other hand, unsynchronized Java classes were much more efficient, however they resulted in inaccurate results as a result of having race conditions – two threads could access and alter the same resource at once.

1. Testing Platform

To check the version of java I was using I used the command java –version. There were two versions of java that we were to test our code on the UCLA SEASnet server with: version 11.0.2 which is the default, and openjdk version 9. By preappending /usr/local/cs/jdk-9/bin:" to path we could temporarily switch to version 9.
I did all of my testing on seasnet server9.
As per the spec's instructions, I inspected the files /proc/cpuinfo and /proc/meminfo to gather some hardware data on my testing platform so that others could recreate my datapoints if they needed to.
The SEASnet server on which I tested has 32 Intel Xeon E5-2640 v2 2.00GHz CPUs each with 8 cores, with a cache size of 20480 KB in L1, and 46 bit physical addresses, 48 bit virtual addresses, and 64 cache alignment.
Using /proc/meminfo I found some more important information. The server memory total is 65755884 kB and MemFree is 53104268 kB.
The size of the kernel stacks is 15328 kB and the page tables are 116524 kB.

3. Analysis of Packages

We had four different packages from Java that were suggested to us to implement BetterSafe – a class intended to achieve better performance than Synchronized while still being Data Race Free(DRF).

1. Java.util.concurrent

   Java.util.concurrent's resource control is simultaneously its advantage and disadvantage. With the concurrent package, we can become very specific with the concurrency implementation – ie Executors-- and the data structures(queues,dequeues) we use to synchronize our threads. However, this level of specificity would knowing the exact ordering of the threads so that they can be placed into the corresponding data structure in the correct order. Therefore the main disadvantage of this package is that the level of complexity makes it very difficult to organize a large nuber of threads. This solution did not seem simple enough to suit my needs to implement better safe, so I did not go with this package.

2. Java.util.concurrent.atomic

This package gives classes that do not use locks to restrict access to resources. The pro of this package is that programmers do not have to concern themselves with locking and then unlocking the data resources as the class does that on its own. Additionally, the implementation of the "locking" for this package is done in such a way that separate data structures are locked separately, so locking one array would not affect the other. But, this is not the case with our swap function, where we deal with a single array and would like to swap values available in our array. The mechanisms of the atomic package would still allow a data race in our critical section if one thread interrupted another which would mess with our result.

3. Java.util.concurrent.locks

The locks package provides classes and tools for locking and waiting for conditions which are separate from built-in synchronization and monitors. The package provides a lot more flexibility in terms of restricting resource. However, like the concurrent package, this flexibility comes with complexity in needing to know exactly which area of the program needs to be done sequentially and without shared resources. Another con is that locks work like the keyword Synchronized, in that they restrict the entire object, even though modifying a different part of the object may be completely acceptable. However overall they win out in comparison to the other packages as we know exactly which part to our program needs to have restricted access and locks are incredibly simple. Therefore, we can lock the critical section of our program while reducing the overhead of the Synchronized keyword and keeping our results data race free.

4. Java.lang.invoke.VarHandle

   Is a package that allows us to create variables to which access is supported only under certain access modes. The pro of VarHandle is that with it, we can restrict access to specific types besides just integers. However, since we are dealing with just integers in our program, the level of specificity is not necessary. Additionally, it would not completely restrict access to the critical section and would still allow another thread to interrupt that area.

3. Testing Method

In order to compare how the different concurrency methods compared to each other, I ran each different concurrency method with varying thread numbers, swap numbers, and element numbers for each trial. The same configurations were done for both versions of Java 11.0.2 and Java 9.

Configurations:

2 threads, 100 swaps, 5 items
6 threads, 1,000,000 swaps, 5 items
6 threads 10,000,000 swaps, 5 items
12 threads 1,000,000 swaps, 5 items
12 threads, 10,000,000 swaps, 5 items
6 threads 1,000,000 swaps, 700 items
12 threads 1,000,000 swaps, 700 items
40 threads 1,000,000 swaps 5 items

Figure 1. Java 11.02: 5 items, 1,000,000 swaps, varying thread numbers

(Time in Ns/transition)

Class				DRF
#thread	6	12	40	
Null	1298.66	4161.31	9663.68	Y
Sync	2138.38	5184.1933	29927.4	Y
Un-sync	FAIL	FAIL	FAIL	N
GetNSet	542686.22	119390.6	475533	N
BetterSafe	2520.06	4816.54	20433.83	Y

Figure 2. Java 9: 5 items, 1,000,000 swaps, varying thread numbers

Class				DRF
#thread	6	12	40	
Null	1390.66	3150.64	10068.0	Y
Sync	1055.91	7132.773	15792.6	Y
Un-sync	FAIL	FAIL	FAIL	N
GetNSet	48579.5	96002	352106	N
BetterSafe	2493.74	4431.91	19125.9	Y

Figure 3. Java 11.02: 5 items, 10,000,000 swaps, varying threads

Class			DRF
#thread	6	12	
Null	48.1498	107.772	Y
Sync	2604.393	4635.99	Y
Un-sync	FAIL	FAIL	N
GetNSet	61349.2	FAIL	N
BetterSafe	2376.78	442.493	Y

Figure 4. Java 9: 5 items, 10,000,000 swaps, varying threads

Class			DRF
#thread	6	12	
Null	118.17	399.99	Y
Sync	1711.06	3515.22	Y
Un-sync	FAIL	FAIL	N
GetNSet	FAIL	108475	N
BetterSafe	2185.24	4221.68	Y

Figure 5. Java 11.02: 700 items, 1,000,000 swaps, varying threads

Class		Ns/transition	DRF
#thread	6	12	
Null	4.94	16.11	Y
Sync	4.60	16.38	Y
Un-sync	FAIL	FAIL	N
GetNSet	5.12	14.40	N
BetterSafe	5.31	14.12	Y

Figure 6. Java 9: 700 items, 1,000,000 swaps, varying threads

Class		Ns/transition	DRF
#thread	6	12	
Null	5.09	15.57	Y
Sync	5.20	16.83	Y
Unsync	4.40	15.17	N
GetNSet	4.56	15.74	N
BetterSafe	4.65	14.97	Y

Evaluation and Analysis

Explain whether your BetterSafe implementation is faster than Synchronized and why it is still 100% reliable. Characterize your implementation's basic idea using terminology taken from Lea's paper.

While BetterSafe in some data points is actually slower, overall BetterSafe runs faster. Its speed is more clear at larger swap and element counts. Overall BetterSafe scales better because it has less overhead in comparison to Synchronized. In addition to this, it is completely Data Race Free and never results in a sum mismatch. This is because it locks the critical section of the program at a finer level than Synchronized – instead of locking the entire function, we are only locking the critical area in the function. But it must be noted that GetNSet is faster than Better Safe with a large number of elements because it uses AtomicIntegerArray, and uses atomic operations instead of locks and as a result has less overhead with reads and writes.

As a result, since Better Safe is completely DRF, and is faster than Synchronized and scales better than Synchronized because of the lessened overhead it is the better implementation for the specified purpose.

A few things to note:

The majority of the time, unsychronized either ran infinitely or failed to run at all. This makes sense as it is simple the same implementation as synchronized except without the synchronized keyword, leaving the program open to race conditions, various exceptions, and potentially infinite loops. However, at higher element numbers, we see unsynchronized is actually functional, and this is because the number of elements is so high that the likelihood of collision severely decreases.

While GetNSet would be able to be faster than Synchronized, it was not completely DRF and would often result in sum mismatches even though the program would be able to finish with GetNSet. GetNSet is not DRF because the atomic operations do not restrict access to the entire critical section, access to all the array elements isn't restricted.

Challenges

A major challenge of this project was simple testing and collecting all the data. I wanted to have a large number of data points so that I would be able to compare the concurrency methods effectively.

Another challenge I had to address was creating a testcase in which there were 700 elements to the array instead of a more negligible number like the standard 5. This I was able to do by using a command similar to:

mapfile -t myArray < randomnums.txt | java UnsafeMemory GetNSet 12 1000000 126 $myArray

where randomnums.txt is a file of 700 randomly generated numbers, each on their own line, between 1 and 126. mapfile maps each line in a file to a specified array. Once I had the array I simply passed the array into the corresponding test I wanted to run.