Homework 3: Java Shared Memory Performance Races

Abstract

In this lab, we measure 5 different classes, Synchronized, Unsynchronized, Null, GetNSet, and BetterSafe to notice which performs the best over different contexts and which is ultimately more reliable. Later, I discuss why I chose to use the package java.util.concurrent.locks over the others for my implementation of BetterSafe Synchronized and why it is the best choice for GDI applications.

1 Testing Platform

In order for others to one day replicate similar results as me, here I describe the specifications of my testing platform. First off, after running the command

```
java -version
```

on SEASnet server 09, I discover that I am using jdk version "1.8.0_191" and that I'm running on an OpenJDK Runtime Environment (build 1.8.0_191-b12) and OpenJDK 64-Bit Server VM (build 25.191-b12, mixed mode).

After running the command

```
/proc/cpuinfo
```

I notice that the SEASnet server I perform my measurements on has 32 Intel(R) Xeon(R) E5-2640 v2 CPUs at 2.00GHz. Each of the processor has an L1 cache size of 20480 KB and 8 cpu cores. Additionally, each processor has 16 siblings, 46 bit physical and 48 bit virtual addresses. Next I run the command

```
/proc/cpumeminfo
```

We notice that the SEASnet server as 65.75 GB RAM with 49.93 GB RAM free and 62.33 GB RAM available. We have 200 GB of Swap space, a Kernel stack of size 12560 kB, and Page Tables of size 80164 kB.

2 Performance and Reliability

I decided to run the four methods, Synchronized, Unsynchronized, Null, BetterSafe, and GetNSet for a varying number of threads and swap conditions.

2.1 Performance Plots

Here I plot how each method performed against 2,4,8,16,24, and 32 while also comparing against 1 million, 100K, and 10K swap transitions on an array of random values with max-val = 127. I plotted these using the language R, and library ggplot2.

Figure 1

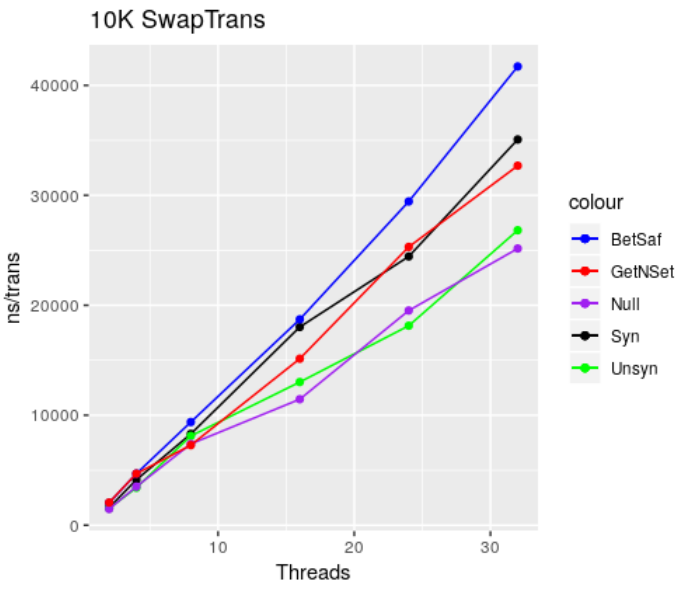

Figure 2

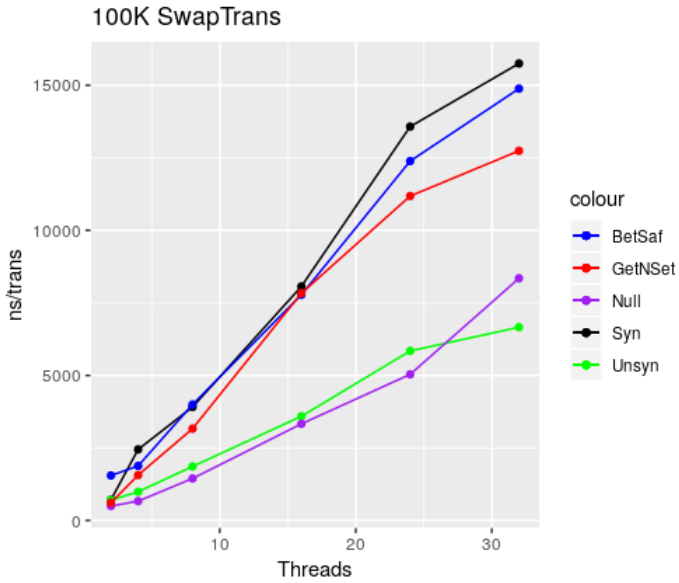

Figure 3

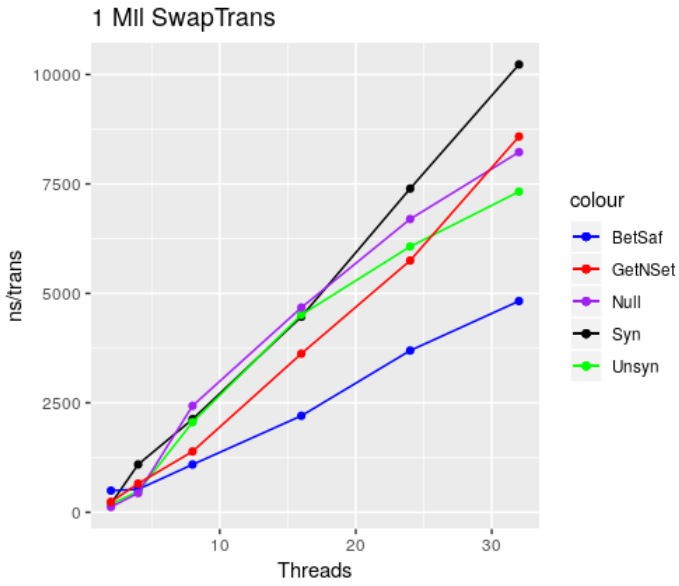

Figure 4

	Unsyn	Syn	Null	BetSaf	GetNSet
DRF	No	Yes	Yes	Yes	No

2.2 Analysis

Across each figure, we first notice that Unsynchronized always performs faster than Synchronized as is expected since there is less overhead to check for synchronization. However, there's no free lunch, so Unsynchronized has a weaker memory model as indicated in Figure 4. It experienced a sum mismatch very often, making it unreliable.

Next, we notice that Null performed particularly well for fewer swap transitions without ever making sum mismatch mistakes. However, for a larger number of swap transitions, it drastically performed worse.

GetNSet follows a similar trend to synchronized, but performs just a little faster because it contains no synchronization keyword that would otherwise slow it down. It consistently performed worse than Null and Unsynchronized due the fact that it must perform an ounce of synchronization when updating the AtomicIntegerArray. Comparing GetNSet to BetterSafe, we notice it only is faster for a smaller number of swap transitions but BetterSafe performs better for larger numbers of swap transitions.

Lastly, BetterSafe performs drasticallly better than the rest at scale since it is able to synchronize the critical section pnly using more granular locking mechanisms. This proves much faster than synchronizing an entire function, and also maintains reliability.

Overall, we notice that BetterSafe is the best performer at scale (large number of swap transitions and threads) while making no sum mismatches and is recommended for GDI applications.

3 Comparing Packages

3.1 java.util.concurrent

Pros: This package contains advanced synchronization data structures which can allow a developer to be more detailed in terms of their synchronization of various threads. For example, we can create a BlockingDeque which can perform thread-safe operations using internal locks and other forms of concurrency control.

Cons: The choice of using this package for implementing BetterSafe comes at a major cost of complexity. At this level of hierarchy and using the classes it has to offer, it is extremely difficult if not careful for synchronization of threads being run from an object method. I chose not to implement this package due to these reasons, and performed further exploration.

3.2 java.util.concurrent.atomic

Pros: This package contains a variety of classes that do a lot of the work in terms of synchronization without having the programmer explicitly placing locks. The package allows for more granular mutual exclusion as some of the package's data structures will lock themselves to allow for accesses.

Cons: This package provides functionality for protecting reads and writes but only for one or the other. In the base

of BetterSafe, we are performing reads and writes so this package will not suffice.

3.3 java.util.concurrent.locks

Pros: This package provides flexible lock classes allowing the programmer to be free to decide where they would like to protect critical sections. Once a type of lock is identified, we can explicitly put the necessary guard around just the critical section rather than the entire function. I chose to use the Reentrant lock as it allows for mutual exclusion giving us full reliability. I used this package to implement BetterSafe for these reasons.

Cons: Since we are given many choices about which locks to use and freedom to place them wherever we want, it adds additional difficultly in simply deciding which type of lock to choose. Locks in general do not allow for multiple threads to modify data that is already owned by another thread, so it may not be application under different circumstances. In our case, it is a perfect fit.

3.4 java.lang.invoke.VarHandle

Pros: The package came after the java.util.concurrent.atomic package and has wider applicability. It allows for one to synchronize not only an array of integers, but also objects.

Cons: Most importantly, this package does not provide mechanisms to protect the critical section, so an untimely interrupt could erase BetterSafe's reliability. Also, it's not necessary since the only objects we are only dealing with are integers.

3.5 BetterSafe $>$ Synchronized

My BetterSafe implementation is faster than Synchronized due to its ability to only place locks around the critical section rather than the entire function. The synchronized keyword adds a lot of additional overhead, and as we saw from Figure 3, BetterSafe will perform better at scale for GDI applications.

Using some terminology from Lea's paper, my implementation of BetterSafe has an memory order mode with the property of partial ordering, meaning two events do not need to be related at all but there are no cycles. My critical section is able to be atomically executed since the linear extension of a partial order obeys all ordering constraints allowing for shared memory within the critical section to be modified by one thread at a time. Also, threads will still maintain their locks through an interrupt, thus we still able to maintain reliability. By simply creating a private reentrant lock variable and placing it around the critical section of the swap function, we can acheive all this.

4 Problems

Gathering all these measurements proved extremely tedious executing them on the command line. Instead, I created a large makefile which would allow me to write all my tests in one location. The only issue that came out of this, would be that the operation would hang on a non DRF method, thus stopping the script. Interestingly, whenever a particular method would not complete, and I interrupted to try another test, the properties of the terminal slowly *melted* in the sense that the view of the command line would get overwritten by other text. It was extremely odd, and I had to create a new terminal often. Eventually, I was able to obtain the measurements I desired.

5 DRF

DRF Classes: Synchronized is DRF because it locks the entire object until completion using the synchronized keyword, thus allowing for no mismatch of sum. Null is also DRF as it does not even modify any shared memory making it impossible for a sum mismatch to occur. BetterSafe is DRF since its critical section is protected using a reentrant lock, only allowing for shared memory to be modified by one thread at a time.

Non-DRF Classes: Unsynchronized is not DRF as it provides no guard whatsoever over the critical section. A command it is likely to fail on the SEASnet GNU/Linux Servers is "`java UnsafeMemory Unsychronized 32 1000000 3 {large arr}`" since it is likely for a sum mismatch and data race to occur with a low maxvalue, large number of threads, and large number of swap transitions. GetNSet is not DRF since it only gives independently protected accesses, meaning that it does not fight against circumstances where a thread preempts for example. It only protects certain pieces of the critical section. The same command used to most likely fail for Unsynchronized should also cause a sum mismatch for GetNSet.

6 Conclusion

In this assignment, we performed performance and reliability measurements on the sequential-consistency-violating test program we built. We discovered that among all methods, BetterSafe was the best choice in terms of reliability and performance at scale. We also discussed pros and cons of different java packages I considered when implementing BetterSafe and why it outperforms Synchronized. In the end, I recommend BetterSafe for GDI applications.