Computer Science 131 (Programming Languages): Homework 3
Java Shared Memory Performance Races

1 Introduction

This lab was conducted as a consultation to Ginormous Data Inc. (GDI) for their concurrency application. They experience significant overhead from using their synchronized Java application and asked us to come up with a way to improve efficiency regarding this idea while doing our best to not sacrifice correctness (although they are ok with some errors). In order to demonstrate this idea to them, we have 3 applications: SynchronizedState, UnsynchronizedState, and AcmeSafe. SynchronizedState is the company's current application, UnsynchronizedState is a free for all with no protection against race conditions, and AcmeSafe is an approach we designed to minimize overhead without sacrificing correctness. As we will demonstrate through the measurements, AcmeSafe still operates with 100% correctness with significantly reduced overhead times. We designed AcmeSafe using the ReentrantLock.

2 Pros/Cons of AcmeSafe Approaches

There were many packages and classes recommended to us. We explore the pros and cons of each of these packages or classes below.

2.1 java.util.concurrent

This package has many capabilities that we can use to create concurrency, namely there are 5 synchronizer classes that we can use. While there are queues and timing units, these are unsuitable for our particular problem. If the problem were slightly different we could use the concurrent collections, but we are using an array and not an ArrayList so this is also not a suitable option. The 5 synchronizer classes are the Semaphore, CountDownLatch, CyclicBarrier, Phaser, and Executor. The Semaphore class implements a classic semaphore which can limit access to a shared resource. This overhead however seems unnecessary to adapt a semaphore for our limited use case. A CountDownLatch is similar but based on some incremental conditions becoming true. Again, it could work but provides extra overhead. A CyclicBarrier is a rolling access method where a number of threads must wait for a common convergence point before the resource is acquired. Since we only want one thread accessing at a time, this doesn't make much sense either. A Phaser is analogous to a CyclicBarrier and CountDownLatch with added capabilities, so it would add even more overhead. Finally, an Executor can be used to exchange objects between two threads at rendezvous points which also would be expensive to adapt for our use. None of the classes inside this package seem to be particularly helpful in our case, however, the subpackages inside this package will prove to be more promising.

2.2 java.util.concurrent.atomic

This package provides lock-free thread-safe programming on specific objects. It provides support for an AtomicBoolean, an AtomicInteger, an AtomicLong, and an AtomicReference for any object. While these single variable classes wouldn't help us since we are modifying an array, this package does offer subclasses that extend the ideas here for arrays. We can analyze this idea in a later section.

2.3 java.util.concurrent.locks

This package provides locking mechanisms with better flexibility and capabilities than the simply synchronized syntax or a semaphore. The idea behind the locks is fairly similar and there are several types of locks for different purposes. For example, the StampedLock provides three modes based on capabilities to determine read and write access. While this lock would be overkill for us, it just goes to show how versatile this package is. The ReentrantLock is one of the locks that is defined using this framework and is best suited for our purposes. It is owned by the last thread that locked it but didn't unlock it yet. When the lock is held by another thread, the current thread is put to sleep so other threads can try to get the lock or can keep working if they already have the lock. This saves a ton of resources. When a thread calls this lock and it isn't own by another thread, it returns and acquires the lock. Otherwise, it returns automatically and immediately. Since this lock has high granularity and is only used when needed, while limiting CPU/memory usage, it will increase the efficiency and decrease the overhead without sacrificing correctness. All of these factors make the Reentrant Lock the best suited mechanism we can use for AcmeSafe and provide GDI with a faster solution that is as safe as the synchronized version.

2.4 java.util.concurrent.atomic.AtomicIntegerArray

This class is another valid option that can be used for AcmeSafe. In fact, based on its description it would seem like the perfect candidate for GDI. The entire array can be atomically updated so it would either update entirely or not at all. This would also be the best granular level that we desire so efficiency would be maximized. However, despite appearances, this class would not necessarily provide 100% correctness for us. Since swapping is not an atomic function supported by this array and is actually two atomic steps, if something were to occur between these two steps, then consistency would be lost. Since I'm aiming for a more efficient solution that does not sacrifice consistency, I would not prefer this class over a Reentrant Lock.

2.5 java.lang.invoke.VarHandle

This class provides a reference to a variable with specific access modes. While we can try to adapt this class to use with our array, it would add a ton of unnecessary overhead that can be solved with a simple lock. Instead of defining access modes and the compare-and-set method, it would make more sense to use a lock.

3 AcmeSafe Implementation

Since the goal of AcmeSafe was to create a state that provides "better performance than Synchronized, better safety than Unsynchronized, or (preferably) both," I realized we could do so by using locks. More specifically, I used the Reentrant Lock (java.util.concurrent.locks.ReentrantLock) to accomplish this objective. My reasoning behind using this lock is fairly simple: I would lock only the critical parts of the code and allow the rest to run in parallel. Thus, I lock the Reentrant Lock when the values are accessed and compared, and unlock if the if condition is true or if the condition is false, after the swap has been made. Thus, the whole method is not blocked but only the critical memory sharing components are. So the correctness of the state is still 100% but the performance will improve significantly.

In terms of being Data Race Free (DRF), my implementation is DRF. The reasoning behind this is very simple: AcmeSafe uses the Reentrant Lock to maintain total ordering of the data accesses. While the rest of the program engages in instruction parallelism, the data accesses are strictly regulated by acquiring the lock and releasing it after the shared memory portions are complete. Thus, no races will emerge.

4 Measurements

For each state, I ran several variations on number of threads, maxval, size of the array, and number of swap transitions. While all of these can be shown here as valid measurements, the best demonstration of the different states capabilities can be shown through number of threads. For the table below, I ran 1000000 swap transitions on a 6-element array with a maxval of 20.

Number of Threads	Null	Synchronized	Unsynchronized*	AcmeSafe
1	34.0593	65.4901	53.7251	98.0570
2	112.474	369.828	196.200	512.193
4	296.548	1345.66	442.674	760.513
8	1496.21	2797.15	1097.78	1216.49
16	3677.40	5543.27	3856.94	2428.34
32	5033.50	12295.9	8721.66	5579.73

*The numbers above were obtained from the average of several runs on two separate machines. While the Null, Synchronized, and AcmeSafe ran with 100% consistency each time, the Unsynchronized state often failed to complete. After the number of threads was increased past 1, it would often hang, so the numbers shown are from when it did complete but it would always lose out on correctness.

As shown from this table, AcmeSafe performs the best without sacrificing correctness. After increasing past 2 threads, AcmeSafe was performing significantly better than Synchronized and at times, even better than Null. This means that for high parallelizability, AcmeSafe is the best solution. The other metrics indicated similar conclusions, where changing swap transitions, number of elements, value of maxval, and so on all directly correlated with AcmeSafe performing better than the other states without sacrificing consistency.

5 Testing Platform Information

Java Version:
openjdk version "13.0.1" 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)

OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)

32 Processors of the Following Type:
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 62
model name : Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
stepping : 4
microcode : 0x42e
cpu MHz : 2300.537
cache size : 20480 KB
physical id : 0
siblings : 16
core id : 0
cpu cores : 8
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 ibrs ibpb stibp
bogomips : 4000.08
clflush size : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:

Memory Info:
MemTotal: 65755732 kB
MemFree: 34023376 kB
MemAvailable: 56550508 kB
Buffers: 432188 kB
Cached: 18988536 kB
SwapCached: 6096 kB
Active: 16050732 kB
Inactive: 10934184 kB
Active(anon): 7579872 kB
Inactive(anon): 227432 kB
Active(file): 8470860 kB
Inactive(file): 10706752 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 20479996 kB
SwapFree: 17700196 kB
Dirty: 140 kB
Writeback: 0 kB
AnonPages: 7561604 kB
Mapped: 114304 kB
Shmem: 243100 kB
Slab: 4122476 kB
SReclaimable: 3841420 kB
SUnreclaim: 281056 kB
KernelStack: 23712 kB
PageTables: 102664 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 53357860 kB
Committed_AS: 13674420 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 395540 kB
VmallocChunk: 34325397500 kB
HardwareCorrupted: 0 kB
AnonHugePages: 1552384 kB
CmaTotal: 0 kB
CmaFree: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
DirectMap4k: 318400 kB
DirectMap2M: 8024064 kB
DirectMap1G: 58720256 kB

6 Conclusion

As we can see from the measurements and logic provided for each of the classes/packages, the Reentrant Lock approach for AcmeSafe performs the best. It performs faster than the Synchronized State and at times, even the Null State. It also does not sacrifice consistency like the Unsynchronized State, which is so unreliable that it would often mismatch the sum or not even complete. Therefore, it is my recommendation to GDI to explore the concept of using Reentrant Locks in their code instead of the Synchronized keyword and if their performance improves significantly, I will be glad my recommendation was quite helpful. Note: monetary forms of thanking us will be kindly accepted by check to Professor Paul Eggert.