CS131 Homework 3 Report

Arnav Garg
304911796

1. Introduction

Like most programming languages, Java can run into race conditions when writing concurrent programs. To solve this issue, the developers of Java created the Java Memory Model (JMM) that is used internally in the Java Virtual Machine (JVM), which defines a set of rules and guidelines how a program can access shared memory while preventing the onset of race conditions. The JMM is responsible for determining how to divide memory for a process or program in terms of thread stacks and heaps.

One of the ways to make a program thread-safe is to use the "synchronized" keyword while defining a method within a class, indicating that only a thread can access that function or block of code at a particular instance of time. While this is a common way to get around the issue of races, it is often slow. Therefore, Java provides many other packages and classes to create synchronization mechanisms for concurrent programming, some of which do exhibit much better performance than synchronized functions, while others are not data-race free (DRF).

This report explores the pros and cons of the different synchronization mechanisms that Java provides in the form of packages and classes. In particular, we will analyze the performance of multithreaded programs without any synchronization, with the "synchronized" keyword, with volatile variables and lastly, with locking mechanisms.

2. Packages and Classes for Synchronization

Java provides a wide variety of packages and classes that enable to creating of thread-safe concurrent programs.

2.1 java.util.concurrent

This is a Java utility class that is commonly used in concurrent programming. It consists of a set of interfaces whose implementations execute tasks. The interfaces include Executor, ExecutorService, Cyclic Barriers, Semaphores and Count Down Latches amongst many others.

One of the main cons of the utility packages is that it is often difficult to implement or use, especially in the case of cyclic barriers and count down latches. In the case of semaphores, blocked threads may need to catch an *InterruptedException*, which adds another layer of complexity since the programmer has to catch this and decide how to handle it. This is also because it gives programmers control over certain situations and behaviors, which might be unnecessary for our task. However, the range of options this package provides in terms of classes is extensive and gives programmers who need to write concurrent program with tremendous flexibility for more complex concurrent programs.

2.2 java.util.concurrent.atomic

The set of classes defined in this package allow for thread-safe, lock-free, programming on single variables. It contains classes that allow for the creation of AtomicBooleans, Atomic Integers and Atomic Integer Arrays. Instances of atomic classes maintain values that are accessed and updated using methods from the seemingly synchronous GetNSet class.

However, in our case, this doesn't prevent races because it does allow us to lock up the code to ensure atomic read or write requests, but does not give us the ability to control what sections of code are really locked up, and therefore, threads can perform read and write operations during the time between another thread's read/write operations. Therefore, it is possible for the classes from this package to cause race conditions during the execution of multithreaded code.

2.3 java.util.concurrent.locks

This class provides locking mechanisms that enable the programmer to place locks at the beginning and end of critical sections that the programmer can decide on. Therefore, it has the potential to lock up code that can result in race conditions, making this a likely DRF candidate.

While this class has an extensive set of locking mechanisms, for our purpose, Reentrant locks are the simplest to use and also guarantee that our code will be DRF, which is why I used this in the BetterSafe class. All the other classes, which would also provide the same locking mechanism, appear to be a bit more convoluted and provide functionality and freedom far beyond what we need for this program. The Reentrant lock has its own variants that give separate locking mechanisms for read and write operations, but this was again far more than what we need for this program. Since the Reentrant lock is a mutex lock, it will also provide performance gains since the threads don't keep spinning/waiting for their time but instead sleep until

another thread is done executing the critical section.

2.4 java.lang.invoke.VarHandle

The class VarHandle within this package provides dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables that allow *access modes*, including plain read/write access, volatile read/write access, and compare-and-set.

Although not obvious at first, this package provides the user with classes that function in a very similar way to the class in util.concurrent.atomic, and therefore would not fit our use case since these would not only be slow, but also be subject to data races.

3. Testing

3.1 Testing Environment

All tests were conducted on UCLA's *lnxsrv10*. This server appears to have the following specifications:

Total Memory	65799788 kB
Free Memory	1070736 kB
Number of Processors	4
Processor Model	Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
Cache Size/Processor	16896 KB
Cores/Processor	4

3.2 Number of Threads vs Runtime

Sync. Method	**Time per transition (ns)**			**Data Race Free (DRF)**
	8 Threads	**16 Threads**	**32 Threads**	
Synchronized	2303.73	5103.69	15456.7	Yes
Unsynchronized	N/A	N/A	N/A	No
GetNSet	2626.79	5486.94	21294.5	No
BetterSafe	3895.43	5075.04	16296.2	Yes

Contrary to normal expectations for multithreaded thread-safe programs, the data indicates an increase in the average time per transition as the number of threads increase from 8 threads to 32 threads when the number of elements in the list is 100. The number of threads seem to have a poor correlation with the time per transition irrespective of the type of synchronization because there appears to be no real correlation since all of them have a similar time per transition for a large number of threads. This could be because a lot of time is spent creating threads, or could just be time spent in context switches between threads which seems to be fairly consistent. Another reason for the lack of an obvious trend can be that many users were running their code simultaneously on *lnxsrv10*, causing the speeds to change depending on the number of programs running on the server at the time these results were tested.

3.3 Number of Swaps vs Runtime

Sync. Method	**Time per transition (ns)**		
	100K Swaps	**500K Swaps**	**1000K Swaps**
Synchronized	2303.73	1089.52	724.972
Unsynchronized	N/A	N/A	N/A
GetNSet	2626.79	1000.79	686.137
BetterSafe	3895.43	930.538	642.030

The data suggests that irrespective of the synchronization method (barring the case where there is no synchronization), as the number of swap operations increase, the average time per transition decreases. BetterSafe appears to have the best time per transition as the number of swap operations increase, but also appears the have the highest time per transition for a smaller number of swap operations. This could be because the time to initialize a Reentrant Lock could be expensive, or because the time taken to lock and unlock a Reentrant lock is expensive. However, once this initial cost occurs, it appears to perform the best suggesting a faster method of dealing with thread contentions.

3.4 Runtime vs Number of Elements in the List

Sync. Method	**Time per transition (ns)**		
	100 Elements	**500 Elements**	**1000 Elements**
Synchronized	3846.73	3109.51	2911.12

Unsynchronized	N/A	N/A	N/A
GetNSet	2804.76	3466.74	3820.61
BetterSafe	2797.23	4911.65	2833.61

The data suggests a multitude of different things. Firstly, for synchronized functions, as the number of elements increase, the total time per transition decreases. This suggests that synchronized methods do scale well with the number of elements/operations being performed. GetNSet appears to take longer with an increase in the number of elements, probably due to the need to slow down operations to ensure atomicity with respect to read and write requests. However, the most interesting trend is observed with BetterSafe, which has roughly the same average time per transition for 100 and 1000 elements. This suggests that locks scale much better than synchronized operations with respect to the number of operations. However, there seems to be an anomaly for the time taken per transition in the case where better BetterSafe ran on 500 elements in the list. This could be because the hardware was unable to exploit spatial locality by loading in elements before and after into the processor cache.

4. Data Race Freeness of our Classes

The class that employs the "synchronized" keyword is data-race free (DRF) because it locks up the function while one thread is executing the code (the method swap) within that function so that no other thread can access/use/run this function at the same time.

The Unsynchronized class is not DRF because it doesn't time thread read and writes, and therefore is subject to potential race conditions as soon as the number of threads is greater than 1, since it executes code sequential and two threads can write to the same memory location concurrently. This also lacks atomicity, so it has the highest chance of running into race conditions.

The GetNSet class is not DRF because it only ensures read and writes are atomic, not that they happen in a certain sequence, or rather, a thread at a time. Therefore, it is still possible for threads to interleave their write operations in between another thread's read and write operations, causing race conditions to still occur. This still maintains the promise of atomicity, but does not ensure memory access is done in a particular, race-free manner.

Lastly, BetterSafe is DRF because it uses the idea of mutex locks, short for mutually exclusive locks, where it locks up sections of code where race conditions could arise, only allowing one thread to execute that section of code at a time. This ensures there are no race conditions.

5. Conclusion

Based on the data and the analysis of each synchronization mechanism, it appears that Reentrant Locks, implemented in the BetterSafe class, would fit GDI's use-case the best. This is because it displays superior performance in terms of average time per transition in terms of number of operations, and well as number of elements in the list. This works well because it will scale for big data applications while having the lowest time-per-transition. Additionally, BetterSafe is data race free, promising that race conditions will not exist, and improving the chances that the results produced during big data analysis will be fairly accurate while not compromising on performance.

However, it is interesting to note Synchronized methods, although slightly slower than Reentrant Locks, do provide reliable performance and scale better than most other classes. One possible reason for this is in the way both of these mechanisms work. Reentrant Locks work like mutex locks, and have only been used to lock up the sections of code that are subject to race conditions. On the other hand, the "synchronized" keyword is normally used with functions and locks up the entire function, preventing any other thread from entering the function at the same time. Therefore, each thread has to wait longer, on average, to execute the swap method within the class.

6. Citations

1. Java API Reference. (2018, September 07). Retrieved from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html
2. Java API Reference. (2018, September 07). Retrieved from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/VarHandle.html
3. Java API Reference. (2018, September 07). Retrieved from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/package-summary.html
4. Java API Reference. (2018, September 07). Retrieved from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html