Homework 3 Report: Reliability vs Speed

Andrew Yong
604905807

Introduction:

Computer science is all about tradeoff. For an example, an increase in memory might require that there is slower access and I/O. On the other hand, having a faster CPU is more expensive and power consuming. In this report, we are taking a look at the tradeoff between reliability and speed.
Multithreading is a programming technique that can make programs run significantly faster because they take advantage of multiple cores that are in many computers, resulting in parallel computing. However, because different threads all share the same memory space, they can inadvertently cause errors in computations by getting in the way of one another (called race conditions). An example of this is when there is a shared variable. One thread can be in the middle of updating a value while another thread is viewing it, leading to the second thread to read a outdated value. This leads to errors in computation, thus making a program unreliable. This report will explore the ways a Java program can increase reliability of the program, but will also consider the tradeoffs of implementing different types of reliability.

Shell Script:

In order to test the program, I created a Bash script to test the different variations in iterations and implementations:

```
#!/bin/bash

tests=(Synchronized Unsynchronized AcmeSafe)
iter=(1000 10000 100000 1000000)
for i in "${tests[@]}"
do
      echo "testing $i"
      for j in "${iter[@]}"
      do
            echo -ne "$j iterations: \t"
            java UnsafeMemory $i 8 $j 127 1 2
3 4 5 6 7 8 9 10 2> /dev/null
      done
done
```

As you can see, I only test the number of iterations. I kept the number of threads constant at 8 in order simply to test speed on reliability. Additionally, I ran this script 10 times to observe the reliability of each version.

Implementation 1: Unsynchronized State

Starting out with the most unreliable implementation, we also understandably have the fastest program. Unsynchronized State does calculations but doesn't make any attempt to keep the threads in sync with each other. As expected, each time I ran Unsynchronized State, it threw a pointer error, and the sums never matched up. But note the data as we look at the other implementations.

Number of Transitions	Average Time for (ns/transition) threads
1000	54801.8
10000	4782.94
100000	499.138
1000000	45.2803

Implementation 2: Synchronized States

Next we take a look at Synchronized State, which uses the *synchronized* keyword in Java on the calculation function. This forces the implementation to only allow one thread to use the calculation function at a time. As shown in the data, this significantly slows down the program at greater Transactions because the threads have to now wait for the other threads to finish first. But now the program is extremely reliable.

Number of Transitions	Average Time for (ns/transition) threads
1000	31082.5
10000	10153.3
100000	5674.07
1000000	2546.59

Note that at lower iterations, this program is actually faster than the unsynchronized program, but with higher and higher iterations, it becomes slower and slower.

Java Libraries Discussion

After looking at these implementations, they seem to be two extremes of what we are looking for. We want both reliability and speed. Thankfully, others have already foreseen this problem, and the Java programming language already has libraries to deal with this problem. Let's consider some of these libraries designed to deal with this problems

java.util.concurrent.atomic
This library uses atomic operations, which means that the computer does a read and a wite operation in the same command. Although it is true that this would eliminate the need for race conditions, this is not the case for this program because it does multiple different operations. Using this library would still allow race conditions to occur. Thus this option would not increase reliability and would only increase performance slightly.

java.util.concurrent.VarHandle
This library allows control over the different access modes of a variable, offering both reliability and speed. This is more along the lines of what we're looking for but is still a bit general. We are looking for fine tune control over which sections to look over, and although this library is reliable, we can do better with performance.

java.util.concurrent.locks
This library has locks, which allows us to force the code to only allow a single thread at a time to access a region at a time. Notice that this is the same idea as *synchronized*, except that it doesn't lock the entire method. By using this library, we can pinpoint the part of the code that is the critical section, and lock that section. Using this in our implementation gives us the maximum possible speed to ensure reliability. This way we can get the best of both worlds:

Implementation 3: AcmeSafe

Unsurprisingly, I decided to use the java.util.concurrent.locks library to get the most performance while ensuring reliability. Specifically, this implementation uses *ReentrantLock*, which is a part of this library, to envelope the critical sections of the code as needed. This allows for the code to be fully optimized, while still having proper synchronization. The data for this implementation is listed as follows:

Number of Transitions	Average Time for (ns/transition) threads
1000	75625.8
10000	15085.2
100000	5387.21
1000000	1200.82

Notice that at lower iterations, this code is actually much slower than all of the implementations. This could be because of the time it takes to initialize the lock. But as the iterations increases, the average time decreases, and pretty soon is more than 2 times faster than Synchronized States, while still maintaining 100% accuracy.

Conclusion and Discussion:

As we looked at the different options to implement this code, we quickly realized the tradeoff between reliability and speed. We wanted to maintain the maximum reliability that Synchronized State had while optimizing the performance that Unsynchronized State boasted.

We then looked at the different Java libraries that were designed to tackle this problem, but we ruled out the one that still maintains race conditions. This is because there is no point in having a fast program if everything it does is wrong. We then considered another library that had better reliability, but was still a bit too unoptimized. Lastly, we considered and settled on the locks library, which allowed us to implement synchronization in exactly the places we want. Thus we can pinpoint the exact locations we needed to regulate to ensure that this program wouldn't have race conditions. This way, AcmeFree can still be data-race free (DRF) while achieving faster speeds at large iterations.

Looking at the data, we can conclude that using Synchronized State is better for smaller operations while AcmeFree is better for programs with large operations. Thus, in this scenario, GDI should use AcmeFree because they use large amounts of data.