Java Shared Memory Performance Races

Nathan Tjoar

University of California - Los Angeles

Abstract

In any multithreaded programming, it is often customary to avoid race conditions, as they can lead to unpredictable behavior. Java is a language which has the *synchronized* key word, in which we can specify such a component of the program to be a critical section. However, this step can bring us to have delays in the code, which is where data race-free (DRF) programs come in. By taking advantage of Java's libraries to create mutual exclusion, or mutex, locks, we can optimize the program to run faster than the original synchronized program. For such a program to be successful, it is necessary that any and all races be avoided completely by having threads directly lock up and take a resource for itself.

1 Introduction

To start this project, a test Java code was given. The cases contained many Java files that could be run by invoking different key words. All the programs were set so that they could be multithreaded in the UnsafeMemory.java file. To run and test each memory and each point, I started by developing a shell file that would run a variety of test cases and sum up the time it took to complete all said test cases. In addition to this, we were to implement the same class but make it unsynchronized. Now, this would definitely break the program and increase likelihood of race conditions. As a fix, we were asked to also implement an additional code that would run with the race conditions and continue to keep the race conditions under check in however way possible. As a test, I used the java.util.concurrent.locks library. With the program, I simply mutex locked our critical section and ran the program. The goal in this is first and foremost, safety of the program, so for my analysis and decision, I will be prioritizing safety above speed. Speed is another factor that we wish to see.

2 Findings

The tests were run on both Linux servers lnxsrv0[679] and lnxsrv10. Every test was run on both, summed up and averages taken across a few times to determine average time taken per data process. Below is an estimated value of all the data points taken between 1, 8, 16, and 32 threads. Each thread count was tested across 32, 64, and 128 array sizes. Additionally, each array size was tested with 100 to 105 transition counts. From here, the average time was taken for each operation and averaged with respect to the transition counts and array sizes. Below are the data values found.

Synchronized

Real time	User time	Sys time	Error Observed
0.152s	0.110s	0.042s	0%

Table 1: Average data from lnxsrv0[679]

Real time	User time	Sys time	Error Observed
0.152s	0.112s	0.040s	0%

Table 2: Average data from lnxsrv10

Average swapping time (lnxsrv0[679]): 4.58201e +06 ns

Average swapping time (lnxsrv10):

4.85911e+06 ns

This program was a DRF model. This thread approach allowed one swap at a time, which in turn gave a high security performance. However, this was running very slow as threads would need to wait.

This program seemed to do particularly well with small scale programs in which threads would have to wait regardless. However, in any other case, this program fails to speed up.

Unsynchronized

Real time	User time	Sys time	Error Observed
0.130s	0.100s	0.029s	1%

Table 1: Average data from lnxsrv0[679]

Real time	User time	Sys time	Error Observed
0.133s	0.102s	0.048s	1%

Table 2: Average data from lnxsrv10

Average swapping time (lnxsrv0[679]): 3.55657e+06 ns

Average swapping time (lnxsrv10):

3.42347e+06 ns

This was the same class as the synchronized class, but with the synchronized key words taken out. By doing so, we allow races to happen, which will cause unpredictable behavior and very often cause the program to spit out values which should not make sense in the first place.

In general, this approach seems to take a fast, but unreliable approach. In many tests, this gave different errors in different commands.

AcmeSafeState

Real time	User time	Sys time	Error Observed
0.148s	0.115s	0.040s	0%

Table 1: Average data from lnxsrv0[679]

Real time	User time	Sys time	Error Observed
0.143s	0.110s	0.031s	0%

Table 2: Average data from lnxsrv10

Average swapping time (lnxsrv0[679]): 3.99168e+06 ns

Average swapping time (lnxsrv10):

3.89166e+06 ns

This program is a DRF model as it works pretty similarly to the synchronized class. It essentially locks up the array and forces for threads to wait for one another in order to access the same part of an array. By doing so, it allows for more access into the memory.

In comparison to the other two, this program worked quite slowly for the lower end in which

multiple threads could not access concurrently. However, in larger programs, this approach seemed to outperform the Synchronized program in speed and the Unsynchronized program in reliability.

3 Problems

Overall the homework was relatively straight forward. I was able to get through the code writing portion fairly quickly. On the AcmeSafeState, however, I had issues deciding which way to approach the issue from. Ultimately, I was able to come to a decision, as I used some past resources regarding different locks and their benefits. The data gathering section was pretty much the only difficult portion, as it was mainly about consistency and fairness for me to get the correct data. I was able to bypass this by writing a simple shell script to do the calculations for me.

4 Conclusions

Overall, I would default to the Acme Program, as it spends less time overall. It has more time in preprocessing than it does in processing as compared to synchronized and as such gives us much better processing times for even bigger programs. Additionally, as compared to unsynchronized, we have a class that is much better for security. As we previously justified, we want to have the program run as fast as it can in a safe state. With this, although few errors were found in unsynchronized, such errors will propagate and cause more issues for us in the future. With synchronized, the issue was in the focus that it only wants to allow one thread to access an array at a time, forcing times to go up.