Breaking and fixing the Java Memory Model for profit

Ryan Sharif

February 7, 2016

Contents

1 Answers and Summary

For the benefit of the T.A.s, I summarize my results and findings for this homework before any of the material is presented. This is because I have included an extensive write up of the homework. Thus, this summary will attempt to meet the requirements of the homework specification, while the details for these findings are found in the pages that follow.

The `Synchronized` and `Null` models provide a baseline to begin our analysis. Looking at the results in the following pages, we can see that the `Null` model runs around two to three times faster than `Synchronized`. Thus, if our class implementations are done right, we want to find ourselves somewhere in between these two models. The Unsynchronized model, when it does not deadlock should run at about the same speed as `Null`; however, I was never able to get it to complete the entire test suite because of lockups. Hence, it results appear to be significantly slower. But this is only because the number of swaps is significantly smaller than the ones we've done on the previous two models.

We avoid deadlocks and erronous ouput using the `AtomicIntegerArray` API that Java provides for us for our `GetNSet` model. As the specification indicates, it runs at a speed that is in between `Null` and `Synchronized`. This is probably due to overhead that `Synchronized` imposes on the execution of a method. Thus, this is a desirable outcome for our company GDI. The implementation and execution of `BetterSafe` is similarly faster than `Synchronized`. In fact, `BetterSafe` runs almost 2.5x faster than

`Synchronized`. This model uses Java's Reentrant lock API. Since its overhead appears to be much smaller than `Synchronized`, it achieves faster over run time. Still, with this speed up, it is able to maintain error-free results.

Finally, if our company GDI wants speed at the expense of a little accuracy, the best model is `BetterSorry` as it achieves speeds as fast as `Null`, while sacrificing a little bit of accuracy. I achieved this by creating a `static volatile` boolean that gets flipped on and off when the program is executing critical portions of code, i.e., writing to memory. It still contains race conditions during reads, though these have been minimized to a maximum of two instructions. A program that increases the liklihood of race conditions is one that invokes a higher and higher number of threads.

2 Introduction

2.1 Checking environment variables and Java version

2.1.1 Java version

Before we get started doing anything else in the assignment, let's make sure that we have our environment variables and an appropriate version of Java. I'll be using a local copy of JDK version 1.8.071:

```
java -version
```

Java(TM) SE Runtime Environment (build $1.8.0_{71}$-b15) Java HotSpot(TM) 64-Bit Server VM (build 25.71-b15, mixed mode)

2.1.2 CPU information

We can look at our CPU information using `less` to see that we will be using an Inter Core i3 Processor running at 3.3GHz. My particular machine reports a single core. This may or may not affect the way interleaved CPU instructions are executed on several cores. Thus, I will run my code locally and on UCLA's SEASNET servers.

```
less /proc/cpuinfo
```

2.1.3 Memory Information

We can similarly inspect our machine's memory information by look at the '/proc/meminfo' file, which tells me that I have 764300 kB of memory total.

3 Running the tests

3.1 Extracting jmm.jar

The files that we'll need for this assignment are compressed in the 'jmm.jar' file. Thus, we'll need to decompress the jar file before we can do anything else:

```
jar -xvf executables/jmm.jar
```

created: META-INF/ inflated: META-INF/MANIFEST.MF inflated: NullState.java inflated: State.java inflated: SwapTest.java inflated: SynchronizedState.java inflated: UnsafeMemory.java

3.2 Makefile

Compiling the source files into executable class files will become tedious. We can automate this process using a Makefile:

```
JC = javac
OUTPUT = executables/

all: nullstate swaptest synchronized_state state unsafe_memory

# All classes for this assingnment
nullstate : NullState.java
        $(JC) -d $(OUTPUT) NullState.java
swaptest : SwapTest.java
        $(JC) -d $(OUTPUT) SwapTest.java
synchronized_state : SynchronizedState.java
        $(JC) -d $(OUTPUT) SynchronizedState.java
state: State.java
        $(JC) -d $(OUTPUT) State.java
unsafe_memory: UnsafeMemory.java
        $(JC) -d $(OUTPUT) UnsafeMemory.java

clean:
        rm $(OUTPUT)*
```

3.3 Testing

With the prerequisite files and environment variables all in order, we can begin testing and working on the assignment. We'll begin our tests on the Synchronized implementation, moving over to the Null model afterwards.

3.3.1 Synchronized model

Here we test the synchronized model first. The initial results on my local machine indicate that this program is not benefiting from more threads. In fact, the more cores we add, the worse our program performs. Given, these results, I will move over to the SEASNET servers to test application peformance on a multicore machine.

```
Threads average 69.9444 ns/transition
Threads average 513.126 ns/transition
Threads average 1511.14 ns/transition
Threads average 3254.82 ns/transition
Threads average 6098.57 ns/transition
Threads average 12087.3 ns/transition
```

Below are the tests and results from running the same application on the SEASNET servers. The results from the SEASNET servers run approximately twice as fast as the results my local machine produced.

```
Synchronized tests
first test set
01: Threads average 70.2679 ns/transition
02: Threads average 425.178 ns/transition
04: Threads average 1637.13 ns/transition
08: Threads average 2989.45 ns/transition
16: Threads average 5828.51 ns/transition
32: Threads average 14677.7 ns/transition

second test set
01: Threads average 108.989 ns/transition
02: Threads average 740.129 ns/transition
04: Threads average 2763.78 ns/transition
08: Threads average 4828.20 ns/transition
16: Threads average 7848.66 ns/transition
32: Threads average 21089.5 ns/transition
```

```
thirds test set
01: Threads average 90.7319 ns/transition
02: Threads average 530.770 ns/transition
04: Threads average 2188.73 ns/transition
08: Threads average 4002.67 ns/transition
16: Threads average 8794.03 ns/transition
32: Threads average 18009.1 ns/transition
```

3.3.2 Null model

As indicated by the specification for this assignment, the Null model does not yet work but still passes the test, thus it runs to completion much faster than the synchronized model. We should note the overhead of creating threads at least on this local machine adds considerable running time to our program despite the fact that no actual work is being done.

```
01 Threads average 36.7658 ns/transition
02 Threads average 132.399 ns/transition
04 Threads average 455.818 ns/transition
08 Threads average 2336.21 ns/transition
16 Threads average 4654.78 ns/transition
32 Threads average 8696.07 ns/transition
```

```
cd files/executables;
echo "second test set";
echo -n "01: "; java UnsafeMemory Null 1 1000000 2 1 1 0 0 1
echo -n "02: "; java UnsafeMemory Null 2 1000000 2 1 1 0 0 1
echo -n "04: "; java UnsafeMemory Null 4 1000000 2 1 1 0 0 1
echo -n "08: "; java UnsafeMemory Null 8 1000000 2 1 1 0 0 1
echo -n "16: "; java UnsafeMemory Null 16 1000000 2 1 1 0 0 1
echo -n "32: "; java UnsafeMemory Null 32 1000000 2 1 1 0 0 1
```

```
thirds test set
01: Threads average 36.5781 ns/transition
02: Threads average 138.304 ns/transition
04: Threads average 439.491 ns/transition
08: Threads average 2372.14 ns/transition
16: Threads average 4792.37 ns/transition
32: Threads average 8229.13 ns/transition
```

4 Unsynchronized implementation

We can begin implementing the unsynchronized model by bringing over the code from the synchronized model and tinkering with it. We will start with a basic class definition, naming the class UnsynchronizedState and letting the Java compiler know that we'll be implementing the class State. This means we'll have to take all the method signatures from State and actually implement them here:

```
class UnsynchronizedState implements State {
    private byte[] value;
    private byte maxval;
```

Similar to the synchronized version, we'll have two constructors: a constructor that receives an array to initialize to some value, and sets the maximum value for the object to 127. We also have a second constructor that similarly takes in an array but also takes in a byte, setting the maximum value for this object to m.

```
UnsynchronizedState(byte[] v) { value = v; maxval = 127; }

UnsynchronizedState(byte[] v, byte m) { value = v; maxval = m; }
```

The key change to the class is simply a removal of the keyword `synchronized` from the definition of the swap method:

```
    public int size() { return value.length; }

    public byte[] current() { return value; }

    public boolean swap(int i, int j) {
        if (value[i] <= 0 || value[j] >= maxval) {
            return false;
        }
        value[i]--;
        value[j]++;
        return true;
    }
}
```

We can compile our class and test it like the other two we've tested before:

```
cd files;
make unsynchronized_state
```

Finally, before we can run our program again, we need to ensure that our program knows how to use the new class by adding two lines of code:

```
else if (args[0].equals("Unsynchronized"))
    s = new UnsynchronizedState(stateArg, maxval);
```

4.1 Running Unsynchronized

There is a problem with the way that unsynchronized works. When we increase the number of threads or swaps beyond an arbitrary value the likelihood that the program will become deadlocked increases. Thus, for these tests we used orders of magnitude smaller swaps than previous tests:

01 Threads average 3412.14 ns/transition 02 Threads average 6278.02 ns/transition sum mismatch (17 != 12) 04 Threads average 13747.9 ns/transition sum mismatch (17 != 18) 08 Threads average 26983.3 ns/transition sum mismatch (17 != 19) 16 Threads average 56221.0 ns/transition sum mismatch (17 != 16) 32 Threads average 152139 ns/transition sum mismatch (17 != 16)

As expected, our unsynchronized class runs into race conditions, where we get unexpected unreliable values.

5 GetNSet

5.1 Writing the Class

With the problematic *unsynchronized* class implemented, we want to achieve similar speed but without the race conditions. Is that possible? Lets implement Java's atomic integer array and see if we can do any better. A definition provided on Wikipedia states that an atomic operation is one that is a guarantee of isolation from concurrent processes. Since we'll be using the AtomicIntegerArray class, lets include it in our file and declare a variable `valueIntegerArray` that we'll instantiate in our constructor:

```
import java.util.concurrent.atomic.AtomicIntegerArray;

class GetNSet implements State {
    private int[] value;
    private byte maxval;
    private AtomicIntegerArray valueIntegerArray;
```

With the variable declared above, we'd like to instantiate an instance of the class; however, looking at the documentation for AtomicIntegerArray shows us that we need to pass in an integer array, not a byte array. Thus, we'll want to repurpose `value` as an `int` array and run a loop that will set each element its equivalent in the byte array:

```
GetNSet(byte[] v) {
    value = new int[v.length];

    for(int i = 0; i < value.length; i++){
        value[i] = v[i];
    }

    maxval = 127;
    valueIntegerArray = new AtomicIntegerArray(value);
}

GetNSet(byte[] v, byte m) {
    value = new int[v.length];

    for(int i = 0; i < value.length; i++){
        value[i] = v[i];
    }

    maxval = m;
    valueIntegerArray = new AtomicIntegerArray(value);
}
```

With the constructors that correctly instantiate our AtomIntegerArray we can change the size method so that it gets the AtomicIntegerArray length. We just call its `length` method. The `current` method requires us to return a `byte` array, so we'll need to create a temporary byte array and return it:

```
public int size() { return valueIntegerArray.length(); }

public byte[] current() {
    byte[] tmp = new byte[value.length];

    for(int i = 0; i < tmp.length; i++){
        tmp[i] = (byte) value[i];
```

```
    }

    return tmp;
}
```

Finally, the `swap` function needs to use the `get` and `set` methods provided by the AtomicIntegerArray class:

```
    public boolean swap(int i, int j) {
        if (valueIntegerArray.get(i) <= 0 || valueIntegerArray.get(j) >= maxval) {
            return false;
        }
        valueIntegerArray.getAndDecrement(i);
        valueIntegerArray.getAndIncrement(j);
        return true;
    }
}
```

5.2 Results

Let's run this class, the same way we've done before:

```
cd files/executables;
echo -n "01 "; java UnsafeMemory GetNSet 1 1000000 6 5 6 3 0 3
echo -n "02 "; java UnsafeMemory GetNSet 2 1000000 6 5 6 3 0 3
echo -n "04 "; java UnsafeMemory GetNSet 4 1000000 6 5 6 3 0 3
echo -n "08 "; java UnsafeMemory GetNSet 8 1000000 6 5 6 3 0 3
echo -n "16 "; java UnsafeMemory GetNSet 16 1000000 6 5 6 3 0 3
echo -n "32 "; java UnsafeMemory GetNSet 32 1000000 6 5 6 3 0 3
```

Like our previous results, we'd expect that the more threads we add the faster our program should run; however, it looks like the overhead of creating the threads is too costly for this simple swap function. On a positive note, we are no longer getting bad results, even testing on an array two and three orders of magnitude larger produces no bad results:

```
32 Threads average 3621.75 ns/transition
32 Threads average 3904.25 ns/transition
```

6 BetterSafe

6.1 Writing the class

We can now move to the BetterSafe model, which will achieve better performance than *Synchronized* but still maintain 100% reliability. We will be able to do this by implementing a system of locks and unlocks.

We begin with our familiar code from *Synchronized*, maintaining a majority of the code. Thus, we only change the name of the class along with the constructor names to reflect this change. Finally, we'll add a lock to use when we are performing a swap:

```
import java.util.concurrent.locks.ReentrantLock;

class BetterSafe implements State {
    private byte[] value;
    private byte maxval;
    private final ReentrantLock swapLock;

    BetterSafe(byte[] v) {
        value = v; maxval = 127;
        swapLock = new ReentrantLock();
    }

    BetterSafe(byte[] v, byte m) {
        value = v; maxval = m;
        swapLock = new ReentrantLock();
    }
```

We'll remove the `synchronized` keyword from the swap function and implement a use of locks to make sure that no thread steps on anyone else's toes:

```
public int size() { return value.length; }

public byte[] current() { return value; }

public boolean swap(int i, int j) {
    swapLock.lock();

    if (value[i] <= 0 || value[j] >= maxval) {
```

```
            swapLock.unlock();

            return false;
        }
        value[i]--;
        value[j]++;

        swapLock.unlock();

        return true;
                                }
}
```

6.2 Testing BetterSafe

Let's test our BetterSafe class by performing the same tests that we've done in the past:

orginal	test:			
1	Threads	average	78.0822	ns/transition
2	Threads	average	549.396	ns/transition
4	Threads	average	624.036	ns/transition
8	Threads	average	1160.22	ns/transition
16	Threads	average	2405.32	ns/transition
32	Threads	average	5874.64	ns/transition
larger	test:			
1	Threads	average	78.6375	ns/transition
2	Threads	average	582.671	ns/transition
4	Threads	average	562.409	ns/transition
8	Threads	average	1207.74	ns/transition
16	Threads	average	2455.95	ns/transition
32	Threads	average	5522.98	ns/transition

7 BetterSorry

7.1 Writing BetterSorry

```
import java.util.concurrent.TimeUnit;

class BetterSorry implements State {
    private volatile byte[] value;
```

```
    private byte maxval;
    private static volatile boolean inCritical = false;
```

Similar to the synchronized version, we'll have two constructors: a constructor that receives an array to initialize to some value, and sets the maximum value for the object to 127. We also have a second constructor that similarly takes in an array but also takes in a byte, setting the maximum value for this object to m. We'll use a psuedo-lock by creating a boolean that lets us know when we're in a critical part of the execution, i.e., when we're writing to our array.

```
BetterSorry(byte[] v) { value = v; maxval = 127; }

BetterSorry(byte[] v, byte m) { value = v; maxval = m; }
```

To make sure we don't have any deadlocks, we'll check to make sure we are not in a critical section, i.e., writing to our array. If we are, we'll wait our turn. If not, then the thread will write what it needs to the array.

```
    public int size() { return value.length; }

    public byte[] current() { return value; }

    public boolean swap(int i, int j) {
        int v_i = value[i], v_j = value[j];

        if (v_i <= 0 || v_j >= maxval) {
            return false;
        }
        while(inCritical) {
            try {
                TimeUnit.NANOSECONDS.sleep(1);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        inCritical = true;

        value[i]--;
```

```
        value[j]++;

        inCritical = false;


        return true;
    }
}
```

7.2 Testing BetterSorry

```
                    01  Threads  average  69.8024  ns/transition
                    02  Threads  average  141.296  ns/transition
                    04  Threads  average  295.705  ns/transition
                    08  Threads  average  644.861  ns/transition
  sum mismatch (17 != 18)
                    16  Threads  average  1722.14  ns/transition
  sum mismatch (17 != 23)
                    32  Threads  average  4318.05  ns/transition
  sum mismatch (17 != 24)
```