Breaking and fixing the Java Memory Model for profit

Ryan Sharif

February 7, 2016

Contents

1 Introduction

1.1 Checking environment variables and Java version

1.1.1 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)

1.1.2 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
```

1.1.3 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.

2 Running the tests

2.1 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

2.2 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)*
```

2.3 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.

2.3.1 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	70.7651 ns/transition
Threads average	183.874 ns/transition
Threads average	428.767 ns/transition
Threads average	843.679 ns/transition
Threads average	1842.09 ns/transition
Threads average	3631.31 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	85.2718	ns/transition
02:	Threads	average	192.319	ns/transition
04:	Threads	average	479.501	ns/transition
08:	Threads	average	791.218	ns/transition
16:	Threads	average	1634.2	ns/transition
32:	Threads	average	4202.26	ns/transition

second	test	set		
01:	Threads	average	102.025	ns/transition
02:	Threads	average	229.165	ns/transition
04:	Threads	average	543.071	ns/transition
08:	Threads	average	1278.21	ns/transition
16:	Threads	average	2768.9	ns/transition
32:	Threads	average	4956.99	ns/transition

thirds	test	set		
01:	Threads	average	86.214	ns/transition
02:	Threads	average	205.809	ns/transition
04:	Threads	average	491.793	ns/transition
08:	Threads	average	1030.86	ns/transition
16:	Threads	average	2238.65	ns/transition
32:	Threads	average	5230.2	ns/transition

2.3.2 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.

1 Threads	average 42.6443 ns/transition
2 Threads	average 97.9121 ns/transition
4 Threads	average 256.354 ns/transition
8 Threads	average 481.088 ns/transition
16 Threads	average 1400.48 ns/transition
32 Threads	average 1701.61 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	41.2914	ns/transition
02:	Threads	average	118.593	ns/transition
04:	Threads	average	290.362	ns/transition
08:	Threads	average	520.991	ns/transition
16:	Threads	average	1633.16	ns/transition
32:	Threads	average	2000.38	ns/transition

3 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);
```

3.1 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:

sum mismatch (17 != 21) sum mismatch (17 != 18) sum mismatch (17 != 21) sum mismatch (17 != 19) sum mismatch (17 != 11)

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

4 GetNSet

4.1 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;
        }
```

```java
        valueIntegerArray.getAndDecrement(i);
        valueIntegerArray.getAndIncrement(j);
        return true;
    }
}
```

4.2 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	6963.54	ns/transition
32	Threads	average	3769.83	ns/transition

5 BetterSafe

5.1 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:

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

```java
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:

```java
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;
                            }
}
```

5.2 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	79.9405	ns/transition
2	Threads	average	1003.66	ns/transition
4	Threads	average	572.069	ns/transition
8	Threads	average	1194.61	ns/transition
16	Threads	average	2515.44	ns/transition
32	Threads	average	5913.67	ns/transition
larger	test:			
1	Threads	average	80.0281	ns/transition
2	Threads	average	1067.41	ns/transition
4	Threads	average	601.73	ns/transition
8	Threads	average	1198.49	ns/transition
16	Threads	average	2542.19	ns/transition
32	Threads	average	5915.93	ns/transition

6 BetterSorry

6.1 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;
    }
}
```

6.2 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)				