HW #3: Java Shared Memory Performance Races

Computer Science 131, Spring 2019
Professor Paul Eggert

Taylor Lee, *604790788*

Abstract

This report details the implementation and testing of four classes that extend the *State* class in an attempt to use multi-threading to speed up performance and limit any reliability mistakes. The four classes used are *Synchronized, UnsynchronizedState, GetNSetState, BetterSafeState*.

1. The Four Classes

1.1. *SynchronizedState*

The Synchronized state is safe but slow. This state is consistently reliable and is a DRF, simply because the *synchronized* keyword does not allow for multi-threading to happen within the entire class. Multi-threading does not increase the performance or speed of this state.

1.2. *UnsynchronizedState*

Looking at the test results found in Appendix A, it is clear that this state increases performance due to its removal of the *synchronized* keyword, which allows the class to allow multithreading. However, this increase in speed performance comes at a huge disadvantage in reliability, as this class throws several *sum mismatch* errors in multiple test cases. This is due to the unprotected multithreading. Hence this class is not a DRF.

The following shell command will result in a sum mismatch bug or an infinite loop:

```
java UnsafeMemory UnsynchronizedState 8 100000 20 19 19 19 19
```

1.3. *GetNSetState*

Similar to *UnsynchronizdState,* this class is not a DRF, as many of the tests returned a *sum mismatch* error or an infinite loop on cases with high transition values. This infinite loop occurs when there are so many bad swaps in the array that all the elements in the array eventually are below 0 or at maxVal. While this scenario is unlikely given a small sample of swaps, given an input of 8 threads and 50,000 swaps, this state entered an infinite loop. Such unreliability proves this state insufficient for our purposes. The same shell command as *unsynchronized* will also result in a *sum mismatch* or an infinite loop.

1.4. *BetterSafeState*

Performance

Overall, BetterSafeState was the most efficient and reliable out of all the states. When looking at the sample test cases *BetterSafeState* excelled the most on test cases involving high amounts of swaps. Given a high volume of swaps (e.g. 50,000), and 8 threads to work with, this state even outperformed the *Null* state, which did no computation. This optimization is due to the multi-threading that is supported in this state. Additionally, this state is DNF due to the locks that are implemented in its swap function. This provides 100% reliability with even greater performance than the clunky *SynchronizedState class*.

Implementation:

When reading the four different given modules to use to implement this class, I decided on using the *ReadWriteReentrantLock* feature in the *java.util.concurrent.locks*. I decided to use this implementation because using the *ReadLock().Lock* function, this lock allowed for multiple threads to read from a locked data structure—so long as the data structure is not currently being written on by another thread. This optimization allows multiple threads to handle the first *if statement* that checks whether a swap is allowed. Additionally, the *writeLock().lock()* function locks the *value* byte array such that no two threads can write to the *value* data structure at the same time. This prevent race conditions and ensures that the class is a DRF.

2. Data Analysis

I ran the same test cases on two different version of Java: Java 9 and Java 11.0.2. Based on the data collected from the two different versions, it is clear that the latest version optimizes multi-threading to a higher degree than Java 9. Additionally, Java 9 resulted in more *sum mismatch* errors or infinite loops.

Additionally, it should be noted that *BetterSafeState* optimization reaches greater performance at 8 threads compared to 16 threads. This is most likely due to the fact that for small sample cases, a high level of threads will not increase performance to the degree it would with a large volume of sample cases. Generally speaking, the higher the amount of swap cases, the greater the margin of performance optimization *BetterSafeState* had compared to the other classes.

Appendix A

Using Java Version 11.2.08 (All values ns/transition)

16 Threads | MaxVal = 50 | Input = 22 4 15 23 19 11				
	50 Swaps	*500 Swaps*	*50000*	*500000*
Null	6.66527e+07	7.75568e+06	76527.7	6234.74
Synchronized	7.13093e+07	8.20249e+06	63470.6	8339.06
Unsynchro-nized	6.93669e+07	7.29086e+06	62504.5	7306.70
GetNSet	6.77907e+07	7.59423e+06	71307.2	7786.81
BetterSafe	7.11840e+07	7.72362e+06	65910.8	9551.97

8 Threads | MaxVal = 20 | Input = 12 4 5 6 19 10				
	50 Swaps	*500 Swaps*	*5000 Swaps*	*50000*
Null	6.70286e+07	7.38446e+06	753371	54516.3
Synchronized	7.89881e+07	6.55844e+06	788450	63792.5
Unsynchro-nized	6.40803e+07	5.96644e+06	676383	70030.2
GetNSet	7.29124e+07	6.47699e+06	733949	72179.3 **sum mismatch**
BetterSafe	6.47736e+07	6.95478e+06	719487	69752.0

Using Java Version 9 (all values ns/transition)

8 Threads | MaxVal = 20 | Input = 12 4 5 6 19 10				
	50 Swaps	*500 Swaps*	*5000 Swaps*	*50000*
Null	388622	53117.2	9729.64	90343.3
Synchronized	317628	45324.9	15279.3	76744.9
Unsynchro-nized	354429	64442.4 **Sum Mismatch**	10921.0 **sum mismatch**	66328.7
GetNSet	356221	51195.5 **sum mismatch**	16975.7 **sum mismatch**	**Infinite Loop**
BetterSafe	551220	138775	26368.8	6552.98

16 Threads	MaxVal = 50	Input = 22 4 15 23 19 11			
	50 Swaps	*500 Swaps*	*5000 Swaps*	*50000*	*500000*
Null	2.70715e+08	2.83398e+07	2.70939e+06	272032	26354.8
Synchronized	2.71390e+08	2.53370e+07	2.68989e+06	263867	26865.9
Unsynchro-nized	2.80555e+08	2.66729e+07	2.64839e+06	297113	23927.2
GetNSet	2.62272e+08	2.79238e+07	2.91713e+06	282754	27411.8
BetterSafe	2.89480e+08	2.48223e+07	2.92714e+06	309643	32479.5