CS131 Homework 3 Report

Hermmy Wang
704978214

1. Testing Platform

The program is tested on lnxsrv09.seas.ucla.edu.
CPU model name: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
CPU cores: 8
Total memory: 65755880 kB

Java versions:
(1) openjdk 11.0.2 2019-01-15
OpenJDK Runtime Environment 18.9 (build 11.0.2+9)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.2+9, mixed mode)
(2) openjdk 9
OpenJDK Runtime Environment (build 9+181)
OpenJDK 64-Bit Server VM (build 9+181, mixed mode)

2. Performance and reliability

```
$ java UnsafeMemory [class] [threads] [swaps] 127 1 2 3 4 1 2 3 4
```

The max value is set to 127, and the array is initialized to [1, 2, 3, 4, 1, 2, 3, 4].

Testing platform: OpenJDK 11.0.2

Swaps	Class	Threads		
		8	12	16
1000	Null	36664.5	59776.5	82937.8
	Synchronized	40710.7	67069.8	95037.3
	Unsynchronized	36100.3	49229.9	80659.1
	GetNSet	45924.8	70748.3	113453
	BetterSafe	81420.9	139504	197141
10000	Null	7488.6	11057	16887.7
	Synchronized	11528.2	18204.7	24438.3
	Unsynchronized	8794.48	12184.4	16555.7
	GetNSet	12127.7	18209.3	24443.8
	BetterSafe	16494.8	23437.4	34743.9
100000	Null	1936.62	2457.25	3613.77
	Synchronized	6336.21	7571.82	12968.4
	Unsynchronized	1798.19	2215.6	4022.67
	GetNSet	3991.28	4749.36	8248.8
	BetterSafe	5199.43	7720.87	10452
1000000	Null	2281.88	3434.86	4583.94
	Synchronized	2938.06	3927.25	6033.87
	Unsynchronized	1744.75	3309.46	4559.45
	GetNSet	1545.44	2875.58	4057.58
	BetterSafe	1440.48	1865.89	2854.44

Table 1. Nanosecond per transition running under OpenJDK 11.0.2.

Class	Success rate
Null	100%
Synchronized	100%
Unsynchronized	0%
GetNSet	0%
BetterSafe	100%

Table 2. Reliability of each class running under OpenJDK 11.0.2.

Testing platform: OpenJDK 9

Swaps	Class	Threads		
		8	12	16
1000	Null	20726.5	38763.1	55929.7
	Synchronized	27207.4	38796.9	50713.6
	Unsynchronized	21830.3	33020.4	51950.7
	GetNSet	34929.6	55483.5	75903.7
	BetterSafe	63648.8	115143	158321
10000	Null	6381.65	9777.05	11407.5
	Synchronized	9554.92	13687.3	18627.7
	Unsynchronized	7336.74	10869.9	12720.1
	GetNSet	11746.4	17212	23731
	BetterSafe	14588.7	24542.5	31797
100000	Null	1219.66	1770.43	2874.89
	Synchronized	6781.17	9949.06	11227.4
	Unsynchronized	1978.45	2228.88	3320.96
	GetNSet	3492.24	4895.18	6092.57
	BetterSafe	4849.29	7316.77	9778.92
1000000	Null	1905.22	2944.66	3977.56
	Synchronized	2492.36	4203.37	6140.93
	Unsynchronized	1365.57	2678.4	5892.92
	GetNSet	1316.1	2424.92	4103.48
	BetterSafe	1247.37	1735.13	2633.01

Table 3. Nanosecond per transition running under OpenJDK 11.0.2.

Class	Success rate
Null	100%
Synchronized	100%
Unsynchronized	0%
GetNSet	0%
BetterSafe	100%

Table 4. Reliability of each class running under OpenJDK 9.

According to the above test cases, the Synchronized class has shown 100% reliability and good performance for small amounts of transitions (<=10000 swaps). This class is

implemented by the "synchronized" reserved keyword in Java. The entire block of the swap() method is synchronized.

The unsynchronized class, although showing faster performance than the Synchronized class, is not reliable. It is guaranteed in the provided test cases that race conditions will always happen, resulting in a sum mismatch error. This class is implemented by removing the synchronized keywords from the code.

Although the BetterSafe class has the worst performance for lower number of transitions, it has demonstrated the best performance when performing more than100000 swaps. When the number of swaps is as high as 100000, the BetterSafe class has consistently outperformed the Synchronized class under OpenJDK 9 and almost always outperforms the Synchronized class under OpenJDK 11.0.8.

Out of the four classes, GetNSet has the worst performance. It is neither faster nor more reliable than the Unsynchronized class.

The test result for running the program on OpenJDK 9 is consistent with the one on 11.0.8. Nanosecond per transition is generally lower for OpenJDK 9, but this is probably due to how busy the server was during the test. It is more obvious on OpenJDK 9 that the BetterSafe class has faster performance when the number of swaps exceeds 100000 times. The BetterSafe always outperforms the Synchronized class in this case.

Therefore, on the two testing platforms, we yield the same conclusion that the BetterSafe class has demonstrated the best performance suitable for GDI that specializes in processing large data.

3. BetterSafe implementation

BetterSafe has better performance with increasing number of threads and swaps. When the number of swaps is low, there is a heavy overhead cost for the BetterSafe which affects its performance. According to the test cases, when the program performs swaps for more than 10000 times, BetterSafe outperforms Synchronized in 91.67% and 100% of the time on two testing programs OpenJDK 11.0.8 and OpenJDK 9 respectively. Therefore, it is apparent that BetterSafe is faster than Synchronized.

I used the ReentrantLock in java.util.concurrent.locks to implement the BetterSafe class. The lock is acquired at the beginning of the swap method to guarantee that the access of the array elements is secure. Then the lock is released upon the completion of swapping. The ReentrantLock makes sure that there is always one thread only holding the lock and is able to change the array. Hence, BetterSafe is 100% reliable. The reason that the ReentrantLock is faster than the synchronized method is due to the fact that it is more flexible and provides a non-block context. I can lock only a part of the code in a function instead of locking the entire block as the synchronized method does.

Choosing packages:

`java.util.concurrent`
This package provides frameworks that are commonly used in concurrent programming. The provided APIs include queues that are thread-safe and non-blocking, timing for controlling time-out-based operations, synchronizers as common concurrency tools, and many other concurrent collections. [1] Although these frameworks are extensible and powerful, the implementation is complicated and not straight-forward. Since code readability is also taken account in writing programs, I did not choose this package.

`java.util.concurrent.atomic`
This package supports lock-free and thread-safe programming and extensible atomic operations on variables. Atomicity is guaranteed on these variables such that changing a variable is either done or not done; nothing unpredictable in-between state will occur to it.[2] Although the operations on this package is lock-free and thread-safe, we do need to copy the given array to a specific atomic object provided by the package. The copying operation introduces more risks to the reliability of the program, so I decided not to use this package.

`java.util.concurrent.locks`
This package allows a flexible use of locks and conditions beyond the build-in synchronized methods. I decided to use this package to implement BetterSafeState.java. Unlike the synchronized methods which synchronize an entire block, the ReentrantLock available in this package can be used in a non-block-structured context. Therefore, besides the basic mutual exclusion functionality as the synchronized methods, ReentrantLock also has extended capabilities such as locking across methods or locking only a part of a method. [3][4]

`java.lang.invoke.VarHandle`
A VarHandle is a type reference to a variable. JDK 9 versions of java.util.concurrent.atomic classes include methods corresponding to these VarHandle construction. VarHandle methods are applied to single Atomic objects. VarHandles are immutable, have no visible state, and cannot be sub-classed. VarHandle provides for read access, write access, and atomic update access which can be used for concurrent programming. VarHandle aims to perform low-level operations, so it should not be used unless unnecessary. [5][6]

4. Difficulties and potential failed test cases

The test has shown that the program is likely to fail when the number of threads is low, the number of swaps is high, and the initial array contains values that are very close to the maximum value 127. The following two classes are not data-race-free (DRF):

UnsynchronizedState: not DRF

```
$ java UnsafeMemory Unsynchronized 4
1000000 127 126 125 126 127 125 126
```

[infinite loop]

GetNSetState: not DRF

```
$ java UnsafeMemory GetNSet 4 1000000
127 126 125 126 127 125 126
```

[infinite loop]

In the BetterSafe class implementation, I did not unlock the Reentrantlock before returning false at the beginning, therefore it always enters the deadlock situation. I have also noticed that OpenJDK 9 does not support sprouting more than 20 threads. The compiler displays the following error message:

```
[0.161s][warning][os,thread] Failed to start thread - pthread_create
failed (EAGAIN) for attributes: stacksize: 1024k, guardsize: 0k, detached.
Exception in thread "main" java.lang.OutOfMemoryError: unable to create
native thread: possibly out of memory or process/resource limits reached
          at java.base/java.lang.Thread.start0(Native Method)
          at java.base/java.lang.Thread.start(Thread.java:813)
          at UnsafeMemory.dowork(UnsafeMemory.java:62)
          at UnsafeMemory.main(UnsafeMemory.java:28)
```

Therefore, I limit the number of threads to 8, 12, and 16.

5. Conclusion

Among the four choices of implementations, BetterSafe is the best candidate for GDI than synchronized with respect to performance and reliability. It provides 100% reliability and fast performance for large amounts of data. BetterSafe is implemented by the ReentrantLock provided in java.util.concurrent.locks. A lock is acquired in beginning of the data operation method and is released after the operation is complete. The critical section is accurate and minimized. Therefore, the BetterSafe class provides both performance and reliability and is most suitable for GDI.

6. References

[1] Package java.util.concurrent

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html

[2] Package java.util.concurrent.atomic

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

[3]Package java.util.concurrent.locks

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/package-summary.html

[4] Difference Between ReentrantLock and Synchronized in Javahttps://netjs.blogspot.com/2016/02/difference-between-reentrantlock-and-synchronized-java.html

[5] Class VarHandle.
https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html

[6] Java 9 Variable Handles Demystified.
https://www.baeldung.com/java-variable-handles