Fang Yujie
UID: 404796809
Dis 1B

Abstract:
This homework investigates the reliability and performance of four classes: Synchronized, Unsynchronized, GetNSet and BetterSafe. Different number of threads are used to test the performance and reliability for all classes. During the process, the differences between *java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks, java.lang.invoke.VarHandle,* and how they are implemented to avoid race condition are discussed.

1. Instructions for HW3:

Java version check:

```
$ export PATH=/usr/local/cs/bin:$PATH
$ java –version
java version "9.0.4"
Java(TM) SE Runtime Environment (build
9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build
9.0.4+11, mixed mode)
```

Convert to byte code:

```
javac UnsafeMemory.java
```

Compilation:

```
java UnsafeMemory
```

test:

```
java UnsafeMemory <state> <thread_number>
1000000 100 5 6 3 0 3
```

2. Compare Between Different Implementations:

State Name	Threads average(ns/transition)	
	2	4
Synchronized	454.496	1317.78
Unsynchronized	190.662	505.171
GetNSet	270.731	666.097
BetterSafe	462.503	1027.5

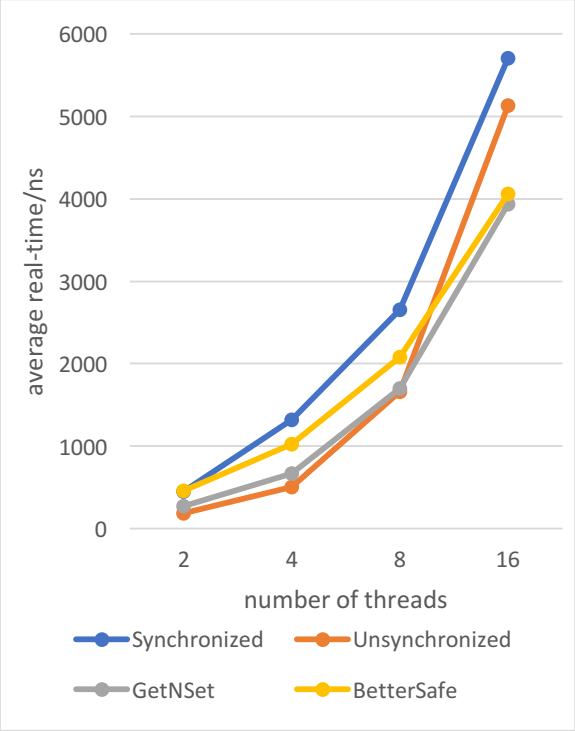
Table 1: average time over different number of threads and different classes

State Name	Threads average(ns/transition)	
	8	16
Synchronized	2656.02	5702.6
Unsynchronized	1664.46	5126.32
GetNSet	1700.94	3932.59
BetterSafe	2079.34	4062.32

Table 2: average time over different number of threads and different classes

State Name	DRF
Synchronized	Yes
Unsynchronized	No
GetNSet	No
BetterSafe	Yes

Table 3: whether a class is DRF (Data-Race Free)



Graph 1: average real-time of the performance of different classes.

3. class's performance and reliability

3.1.Synchronized:

Among all classes, synchronized shows the lowest performance score, especially for higher number of threads, while giving 100% reliability. The keyword “synchronized” ensures that it gives no race condition.

3.2. Unsynchronized:

unsynchronized class fails and does not give any output. As all keywords “synchronized” are removed, it does not guarantee race free, leading to infinite loop. Therefore, it is the most unreliable class among all.

3.3. GetNSet:

GetNSet gives the second-best performance as shown by Graph 1. GetNSet uses *java.util.concurrent.atomic.AtomicIntegerArray*, which allows each element to be accessed and updated atomically. However, during testing, it always prints output in the form of "`negative output (-73 != 0)`", which indicates that GetNSet state is not data race free since the array should maintain the same sum throughout the program if it is data race free. As I look into the function for *get* and *getAndIncrement*, I realize that there is a if statement that ensures thread-safe between access and update. It is possible that some threads may access the same element of the array at different times and then updating it more than once using the incremented value, resulting in race conditions.

3.4. BetterSafe:

BetterSafe class gives the better performance while retaining 100% reliability. As shown by Graph 1, its efficiency is twice of the efficiency for synchronized. It implements *java.util.concurrent.locks*, which gives only one access to a thread and then locks it. It is only unlocked by the same owner after updating the value and then allows other threads to access the same element. While an element is "locked", other threads that tries to access or update the same element have to wait for it to be unlocked. This ensures reliability.

4. BetterSafe implementation

4.1. Why not semaphore in java.util.concurrent?

The benefit of semaphore is that it avoids race condition by allowing only one thread to read and write before another thread starts. However, it does not protect shared variables from being accessed by different threats at the same time.

To be more specific, the main difference between synchronized and *java.util.concurrent.locks*, is that the former is a synchronization primitive whereas the latter a higher level locking construct which provides more elaborate operations than synchornized.

Semaphore, as a concurrent tool for synchronized, has the following properties:

- Has no concept of ownership
- Any thread can invoke P or V operations Consecutive P (or V) operations will be blocked
- Need to specify an initial value

In contrast, lock has the following properties:

- A lock can be owned by at most one thread at any given time
- Only the owner can invoke unlock operations
- The owner can invoke lock (or unlock) operations in a row.
- Does not have to be initialized

For a binary semaphore, consecutive P operations will be blocked. But a thread that owns a lock can invoke lock operations again without being blocked. The owner for calls to lock and unlock must be the same thread. But calls to P and V can be made by different threads.

4.2. Why not java.util.concurrent.atomic?

GetNSet uses concurrent.atomic, but it still has race condition. Why? Although each individual operation for a thread involved is atomic, the combined is not. The operations for multiple threats can interleave, so it is possible that just after one thread access an element, the other thread immediately updates the same element to another value, resulting in race condition. Therefore, as concurrent.atomic does not guarantee 100% reliability, it is not a good implementation for BetterSafe.

I tested GetNSet state for 2, 4, 8 and 16 threads; each one was tested 20 times. The error rate is recorded as follows:

Number of threats	2	4	8	16
Error rate	50%	55%	45%	45%

4.3. Why not VarHandle?

VarHandles should be declared as static final fields and explicitly initialized in static blocks better performance results than Atomic classes. In theory, it is faster than atomic.

Since VarHandle is a generic version of AtomicInteger, it shares the downside of atomic such as separate reads and writes that might cause race condition.

The following code is used to test for VarHandle:

```
import java.lang.invoke.MethodHandles;
import java.lang.invoke.VarHandle;

class BetterSafe implements State {
	private byte[] value;
	private byte maxval;
	private VarHandle x =
MethodHandles.arrayElementVarHandle(byte[]
.class);

	BetterSafe(byte[] v){
		value = v;
```

```
            maxval = 127;
        }

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

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

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

        public boolean swap(int i, int j){
           Byte ci = (Byte)x.get(value, i);
           Byte cj = (Byte)x.get(value, j);
           if (ci.byteValue() <= (byte)0 ||
cj.byteValue() >= (byte)maxval){
            return false;
           }
           x.getAndAdd(value, i, (byte)-1);
           x.getAndAdd(value, j, (byte)1);
          return true;
        }
}
```

For 1000000 successful swap transitions, the error rate is very low, but when I change 1000000 to 1000, the error rate increases.

The error rate for 1000 successful swap transitions is recorded as follows:

Number of threats	2	4	8	16
Error rate	5%	0%	10%	5%

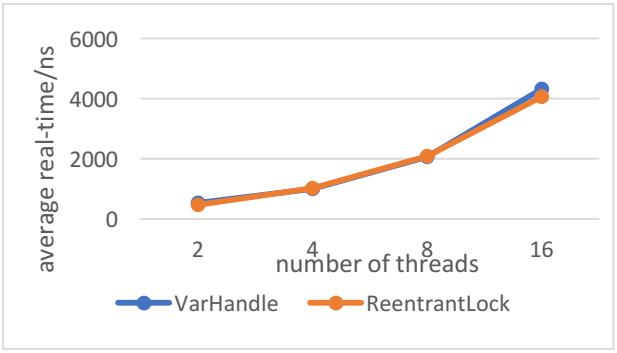
I tested

```
java UnsafeMemory BetterSafe 4 1000000 100
5 6 3 0 3
```

for both VarHandle and ReentrantLock and compared their performance:

Classes used	Thread Average real time(ns/transition)			
	2	4	8	16
VarHandle	525.827	1011.24	2067.65	4316.43
ReentrantLock	462.503	1027.5	2079.34	4062.32

Table 4: performance comparison between using ReentrantLock and using VarHandle.



Graph 2: performance comparison between using ReentrantLock and using VarHandle.

As shown by the graph, the performance difference is not very distinct. But since VarHandle does not guarantee 100% reliability, it is not chosen for BetterSafe.

4.4. Why not synchronize?

The major difference between lock and synchronized is the following:

- with locks, you can release and acquire the locks in any order;
- with synchronized, you can release the locks only in the order it was acquired.

This explains why reentrant shows better performance than synchronize.

4.5. Why ReentrantLock?

A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities. The framework permits much greater flexibility in the use of locks and conditions, allowing release and acquire the locks in any order.

5. Conclusion

problems I overcame:

- VarHandle is very difficult to write. It is passed by object and cannot be converted to byte by coercion. Every time I access it, I need to know what object is passed before I give it the correct built-in function to call.
- I run into inifinite loop as I run unsynchronized class. Since the race condition is deliberately added, the only way to produce the average real-time per transition is to reduce the number of thread and increase maxval to above 100. Therefore, to ensure consistency between the parameters that I passed, I made maxval to be 100.