Assignment 4

Week 7

7.1.1) *Write a class CasHistogram implementing the above interface. In your implementation, ensure that: i) class state does not escape, and ii) safe-publication. Explain why i) and ii) are guaranteed in your implementation, and report any immutable variables*

we have only one state, which is defined as `private final AtomicInteger[] counts;`

i) class state does not escape: as it is set as private and no methods return a reference to it.

ii) safe-publication: as it is defined as final, The initialization of a final field happens-before any other actions of a program (after the constructor has finished its execution)

7.1.2) *Write a parallel functional correctness test for CasHistogram to check that it correctly stores the number of primes in the range (0, 4999999); as you did in exercise 6.3.3 in week 6. You must use JUnit 5 and the techniques we covered in week 4. The test must be executed with 2 n threads for n ∈ {0, . . . , 4}. To assert correctness, perform the same computation sequentially using the class Histogram1 from week 6. Your test must check that each bin of the resulting CasHistogram (executed in parallel) equals the result of the same bin in Histogram1 (executed sequentially).*

See *TestHistograms.java*

7.1.3) *Measure the overall time to run the program above for CasHistogram and the lock-based Histogramweek 6, concretely, Histogram2. For this task you should not use JUnit 5, as it is does offer good supportto measure performance. Instead you can use the code in the file TestCASLockHistogram.java. Itcontains boilerplate code to evaluate the performance of counting prime factors using two Histogram classes.To execute it, simply create two objects named histogramCAS and histogramLock containing yourimplementation of Histogram using CAS (CasHistogram) and your implementation of Histogram usinga single lock from week 6 (Histogram2).*

What implementation performs better? The (coarse) lock-based implementation or the CAS-based one? Is this result you got expected? Explain why.

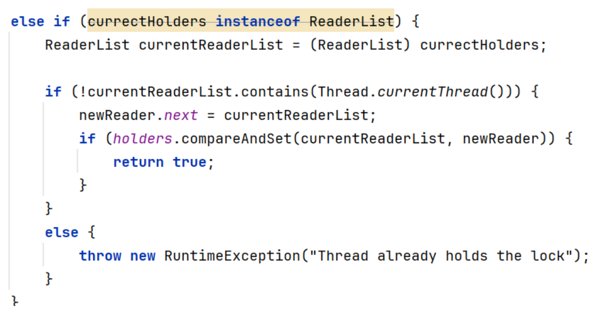
In this case, the compare-and-swap implementation performs significantly faster than the single-lock implementation. Note that the two implementations performs equally when executing on one thread only, but when reaching a thread count of 8, the compare-and-swap is approximately twice as fast as the single-lock implementation.

Lock is a blocking algorithm; when a thread cannot acquire the lock, it enters a lock-wait queue, waiting to be awakened by other threads, which have a context-switching overhead. When update operations are very fast, using locks may result in the thread spending more time transitioning from a Blocked state back to a Runnable state than actually performing the update.

CAS is non-blocking. When an update conflicts, it retries without blocking So there is no switch cost. When contention is not very high, the efficiency of CAS is higher.

```
j: 0
CAS with 1 thread              3616941420,0 ns 60194633,63  2
Lock with 1 thread             3616363685,0 ns 88325973,46  2

j: 1
CAS with 2 thread              2242904970,0 ns 15467367,11  2
Lock with 2 thread             2299372725,0 ns 32074161,65  2

j: 2
CAS with 4 thread              1204195055,0 ns 6551503,64   2
Lock with 4 thread             1339010580,0 ns 20264616,49  2

j: 3
CAS with 8 thread               708045330,0 ns 46750365,78  2
Lock with 8 thread             1303002040,0 ns 9041905,67   2

j: 4
CAS with 16 thread              513623825,0 ns 85524044,30  2
Lock with 16 thread            1186199575,0 ns 77376515,16  2
```

***(Challenging) 7.2.7**) Improve the readerTryLock method so that it prevents a thread from taking the same lock more than once, instead an exception if it tries. For instance, the calling thread may use the contains method to check whether it is not on the readers list, and add itself to the list only if it is not. Explain why such a solution would work in this particular case, even if the test-then-set sequence is not atomic.*



```java
else if (currectHolders instanceof ReaderList) {
    ReaderList currentReaderList = (ReaderList) currectHolders;

    if (!currentReaderList.contains(Thread.currentThread())) {
        newReader.next = currentReaderList;
        if (holders.compareAndSet(currentReaderList, newReader)) {
            return true;
        }
    }
    else {
        throw new RuntimeException("Thread already holds the lock");
    }
}
```

Between the if(xx) check and the holder.compare operation, there may be interleavings.

Case 1: Unlocking occurs. Only the thread itself can unlock the lock it holds. If it is the same thread attempting to unlock, it is impossible because single-threaded operations can only be executed sequentially.

Case 2: Locking occurs. If it is the same thread attempting to lock again, this is also impossible. If other threads attempt to acquire the lock and modify the holders, the compareAndSet here will fail and will be retried.

Assignment 4

Week 8

8.1.1) *Is this execution sequentially consistent? If so, provide a sequential execution that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not sequentially consistent.*

```
A: ---------------|q.enq(x)|--|q.enq(y)|->
B: ---|q.deq(x)|------------------------->
```

Yes it sequentially consistent as we have the sequential re-ordering that satisfies specifications:

<q.enq(x) ,q.enq(y) ,q.deq(x)>

1. Operations happen one-at-a-time 2. Program order is preserved (for each thread) 3. The execution satisfies the specification of the object

8.1.2) *Is this execution (same as above) linearizable? If so, provide a linearization that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not linearizable.*

It is not linearizable as q.deq(x) and q.enq(x) does not overlap, meaning we can't have a sequential execution of the linearzation points where q.enq(x) happens before q.deq(x).

8.1.3) *Is this execution linearizable? If so, provide a linearization that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not linearizable.*

```
A: ---|        q.enq(x)           |-->
B: ------|q.deq(x)|--------------->
```

It is linearizable as the linearizable points can be mapped to this sequential execution that satisfies the specifications.

<q.enq(x) ,q.deq(x)>

linearization points within each method call, which map to sequential execution that satisfy the specification of the object

8.1.4) *Is this execution linearizable? If so, provide a linearization that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not linearizable.*

```
A: ---|q.enq(x)|-----|q.enq(y)|-->
B: --|        q.deq(y)         |->
```

No it is not linearizable, as there is no sequential ordering of the linearization points where q.enq(y) happens before q.enq(x), so it will always violate the FIFO specification, when q.deq(y) is executed.

Challenging 5.) *Is this execution sequentially consistent? If so, provide a sequential execution that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not sequentially consistent.*

A: -|p.enq(x)|------------|q.enq(x)|------------|p.deq(y)|------------>
B: ------------|q.enq(y)|------------|p.enq(y)|------------|q.deq(x)|->

q.enq(y), p.enq(x), p.enq(y) q.enq(x) p.deq(y) q.deq(x)

Case 1:

if we want to deq(y) in p first, then we need first enq(y) in p. Then it must be
q.enq(y) p.enq(y)
Then we will deq(x) in q. So in q we should first enq(x). It is conflict.

Case 2:

if we want to deq(x) in q first, then in q we should first enq(x). Must be
p.enq(x)------q.enq(x)
Then we will deq(y) in p. So in p we should first enq(y). It is conflict.

So it isn't exist a reordering of operations producing a sequential execution.

8.2.1) *Define linearization points for the push and pop methods in the Treiber Stack code provided in app/src/ main/java/exercises08/LockFreeStack.java. Explain why those linearization points show that the implementation of the Treiber Stack is correct. For the explanation, follow a similar approach as we did in lecture 8 for the MS Queue (slides 44 and 45).*

push() has one linearization points:

- P1: If successfully executed, the element has been pushed into the stack.

```
    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));    //P1
    }
```

Correctness:

- If two threads execute enqueue concurrently before top is updated, then only one them will succeed in P1, while the other will fail and repeat the push.
- If top was updated by another Thread P1 will fail, and the thread will try again using the new top. ? (根据slide只conve if branch, 分析了所以if 失败的情况. 我觉得这里可能不需要这个, 这应该时说top被pop的修改了)

pop() has two linearization point:

- P1: If the stack is empty. After its execution, the evaluation of if(oldHead ==null) is determined and the method will return null.
- P2: If successfully executed, the element has been popped from the stack.

```
    public T pop() {
        Node<T> newHead;
        Node<T> oldHead;
        do {
            oldHead = top.get();                        //P1
            if(oldHead == null) { return null; } // A1
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead,newHead));  //P2

        return oldHead.value;
    }
```

Correctness:

- If two threads try to execute pop() concurrently before top is updated and the stack is not empty(A1 fails), then P2 succeeds for only one of them, while the other will repeat pop.
- If a thread executes pop() after top has been changed by another thread P2 will fail and attempt again to remove the current top, until it succeeds or the stack is empty.

8.2.2-3)

See TestLockFreeStack.java

8.3.1) *Consider the reader-writer locks exercise from week 7. There are four methods included in this type of locks: writerTryLock, writerUnlock, readerTryLock and readerUnlock. State, for each method, whether they are wait-free, lock-free or obstruction-free and explain your answers*

writerTryLock:

```java
return holder.compareAndSet(null, new Writer(Thread.currentThread()));
```

It is Wait-Free, as the method is executed in a finite number of steps, no matter the contention from other threads.

writerUnlock:

```java
if (!holder.compareAndSet(new Writer(Thread.currentThread()), null)) {
    throw new RuntimeException("Not the current Thread");
}
```

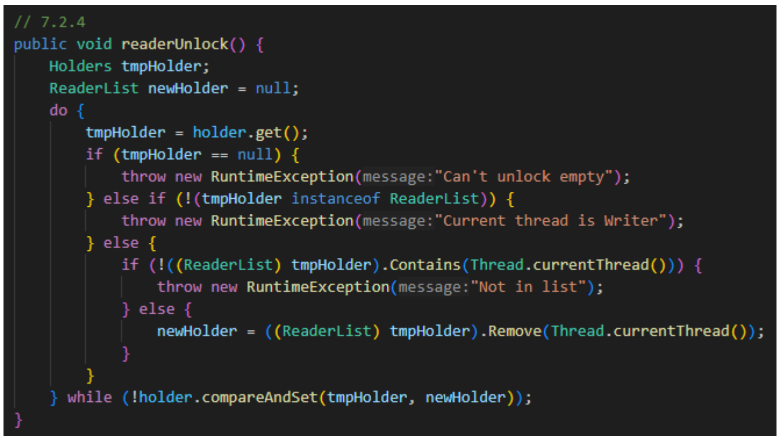
It is Wait-Free, as the method is executed in a finite number of steps, no matter the contention from other threads.

readerTryLock:

```java
Holders currentHolder;
while (holder.get() instanceof ReaderList || holder.get() == null) {
    currentHolder = holder.get();
    if (holder.compareAndSet(currentHolder,
        new ReaderList(Thread.currentThread(), (ReaderList) currentHolder)))
        {return true;}
}
return false;
```

It is Lock-Free as the method guarantees that at least one thread will make progress in a finite number of steps, no matter the contention from other threads. One thread will succeed in updating in the compareAndSet, while other contending threads will re-attempt to update the holder value with the new information.

readerUnlock:



```java
// 7.2.4
public void readerUnlock() {
    Holders tmpHolder;
    ReaderList newHolder = null;
    do {
        tmpHolder = holder.get();
        if (tmpHolder == null) {
            throw new RuntimeException(message:"Can't unlock empty");
        } else if (!(tmpHolder instanceof ReaderList)) {
            throw new RuntimeException(message:"Current thread is Writer");
        } else {
            if (!((ReaderList) tmpHolder).Contains(Thread.currentThread())) {
                throw new RuntimeException(message:"Not in list");
            } else {
                newHolder = ((ReaderList) tmpHolder).Remove(Thread.currentThread());
            }
        }
    } while (!holder.compareAndSet(tmpHolder, newHolder));
}
```

It is Lock-Free as the method guarantees that at least one thread will make progress in a finite number of steps, no matter the contention from other threads. One thread will succeed in updating in the compareAndSet, while other contending threads will re-attempt to update the holder value with the new information.