Assignment 1

Week 2

2.1.2) No. Readers can come as long as there are no writers, but writers need to wait until there are 0 readers, and new readers can come in while a writer is waiting (in other words skipping the queue). If readers keep coming infinitely, writers will be blocked forever.

After change, the answer is yes. In this case, even if readers keep coming, due to select thread from lock queue is non-deterministically, once writers is being selected, it will set the write variable to 'true'. Then next time writer can execute and readers will be blocked. So both writer and reader will run eventually once they are ready.

2.2.1) Yes, it is stuck in the "*while (mi.get() == 0)*" loop. *mi* object is visible to the new thread, but it does not get the new state of mi when the main thread sets its value state to 42. Therefore the Thread can loop forever as mi.value is stuck as 0, meaning the while loop will keep running. It does depend on the hardware as a slow computer might first start the new thread after the main Thread has slept for 500 milliseconds and executed *mi.set(42)*.

2.2.2) By using synchronized/locks in the methods we ensure that the *mi* object and its state are visible to all Threads regardless of which CPU they are running on, as the unlock method flushes the object from CPU cache to the memory, which is available to all threads.

2.2.3) No. If we use synchronized methods,before each thread unlocks, it will refresh the latest value of the shared variable to the shared memory. When a thread locks, it will retrieve the latest shared variable from the shared memory. If we don't use synchronized for get(), even though the main thread refreshes the value in the shared memory, thread T will still access the old value stored in the cache.

2.2.4) Yes, because volatile is stored in memory available to all threads, so the main thread can set the value to 42 and the new thread will see this value and get out of the loop.

2.3.1)

> “Sum is 1348167,000000 and should be 2000000,000000”
>
> “Sum is 1875074,000000 and should be 2000000,000000”
>
> “Sum is 1448499,000000 and should be 2000000,000000”

Yes, there is a race condition. They try to increment sum by 1 at the same time, so instead of incrementing by 2 it is only incremented by 1.

2.3.2) It is because the two methods do not use the same lock, as the addStatic method uses the Class-level, and the addInstance method uses the Instance-level lock. So they can both increment the sum variable simultaneously, creating a race condition.

2.2.3) By using a reentrant lock we ensure the two Threads can never read and write to the sum state in the critical section simultaneously, so interleaving such as:

```
T1(read), T2(read), T1(write), T2(write)
```

can never happen, eliminating race conditions.

2.3.4) In the given case, removing the synchronized keyword from “sum()” will not create any race conditions, as “sum()” is not called until both t1 and t2 have finished executing, so we do not need to prevent possible reordering.

However, the synchronized keyword is essential. Even if 'sum' is a static variable, without using the synchronized keyword, it's still possible for “sum()” to read the outdated value that's thread-private.