**Assignment 2**

**Answer No 1:**

a)

On SingleLockConcurrentCertainBookStore we take a global lock at the start of our function calls, thus at top level we achieve correct before-or-after atomicity.  
For TwoLevelLockingConcurrentCertainBookStore, at top-level the takeGlobal- Lock is acquired on exclusive mode when we perform addBooks and remove- Books. globalReadLock is taken whenever a local lock is taken. This ensures that a global exclusive lock can’t be taken before all local locks are released.

At the bottom level, there is one read-write lock for each book in the bookstore. We use takeLocalWriteLock on data items that are modified and takeLocal- ReadLock on data items that are only read. We do this during the execution of the transaction as needed. We release locks on objects at the end of the transaction or in case of some exception being thrown.

At each level the transactions are executed as if in serial order, so the TwoLevel- LockingConcurrentCertainBookStore has the correct before-or-after atomicity.

b)

We have used two general strategies for testing the concurrency code. The first strategy is to have two different threads doing opposite operations. At the end, we test that the environment is unchanged. This can for example be seen in testBuyAndAddConcurrent and testUpdateEditorPicks.

The other strategy we use is to have some number of threads repeatedly doing an ac- tion and a testing thread repeatedly reads the updates and checks consistency. This is to ensure that threads can only observe legal states. Such tests are for example testBuyAndAddConsistency, testRatedBooks and testUpdateEditorPicks. The latter of the two uses two threads doing updates and one checking consistency.

c)

We don’t have to consider different testing. Because the scheme in SingleLock- ConcurrentCertainBookStore is the same as the scheme in TwoLevelLockingCon- currentCertainBookStore  
The use of different strategies would not be a violation of modularity, but it might be helpful to still consider it. Different implementations have different edge cases where there might be bugs. Taking this into consideration when writing tests is a good idea.

**Answer No 2:**

The SingleLockConcurrentCertainBookStore implementation is not only equivalent with strict 2PL, it is also equivalent with conservative 2PL. This is because all functions im- mediately take all locks (the only lock) they need and release it at the very end of the call.

For TwoLevelLockingConcurrentCertainBookStore, things are a little more complicated. In 2PL, there is a growing phase and a shrinking phase. Locks are taken in the growing phase and released in the shrinking phase. Strict 2PL also requires that all exclusive locks are released at the very end of the transaction, when either commiting or aborting.

The growing phase is encoded in our implementation simply as part of the function body. The thread acquires locks on objects as it needs them. The shrinking is embedded in the finally blocks that wrap the function body. Here, all locks taken by the transaction are released. Since this includes all exclusive locks, the implementation is strict.

Since we implement a shrinking and growing phase and all exclusive locks are released only upon commit or abort, our implementation is equivalent to strict 2PL.

**Answer No 3:**

For SingleLockConcurrentCertainBookStore, our locking protocol will not lead to deadlocks. We first take writeLock on data items that are modified and take readLock on data items that are read, then execute transaction, finally release all locks. Therefore there is no deadlocks.

• For TwoLevelLockingConcurrentCertainBookStore, our locking protocol will lead to deadlocks. At top-level the takeGlobalLock is acquired on exclusive mode when we perform addBooks and removeBooks, globalReadLock is in all the other operations. At the bottom level, there is one read-write lock for each book in the bookstore.

takeLocalWriteLock on data items that are modified and takeLocalReadLock on data items that are read, but do this during execution of transaction (as needed). Release locks on objects no longer needed during execution of transaction. There- fore we need to know when to release locks, which may leads to deadlocks.

**Answer No 4:**

We have no thrashing handling in the implementation, so a large amount of requests could simply bog down our store in overhead computations. This is especially true for SingleLockConcurrentCertainBookStore because of the low amount of concurrency in the implementation.

**Answer No 5:**

The concurrency for SingleLockConcurrentCertainBookStore is limited by the number of operations that modify state, since these need exclusive locks and put everything else on hold. In a bookstore, we can expect there to be less requests for writing than reading. Since the overhead is pretty small, this tradeoff is probably good.

With the two-level locking scheme we get to pay quite a bit more overhead than with the single-lock: Our locks live in a map and take O(n) time to find every time we need one. If our bookstore has an enormous amount of books and a large amount of requests, this might actually have impact on performance.

On the other hand, we achieve the benefit that a write operation no longer locks the rest of the bookstore from functioning, which is an advantage if the number of requests is large.