ACS Assignment 2

This assignment is due via Absalon on December 4, 23:59. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of two students. Groups may at a maximum include three students. NOTE: The KU IDs of ALL group members MUST be stated on a separate `group.txt` file to ensure all group members get feedback and get the assignment accounted for in Absalon.

A well-formed solution to this assignment should include a PDF file with answers to all theory questions, as well as all questions posed in the programming part of the assignment. In addition, you must submit your code along with your written solution. Evaluation of the assignment will take both into consideration.

Note that all homework assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Learning Goals

This assignment targets the following learning goals:

- Analyze the serializability of transaction schedules, explaining why certain schedules are serializable and some are not.
- Predict decisions that would be made by different concurrency control protocols, e.g., variants of two-phase locking (2PL) or optimistic concurrency control.
- Design concurrency control mechanisms for a particular system in which operations must be guaranteed to be atomic, and argue for the correctness of these mechanisms by equivalence to a 2PL variant, while considering issues such as predicate reads, deadlocks, and the amount of concurrency achieved.
- Implement a concrete locking strategy in a modular service to guarantee the atomicity of operations in multithreaded executions.

Theory questions

Question 1: Serializability & Locking

Consider the following two transaction schedules with the time increasing from left to right. C indicates commit.

Schedule 1

```
T1: R(X)                                                W(Y)   C
T2:         W(Z)  W(X)  C
T3:                                 R(Z) R(Y)  C
```

Schedule 2

```
T1: R(X)                            W(Y)  C
T2:                     R(Z)                        W(X) W(Y)  C
T3:         W(Z)  C
```


Now answer the following questions:

- Draw the precedence graph for each schedule. Are the schedules conflict-serializable? Explain why or why not.
- Could the schedules have been generated by a scheduler using strict two-phase locking (strict 2PL)? If so, show it by injecting read/write lock operations in accordance to strict 2PL rules. If not, explain why.

Question 2: Optimistic Concurrency Control

Consider the following scenarios, which illustrate the execution of three transactions under the Kung-Robinson optimistic concurrency control model. In each scenario, transactions T1 and T2 have successfully committed, and the concurrency control mechanism needs to determine a decision for transaction T3. The read and write sets (RS and WS, respectively) for each transaction list the identifiers of the objects accessed by the transaction.

Scenario 1

```
T1: RS(T1) = {1, 2, 3}, WS(T1) = {3},
  T1 completes before T3 starts.
T2: RS(T2) = {2, 3, 4}, WS(T2) = {4, 5},
  T2 completes before T3 begins with its write phase.
T3: RS(T3) = {3, 4, 6}, WS(T3) = {3},
  allow commit or roll back?
```

Scenario 2

```
T1: RS(T1) = {1, 2, 3}, WS(T1) = {3},
  T1 completes before T3 begins with its write phase.
T2: RS(T2) = {5, 6, 7}, WS(T2) = {8},
  T2 completes read phase before T3 does.
T3: RS(T3) = {3, 4, 5, 6, 7}, WS(T3) = {3},
  allow commit or roll back?
```

Scenario 3

```
T1: RS(T1) = {2, 3, 4, 5}, WS(T1) = {4},
  T1 completes before T3 begins with its write phase.
T2: RS(T2) = {6, 7, 8}, WS(T2) = {6},
  T2 completes before T3 begins with its write phase.
T3: RS(T3) = {2, 3, 5, 7, 8}, WS(T3) = {7, 8},
  allow commit or roll back?
```


For each scenario, predict whether T3 is allowed to commit. If T3 is allowed to commit, state which validation tests were necessary to reach that conclusion. If T3 must be rolled back, state all tests which T3 fails, along with the offending objects from the read and write sets of T1 and T2.

Programming Task

A Concurrent Certain Bookstore

The team of `acertainbookstore.com` has tasted success and their bookstore service has become immensely popular. This has resulted in a lot of clients requesting services from the bookstore. The team has now assigned you the responsibility to increase concurrency in the interfaces implemented by the bookstore without violating application semantics. In particular, the team requires that every operation in the service respect *before-or-after atomicity*. As a consequence, the results of executing operations concurrently in multiple threads should be equivalent to the results of *some* serial order of execution of the same operations.

For this assignment, you are provided with two classes, *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*, which are *almost* similar to the *CertainBookStore* class handed out in Assignment 1. The only difference is that the `synchronized` keyword has been removed from the method signatures. It is your job to design and implement two *lock-based* concurrency control protocols that will produce equivalent before-or-after atomicity for the operations, but achieve higher performance by allowing for greater concurrency.

Note that even though we are using the same underlying code base throughout Assignments 1-4, every assignment will be independent. For Assignment 2, you can (and should)

leave the functionality for `rateBooks`, `getTopRatedBooks`, and `getBooksInDemand`, discussed in Assignment 1, unimplemented. Making your locking protocol work with these extra methods is completely *optional*.

Implementation

Create tests for your concurrent implementation

As always at `acertainbookstore.com`, test-driven development is very much encouraged. In order to focus on concurrency, you are asked to prepare a set of local tests: clients run on the same address space as the *SingleLockConcurrentCertainBookStore* or *TwoLevelLockingConcurrentCertainBookStore* services, while continuing to access it exclusively through the *BookStore* and *StockManager* interfaces; at the same time, clients run in different threads, and share the same thread-safe object for the service.

The team at `acertainbookstore.com` has described to you two test cases that they believe are important for the concurrent implementation of the service:

- **Test 1**: Two clients *C1* and *C2*, running in different threads, each invoke a fixed number of operations, configured as a parameter against the *BookStore* and *StockManager* interfaces. Both *C1* and *C2* operate against the same set of books S. *C1* calls *buyBooks*, while *C2* calls *addCopies* on S. Configure the initial state of the store in such a way that there is a sufficient number of copies in stock to execute a fixed number of operations without exceptions. The net result of the test should be that the books in S end with the same number of copies in stock as they started. If the latter is not verified, the test fails, indicating that operations that perform conflicting writes to S were not atomic.
- **Test 2**: Two clients *C1* and *C2*, running in different threads, continuously invoke operations against the *BookStore* and *StockManager* interfaces. *C1* invokes `buyBooks` to buy a given and fixed collection of books (e.g., the Star Wars trilogy). *C1* then invokes `addCopies` to replenish the stock of exactly the same books bought. *C2* continuously calls `getBooks` and ensures that the snapshot returned either has the quantities for *all* of these books as if they had been just bought or as if they had been just replenished. In other words, the snapshots returned by *getBooks* must be *consistent*. The test fails if any inconsistent snapshot is observed, and succeeds after a large enough number of invocations of `getBooks`, configured as a parameter to the test.

In addition to the tests above, you should extend the set of test cases further to exercise different conditions and scenarios with multiple threads. We expect you to include *at least* 4 test cases related to concurrency in the `com.acertainbookstore.client.tests` package, including the two tests above (i.e., the two test cases above plus two additional test cases minimum). You may extend the test classes given with these additional test cases, or alternatively, create a separate JUnit test class for them.

In your solution, you should explain the test cases you have added, and argue why they help you test different concurrency aspects of your implementation. Your tests will be evaluated by how much they potentially trigger different anomalies or expose the service to different scenarios.

Note: Focusing exclusively on local tests with multiple threads is enough to achieve the learning goals for this assignment. However, if you would like to integrate your concurrent bookstore implementations with RPCs, you may *optionally* do so. But again, we emphasize that local tests are what we expect for this assignment, so that you can solely focus on concurrency aspects.

Implement two locking protocols for *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*

As in Assignment 1, the interface methods in the *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore* classes are implemented (except for `rateBooks`, `getTopRatedBooks`, and `getBooksInDemand`, which you may ignore). For this assignment, you should make the implemented methods thread-safe while maintaining the method semantics. Recall that as described above, you must ensure *before-or-after atomicity* of methods in the bookstore. You should aim at increasing the concurrency in the processing of methods in the bookstore by guaranteeing:

- Read operations for the same object do not block each other, i.e., methods which are just reading should not block each other.
- A *locking* protocol is used as a concurrency control mechanism.
- The assumption that all-or-nothing semantics are defined with respect to application-level logic errors for each method (checked exceptions), and not for unexpected errors coming from the runtime system (runtime exceptions) is maintained.

(In particular, you may assume for simplicity that runtime exceptions simply crash the service, but you must avoid *introducing* runtime exceptions due to concurrency.)

In the *SingleLockConcurrentCertainBookStore*, you should employ a single read-write lock on the entire database, acquiring the lock in shared mode in read-only operations and in exclusive mode in operations including any writes. In the *TwoLevelLockingConcurrentCertainBookStore*, you should implement a slightly more sophisticated two-level strategy. Employ one read-write lock at the level of the database *to simulate* intention locks. This top-level lock should be acquired on exclusive mode when operations perform inserts or deletes, and in intention (in this case, simulated by read) mode in all other operations. At the bottom level, there should be one read-write lock for each item (or optionally group of items, if you wish to have control over granularity) in the database. These locks must be acquired in shared mode in operations that only read the affected items and in exclusive mode in operations that end up writing the affected items.

In your solutions for each of the above, you should explain the design and implementation of your locking protocol. Focus first on explaining *how* your locking protocol works,

documenting any additional data structures you needed to create, the types of locks employed, and the policies you used to acquire and release locks. In the discussion questions below, you will be asked to complement this explanation by providing an argument for *why* your locking protocol achieves atomicity of operations. Your locking protocol will be evaluated on correctness, and the degree of concurrency achieved. We will also pay particular attention to the quality of the argument you provide below.

Tip: It is often helpful to start with the simplest locking protocol that achieves the requirements above that you can think, even if it does not provide for much concurrency, e.g., by implementing the *SingleLockConcurrentCertainBookStore*. After you have that working, make any improvements in a second pass over your implementation, e.g., by implementing the *TwoLevelLockingConcurrentCertainBookStore*. That way, you will get experience with the issues in an implementation you can easily reason about, and also have something working (hopefully) quickly. If the unexpected occurs, this strategy also gives you at least a partial solution for the assignment that you can hand in on time. *You are encouraged to use the java.util.concurrent.* helper classes. You might find java.util.concurrent.locks.ReadWriteLock particularly useful if you need shared and exclusive locks.*

Questions for Discussion on the Concurrent Implementations of Bookstore

In addition to the implementations above, reflect about and provide answers to the following questions in your solution text:

1. Provide a short description of your implementation and tests, in particular focusing on:

   (a) What strategy have you followed in your implementation to achieve before-or-after atomicity for each of *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*?

   (b) How did you test for correctness of your concurrent implementation? In particular, what strategies did you use in your tests to verify that anomalies do not occur (e.g., dirty reads or dirty writes)?

   (c) Did you have to consider different testing strategies for *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*? Since these classes need to provide the same semantics, would the use of different strategies be a violation of modularity?

2. Is your locking protocol correct? Why? Argue for the correctness of your protocol by equivalence to a variant of 2PL. Remember that even 2PL is vulnerable when guaranteeing the atomicity of predicate reads, so you must also include an argument for why these reads work in your scheme. NOTE: You must include arguments here for *each* of *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*.

3. Can your locking protocol lead to deadlocks? Explain why or why not. NOTE: You must include arguments here for *each* of *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*.

4. Is/are there any scalability bottleneck/s with respect to the number of clients in the bookstore after your implementation? If so, where is/are the bottleneck/s and why? If not, why can we infinitely scale the number of clients accessing this service? Also, discuss how scalability differs between the two variants *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*.

5. Discuss the overhead being paid in the locking protocol in the implementation vs. the degree of concurrency you expect your protocol to achieve. Contrast how this trade-off differs between the two variants *SingleLockConcurrentCertainBookStore* and *TwoLevelLockingConcurrentCertainBookStore*.