Programming Assignment 2

Author
Wanjing Hu / fng685@alumni.ku.dk

December 1, 2024

1 Question 1

Provide a short description of your implementation and tests, focusing on:

(a) What strategy have you followed in your implementation to achieve before-or-after atomicity for each of SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore? (1-2 paragraphs)

(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)? (1-2 paragraphs)

(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 sentences)

Answer

(a) The before-and-after atomicity means the methods in the two bookStores cannot have effects that would only arise by interleaving of parts of transactions, and in this case we mainly focus on the operations around *bookMap*, which is a *HashMap* mapping ISBN to the BookStoreBook objects. For **SingleLockConcurrentCertainBookStore**, we use one *java.util.concurrent.locks.ReentrantReadWriteLock* as the lock for the bookMap, and it has both read lock and the write lock provided. Different from what is recommended in the assignment document, the reentrant lock would let a same thread to get the lock if the thread is the last one to get the lock. For read locks we have a small granularity on *getBooks()*, *validate()* and *getEditorPicks()*, and multiple threads can simultaneously hold the read lock the read only operations could operate concurrently. For write locks they are applied on *addBooks()*, *butBooks()*, and *removeAllBooks()*. This lock is exclusive and ensures that only one thread can modify the *bookMap* at a time. The read locks are acquired when the method only needs to ensure data consistency without modifying shared data, and the write locks are acquired when data modifications occur, preventing concurrent reads or writes during the critical section. Both locks are released in finally blocks to guarantee release even if exceptions occur, ensuring the system doesn't enter a deadlocked state. This protocol ensures **before-or-after atomicity** by isolating write operations and serializing them relative to each other and concurrent reads. Readers proceed without interference unless a write is underway, maximizing concurrency. However, the design prioritizes consistency over write throughput, which is a trade-off to achieve high concurrency for read-dominant workloads while maintaining correctness.

For **TwoLevelLockingConcurrentCertainBookStore**, we have a two-level locking mechanism using global locks and row-level locks, which is the top level and the bottom level in the assignment document, and for a better modularity we have wrapped them in to one class *LockManager*. The protocol enforces 4 lock acquisition policies: Global Read Lock (shared): Acquired when reading all books to prevent any concurrent writes (e.g., getBooks() method). Global Write Lock (exclusive): Acquired for operations that modify the entire collection (e.g., removeAllBooks()). Row-Level Read Lock (shared): Acquired on specific rows for read validation or checking book existence (e.g., updateEditorPicks()). Row-Level Write Lock (exclusive): Acquired when modifying individual book data (e.g., updating the number of copies in addCopies()). The protocol employs two types of locks: shared (read) locks and exclusive (write) locks. Global locks are used for operations that affect the entire collection of books (e.g., adding or removing all books), while row-level locks are used for operations on individual books identified by their ISBNs. Also we only release the locks that are previously guaranteed, so that no extract lock is released. The **before-or-after atomicity** is ensured for any two conflicting operations are either executed in sequence or one completely precedes the other, providing both correctness and concurrency. Shared locks allow multiple read operations concurrently, enhancing parallelism, while exclusive locks prevent race conditions during writes.

(b) We have 4 test cases to validate the correctness of concurrency, including designing stress tests that simulate high levels of concurrency while checking for consistency, data integrity, and absence of deadlocks. Specifically, tests like *testConcurrentBuyAndAddCopies* and *testConcurrentSnapshot* demonstrated **key concurrency scenarios** by having one thread repeatedly update the number of copies (simulating a producer) while another either bought copies or observed the state (simulating a consumer). The tests ensured correctness by checking that the final inventory matched the expected total after all operations, thereby ruling out anomalies like dirty reads or writes.

For **dead lock detection** we have *testPotentialDeadlock* test intentionally creating scenarios where threads acquired locks in different orders. This approach ensured that potential circular waits did not occur, as both threads attempted to buy books in conflicting sequences. We also have a **stress test** with adjustable QPS in *testConcurrentAddAndRemoveBooksWithQPS*, and we can assign the duration and QPS to the test so that the test is long enough. There are still improvement that we can also have a timeout design so that we can find a max QPS based on the timeout rate.

(c) Yes we have different tests, and the *testPotentialDeadlock* is specially for the *TwoLevelLockingConcurrentCertainBookStore* because it has different bottom level locks on different ISBN, while the Single version only have one lock on the whole bookMap. The usage will not be violating the modularity because the difference is hide in the implement of the interface.

2 Question 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: Include arguments for each of SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore. (2 paragraphs—1 for each of the two classes)

Answer For SingleLockConcurrentCertainBookStore, a coarse grained ReentrantLock is used to guarantee that only one thread can access or modify the shared state of bookMap at any given time. In the 2PL case, the lock is acquired at the beginning of any operation (growing phase) and released only after the operation completes (shrinking phase). As a result, no conflicting operations (such as simultaneous reads and writes) can interleave, preventing anomalies like dirty reads, lost updates, or inconsistent states. Also, the coarse-grained nature of the lock addresses predicate reads (e.g., using validate() and validateISBNInStock() to checking available stock before buying) by ensuring no other operations can modify the state during the check and subsequent action. Although it sacrifices concurrency and performance due to serializing all operations, it guarantees atomicity and consistency, making it equivalent to a strict form of 2PL where predicate reads are effectively serialized.

For TwoLevelLockingConcurrentCertainBookStore, we have a top level lock(GlobalLock) for adding or removing or reading all the books, and a bottom level lock for adding or removing or reading one ISBN of book. This is a variant of 2PL where there are also individual lock cases, leading to more locks. In the 2PL case, the predicate reads are handled correctly by acquiring all necessary locks before evaluating a predicate and performing the operation atomically. For example, when we call buyBooks, the locks the relevant book objects before checking availability and decrementing stock. The shrinking phase occurs only when the operation completes, ensuring consistency.

3 Question 3

Can your locking protocol lead to deadlocks? Explain why or why not. Note: Include arguments for each of SingleLockConcurrentCertainBookStore and TwoLevelLockingConcurrentCertainBookStore. (2 paragraphs—1 for each of the two classes)

Answer For SingleLockConcurrentCertainBookStore, there is no deadlocks because our implementation uses a single coarse-grained lock to synchronize all operations on the bookstore. And for the exclusive write lock, only one thread can acquire the lock at a time, so there is no possibility of circular waiting, and all the actions are serialized.

For TwoLevelLockingConcurrentCertainBookStore, there IS a potential of dead locks. There will be deadlocks if the threads acquire locks in different orders, leading to circular waiting. We need to specify the lock acquiring order to prevent the dead lock, for example always give out the lock in the same order as the ISBN increasing order.

4 Question 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. (1-2 paragraphs)

Answer For SingleLockConcurrentCertainBookStore, there is a scalability bottleneck, and the scalability bottlenecks lies in the the single coarse grained lock. As the number of clients increases, the contention for the single lock will intensify, and there will be blocking threads waiting for the lock to release, and the throughput will reach a limit.

For TwoLevelLockingConcurrentCertainBookStore, there is a better scalability, but there will still be a bottle neck, because there will be more locks on different ISBN and there are less possibility on two thread queuing for a same lock. However its scalability is still bounded by how frequently locks on the same resources are contended, for example when there are a bunch of queries on a same ISBN of book, the bottle neck will be the same as the SingleLockConcurrentCertainBookStore.

5 Question 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. (1 paragraph)

Answer For the SingleLock, the overhead, meaning the extra computing cost, is minimal since there is only one lock acquiring and releasing for all the operations, and the lock management will not do much. However this simple strategy will limit the concurrency as all the threads are serialized and there will be no parallel write operations. For the TwoLevel, there will be greater overhead on lock maintaining because the bottom level locks are stored in the HashMap, and for the numbers of locks we need to track the lock acquisition order, and also do the dead lock prevention strategies. The complexity on overhead trades off a higher level of concurrency, and for different ISBN of books there can be concurrent read and write, which is a better balance.