ACS Assignment2

1 Serializability & Locking

Scheduel 1

$$\begin{tikzcd}
T1 \arrow[rr] & & T2 \arrow[dl] \\
& T3 \arrow[ul] &
\end{tikzcd}$$

Scheduel 2

$$\begin{tikzcd}
T1 \arrow[rr] & & T2 \\
& T3 \arrow[ur] &
\end{tikzcd}$$

To be conflict serializable the graph must be acyclic. Knowing that we conclude that only the second scheduel is conflict serializable.

Since 2PL ensures that the graph is acyclic, only scheduel 2 could have been generated in that way:

T1:	S(x) R(x)			X(y)W(y)C	
T2:			S(z)R(z)		X(x)W(x)X(y)W(y)C
T3:		X(z)W(z)C			

2 Optimistic Concurency Control

Before answering this question there are 3 validation tests to keep in mind:

Test 1:
For all i and j such that $T_i < T_j$ check that T_i completes before T_j begins.

Test 2:
For all i and j such that $T_i < T_j$ check that:
T_i completes before T_j begins its write phase and
WriteSet(T_i) $\cap$ ReadSet(T_j) is empty

Test 3:
For all i and j such that $T_i < T_j$ check that:
T_i completes read phase before T_j does and
WriteSet(T_i) $\cap$ ReadSet(T_j) is empty and
WriteSet(T_i) $\cap$ WriteSet(T_j) is empty

Scenario 1
T_3 is not allowed to commit because it fails on following tests:
Test 1, since $T_2 < T_3$, T_2 should be completed before T_3 but that does not happen. Offending objects in this case is the WS of T_2 since it is not completed by the start of T_3.
Test 2, even though the first part occurs (since T_2 completes before T_3 begins its Write phase) the WS(T_2) $\cap$ WS(T_3) is not empty. The offending job in here is 4 since it is the intersection between the two sets.
Test 3, fails in the same step as test 2 so it has the same offending job.

Scenario 2
T_3 is allowed to commit because it passed the follwing tests:
Test 1, T_1 completes before T_3 even starts.
Test 3, T_2 completes the read phase before T_3 and WS(T_22) $\cap$ RS(T_3) is empty and so is WS(T_2) $\cap$ WS(T_3).

Scenario 3
T_3 is allowed to commit because it passes the follwoing tests:
Test 2, both T_1 and T_2 completes before the read phase of T_3 and
WS(T_2) $\cap$ RS(T_3) is empty and so is WS(T_1) $\cap$ RS(T_3).
Test 3, both T_1 and T_2 completes before the read phase of T_3, Test 2 passed and WS(T_1) $\cap$ WS(T_3) and WS(T_1) $\cap$ WS(T_3) are empty.

3 Implementation

3.1 Tests

We added two more tests:
test3_concurency: stockmanager1 gets a book and updates it as !isEditorPick and stockmanager2 gets the same book and updates it as !isEditPick. The test succeeds if the snapshot before the two transactions of the book is the same as the snapshot of the book after the two transactions.

test4_concurency: we count all of the books before the test, then we add a new book (isbn); C1 adds books while C2 removes the same books; after the transactions we count the total number of the books in the snapshot and the tests succeeds if the numbers coincide. This tests help us ensure the atomicity of our program.

4 Questions for Discussion on the Concurrent Implementation of Bookstore

4.1

A) We used a readwritelock so we could achieve before-or-after atomicity this way each time there was a writing request to one of our methods on the server the server would give him a lock and unlock it in the end this way we could make sure only one user would write at the time. To do this we used trylocks, this way only if the lock is not being used it would acquire it. For read locks the same implementation was done, a read lock is a shared lock that should allow reentrant procedure, this way even if there is different read operations to the same object it should work fine.
The trylocks also work as a semaphore, this way it achieves the atomicity of lock/unlock, it also helps to simplify the implementation of lock/unlock requests instead of doing a queue of requests and check if they are empty each time and check if the object has currently an exclusive lock, the ReadWriteLock library does that instantly and with the trylock method we check if those conditions are met automatically and acquire it if so.

B) On all tests we forced reads and writes in both clients so they would make a dirty read and try to dirty write again. Asserting the results in the end made sure they would verify the same results and this way assuring that

they are not making those dirty operations.
More specifically on test 1 one adds books and another buys books, if one reads while the other did not add any books than he would not be able to buy it, or if they both do it on the same time then both would fail and the result would be different For test 2 tests only for dirty reads while we add copies the other keeps reading while doing this we check if he is reading correct values.
For test 3, we test we check for dirty writes on editor picks while one client tries to update the editor pick while another updates it to false.
For test 4 we have one client deleting books while the other adds it and check if we have the same number of books in the end this way we can check for dirty writes and reads while removing books

4.2

Yes, we implemented Strict 2PL, using a try catch we could achieve the first rule such that each time a transaction wanted to write it asked for a lock (Write lock), and in the end of each try catch we would do a finally to unlock the lock, this way even if the method failed it would still release the lock. To achieve the read atomicity we used the read lock (shared locks) they work as semaphores but they are reentrant locks this way even if there are more read operations to the same object they would still be allowed while maintaining the atomicity.

4.3

Yes it can, because each time it tries to acquire a lock it times out if it can not acquire for 1 second. For this we used the try lock method with timeout arguments.

4.4

The single server in the architecture is the bottleneck. Also the DB can be viewed as a bottleneck: it takes time to do writes and reads especially with the locking mechanism. Even if there are 10 clients and they all request the same resource if one does a write, all must wait for it to finish after they can do a write.

4.5

The overhead being paid seems to be high, as the resource access is being denied to other clients while there is a write lock set. Even if we have multiple threads they are still blocked untill the write is done. But at least we assure atomicity.