Principles of Computer Systems Design
Assignment 2

Tudor Dragan
Gabriel Carp

December 2, 2014

Question 1: Serializability & Locking

Conflict-serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

A schedule is conflict-serializable if and only if it's precedence graph has no cycles. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions.

Schedule 1

Schedule 1 *is not* conflict serializable because the graph is *cyclic*:

- T_1 - T_2: read-write conflict on X.
- T_2 - T_3: write-read conflict on Z.
- T_3 - T_1: read-write conflict on Y.

Schedule 2

Schedule 2 *is* conflict serializable because the precedence graph is *acyclic*:

- T_1 - T_2: X is accessed by T_2 after T_1 has committed.
- T_2: Y is only accessed by T_2.
- T_3 - T_2: Z exclusively locked by T_3 is released prior to T_2 acquiring a shared lock.

Question 2: Optimistic Concurrency Control

Scenario 1

Because T_1 finishes before T_3 starts, the 1^{st} condition holds. We have to check that the 2^{nd} condition holds for T_2 and T_3, but because T_2 writes the object that T_3 reads from, this condition does not hold. Therefore T_3 has to *rollback*.

Scenario 2

The 2^{nd} validation condition does not hold for T_1 and T_3 because T_1 writes to object 3 and T_3 reads from it. Therefore T_3 has to rollback. We can observe that the 3^{rd} holds for T_2, because T_3 does not access object 8.

Scenario 3

The only object that T_1 writes to is object 4 and because T_3 does not read or write to that object, the 2^{nd} validation condition holds. Then we check is the 2^{nd} condition holds for T_2 and T_3. T_2 only writes to object 6 and T_3 does not access that object in any way, therefore T_3 can commit.

Programming Task

For the implementation of the locking protocol used in the ConcurrentCertainBookStore we used the *ReentrantReadWriteLock* class available in the *java.util.concurrent.locks* package.

A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities. A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.

The ReentrantReadWriteLock allows both readers and writers to reacquire read or write locks in the style of a ReentrantLock. Non-reentrant readers are not allowed until all write locks held by the writing thread have been released. Additionally, a writer can acquire the read lock, but not vice-versa. Among other applications, reentrancy can be useful when write locks are held during calls or callbacks to methods that perform reads under read locks. If a reader tries to acquire the write lock it will never succeed.

We implemented the strict two-phase locking (S2PL) by requiring a write lock when modifying the data at the start of the method. Some methods only require read locks such as getBooks and getBooksInDemand.

Lock type	**read-lock**	**write-lock**
read-lock		**X**
write-lock	**X**	**X**

Table 1: Lock compatibility table

Discussion on the Concurrent Implementation of Bookstore

Is your locking protocol correct? Why? Argue for the correctness of your protocol by equivalence to a variant of 2PL.

Our protocol is implemented by using the conservative S2PL by acquiring a lock at the beginning of each method and releasing it before the method returns.

Can your locking protocol lead to deadlocks? Explain why or why not.

The deadlock cannot occur because we acquire the locks right at the start of the methods, transforming every call into an *atomic* one. This approach impacts the concurrency but ultimately assures proper synchronization.