Assignment 2

Nicolai Jørgensen and Yiran Zhang

December 3, 2017

1 Question 1

1. the precedence graph for each schedule are showing below , these two schedules are conflict- serializable, because their the precedence graph are acyclic.

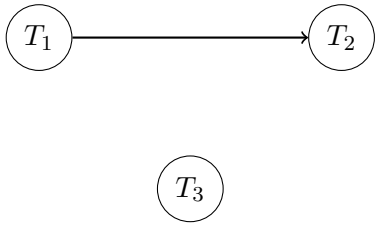

Figure 1: Precedence graph for schedule 1

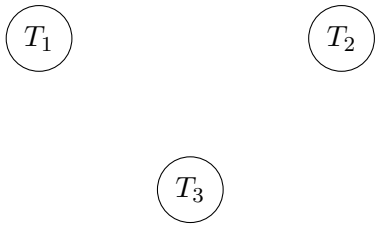

Figure 2: Precedence graph for schedule 2

2. The **schedule 1** couldn't have been generated by a scheduler using strict two-phase locking (strict 2PL), because there is a read lock in X in T_1, and then T_2 want to set a write lock on X, it must wait for until T_1 release the read lock.

   The **schedule 2** could have been generated by a scheduler using strict two-phase locking (strict 2PL), because there

2 Question 2

1. T3 must be rolled back.

   - test1 fails, because T2 not completes before T3 starts.
   - test2 fails, because T2 completes before T3 begins with its write phase. but the WriteSet(T2) ∩ ReadSet(T3) is not empty, and the offending object is 4.
   - test3 fails, no one meets the conditions of test3

2. T3 must be rolled back.

   - test1 fails, because no one meets the conditions of test1
   - test2 fails, because T1 completes before T3 begins with its write phase. but the WriteSet(T1) ∩ ReadSet(T3) is not empty, the offending object is 3.
   - test3 fails, because T1 doesn't meets the conditions of test3.

Nicolai Jørgensen
Yiran Zhang

3. T3 is allowed to commit, test2 is necessary to reach the conclusion. T1 completes before T3 begins with its write phase, and the WriteSet(T1) ∩ ReadSet(T3) is empty. T2 completes before T3 begins with its write phase, and the WriteSet(T2) ∩ ReadSet(T3) is empty.

3 Question 3

1. a)
   - For *SingleLockConcurrentCertainBookStore*, first we 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 each newly created transaction i must, before reading or writing any data, waiting until the preceding transaction has either committed or aborted. The scheme forces all transactions to execute in the serial order. Since that order is a possible serial order of the various transaction, by definition simple serialization will produce transactions that are serialized and thus are correct before-or- after actions.
   - For *TwoLevelLockingConcurrentCertainBookStore*, at top-level the **takeGlobalLock** is acquired on exclusive mode when we perform **addBooks** and **removeBooks**, **globalReadLock** is in all the other operations. This scheme is also the scheme on *SingleLockConcurrentCertainBookStore*, thus at top level we achieve correct before-or-after atomicity.
     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. This solution can also forces all the transaction at bottom level to execute in serial order.
     At each level the transactions are executed in serial order, so the *TwoLevelLockingConcurrentCertainBookStore* has the correct before-or- after actions.

   b) **Test 1** tests that concurrently adding copies and buying books maintain consistent state. We initiate the number of copies of the default book. Two threads C1 and C2 run concurrently and iterate buyBooks and addCopies 10000 times respectively. We finally check whether the finally value is equivalent to the initial value. When we take write or read lock on the data items the final value is equal to the initial value, otherwise the final value varies. The anomalies occurs if the final value varies, in this case, buyBooks happens during the addCopies or the addCopies happens during the buyBooks, and this will lead to dirty reads or dirty writes and make the final value to vary.

   c) We don't have to consider different testing. Because the scheme in the *SingleLockConcurrentCertainBookStore*, is the same as the scheme in the top-level in the *TwoLevelLockingConcurrentCertainBookStore*

   The use of different strategies would not be a violation of modularity.

2.

3. • For *SingleLockConcurrentCertainBookStore*, our locking protocol will not lead to deadlocks. We first take **writeLock** on data items that are modified and take

Nicolai Jørgensen
Yiran Zhang

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. Therefore we need to know when to release locks, which may leads to deadlocks.

Nicolai Jørgensen
Yiran Zhang