Assignment 2

Nicolai Jørgensen and Yiran Zhang

December 4, 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.

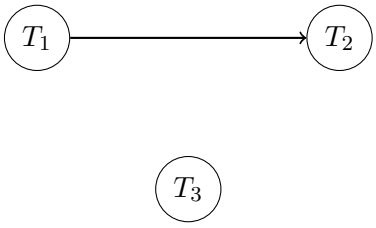

Figure 1: Precedence graph for schedule 1

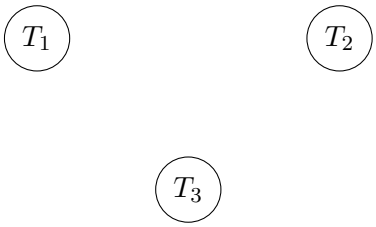

Figure 2: Precedence graph for schedule 2

2. The **schedule 1** couldn't have been generated by a scheduler using strict two-phase locking, because T_1 takes a read-lock on X and does not release it before T_2 wants to take a write-lock on X. Therefore, under strict 2PL it would have to wait for T_1 to release its locks.

   The **schedule 2** could have been generated by a scheduler using strict two-phase locking (strict 2PL), because at no point does a thread request a lock on an already locked resource. The schedule with explicit lock-taking looks like this:

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

   Keep in mind that `C` releases all locks held by the thread.

2 Question 2

We recall the three validation conditions from the text:

1. T_i completes (all three phases) before T_j begins.
2. T_i completes before T_j starts its Write phase, and T_i does not write any database object read by T_j

Nicolai Jørgensen
Yiran Zhang

3. T_i completes its Read phase before T_j completes its Read phase, and T_i does not write any database object that is either read or written by T_j

So for each of the different scenarios we have:

1. T3 must be rolled back. T1 passes the first condition, but for T2:
   - Condition 1 fails, because T2 does not complete before T3 starts.
   - Condition 2 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.
   - Condition 3 fails, none of the conditions are met.
2. T3 must be rolled back. For T1:
   - Condition 1 fails, because T1 does not complete before T3 begins.
   - Condition 2 fails. T1 does complete before T3 begins with its write phase, but the WriteSet(T1) ∩ ReadSet(T3) is not empty, the offending object is 3.
   - Condition 3 fails, because T1 doesn't meet the initial condition.
3. T3 is allowed to commit and condition 2 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) On *SingleLockConcurrentCertainBookStore* we take a global lock at the start of our function calls, thus at top level we achieve correct before-or-after atomicity.

   For *TwoLevelLockingConcurrentCertainBookStore*, at top-level the **takeGlobalLock** is acquired on exclusive mode when we perform **addBooks** and **removeBooks**. **globalReadLock** is taken whenever a local lock is taken. This ensures that a global exclusive lock can't be taken before all local locks are released.

   At the bottom level, there is one read-write lock for each book in the bookstore. We use **takeLocalWriteLock** on data items that are modified and **takeLocalReadLock** on data items that are only read. We do this during the execution of the transaction as needed. We release locks on objects at the end of the transaction or in case of some exception being thrown.

   At each level the transactions are executed as if in serial order, so the *TwoLevelLockingConcurrentCertainBookStore* has the correct before-or-after atomicity.

   b) We have used two general strategies for testing the concurrency code. The first strategy is to have two different threads doing opposite operations. At the end, we test that the environment is unchanged. This can for example be seen in `testBuyAndAddConcurrent` and `testUpdateEditorPicks`.

   The other strategy we use is to have some number of threads repeatedly doing an action and a testing thread repeatedly reads the updates and checks consistency. This is to ensure that threads can only observe legal states. Such tests are for example `testBuyAndAddConsistency`, `testRatedBooks` and `testUpdateEditorPicks`. The latter of the two uses two threads doing updates and one checking consistency.

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

The use of different strategies would not be a violation of modularity, but it might be helpful to still consider it. Different implementations have different edge cases where there might be bugs. Taking this into consideration when writing tests is a good idea.

2. The *SingleLockConcurrentCertainBookStore* implementation is not only equivalent with strict 2PL, it is also equivalent with conservative 2PL. This is because all functions immediately take all locks (the only lock) they need and release it at the very end of the call.

   For *TwoLevelLockingConcurrentCertainBookStore*, things are a little more complicated. In 2PL, there is a growing phase and a shrinking phase. Locks are taken in the growing phase and released in the shrinking phase. Strict 2PL also requires that all exclusive locks are released at the very end of the transaction, when either commiting or aborting.

   The growing phase is encoded in our implementation simply as part of the function body. The thread acquires locks on objects as it needs them. The shrinking is embedded in the `finally` blocks that wrap the function body. Here, all locks taken by the transaction are released. Since this includes all exclusive locks, the implementation is strict.

   Since we implement a shrinking and growing phase and all exclusive locks are released only upon commit or abort, our implementation is equivalent to strict 2PL.

3. - For *SingleLockConcurrentCertainBookStore*, our locking protocol will not lead to deadlocks. We first 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 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.

4. We have no thrashing handling in the implementation, so a large amount of requests could simply bog down our store in overhead computations. This is especially true for *SingleLockConcurrentCertainBookStore* because of the low amount of concurrency in the implementation.

5. The concurrency for *SingleLockConcurrentCertainBookStore* is limited by the number of operations that modify state, since these need exclusive locks and put everything else on hold. In a bookstore, we can expect there to be less requests for writing than reading. Since the overhead is pretty small, this tradeoff is probably good.

With the two-level locking scheme we get to pay quite a bit more overhead than with the single-lock: Our locks live in a map and take $O(n)$ time to find every time we need one. If our bookstore has an enormous amount of books and a large amount of requests, this might actually have impact on performance.

On the other hand, we achieve the benefit that a write operation no longer locks the rest of the bookstore from functioning, which is an advantage if the number of requests is large.

Nicolai Jørgensen
Yiran Zhang