Homework 4, Q1 Solution:

T1: R(X), W(X), R(Y), W(Y)

Assumption: R(X) and W(X) if happening immediately one after another within the same transaction are considered in one SQL statement. With this assumption, DBMS sees both actions and can immediately get an exclusive lock.

The solution that does not use the above assumption also works.

An X lock is used when a transaction wants to write an object. A read action on the object is allowed when a transaction holds an exclusive lock on the object. S lock is used when a transaction wants to only read from the object.

For each schedule, the order of actions in each transaction must not be changed, for instance, T1 cannot do W(X) before R(X) as this changes the logic of the program.

Same assumptions for the object Y.

There are multiple correct solutions

a. Example: T2:R(O), W(O)

As long as T2 reads or writes a different object except X and Y, there is no conflict. It is not necessary that Commit or Abort is shown in the given schedule.

Explanation: T2 does not have any conflict with T1 since T2 does read and write actions on objects that T1 does not read from or write to. To be in a conflict, the two transactions must perform actions on the same object and one of the actions must be a write action.

Note: If T2 has R(X) or R(Y), a conflict can occur since T1 has W(X) as well as W(Y).

b. Answer: Example: T3: R(X); W(X)

An example of a schedule with a write read conflict (dirty read):

T1:R(X), T1:W(X), T3:R(X), T3:W(X), T1:R(Y); T1:W(Y)

Here, T3 reads X made dirty by T1:W(X)

Note: A schedule that shows that T1 reads after T3 writes on the same object is also fine as long as the order of actions within the same transaction does not get changed.

Show how Strict 2PL prevents interference on the above schedule

- The solution below does not use the assumption that R(X) and W(X) that happen right after each other is within the same SQL statement.

  T1:S(X); T1:R(X); T1:X(X); T1:W(X); T3 wants to do R(X), a shared lock is requested for T3, but cannot be granted since T1 holds an exclusive lock on X. T3 is put in the queue for the lock on X. T1 proceeds to T1:S(Y); T1:R(Y); T1:X(Y); T1:W(Y); T1:Commit; at this point, all the locks held by T1 are released. DBMS picks the first transaction that is in the queue for these locks to run. T3:S(X); T3:R(X);

T3:X(X): T3:W(X); T3:S(Y); T3:R(Y); T3:X(Y); T3: W(Y); T3: Commit. Once T3 commits, all locks held by T3 are released.

- The solution below uses the assumption that R(X) and W(X) that happen right after each other is within the same SQL statement.

  T1:X(X); T1:R(X); T1:W(X); T3 attempts to R(X), since T3 also wants to do W(X), it asks for an exclusive lock (X-lock) on X, but cannot get it since T1 is currently holding the exclusive lock on X. T3 is blocked waiting. Only T1 can proceed with T1:X(Y); T1:R(Y): T1:W(Y);T1:Commit;

  After T1 commits, the X locks on X and Y held by T1 are released. The waiting T3 can get the X lock on X and continues with the rest of its actions.

  T3:X(X); T3:R(X); T3:W(X); T3:Commit;

  This schedule produces the same result on the database as running actions of T1 completely first before running the actions of T3.

  It is ok if students do not show the commit or abort for one of the transactions as long as the Strict 2PL protocol is correct for the other transaction and students show how the blocked transaction can proceed.

c. **An example of a transaction that can create a schedule** with a read write conflict (unrepeatable read):

T4: R(X); W(X)

A conflicting schedule example: T1: R(X); T4:R(X); T1:W(X); T4:W(X); T1:R(Y); T1:W(Y); T4:R(X); T4:W(X)

If T4 were to do R(X) again after T1:W(X) before T4:W(X), T4 would find that the value of X is different from the first time it reads X even though T4 does not change the value of X in its code. This conflict is therefore called “unrepeatable read.”

Show how Strict 2PL prevents interference on the above schedule

- The solution does not use the assumption that R(X) and W(X) that happen right after each other within the same transaction is within the same SQL statement so that DBMS asks for an exclusive lock right away.

  T1:S(X); T1:R(X); T4:S(X); T4:R(X); T1 wants to write X, but cannot upgrade its lock to an exclusive lock on X since T4 has a shared lock on X. T1 is blocked waiting for the lock held by T4. T4 wants to upgrade its lock to an exclusive lock on X in order to do W(X), but it cannot get it because T1 holds a shared lock on X. T4 is blocked waiting for T1. A deadlock occurs. A deadlock detection that runs periodically detects the deadlock. A deadlock resolution policy is used. One of the transaction is killed and all of its actions must be rolled back and locks that they hold are released. Let’s assume that T4 gets aborted and will be restarted at a later time. The shared lock on X that T4 holds is

released upon its abortion. T1 gets the exclusive lock on X and proceeds T1:X(X); T1:W(X); T1:S(Y); T1:R(Y); T1:W(Y); T1:Commit. It is possible that T4 may be restarted while T1 is still running.

- This solution uses the assumption that R(X) and W(X) that happen right after each other within the same transaction is within the same SQL statement so that DBMS asks for an exclusive lock right away.

  T1:X(X); T1:R(X); T4 attempts to R(X), but is blocked since T1 holds an exclusive lock on X. T1 proceeds. T1:W(X); T1:X(Y); T1:R(Y); T1:W(Y); T1:Commit; the locks held by T1 are released per Strict Two Phase Locking protocol; T4:X(X); T4:R(X); T4:W(X). This schedule produces the same result as running T1 actions first followed by running T4 actions.

  Note: It is ok to also write T1: Abort instead of T1: Commit. But we need to know that T1 either commit or abort before the lock can be released.