Assignment 2

Björn Sigurðsson
Leif Willerts
Marc-André Poulin

December 1, 2015

Exercises

Question 1: Serializability & Locking

Schedule 1

Schedule 1 is not conflict-serializable because the $W(Y)$ operation in thread **T1** conflicts with $R(Y)$ in **T3**, $R(Z)$ in **T3** conflicts with $W(Z)$ in **T2** and $W(X)$ in **T2** conflicts with $R(X)$ in **T1**. This schedule has a precedence cycle as can be seen clearly in the precedence graph. A schedule is conflict-serializable if and only if its precedence graph is acyclic.

This schedule could not have been generated by a scheduler using strict 2PL. **T1** gets a shared lock on X in the beginning and does not unlock until the end when it commits. **T2** Requests an exclusive lock on X but must wait until **T1** releases its shared lock first. According to the schedule **T2** commits ahead of **T1** but that cannot happen because **T2** must wait for **T1** to finish.

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

Figure 1: Schedule 1 with shared/exclusive lock operations in accordance to strict 2PL rules. Some of the locks contradict each other.

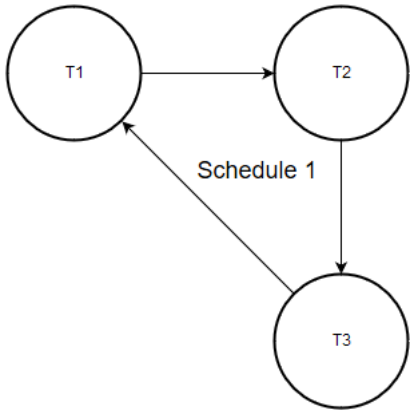

Figure 2: Precedence graph for schedule 1.

Schedule 2

Schedule 2 is conflict-serializable because its precedence graph is acyclic. Operations in **T2** conflict with operations in both **T1** and **T3** but operations in those threads do not conflict with anything so there is no cycle in the schedule.

This schedule could have been generated by a scheduler using strict 2PL. All threads release their locks before another thread tries to take hold of it. **T1** unlocks X and Y before **T2** locks them again. **T3** unlocks Z before **T2** locks it.

```
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
```

Figure 3: Schedule 2 with shared/exclusive lock operations in accordance to strict 2PL rules.

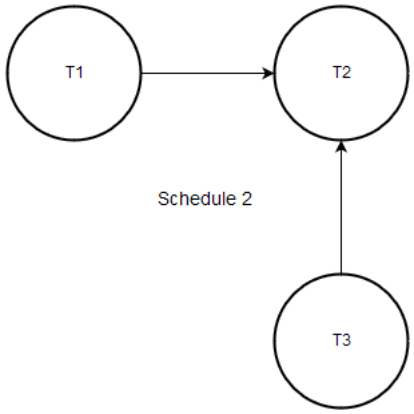

Figure 4: Precedence graph for schedule 2.

Question 2: Optimistic Concurrency Control

Scenario 1

T1 completes before **T3** starts so test 1 ensures that no conflict occurred between **T1** and **T3**.

T2 completes before **T3** begins its write phase. That means test 1 does not necessarily hold. We check if test 2 holds but see that it does not because 4 is in **WS(T2)** and **RS(T3)**. **T3** might be reading dirty data from **T2**. If **T2** were to write something to 4 and abort after **T3** reads it then **T3** will have read data in 4 that should never have existed. Therefore **T3** must roll back.

Scenario 2

T1 completes before **T3** begins its write phase. Test 1 does not hold. Test 2 holds because the intersection of **WS(T1)** and **RS(T3)** is empty and guarantees that no conflict occurred between **T1** and **T3**.

T2 completes its read phase before **T3** does, that's all we know. For test 3 to hold, then, no element in **WS(T2)** can be in either **RS(T3)** or **WS(T3)**. **WS(T2)** contains only 8 which is in neither **RS(T3)** or **WS(T3)** so test 3 does in fact hold and **T3** is allowed to commit.

Scenario 3

T1 completes before **T3** begins its write phase. Test 2 holds because the intersection of **WS(T1)** and **RS(T3)** is empty.

T2 completes before **T3** begins its write phase. Test 2 holds for the same reason. **T3** is allowed to commit.

Programming Task

Questions for Discussion on the Concurrent Implementation of Bookstore

1. (a) We have a map of ReadWrite locks that map each book's ISBN to a lock. To access information about the book, a thread gets a read lock and to change the book it needs a write lock. The map that holds the books also has a ReadWrite lock so that when a thread needs to add or remove books it gets a write lock on the map itself. Whenever a thread needs to access information about a book or change a book's information the thread gets a read lock on the map holding the books. A thread cannot have a write lock on a book while another thread has a write lock on the map.

   We use a conservative strict locking protocol to prevent deadlocks and take advantage of the fact that we always know all the books we need to access at the beginning of each transaction so we can lock them all before we do any reads or writes. The locks make sure that while a thread has a write lock on a book no other thread can access that book. Multiple threads can have read locks at the same time. Atomicity is ensured because all locks are acquired before any reading/writing starts in the order of increasing ISBN and released in the same order.

   (b) We test the correctness of the concurrent implementation by having several different threads operate on the bookstore at the same time. We check both while the threads are running and after all transactions are completed that the database is consistent and the results of transactions is as expected.

2. The tests suggest that the locking protocol is correct. Tests were first run before implementing the locking protocol to see that they did indeed fail for multiple threads, then with the trivial solution of synchronizing everything and lastly with the locking protocol in place. They passed indeed and the time they took compared to the synchronized solution show that the locking protocol was actually an improvement.

   The locking protocol is conservative and strict and locks are always acquired and released in the same order. We use a strict locking so that even if a write transaction is aborted another thread cannot read something dirty because the aborting thread does not release anything until it is finished and nothing has failed.

3. No, the locking protocol cannot lead to deadlocks because it is conservative and acquired all locks in a predetermined order, the increasing order of ISBNs.

4. If there are many managers adding and removing books then we will have a huge bottleneck because that locks down the whole store and no customer can buy or even look at books. Many customers can buy different books at the same time but block each other when they buy the same book (shouldn't happen often in a big store). We could improve the performance by making the protocol non-strict but then we would have to take care of possible dirty reads with rollbacks.

	single-threaded	multithreaded
CertainBookStore	7.3s	impossible
ConcurrentCertainBookStore	18.2s	16.7s

Figure 5: Performance of several implementations of the book store in different environments.

So our implementation is scalable in the number of 'clients' as in buying customers, but not necessarily so well in the number of StockManager 'clients'.

5. The locking protocol introduces a significant overhead. The single-threaded bookstore is much faster in a single-threaded environment, that is when all the customers line up in a neat queue. The benefits of using multiple threads are by far not enough to let ConcurrentBookStore compete with CertainBookStore in our test (lots and lots of writes on 4 books). It would probably be faster on a system with more CPU cores than our test computer having 4 (that is then able to run more than 4 threads *actually* at the same time).