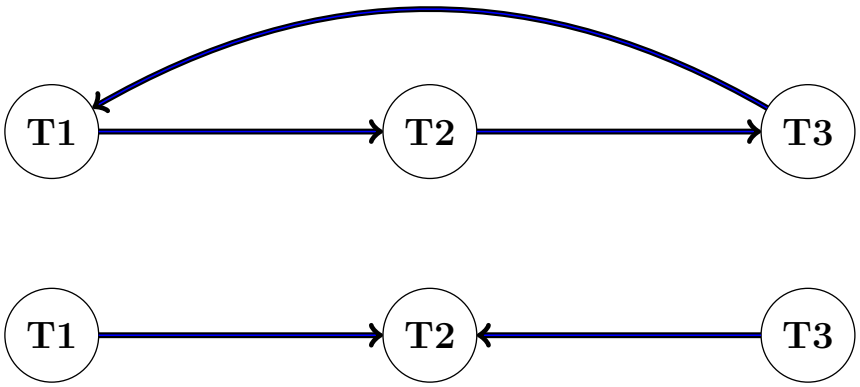
ACS Assignment 2

Arinbjörn Brandsson, Mads Rogild

December 2015

1 Question 1: Serializability & Locking

1.1 Precedence Graphs



The first is not conflict serializable, due to the fact that it contain cycles. The second is conflict serializable.

1.2 Two-phase locking

T1	X(X)	R(X)																X(Y)	W(Y)	C
T2			X(Z)	X(X)	W(Z)	W(X)	C													
T3								X(Z)	X(Y)	R(Z)	R(Y)	C								

T1	X(X)	R(X)										X(Y)	W(Y)	C						
T2						X(Z)	R(Z)	C							X(X)	X(Y)	W(X)	W(Y)		
T3			X(Z)	W(Z)	C															

2 Question 2: Optimistic Concurrency Control

For this part, we refer to page 91 of the course book, for the validation conditions.

Secnario 1:

T1 read 1,2,3 and write 3.
T2 read 2,3,4 and write 4,5.
T3 read 3,4,6 and write 3.

For this scenario T3 is not allowed to commit. There is no offending objects from T1 as it passes the first validation conflict. However it fails the second validation condition on T2, on the write object 4.

Scenario 2:

T1 read 1,2,3 and write 3.
T2 read 5,6,7 and write 8.
T3 read 3,4,5,6,7 and write 3.

For this scenario T3 is not allowed to commit. There is no offending objects from T2 as it passes the third validation conflict. However it fails the second val idation condition on T1, on the write object 3.

Scenario 3:

T1 read 2,3,4,5 and write 4.
T2 read 6,7,8 and write 6.
T3 read 2,3,5,7,8 and write 7,8.

For this scenario T3 is allowed to commit, in both T1 and T2 it passes the second validation condition.

3 Questions for Discussion on the Concurrent Implementation of Bookstore

3.1

3.1.1

The strategy used for our implementation of before-or-after atomicity is similar to how two-phase locking works. We have an exclusive lock, which we call writeLock, and a shared lock, called readLock, which keeps track of how many threads are currently reading bookMap. When getReadLock() is called, a check for whether writeLock is true is made, and if it is, it puts the thread in a spinlock until the exclusive lock is released. It then increments the readLock counter by one. When releaseReadLock() is called, it decrements the readLock counter.
When an exclusive lock is requested with getWriteLock, it adds the requesting thread to a thread array which keeps track of requested writes. It then waits in a spinlock until both the writeLock is released and it is the thread's turn. It then locks the writeLock, and waits for the readLock to be released. When the exclusive lock is to be released, it simply sets writeLock to false.
The reason spinlocks are used is because we assume that access to the bookMap variable is fast enough that it is faster for the thread to wait for a lock than to fully sleep.

3.1.2

We have made 4 test cases in total. Besides the one defined in the assignment, we have made one to test that the write lock puts further read requests on hold, and one to make sure we cannot read a value that has not fully been written yet. In the first of them many read requests is made and a single write request is made. If the write locks correctly the execution of the reads should stop and let the write execute properly, otherwise it will fail. To make sure we are able to test that the write is done fully before reading, 2 books is added simoultanously to the store. A similar approach has been used for the second test. Here a write is made if the locking works properly the amount of books will either be the same when read or 2 higher.

3.2

Our locking protocol is correct. It guarantees that deadlocks can not occur, write functions are protected from read locks, and read locks can happen concurrently. It is very similar to two-phase locking, as explained before. Our protocol protects against dirty reads, as as soon an exclusive lock is requested, all reads are blocked until the lock is released. This also means that a read request may have to wait for a while, as all write requests are handled first before read requests are allowed again.

3.3

Our protocol can not lead to deadlocks. This is because there are no dependencies on other variables to read/write to, so deadlocks can't occur because of two threads that each has an exclusive lock on a variable tries to get a lock on the others object. So the only way for program to deadlock is if a thread doesn't release a lock, however it has been ensured that if any standard exceptons takes place, then whichever locks a thread has control of gets released.

3.4

When many clients wishes to read, there are no bottlenecks, as the shared lock is made to be able to go concurrently. The bottleneck occurs when many write operations are to take place, as they are made to not only block, but also to take priority.

3.5

The overhead being paid is relatively small. The cost of implementing and running the locks are pretty small, and while the speed to write is the same as if we were to serialize the requests (since we have to wait in any case), the read operation, which allows for unlimited read requests, means that if a minimum of two clients were to request a read, the second client would receive a reply much faster than if we were to serialize the process, while the first client has lost minimal time.