Advanced Computer Systems
Assignment 2

Christian Grüner - htz154
Jesper Lundsgaard - qhk359

1st of December 2015

1 Serializability & Locking

1.1 Scheduler 1

$$T1 -> T2$$

$$T2 -> T3$$

$$T3 -> T1$$

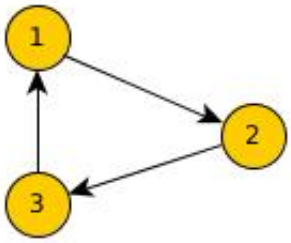

Figure 1: Precedence graph of schedule 1

Because we are having a cycle in our graph this schedule isn't conflict-serializable. This is also the same reason why we cannot generate the schedule using a strict 2P locking, as it would never create a cycling schedule.

1.2 Scheduler 2

$$T1 -> T2$$

$$T3 -> T2$$

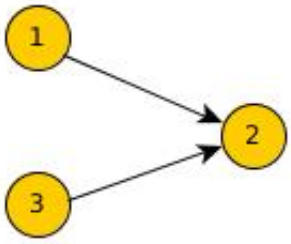

Figure 2: Precedence graph of schedule 2

Because we havn't got any cycles in our graph it means that the schedule is conflict-serializable.
To check whether it also can be generated using strict 2PL we need to check the order of how the transactions needs to be committed.
In this schedule the commits are in following order: $T3 -> T1 -> T2$.
When we look at our graph we can see that both T1 and T3 are supposed to come before T2. We can then choose whether T1 or T3 should come first.
Because we can order the transactions to be : $T3 -> T1 -> T2$, the schedule could be generated using 2P locking.

2 Optimistic Concurrency Control

2.1 Scenario 1

In this scenario there is a possible conflict in T2 write phase and T1 read phase. Because they are both writing and reading the same object *4*, this is a conflict and we will therefore roll back.

2.2 Scenario 2

In this scenario there is a possible conflict in T1 write phase and T3 read phase. But because they don't access the same objects this is not a conflict.
There is also a possible conflict on T2 write phase and T3 read phase. Because both T2 write phase and T3 read phase are accessing the same object this is a conflict and we will therefore roll back.

2.3 Scenario 3

In the last scenario we are looking at T1 write phase and T3 read phase, but this aren't accessing the same object.
We are also looking at T2 write phase and T3 read phase, but these phases aren't accessing the same object either. We will therefore commit the transactions
.

3 Concurrent implementation of Bookstore

3.1

Our approach to implementing concurrency in the bookstore is to create a *lockMap* which is a map that has the same ISBN numbers as keys, as the *bookMap* has. The values are then individual ReadWrite locks. We have followed this strategy to make sure that functions only lock the exact entries in the *bookMap* that they are using.

We have also made the decision that all the functions should lock down all the resources they need before executing and release the locks at once when it is done. Thus our implementation is conservative and strict. This also makes sure that transactions are fully isolated.

To make sure deadlocks do not happen we also sort the ISBN numbers before acquiring the locks. This way two threads can never wait for each other, since they acquire the locks in the same order and one will inevitably be ahead.

To test our implementation, we implemented the two tests from the assignment text and they both pass.

We then implemented a test that has two threads constantly updating a set of books to be editor picks and then not editor picks. Then a third thread constantly checks that at all times the books in the set are either all editor editor picks or none of them are. This tests the same consistency property as test 2, but also tests that the transactions are isolated from each other in addition to also checking that the concurrency is correctly implemented for all of the methods. It also has a test to check that it is possible to add and remove books without messing up the lock table as we foresaw this to be an issue if we did not implement it correctly

3.2

As mentioned previously, we believe our locking protocol to be correct. We have the tests to show it, and we have followed the conservative 2PL approach all the way.

3.3

As mentioned previously, we are using conservative and strict 2 phase locking. Because of this, we don't need to protect against deadlocks, as long as the ISBN numbers are being locked in order. If we were to make it completely strict, we would have too make sure that isolation failures wouldn't occur. By using conservative it is being taken care of.

3.4

Our *lockmap* scales as well as the *bookmap*, which means that memory-wise, we will not have scalability issues with our locking system until we also have scalability problems with our book storage. Concurrency also slows down the whole system, but there is no way to avoid this to some degree and we have strived to make the system able to use concurrency a lot.

3.5

Our implementation has a pretty big overhead since it needs to go through the *lockmap* to find locks and also needs to do additional tasks like sorting lists. This is the price that we have had to pay, but on the other hand we get a lot of concurrency out of it. We only lock the entries that we need which means a lot more operations can run concurrently.

Another solution could be only have one lock, this would get rid of the overhead, but it wouldn't be possible to buy a certain book and get another book at the same time.

We concluded that operations like *addBooks* that locks the whole bookstore, is not being used as often as *getBooks*, and *buyBooks*. This is why we chose to have locks for every book in the store.