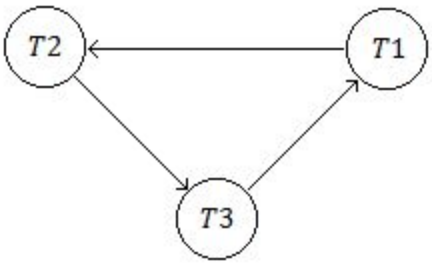
Advanced Computer Systems
Malte von Mehren (fns304)
Nicholas Indrehus (qwp654)

ACS Assignment 2

Question 1: Serializability & Locking

Schedule 1

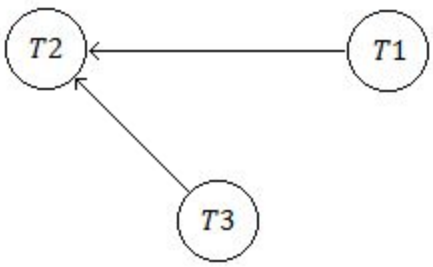
T1 must precede T2 because of the read to X in T1 and the write to X in T2.
T2 must precede T3 because of the write to Z in T2 and the read to Z in T3.
T3 must precede T1 because of the read to Y in T3 and the write to Y in T1.
This makes the precedence-graph for this schedule to be:



This schedule is not conflict-serializable, since the precedence-graph is cyclic.
The schedule could not have been generated by a scheduler using strict 2PL. The reason for this is that when T2 wants an exclusive lock on X before the write to X, T1 already has a shared lock on X because of the earlier read on X in T1.

Schedule 2

T1 must precede T2 because of the write on Y in T1 that occurs before the write on Y in T2.
T3 must precede T2 because of the write on Z in T3 that occurs before the read on Z in T2.
This makes the precedence-graph for this schedule to be:



This schedule is conflict-serializable, since the precedence-graph is acyclic, we just have to make sure that T2 is the last transaction, and this gives us these two possibilities: T1 T3 T2 and T3 T1 T2.

Yes, this schedule could have been generated by a scheduler using strict 2PL. Here we have injected read/write lock operations in accordance to strict 2PL rules:

```
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
```

Question 2: Optimistic Concurrency Control

Scenario 1

There can be no conflict between T1 and T3, since T1 is complete before T3 does anything. The only case to consider is whether there is a read/write-conflict between T2 and T3.

$$WS\{T2\} \cap RS\{T3\} = \{4\}$$

There is a possible conflict, so T3 must be rolled back.

Scenario 2

There can be a read/write-conflict between T1 and T3, and there can be both a read/write-conflict and write/write-conflict between T2 and T3. We check all the cases.

$$WS\{T1\} \cap RS\{T3\} = \{3\}$$
$$WS\{T2\} \cap RS\{T3\} = \{\}$$
$$WS\{T2\} \cap WS\{T3\} = \{\}$$

There is a possible read/write-conflict between T1 and T3, so we roll back.

Scenario 3

There can be read/write conflict between T1 and T2, and there can be a read/write-conflict between T2 and T3. We check the cases.

$$WS\{T1\} \cap RS\{T3\} = \{\}$$
$$WS\{T2\} \cap RS\{T3\} = \{\}$$

There are no possible conflicts, so T3 can commit its changes.

Questions on Code

1.

(a)

Before-or-after atomicity means that there cannot occur effects that would only arise from interleaving threads. This is achieved by our implementation since it isn't possible to have any interleaving methods that write to the *bookMap*, executing their instructions in parallel. Methods that read from the *bookMap* can execute concurrently, but since these only read from the *bookMap*, these methods does not affect the data in any way. Our implementation achieves before-or-after atomicity by using a Strict Conservative 2PL protocol (more to follow in question 2).

(b)

We have only run local tests for testing for successful concurrency, so we have no experience with how it works with RPC. The appearance of anomalies are not guaranteed, but rather based on probabilities proportional to the amount of potentially conflicting operations. Therefore all our tests are based on performing such operations enough times such that they will almost always lead to a conflict.

We have created four tests: *testThread1, testThread2, testThread3, testThread4*. For testing purposes we have created a flag *PROPERTY_RUN_CONCURRENTLY* in *BookStoreConstants*. If this flag is set to *true* all operations are performed using our concurrent implementation. Otherwise our concurrent implementation won't be used. Our tests are always successful when using our concurrent implementation. They all fail with a very high probability if it's not used.

testThread1

The method *buyBooks* reads the number of copies of one or more books, and then writes to the same fields. The methods *addBooks* reads and writes to the same fields. If a client buys books while a store manager adds books, we potentially get a read/write-conflict. The test checks if 5000 sequential *buyBooks* calls performed concurrently with 5000 sequential *addBooks* calls are all performed atomically. We should end up with 5000 copies of the book after all operations. Without our concurrent implementation we usually end up with a difference somewhere between 25 and 100 between the expected remaining amount of books and the observed amount of books. With our concurrent implementation we always have 5000 copies after the operations. This shows that our implementation ensures before-or-after-atomicity.

testThread2

The method *buyBooks* reads and writes to the object *bookMap*. The method *addBooks* adds writes to *bookMap*. The method *getBooks* reads from *bookMap*. This leads to potential read/write-conflicts. The methods *buyBooks* and *addBooks* can remove and add several books at a time. This test checks whether it's possible to observe the *bookMap* in the midst of such an operation. Without our concurrent implementation we do indeed observe such cases, which violates before-or-after-atomicity. With our concurrent implementation we never observe such cases.

testThread3

This test is similar to *testThread2*, except it deals with the methods *updateEditorPicks* and *getEditorPicks*. The method *updateEditorPicks* changes (writes) a property *EditorPick* of one or more books, and *getEditorPicks* retrieves books based on this property (reads). We check if it's possible to observe a situation where *updateEditorPicks* is only partially finished. This is possible without our concurrent implementation, but never the case using our implementation. Our implementation ensures before-or-after-atomicity.

testThread4

This test is similar to *testThread1*, but also checks that it's not possible to buy books when the store has no more copies as a result of a outdated read operation.

2.

Our locking protocol is correct, since we use a Strict Conservative 2PL protocol. We have implemented this with a single global lock for which an exclusive lock and a shared lock can be requested. This means that all methods inside the *ConcurrentCertainBookStore* class either requests a shared lock or an exclusive lock as the first thing in the method. Then, the method performs its actions, and then the lock is released just before returning. This is a Strict 2PL protocol since it acquires all the locks it needs at the start of the program, regardless of whether or not it needs it at this point. It is also a Conservative 2PL protocol, since it waits until just before returning to release its locks.

3.

Having a Strict Conservative 2PL protocol, we cannot have deadlocks, and we also cannot have cascading aborts. The only way that it is possible to have deadlocks in this scheme, is if the locking mechanism cannot detect that a thread first requests the read lock and after that the write lock (the thread will then wait on itself for the read lock to be released). The specific type of lock we are using cannot detect this, but this cannot happen in any of our implementations though, since each method only has one lock at a time, either the read lock or the write lock.

4.

Since all write requests are run sequentially in our implementation, we have a bottleneck here with respect to the number of clients in the bookstore, since some part of the clients will inevitably request methods that write to the *bookMap* (for instance when buying books). This bottleneck could perhaps have been alleviated by introducing more locks for specific parts of the bookMap such that some of these write requests could run concurrently.

5.

The degree of concurrency we expect depends on the number of reading method requests versus the number of writing method requests. If there is a very high number of requests to methods that only reads from the *bookMap*, and a very low number of requests to methods that writes from the *bookMap*, then we expect a high degree of concurrency overall. But if it is the other way around, we expect a very low degree of concurrency. Since we only have one lock, the overhead of our locking protocol isn't very big, so only a small degree of concurrency will be needed to make up for it. Realistically though, we don't expect much concurrency, since only a small number of writes will dominate the total time that the locks are locked, but still, the greater the percentage of requests we have that are reads, the more concurrency we will achieve, and only in the case with no reads will we have achieved no concurrency.