Principles of Computer Systems Design
Assignment 1

Andreas Bock
Asbjørn Thegler
Lasse Dessau

December 8, 2013

Question 1: Serializability & Locking

A schedule is conflict serializable if and only if it's precedence graph is acyclic. The graphs are computed by drawing arrows from operations in a transaction T_j that conflicts with an earlier operation in T_i.

Schedule 1

This schedule is not conflict serializable since the precedence graph is cyclic. In particular:

- T_1 reads X after which T_2 wants to write to X resulting in a read-write conflict.
- T_2 writes Z which T_3 then reads resulting in a write-read conflict.
- T_3 reads Y which is then written to by T_1 resulting in a read-write conflict.

Schedule 1 cannot possible be generated by strict 2PL since T_1 has a shared lock on X when T_2 writes to it. The precedence graph can be seen in figure 1.

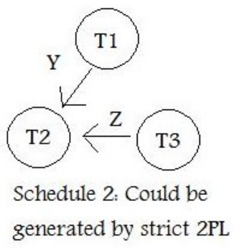


Figure 1: Precedence graph for schedule 1.

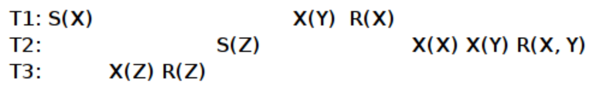
Schedule 2

This schedule is conflict serializable since the precedence graph is acyclic. In particular:

- X locked by T_1 is only accessed in T_2 after T_1 has committed and thus released the lock.

- Y is only accessed by T_2.
- Z exclusively locked by T_3 is released prior to T_2 acquiring a shared lock.

The locks are acquired and released in the order depicted in figure 2, and figure 3 shows the precedene graph.



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

Figure 2: Acquisition on shared/exclusive locks in schedule 2.

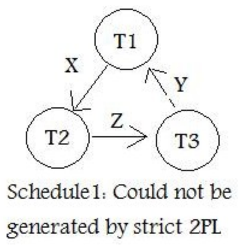


Figure 3: Precedence graph for schedule 2.

Question 2: Optimistic Concurrency Control

For this question, we refer to the course compendium page 91, where the *validation conditions* are described. We will refer to the conditions as 1, 2 and 3, respectfully, as listed in the material.

Scenario 1

T_1 finishes before T_3 even starts, so we see that*validation condition 1* holds for this transaction. We check if *validation condition 2* holds for T_2, but since T_2 writes to *object 4*, which T_3 reads from, it turns out that the condition does not hold. T_3 will have to be *rolled back*.

Scenario 2

We check if *validation condition 2* holds for T_1, but since T_1 writes to *object 3*, and T_3 reads from it, it turns out that the condition does not hold. T_3 will have to be *rolled back*. We were asked for all offending objects, so we will continue checking T_2. We check if *validation condition 3* holds for T_2, and since T3 does not access (read or write) object 8, which is the only object T2 writes to, the condition holds.

Scenario 3

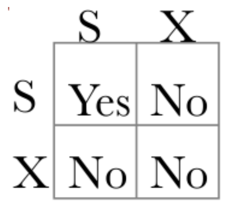
We check if *validation condition 2* holds for T_1, and since T_3 does not read from *object 4*, which is the only object T_1 writes to, the condition holds. We check if *validation condition 2* holds for T_2 , and since T_3 does not read from *object 6*, which is the only object T_2 writes to, the condition holds. T_3 will can be allowed to commit.

Programming Task

In this section we go through our implementation and how our choice of concurrency control protocol adheres to the requirements of `acertainbookstore.com`.

Implementation

As stated in the assignment text, the `synchronized` keyword has been removed to allow for finer grained concurrency control by using appropriate locks. This has been achieved by using a `ReentrantReadWriteLock` , which allows us to declare read and write locks with the desired semantics.
Our concurrency control protocol is *almost* strict two-phase locking (S2PL). The only difference is we are more conservative, and have identified the methods that at *some* point may require a write lock, and made them acquire it from the start. In other words, acquisition of the locks follows the diagram from Marcos' slides (figure 4).



	S	X
S	Yes	No
X	No	No

Figure 4: Locking solution for isolation.

More concretely, this means that methods such as `getEditorPicks` and `getBooks` need only acquire the read lock, while the methods that need to acquire write locks. Unfortunately, the library we have used for synchronization does not allow us to upgrade from a read to a write lock, so one option was to implement our methods in a manner consistent with the pseudocode described in algorithm 1.

Algorithm 1 Pseudocode for methods that need to read and write

```
function READANDWRITER(someParams)
    ACQUIREREADLOCK()
    if requestIsValid then
        RELEASEREADLOCK()        ACQUIREWRITELOCK()
        $result \leftarrow$ DOWRITE($someParams$)
    else
        $result \leftarrow Error$
    end if
    UNLOCKACQUIRED()
    return $result$
end function
```

It is clear that the method is in no way desirable, as a context switch is possible between releasing the read lock and acquiring the write lock and will render the sanity-check useless. Instead, our writers are implemented as described in algorithm 2.

Algorithm 2 Amended pseudocode for methods that need to read and write

```
function READANDWRITER(someParams)
    ACQUIREWRITELOCK()
    if requestIsValid then

        $result \leftarrow$ DOWRITE($someParams$)
    else
        $result \leftarrow Error$
    end if
    UNLOCKWRITELOCK()
    return $result$
end function
```

We can, however, expect higher throughput even with this protocol as we now allow for multiple readers, which is convenient for a book store where you commonly have several customers browsing the wares.

Testing

We have added the two required tests described in the assignment text, along with two other tests for `getInDemand()` and `getAverageRating()` [1].
To use JUnit in a threaded test, we have added two fields, `Exception exception` and `Error error` to our `ConcurrentBookStoreTest` class that are initialized to `null` and server as containers for errors and exceptions thrown by any thread that JUnit has started. These are therefore declared as `volatile` to indicate that the value may never be cached thread-locally so that threads must write directly to memory instead.
This is due to the fact that JUnit will give a false positive if we simply use `assertTrue(...)` in another thread, so we need this mechanism to catch errors in the threads we spawn. The `tearDownAfterClass` will then report the error or exception if one of these are anything other than `null`.

Finally the assignment text states: "*the clients invoke a fixed number of operations, configured as a parameter...*". This is not clear to us what is meant by this, as it seems excessive to create a new class where we translate the parameters (along with parameters such as `Set<StockBook>` for `addBooks`. Instead, we have designed static nested classes that implement `Runnable` to cater for the need for multiple threads.

Tests are performed locally.

Test 1

This test has been implemented using two helper classes through the `Runnable` interface, and perform the necessary operations on the database. The main thread then reports if we the stock is consistent.

Test 2

This tests uses the same logic as in test 1, with the exception that we have used the `volatile` "trick" described in the introduction.

[1]Which we have also implemented in the new concurrent bookstore.

Other tests

In addition to the two above, we have tested `getInDemand()` and `getAverageRating()`, again by creating static nested classes and running them as separate threads.
To properly test the `getInDemand()` method we have had to delete a part of the implementation inherited from `CertainBookStore` class, as it simply throws an error if a sale miss occurs. We of course want to keep the bookstore running even though a customer provokes a sale miss.

Discussion

As mentioned above, our locking protocol follows conservative S2PL and is in that sense correct. This can easily be seen by observing that we acquire all locks in the beginning (for both readers and writers), and release them just before returning from the method.

Furthermore, we cannot deadlock as the necessary locks are acquired immediately upon entry of the methods in `ConcurrentCertainBookStore`. This lowers concurrency, as we could have multiple methods (reader or writers) sharing the read lock for their validation steps. This is a consequence of the implementation of the synchronization library we have chosen.
We cannot observe phantoms (i.e. problems pertaining to predicate reads, see page 66 of the course compendium) using conservative S2PL, as we perform all operations (methods) while holding a read or write lock, and as we are strict we only release them at the end.
However, had we used proper S2PL, phantoms would still not be a problem since we want users to see the newest state of the bookstore at all times. For the sake of argument one solution would to lock the table/database until the reading transaction has finished. This is not feasible, as we would then allow users who browse the bookstore to prevent other users to buy.
Scalability is a problem since we lock *the whole* database when we want to write. The distinction between readers and writers allows for multiple reads but not for multiple writers (clients).
If we had implemented locks per table entry we could have hoped for higher scalability as we could potentially have a writer for each entry, and would amount to higher levels of concurrency. As it is now our database locking protocol is preventing us from achieving the level of concurrency that could theoretically be possible.

Further, if we exhibit an influx in the amount of clients buying books, the write lock will be contended and as a result we could experience thrashing as described in the course compendium. This makes our solution less scalable than if we had a lock per entry.

Finally, ssuming the cost of implementing such a protocol is reasonable, the overhead being paid by is minimal compared to the gains in concurrency. As we mentioned early, this can be seen from the nature of the application we are building, namely a bookstore. We will expect readers to be manifold compared to the number of writers (readers = browsing customer, writer = buyer), and therefore we except the locking protocol to increase our throughput while still providing isolation.