Advanced Computer Systems
Assignment 2

Anders Kiel Hovgaard Shivam Bharadwaj Su Tong

December 1, 2015

1 Serializability & Locking

1.1 Schedule 1

The precedence graph for schedule 1 is shown in Figure 1.

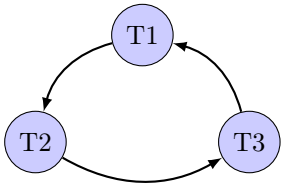

Figure 1: Precedence graph for schedule 1.

Transaction T1 reads from X before T2 writes to X, so T1 must precede T2 at this point. T2 writes to Z before T3 reads from Z, so T2 must precede T3. Finally, T3 reads from Y before T1 writes to Y, so T3 must precede T1.

This schedule is not conflict-serializable since the precedence graph of the schedule contains a cycle. Therefore, the schedule could not have been generated by a scheduler using strict two-phase locking (strict 2PL) either. This can also be seen by trying to insert lock operations and observing that T2 tries to acquire an exclusive lock on X before T1 commits even though T1 has a shared lock on X.

1.2 Schedule 2

The precedence graph for schedule 1 is shown in Figure 2.

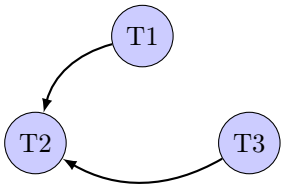

Figure 2: Precedence graph for schedule 2.

This schedule is conflict-serializable since the precedence graph is acyclic. This schedule could have been generated by strict 2PL scheduler which can be seen by injecting lock operations as shown below:

```
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
```

2 Optimistic Concurrency Control

In the following, it is given that transactions T1 and T2 have successfully committed so conflicts between these will not be checked.

2.1 Scenario 1

Since T1 completes before T3 starts, there is no conflict between these. Since T2 completes before T3 begins its write phase, we need to check whether the intersection between the write set of T2 and the read set of T3 is empty, i.e. the following should hold:

$$WS(T2) \cap RS(T3) = \varnothing \ ,$$

but

$$WS(T2) \cap RS(T3) = \{4,5\} \cap \{3,4,6\} = \{4\} \neq \varnothing \ ,$$

so T2 and T3 conflicts and therefore T3 must be rolled back.

2.2 Scenario 2

Since T1 completes before T3 begins its write phase, we need to check that the write set of T1 and the read set of T3 does not intersect. Further more, since T2 completes its read phase before T3 does, we need to check that the write set of T2 does not overlap with the read set of T3 and that the write set of T2 does not overlap with the write set of T3.

$$\begin{aligned} WS(T1) \cap RS(T3) &= \{3\} \cap \{3,4,5,6,7\} = \{3\} \neq \varnothing \\ WS(T2) \cap RS(T3) &= \{8\} \cap \{3,4,5,6,7\} = \varnothing \\ WS(T2) \cap WS(T3) &= \{8\} \cap \{3\} = \varnothing \end{aligned}$$

Since T1 conflict with T3, T3 must be rolled back.

2.3 Scenario 3

Since both T1 and T2 completes before T3 begins its write phase, we need to check that the write set of T1 and T2, respectively, does not overlap with the read set of T3.

$$\begin{aligned} WS(T1) \cap RS(T3) &= \{4\} \cap \{7,8\} = \varnothing \\ WS(T2) \cap RS(T3) &= \{6\} \cap \{7,8\} = \varnothing \end{aligned}$$

Since neither T1 and T3 nor T2 and T3 conflict, transaction T3 should be allowed to commit.

3 Questions for Discussion on the Concurrent Implementation of Bookstore

3.1 Description of implementation and testing

To make the implementation of the *ConcurrentBookStore* class thread-safe, we have used *ReadWriteLock*, specifically *ReentrantReadWriteLock* (which is the only class in the Java API implementing the *ReadWriteLock* interface), to obtain shared and exclusive locks. In the current version of the implementation, these locks lock the whole map of books in the bookstore, so it is a rather crude strategy, but it nonetheless allows for greater concurrency than using just one exclusive lock on the data or using the *syncronized* mechanism on all the methods.

Before-or-after atomicity is achieved, by using a locking strategy very similar to conservative, strict two-phase locking (2PL). Once a lock is acquired, the same lock will never be acquired again during the same execution (hence, two-phases). If a method may need to write to the `bookMap`, it obtains an exclusive lock on the `bookMap` as soon as it starts reading data from the map. If a method only needs to read from the `bookMap`, e.g. as is the case for `getBooks`, `getBooksByISBN`, and `getEditorPicks`, it only acquires a shared lock on the map. This way, concurrent read operations can proceed without blocking each other.

We currently have four tests that are concerned with concurrency. These are implemented in a separate class *ConcurrentTest*. These include the two tests described in the assignment text as well as two other tests, which work with many concurrent threads instead of just two.

The test suite sets up a collection of three books which each have a large number of copies in stock. This way, the threads can run in an arbitrary order and the stock will still be large enough for e.g. all buy operations to happen before all operations adding new copies.

The test `testConcurrentAddBuy` runs two threads which each executes a given number of `addCopies` and `buyBooks` method calls, respectively, on the same book store object. This tests before-or-after atomicity of the method calls since if we don't have before-or-after atomicity, we might observe that two threads read the same number of copies in stock, but that only one of the updates are recorded, i.e. a write-write conflict.

The test `testManyThreadsAddBuy` works similarly except it runs many threads and each thread only buys or adds a single copy of the specific book. We run twice as many threads as there are books in stock, where one half of the threads buys a book and the other half adds a copy of the book. Thus, we expect that the final number of copies returns to the value it initially was.

3.2 Correctness

We have implemented a locking protocol equivalent to conservative, strict two-phase locking (2PL). Since we are locking only one object, the two phases of the 2PL transaction are only going to lock and unlock on that single object and it is therefore just conservative, strict 2PL. Because of this, it ensures before-or-after atomicity. This is also apparent from the fact that all write operations lock the whole book map exclusively and are therefore executed serially.

Consistency of read operations, e.g. the test of consistency of `getBooks`, is ensured since operations are before-or-after atomic and since we are using shared and exclusive locks, we can guarantee that no write will take place concurrently with a read action.

Since the lock works on the entire book map, there is no concern about the atomicity of predicate reads since no write can take place concurrently with other reads.

3.3 Deadlocks or not

The locking protocol is equivalent to conservative, strict 2PL. Because of the conservative property, there cannot be deadlocks. The locking protocol cannot lead to deadlocks since there is only one thing, namely the whole book map, that can be locked. It is not possible that two threads each hold locks on different objects and wait for acquiring a lock on the object that the other thread holds, since there is only one object to lock. The unlocking mechanism is implemented by using a `try...finally` block wherein the try block monitors exceptions and the finally block releases the lock. Thus in case of exceptions, the lock will still be released. There is no way that the lock will not be unlocked when the method returns.

It is of course pretty neat that the solution does not deadlock, but it comes with the price that all write operations are being executed in serial.

3.4 Scalability bottlenecks

The only additional overhead on read-only operations, is the locking and unlocking of the shared lock, so the read-only operations should scale very well with the number of clients accessing the service, since the locking overhead is not expected to increase significantly with the number of threads acquiring shared locks.

However, since the operations involving writing to the book map acquire exclusive lock on the whole book map, these operations are effectively serial and therefore they will scale similarly to a non-concurrent implementation, although they are now thread-safe. In fact, the locking probably adds some overhead (waiting for access, maintaining a queue) compared to the previous, serial implementation, so if the application is dominated by write operations, the implemented concurrency will probably not be very beneficial and might even lower the throughput. On the other hand, if a large fraction of the operations executed by the clients are read-only operations, this implementation should be a great improvement since it allows for completely concurrent reading.

For these reasons, as the number of client increase, operations involving writing will be a scalability bottleneck.

3.5 Overhead vs. concurrency

All the locking that happens when performing write operations are essentially just added overhead compared to the sequential implementation, since operations need to wait to acquire the lock. There is also an overhead of maintaining the locks as the lock manager needs to maintain a queue. For writing, no concurrency is gained. However, in case of reading, the locking overhead will be

relatively small compared to the amount of concurrency gained. If all clients were reading, the number of clients could increase greatly and the overhead is expected to scale very close to linear in the number of clients.