Advance Computer Systems

Assignment 2

Ciprian Florescu (bgc477)

Ehsan Khaveh (ztv821)

December 1, 2015

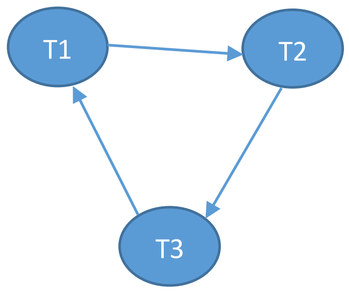
Contents

Question 1: Serializability & Locking

Schedule 1

This schedule is not conflict-serializable because its precedence graph is cyclic, which means it is not conflict equivalent to any serial schedule.
It could not have been generated using strict *2PL* approach as it is not conflict-serializable and strict *2PL* only allows for conflict-serializable schedules. In this case *T2* cannot be committed before *T1*, as scheduled based on the graph in the assignment question.

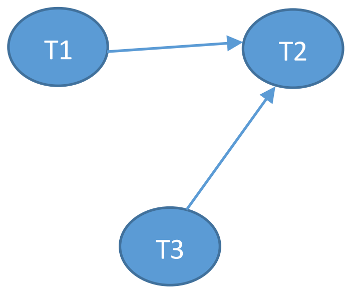
Precedence Graph:



Schedule 2

This schedule is conflict-serializable because its precedence graph is acyclic.
It could have been generated using strict *2PL* approach because it is conflict-serializable and all three transactions could be committed in the sequence they are scheduled, according to the schedule graph in the assignment text.

Precedence Graph:



Schedule with Locks Injects:

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

Validation tests for Optimistic CC:

1. For all *i* and j such that $T_i < T_j$, check that T_i completes before T_j begins.
2. For all *i* and j such that $T_i < T_j$, check that:
   - T_i completes before T_j begins its Write phase
   - $WS(T_i) \cap RS(T_j)$ is empty.
3. For all *i* and *j* such that $T_i < T_j$, check that:
   - T_i completes Read phase before T_j does
   - $WS(T_i) \cap RS(T_j)$ is empty.
   - $WS(T_i) \cap WS(T_j)$ is empty.

Scenario 1:

In this scenario *T3* has to roll back. We know that *T2* completes before *T3* begins with its write phase. So for this scenario to be valid, $WS(T2) \cap RS(T3)$ must be empty, which in this case is {4}. Thus, *T3* fails test 2 and cannot be committed.

Scenario 2:

In this case *T3* fails both test 2 because *T1* completes before *T3* begins with its write phase, and $WS(T2) \cap RS(T3) = \{3\}$, which has to be empty. Hence, *T3* must roll back.

Scenario 3:

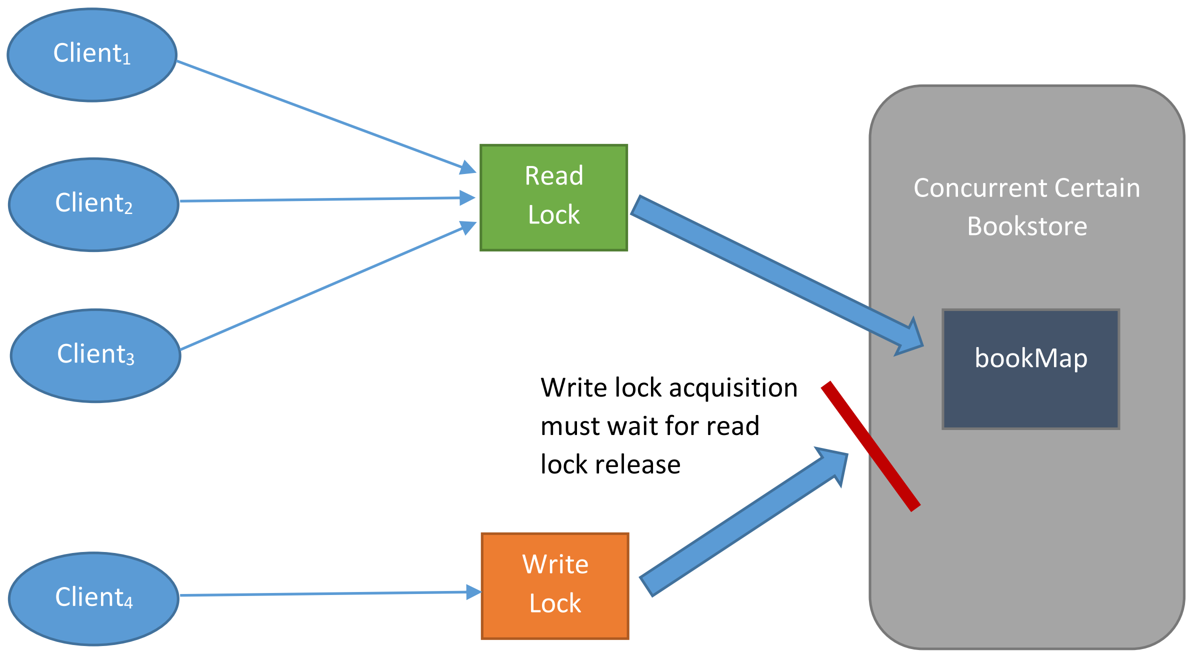
As for the last scenario, *T3* is allowed to commit. By validating the scenario using test 2, we can see that both *T1* and *T2* complete before *T3* begins with its write phase, and their write set does not share any object with *T3* Read Set.

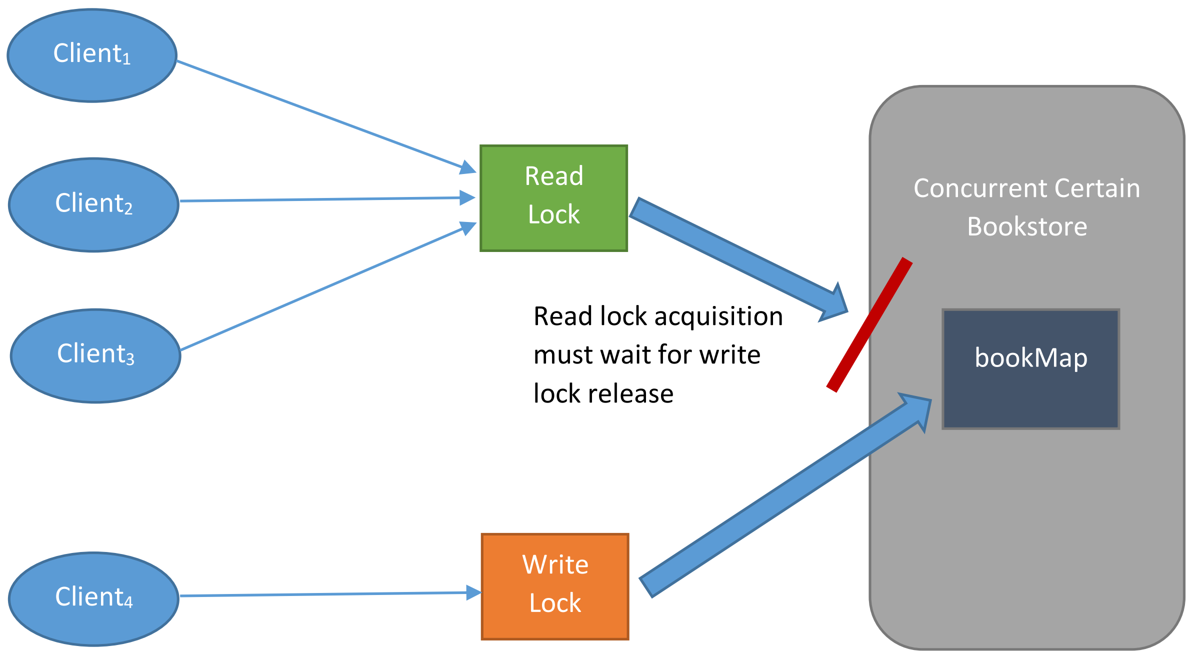
Question 3: A Concurrent Certain Bookstore

Locking Protocol Design

The approach we used for our locking protocol is based on a simple reentrant locking mechanism. In our design, we have two different types of locks, a shared lock for read operations and an exclusive one for writes to the *bookMap* object. This allows multiple threads to acquire multiple read locks and be processed simultaneously, while only a single thread can acquire a mutually exclusive write lock to perform an update. For instance, multiple threads can read our *bookMap* object (e.g. performing *containsKey* on the object), only if the object is not exclusively locked by another writing thread. Also a thread with write operation (e.g. adding a new book to *bookMap*), can only proceed when there are no other ongoing read or write operations are happening. In case there are other operations the writing thread needs to wait until they are complete and all the locks acquired on *bookMap* object is released. With this approach we get a relatively higher degree of concurrency with concurrent read requests while keeping the bookMap object persistent by allowing for sequential writes only.

The following two graphs depict our locking protocol:


Client$_1$
Client$_2$
Client$_3$
Client$_4$
Read Lock
Write Lock
Concurrent Certain Bookstore
bookMap
Write lock acquisition must wait for read lock release


Client$_1$
Client$_2$
Client$_3$
Client$_4$
Read Lock
Write Lock
Read lock acquisition must wait for write lock release
Concurrent Certain Bookstore
bookMap

The main reason for us to choose such design was that it is simple to understand and implement, and can fulfill all of the requirements for this assignment.

Locking Protocol Implementation

To implement our design of locking protocol, we have made use of java *ReentrantReadWriteLock* library, that implements *ReadWriteLock* interface. The reason we decided to choose this library was because it accommodates for having different types of locks for reading and writing operations, which is exactly what we have in our design.

In our *ConcurrentCertainBookStore* we have a single object of this class that provides us with a read and a write lock object. Throughout the code, we acquire shared locks wherever *bookMap* object needs to be accessed for reading and an exclusive lock when it needs to be updated. It is also worthwhile to mention that in cases where the object needs to be first read and then updated, we acquire a write lock on the whole process because we thought there is a chance of dirty read by other threads otherwise. For example, in our *addbooks* method, where duplicate checking is done on book ISBNs before adding them to *bookMap*, multiple writes are followed by a number of reads. If, we acquired separate locks for reading the *bookMap* and then writing it, there could have been the possibility where two writing threads, that are trying to add the same book to the map, get a shared lock on *bookMap* to check for duplicity. They both get a green light to go ahead and add the book to the map. The first book that acquires the lock adds the book while second one has to wait till it is done. When the book is added, the second book tries to adds the book to the map and wen end up with a duplicate in our collection.

Test Cases

We created a new parametrized test class *BookStoreConcurrencyTest*. All of the tests can be configured in *@Parameterized.Parameters.* An example of a parameter is *{100, 100, 2}*. They are called *repeatsC1*, *repeatsC2* and *nrThreads*. Additional parameters can be added, if needed.

The followings are our test cases for testing concurrency of the application:

Test 1

C1 and C2 operate against the same set of books S. C1 calls *buyBooks*, while *C2* calls *addCopies* on *S*. This test uses the first two parameters (*repeatsC1* and *repeatsC2*) which represents the number of transactions executed in each thread. We spawn two threads respectively *C1* and *C2* and call *buyBooks* from *C1* and *addCopies* from C2 on the same amount of books. This test is checking mainly if the implementation respects atomicity and blind write (Overwriting uncommitted data).

Test 2

This test checks mainly for consistency between atomic transactions. We are spawning two threads *C1* and *C2*, *C2* will loop forever and call *getBooks*, *C1* will call *buyBoooks* then *addCopies*. Every time *C2* calls *getBooks* the data should be either the state before *C1* or the state after *C1* in order to be valid and respect atomicity for transactions.

Test 3

In this test case, we are testing consistency between writes and reads, also somewhat stress test. We are spawning a configurable amount of threads for three operations (third parameter). First two are write operations and modify the data, the third operation checks if everything has executed atomically. We are ignoring runtime exceptions for this test, we care only if transactions are executed partially.

Test 4

Test for dirty read. Spawning two threads, and running them a configurable amount of time. We are reading the same object at the same time and then try to modify the value of the copies of the book. Both modifications should show in the result. E.g. *T1* reads *A*, *T2* reads *A*, *T1* writes *A+50*, *T2* writes *A+10*, the result should be *A+50+10*.

Test 5

This test case tests possibility of having deadlocks in the system. Although in theory this is guaranteed not to happen, we made a test for this to demonstrate that in practice as well.

Questions for Discussion

1.

a) Since our protocol respects before-or-after atomicity, transactions cannot be executed partially (it is either all or nothing). This is also evident in our tests. We use exclusive locks on the object in case of an update and shared locks for read operations, we execute the transaction and finally release the locks acquired earlier.

b) We used unit tests to verify the correctness of our implementation. We are spawning threads in order to simulate multiple clients and execute transaction that could potentially break our protocol or semantics. To verify that anomalies do not occur we implemented different unit tests. We are spawning a configurable amount of threads and execute transactions or spawn two threads and execute a configurable amount of transactions. e.g. for dirty reads we spawn two threads and read the same object at the same time and try to modify it and check the result of the write operation (since the write operations are executed sequentially the object should show modifications from both threads).

2.

In our design, we follow a conservative strict two phase locking (*2PL*). We obtain locks on the object for each request (exclusive for updates and shared for read-only), process the request and release the locks. Since the locks are obtained on the collection as a whole and not on every item of it, and we are only dealing with a single lock for each type of request, our protocol is conservative and strict.

This approach makes the *bookMap* collection a thread-safe object by implementing locks on the object. This is to guarantee that out collection behaves sensibly when it is being operated on via multiple threads. The locking mechanism also makes sure that write operations via different threads run sequentially, and that prevents anomalies from occurring when updating *bookMap* object. In that way, our solution honors before-or-after atomicity (serializability). At the same time, our protocol achieves concurrency improvements by allowing multiple read operations to be processed simultaneously.

Having said that, we believe our simple solution to this problem is correct because it meets all the main goal here, which is enhancing the concurrency of the system while respecting before-or-after atomicity.

However, there are certainly changes that can be made to our solution to achieve a higher degree of concurrency. For instance, by having a more complex locking mechanism that allows for locking different parts of the *bookMap* object instead of locking the whole collection.

3.

As discussed in the second question, our locking protocol follows a conservative strict 2PL. Hence, chances that it can lead to a deadlock are none. The reason for this is because in our protocol we obtain all the locks we need (in our case only one lock for each type of operation) before the operation begins. This ensures that a transaction that already holds some locks will not block waiting for other locks.

4.

There could be some scalability bottlenecks in the system after implementing our protocol, especially if there are a lot of concurrent write operations going on in the system. This is because our protocol runs the write operations sequentially and performance will be greatly affected with huge number of writing threads making concurrent updates to our bookstore. Otherwise, the system is expected to scale well with a good degree of concurrency if the number of concurrent write requests is not very high.

5.

While the locking protocol improves the concurrency by managing concurrent operations in the system, it also imposes some overhead that affects the performance of the system in a negative way. How significant this overhead is in a system, mainly depends on the number of locks the system uses. The more locks a program uses, the more overhead associated with the usage. Besides, how frequent the locks are acquired and released can also affect the amount of overhead.

In our locking protocol, since there are only two locks (a read and a write lock) used, this overhead is insignificant compared to the degree of concurrency the protocol is expected to provide.