Alexander Carlqvist

Assignment 2

Question 1

1. The precedence graph of schedule 1 (see Figure 1) is not acyclic. That is, we can go around in the graph. Therefore, schedule 1 is not conflict serializable.

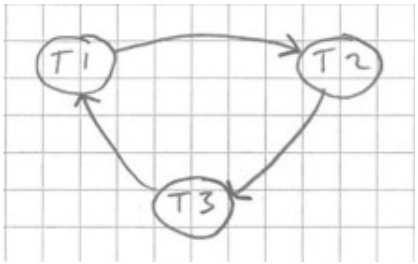

Figure 1. The precedence graph of schedule 1.

The precedence graph of schedule 2 (see Figure 2) is acyclic. That is, we cannot go around in the graph. Thus, schedule 2 is conflict serializable.

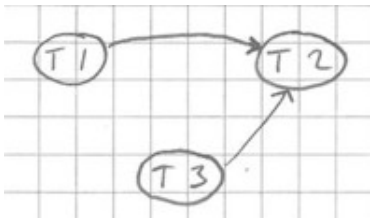

Figure 2. The precedence graph of schedule 2.

2. Strict 2PL only gives conflict serializable schedules. Thus, we cannot have schedule 1 with strict 2PL.

We can have schedule 2 with strict 2PL. The following schedule is an adaptation where SL denotes acquiring a shared lock and EL denotes acquiring an exclusive lock. As we are dealing with strict 2PL, all locks are to be released when a transaction is completed. That is, immediately after C.

T1: SL(X) R(X) EL(Y) W(Y) C
T2: SL(Z) R(Z) EL(X) W(X) EL(Y) W(Y) C
T3: EL(Z) W(Z) C

Question 2

Scenario 1

Test 1 holds for T1 and T3.

Trivially, test 1 does not hold for T2 and T3. Test 2 does not hold for T2 and T3. We see that the intersection of WS(T2) and RS(T3) is {4}, which is nonempty. Test 3 cannot hold because of the previous observation.

We see that no test holds for T2 and T3. By contradiction, T3 must roll back.

Scenario 2

Test 1 obviously does not hold for T1 and T3. Test 2 does not hold. The intersection of WS(T1) and RS(T3) is {3}, which is nonempty. Test 3 cannot hold as well because of that.

Trivially, test 1 and 2 do not hold for T2 and T3. However, test 3 holds. The intersection of WS(T2) and RS(T3) is empty and the intersection of WS(T2) and WS(T3) is empty.

No test holds for T1 and T3. By contradiction, T3 must roll back.

Scenario 3

Test 2 holds for T1 and T3. The intersection of WS(T1) and RS(T3) is empty.

Similarly, test 2 holds for T2 and T3. The intersection of WS(T2) and RS(T3) is empty.

We have found one test that holds for every pair. Thus, T3 is allowed to commit.

Programming task: A Concurrent Certain Bookstore

We have decided to follow the two-phase locking protocol (2PL). Everything we need is in java.util.concurrent.locks. ReadWriteLock provides the interface. We need only modify ConcurrentCertainBookStore. It needs a private manager that takes care of all locks. A ReentrantReadWriteLock is the perfect choice. If we let it be in fair mode, locks will be handed out approximately in the order we receive requests. Thus, a transaction needs not risk waiting forever for a lock.

The mutable data we set out to protect is bookMap (including its books). In effect, bookMap can be considered our database.

We devised a general pattern for the methods to follow. First, if there is any input, we always check if it is null. If so, we throw an exception. We can exploit this and not ask for any lock at this time. In effect, if a method is called with the wrong arguments, it is allowed to abort immediately. This allows for greater concurrency. Now we deal with the locking. We acquire either a read (shared) or write (exclusive) lock depending on the operations of the request. This corresponds to the expanding phase. All work is wrapped in a try-block, and our releasing of the lock is put in a finally-block, which corresponds to the shrinking phase. The reason is to ensure that we still release the lock in case of an exception due to an error. Two-phase locking requires that all locks be released whether a transaction either commits or aborts. Some methods return a value right after the lock is released. This would not violate any two-phase locking scheme, because all return values are constructed by making immutable objects (deep copies) from the books while we have a lock. That is, whatever we return is private data that belongs to the transaction.

In deciding what lock to ask for, we check if a request does any write operation directly against bookMap or indirectly through fetching a book and modifying it. If so, we need a write lock. If we will only make read operations, we ask for a read lock. We see that most methods have a validation phase in which numerous tests are carried out. If there is an error, an exception is thrown and we break out of the program flow. These validation phases only do read operations. The idea was to exploit this by first acquiring a read lock and then upgrading it if there was no error. Unfortunately, ReentrantReadWriteLock does not support lock upgrading. This would have allowed for more concurrency. Maybe someone would be tempted to release the read lock and ask for a write lock. This does not work as intended. Lock upgrading requires special treatment. The thread in question needs the highest priority and no other thread should be granted a lock in between. This is to ensure atomicity. We cannot allow for the possibility of our data changing between our tests and updates.

Deadlocks are possible. The reason is how we acquire locks. The methods we invoke wait indefinitely for a lock. If there are several competing transactions that all want a lock, we can get stuck. One way to combat deadlocks would be to use tryLock instead of lock.

There is a scalability bottleneck with respect to the number of clients. The more

clients, the higher risk of deadlocks.

The overhead we pay for is cheap. We only have one lock manager that guards bookMap. Unfortunately, we do not get the highest degree of concurrency out of our locking protocol. The problem is that we lock our entire database. We would gain in concurrency if we were to have locks for individual books. The overhead would increase dramatically, though.

For our concurrent bookstore to be reliable, we must guarantee ACID. Thus, we have one test for each property of ACID.

Test 1 (BookStoreTest.test1)

The first test is for atomicity. We make a copy of the test book to remember what it was before any concurrent transactions took place. We make one thread for each client in which it will do its actions. When both threads have died, we get the test book again and now we compare if it is identical to what it was before. It should be. The point is that the clients' actions cancel each other out.

Test 2 (BookStoreTest.test2)

The second test is for consistency. We have a set of books as they are when they have been replenished. We also have a set of the books as they are when some copies have been sold. The first thread repeatedly buys and adds copies of the books. The second thread repeatedly asks for the books. Every book must be identical to what it would have been like had it been either replenished or sold. Keep in mind that this relation should hold for each individual book. If it does not hold for some book, the test fails.

Test 3 (BookStoreTest.test3)

The third test is for isolation. We have some threads that concurrently add some books to the bookstore. When the threads have died, we check to see if the books are there as if they had been added in some sequence. All books should be there. If not, the test fails.

Test 4 (BookStoreTest.test4)

The fourth test is for durability. We have a thread that adds a book and then sleeps before it will add another book. Meanwhile, an interrupt signal is sent to the thread to simulate a system crash. What was added before the crash should be in persistent memory. If it is not in the database, the test fails.