Advanced Computer Systems
Assignment 2

Ola Rønning (vdl761)
Tobias Hallundbæk Petersen (xtv657)

December 1, 2015

1 Serializability & Locking

Schedule 1

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

Figure 1: Schedule one.

Schedule one, reproduced above, corresponds to the precedence graph in figure 2. The precedence

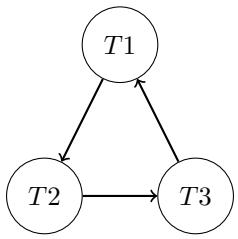

Figure 2: Precedence graph for Schedule one, see figure 1.

graph, see Figure 2, contains a cycle between transactions: `T1,T2,T3` and the schedule is thererfore not conflict serializable. As schedulers using Strict two phase locking allows only conflict serializable schedules. Schedule one cannot have been produced by a scheduler following Strict two phase locking.

Schedule 2

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

Figure 3: Schedule two.

Schedule two, reproduced above, corresponds to the precedence graph in figure 4. The precedence graph, see Figure 4, is acyclic, so Schedule two is conflict serializable. In particular, schedule two is equivalent with a serial schedule where transaction two is performed last. As schedule two is conflict serializable, schedule two could be scheduled by a scheduler following Strict two phase locking. The injection of shared and exclusive locks, required if the scheduler was using Strict two phase locking, is reproduced below in figure 5.

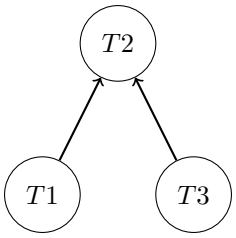

Figure 4: Precedence graph for Schedule one, see figure 3.

```
T1:  S(X)                  E(Y)RS(X)RE(Y)
T2:                   S(Z)                  E(X)E(Y)RS(Z)RE(X)RE(Y)
T3:          E(Z)RE(Z)
```

Figure 5: Shared and exclusive locks that need to be acquired in order for schedule two, see figure 3, to be scheduled using Strict two phase locking. `S(Q)` acquires a shared lock on `Q` and `RS(Q)` releases the lock on `Q`, exclusive locks follow the same semantics, only with an `E`.

2 Optimisitc Concurrency Control

Scenario 1

```
T1:  RS(T1) = {1, 2, 3}, WS(T1) = {3},
T1 completes before T3 starts.
T2:  RS(T2) = {2, 3, 4}, WS(T2) = {4, 5},
T2 completes before T3 begins with its write phase.
T3:  RS(T3) = {3, 4, 6}, WS(T3) = {3},
allow commit or rollback?
```

Figure 6: Scenario one.

In scenario one, see Figure 6, we will have to roll-back. As both the write-set of transaction two and the read-set of transaction three contain object `4`. The conflict occurs because transaction two completes before transactions threes begins its write phase.

$$WS(T2) \cap RS(T3) = \{4, 5\} \cap \{3, 4, 5\} \tag{1}$$
$$= \{4\} \neq \emptyset \tag{2}$$

Scenario 2

```
T1:  RS(T1) = {2, 3, 4, 5}, WS(T1) = {4},
T1 completes before T3 begins with its write phase.
T2:  RS(T2) = {6, 7, 8}, WS(T2) = {6},
T2 completes read phase before T3 does.
T3:  RS(T3) = {2, 3, 5, 7, 8}, WS(T3) = {7, 8},
allow commit or rollback?
```

Figure 7: Scenario two.

In scenario two, see Figure 7, we will also have to roll-back. As both the write-set of transaction one and the read-set of transaction three contain object `3`. The conflict occurs because transaction one completes before transaction three begins its write phase.

$$WS(T1) \cap RS(T3) = \{3\} \cap \{3, 4, 5, 6, 7\} \tag{3}$$
$$= \{3\} \neq \emptyset \tag{4}$$

```
T1:  RS(T1) = {2, 3, 4, 5}, WS(T1) = {4},
T1 completes before T3 begins with its write phase.
T2:  RS(T2) = {6, 7, 8}, WS(T2) = {6},
T2 completes before T3 begins with its write phase.
T3:  RS(T3) = {2, 3, 5, 7, 8}, WS(T3) = {7, 8},
allow commit or rollback?
```

Figure 8: Scenario three.

Scenario 3

In scenario three, see figure 8, transaction three can commit as there are no offending objects.

$$WS(T1) \cap RS(T3) = \{4\} \cap \{2,3,5,7,8\} \tag{5}$$
$$= \emptyset \tag{6}$$
$$WS(T2) \cap RS(T3) = \{6\} \cap \{2,3,5,7,8\} \tag{7}$$
$$= \emptyset \tag{8}$$

3 Discussion on the Concurrent Implementation

1. For the implementation we have used two types of locking, one for the entire bookmap, and one for each individual book in the bookmap, whenever a subset of books are to be written or read from, their locks are taken in order of their ISBN. The lock for the entire bookmap, is only used in cases such as adding, removing or checking if something is contained in it. This results in a strict conservative two-phase locking scheme.

   (a) By ensuring that no reads or writes to a set of books are done unless all locks for the given books are held. This way an operation can only read when others read but not write, ensuring that no reads result in a wrong value. Likewise writes are only able to be executed if all locks for the books to write to are held.

   (b) We implemented two test to check correct behaviour during concurrent execution *testSerilizability* and *testNoDeadLocks* besides the two suggested in the assignment text.

   - *testSerilizability* has three threads attempt to write to the same location, that is buy the same book. In this test two of threads should be throw exceptions as they all attempt to buy more than half of the books left in the book store. This test that our lock produces a conflict free schedule as the threads create a both read-write and write-write dependencies.
   - *testNoDeadLocks* starts a large number of threads that each interact with the same collection of books. This test checks that deadlock does not occur, in particular that locks are taken in the same ordering for all threads.

   The two test from the assignment are implemented with the semantics specified, test named **Test 1** in the assignment is implemented in the method named *testBefAftAtom* and the test named **Test 2** in the assignment is implemented in the method named *testConsistency()*. All test behave as expected.

2. As the locking is partitioned, we have two different locking schemes. To argue the locking scheme for the entire bookmap is correct, we need to look at the functionality. There are three different operations that can be done on the bookmap: adding, removing, looking up. The first two being writes, and the last being a read. Whenever adding one or more books, the write lock is taken, the locks for the new books are initialized, the new books are added, and the write lock is given up. This way no books can be removed, no books can be looked up and no books can be added, resulting in a correct bookmap after the operation.
   Whenever removing one or more books, the write lock is taken, the locks for the books to be removed are taken, the books are removed.And lastly the locks for the books are removed unlocked, and removed from the lockmap. At this point the thread that might be waiting for one of the removed books will crash as the book is not to be found anymore. This behaviour is desired as it is the cleanest way to stop threads waiting to operate on to be removed books.
   When looking up whether or not a book exists in the bookmap, the bookmap read lock is taken,

this ensures that the book is not removed when looking it up, and that it is not added after looking it up.
The other locking scheme which is on book by book level, we use strict conservative two-phase locking, which if implemented correct is correct.

3. Yes, in the case where a thread dies unexpectedly, and its lock is not given up. In all other cases, no two threads can wait for each other. On the book by book level, strict conservative two-phase locking is implemented, which will not result in deadlocking, on the bookmap level, as there is only one lock for read and one for write, no deadlocking can occur. The only place you could think deadlocking could occur would be when the two locking schemes waited for each other. The only operation that uses both locking schemes at the same time is when removing a book, which implies that there will result in no deadlocking.

4. The only issue with scalability is that the add and remove operation become more and more expensive, as no other operations can lookup whether or not a book exists when these are done. When scaling up the system we can assume that more of both will be requested and as they become more expensive it is a bottleneck.

5. We incur an overhead that is of size $O(\lg n)$ when doing an operation on one or more books, as ISBN's needs to be sorted and locked. A harder to measure overhead happens when taking the write lock on the entire bookmap, this will increase all concurrent requests with the time of the write. This overhead is not that bad for the concurrency we get, assuming that adding new books and removing old books are relatively rare operations.