

Advanced Computer Systems

Assignment 2

Anna Sofie Kiehn and Kenneth Jürgensen

December 1, 2015

1 Serializability and Locking

1.1 Precedence graph

The precedence graphs for schedule 1 and schedule 2 can be seen in Figure 1.

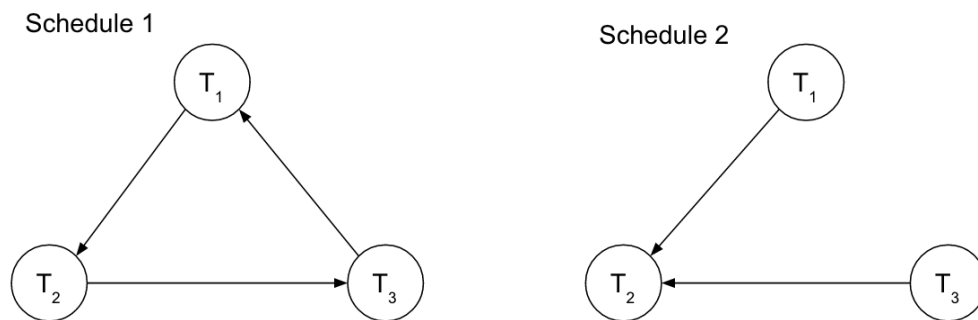


Figure 1: Precedence graphs for schedule 1 and schedule 2

As can be seen in the graphs, schedule 1 cannot be synchronized as there is a cycle in the graph. Schedule 2 can be synchronized as it contains no cycle.

1.2 Can the schedules be generated using strict 2PL

To see if the schedules can be generated using strict 2PL we insert the necessary locks into the schedules, which can be seen in Figure 2 and Figure 3. In 2 we see that when T_2 tries to acquire an exclusive lock on X (marked with red in the figure) it has to wait till T_1 that already has a shared lock on X , finished. This however breaks with the schedule as we see T_2 should finish before T_1 . Schedule 1 therefore could not be generated using strict 2PL. In schedule 2 no problems arise when adding the locks, since T_3 finished before T_2 needs to acquire the shared lock on Z , and T_1 finishes before T_2 needs the lock on X and Y . So schedule 2 could be generated using strict 2PL.

2.3 Scenario 3

All checks are possitive so T3 is allowed to commit.

Test :

```
Check that T1 completes before T3 begins Write phase
Check that T2 completes before T3 begins Write phase
Check that  $WS(T1) \cap RS(T3) = \emptyset$ 
Check that  $WS(T2) \cap RS(T3) = \emptyset$ 
```

3 Programming task

3.1 Short description of implementation and test

We have used the `ReentrantReadWriteLock` class to implement our locking scheme. When running the test on our machines with non-local tests, it takes roughly 2-3 minutes. The local tests uses millions of requests to measure the performance, and those are therefore the most accurate tests. As jetty adds a lot of overhead when using HTTP, and the non-local tests are scaled down versions of the local tests, these are more to show that it works, but the performance measurements are not as precise.

3.1.1 Strategy

We acquire a read lock just before validating the input in the methods that are read only, and release it right after reading the result. In the methods that write to the book store map, we acquire a write lock just before validating the input, and hold the lock until the data has been written to the book store. In every method, if an exception is thrown then the held lock is released before throwing the exception. We implemented the two tests described in the assignment to test the atomicity of our implementation. We also implemented a third test to test the throughput of our implementation compared to the `CertainBookStore` in the previous assignment that uses synchronized in every method. The test starts 1000 threads, each calling the given bookstore with 10,000 `getBooks()` requests. The time is noted before starting all the threads and when all are done, calculating the throughput as number of requests handled per second. Since our implementation allows for concurrent reads it should be faster than the old implementation, and so the test is successful if the calculated throughput for our implementation is higher than the throughput for the old implementation. The last test shows that while our implementation allows for much concurrency with read requests, it is sensitive to the amount of write requests. The test measures the throughput 4 times, each time with 1000 read threads and 10,000 requests sent by each request. The number of write threads are 1, 10, 100 and 1000, each sending a write request to add one copy to all books in the bookstore. When run on our machines, the first three measurements are quite near each other, but on the last measurement, where 50% of the requests are writes, there is a substantial drop in throughput. This is discussed in the scalability section.

3.1.2 Correctness

Since we only hold one lock at any point in any method, and that lock is released just before exiting the method, whether through throwing an exception or returning a result, there is not much that can go wrong with the implementation. This approach makes dirty read and writes impossible, as a method that writes can not be interrupted between validating its input and writing its data. We implemented the test 2 described in the assignment text to test that before-after-atomicity holds up for our implementation, and that no dirty reads happen. Since we validate all input before writing anything, dirty writes are impossible too.

3.2 Correctness of protocol

We want to argue that our implementation, is correct by showing it follows the strict 2PL protocol. We follow the strict 2PL protocol in that we:

- Comply with 2PL as we acquire all locks before we do any work, and releases all locks at the same time, having a acquire and release phase.
- We have shared locks for transactions that only reads
- We have exclusive locks for transactions that writes.

2PL is vulnerable when it comes to predicate reads, but since we lock the entire hashmap each time writes happen, predicate reads cannot happen.

3.3 Deadlocks

Deadlocks are not possible in our implementation. Each method only holds one lock, either a read or a write, and a deadlock requires waiting to acquire one or more locks while holding one or more locks. Since each call to the book store can at most be a series of calls, no thread can hold more than one lock, and that lock will be released before acquiring another, assuring deadlocks are impossible.

3.4 Scalability bottlenecks

Because we have used blocking to implement concurrency there is a scalability bottleneck in regards to the number of writing clients (transactions), when the transactions begin to conflict and block each other. When the number of active transactions increase so will the amount of blocking and therefore the amount of delay, which means that throughput will increase more slowly than the number of active transactions. In our implementation, we keep locks for a relatively long period of time and we lock the entire hashmap so the likelihood of transactions having to wait is fairly big. If we instead had only locked smaller parts of hashmap, for instance by using a concurrent hashmap (from the `java.util.concurrent` library) we could have supported both concurrent reads and writes - if the writes was on different parts of the data. This would have made the implementation much more scalable but the locking protocol more complex.

3.5 Overhead

The amount of concurrency we get depends on whether there is mostly reads or mostly writes in the application. If there are mostly writes there is practically no concurrency, but we still have the overhead due to locking and unlocking. However there is mostly reads or even just some reads the concurrency vs the overhead is pretty good.