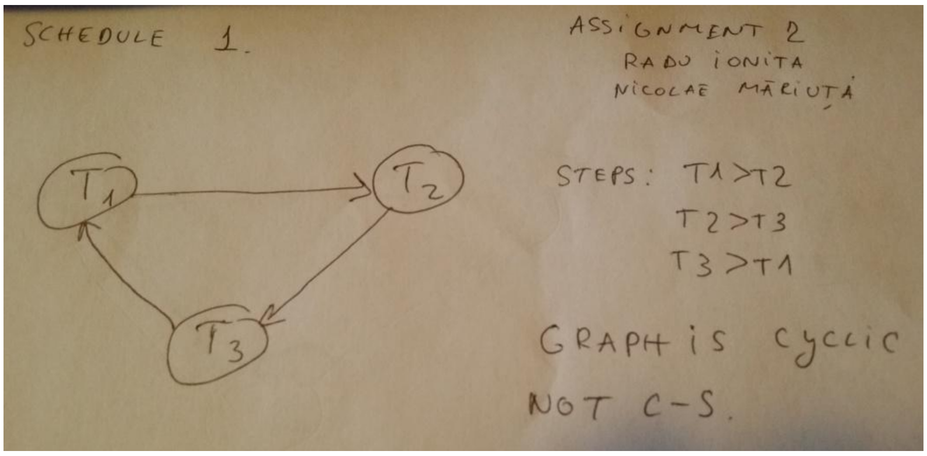
Assignment 2

Ionita Radu, Nicolae Mariuta

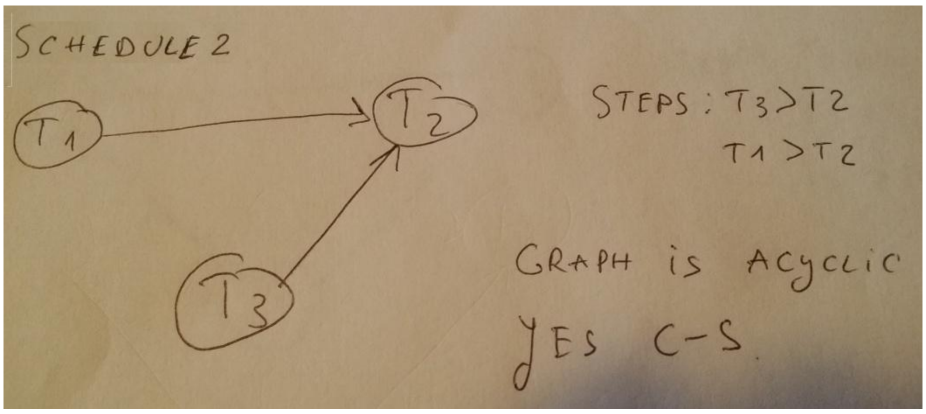
02.12.2014

Q1: Draw the precedence graph for each schedule. Are the schedules conflict-serializable? Explain why or why not.



$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \arrow[dl] \\ & T_3 \arrow[ul] & \end{tikzcd}$$

Picture 1. Schedule 1



$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \\ & T_3 \arrow[ur] & \end{tikzcd}$$

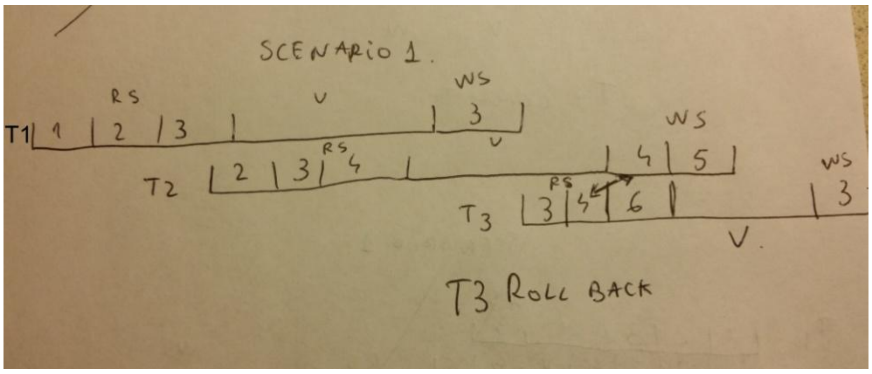
Picture 2. Schedule 2

As seen in the pictures, Graph 1 is Cyclic, making it not conflict serializable, while the Graph 2 is Acyclic, making it Conflict Serializable.

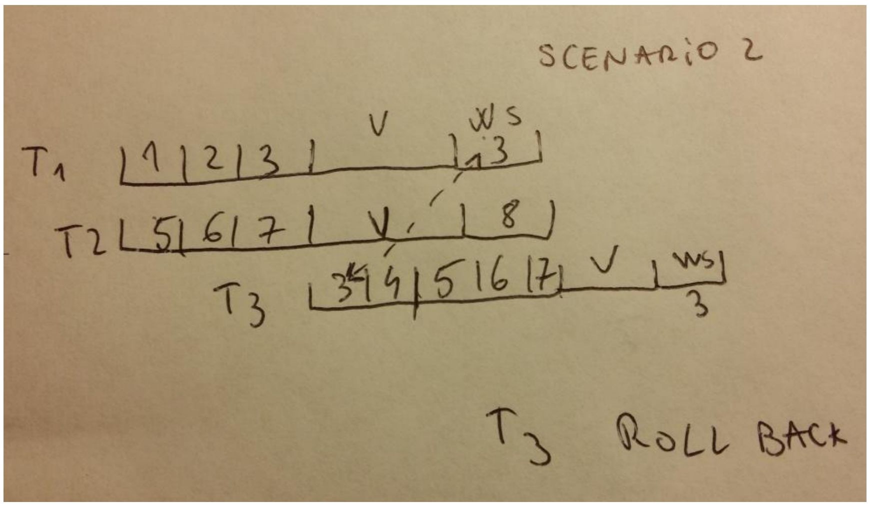
In the first Schedule there is a possibility for a dirty read, where T1 uses READ on X, after that T2 makes a WRITE on X. T1 will not READ the next X until the it commits, showing a READ that is not the most updated one.

Only one of the schedules has been generated by a strict 2PL the one that is acyclic.

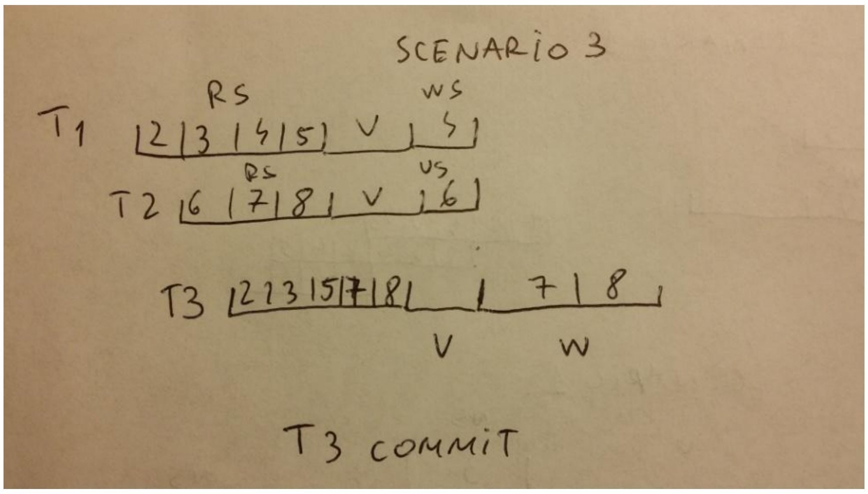
Q2: Consider the following scenarios, which illustrate the execution of three transactions under the Kung-Robinson optimistic concurrency control model. In each scenario, transactions T1 and T2 have successfully committed, and the concurrency control mechanism needs to determine a decision for transaction T3. The read and write sets (RS and WS, respectively) for each transaction list the identiers of the objects accessed by the transaction.



Picture 3. Scenario 1



Picture 4. Scenario 2



Picture 5. Scenario 3

As you can see in the above pictures, in scenario 1 T3 can not commit because it fails Test2 and Test3, namely the intersectation between WriteSet(T2) ReadSet(T3) is not empty. In Scenario 2 T3 can not commit (Rollback) and it fails Test2 and Test3, again the the intersectation between WriteSet(T1) ReadSet(T3) is not empty. In Scenario 3 T3 can succesfully commit because all Tests are fulfilled, the intersections are empty.

Programming Implementation

1. Implement locking protocol

For this task we implemented Conservative 2PL protocol which adds exclusive locks to items that are modified and shared locks to data that is read. We have used the *ReadWriteLock* to add exclusive and shared locks.

Inside the functions that only read data, like *getBooks* or *getTopRatedBooks* we added *readLock* at the beginning of the function and we unlocked the items at the end. In the same way we inserted *writeLock* inside functions that modifiy items in databes like *addBooks* and *rateBooks*. By doing this we ensure that 2 clients that want to read data then they are allowed to access the information but if one of them wants to write somethig, it will be exclusive lock.

We also implemented the functions *rateBooks*, *getTopRatedBooks*, *getBooksInDemand* by simply adding the code from the previous assignment and implemented Strict2PL protocol in the same way that we did for the other functions.

2. Tests for concurrent implementation

For the testing we created classes that implement *Runnable* class and created the operations:

- *ConcurrentBuyBooksThread* which buys a certain number of copies of books.
- *ConcurrentAddCopiesThread* which is for the stock manager client to add copies of a certain book to the store
- *ConcurrentGetBooksThread* thread for getting all the books in the store and compares them to list of books that were in store before starting the thread
- *ConcurrentBuyAndAddThread* which takes a list of book, call method to buy and then method to add them again to the bookstore
- *ConcurrentRatingThread* thread that adds a number of ratings to a certain book

We used these classes to start multiple client threads and make testing functions:

- *testBuyAddCopies* here we answered for the Test1 requirements. We got no failures by having the protocol strict 2PL implemented but the test failed when we removed the locks from *ConcurrentCertainBookStore* which shown that without the protocol, the application does not work
- *testMultipleBuyAddCopies* in this test we answered for the requirements of Test2 and again we did not get failure unless we removed the lockings from bookstore implementations.
- *ConcurrentRatingTest* for this test we create 2 threads: one for adding ratings to certain books and one for reading the ratings. We also check if the thread for reading the rating get the expected result, which is the rating after the first thread has modified the values. We did not get errors by implementing this protocol but we think that it is not so critical for a bookstore, as it is the buy operation
- *testBuyCopiesDemand* starts 2 threads that want to buy a large amount of books from the store. Together, the clients will buy more books that are copies in the store. We wanted to check what happens if multiple clients want to buy at the same time and there is a limited number of book copies. Because the first client has lock over the items that he is accessing then no failure has occurred.

1.Is your locking protocol correct?

Our locking protocol is correct because, as described previously, we added locks at the beginning of each method and released lock at the end of each method; shared locks for methods that read data and exclusive locks for data that is modified. This is a simple approach.

2. Can your protocol lead to deadlock?

Because we add locks at the beginning of each method and releasing them at the end, then the protocol cannot lead to deadlocks. And the protocol conservative 2PL cannot have deadlocks.

3. Scalability bottlenecks

Bottlenecks can occur because it is because if there is a client that modifies data in the database, then it becomes inaccessible for the other clients and all will have to wait for shared lock. Some operations like modifying the ratings and get information about books should be allowed to occur because dirty reads in those situations are not that critical as in case of buy books and add new books to store.

4. Discuss the overhead

The overhead is not a big problem in the application. Because the number of clients that add books to database and add copies (the stock manager) is smaller than the number of clients that want to add rating, view ratings and view data about books.