# Laboratory 6

Title of the Laboratory Exercise: Transactions

1. Introduction and Purpose of Experiment

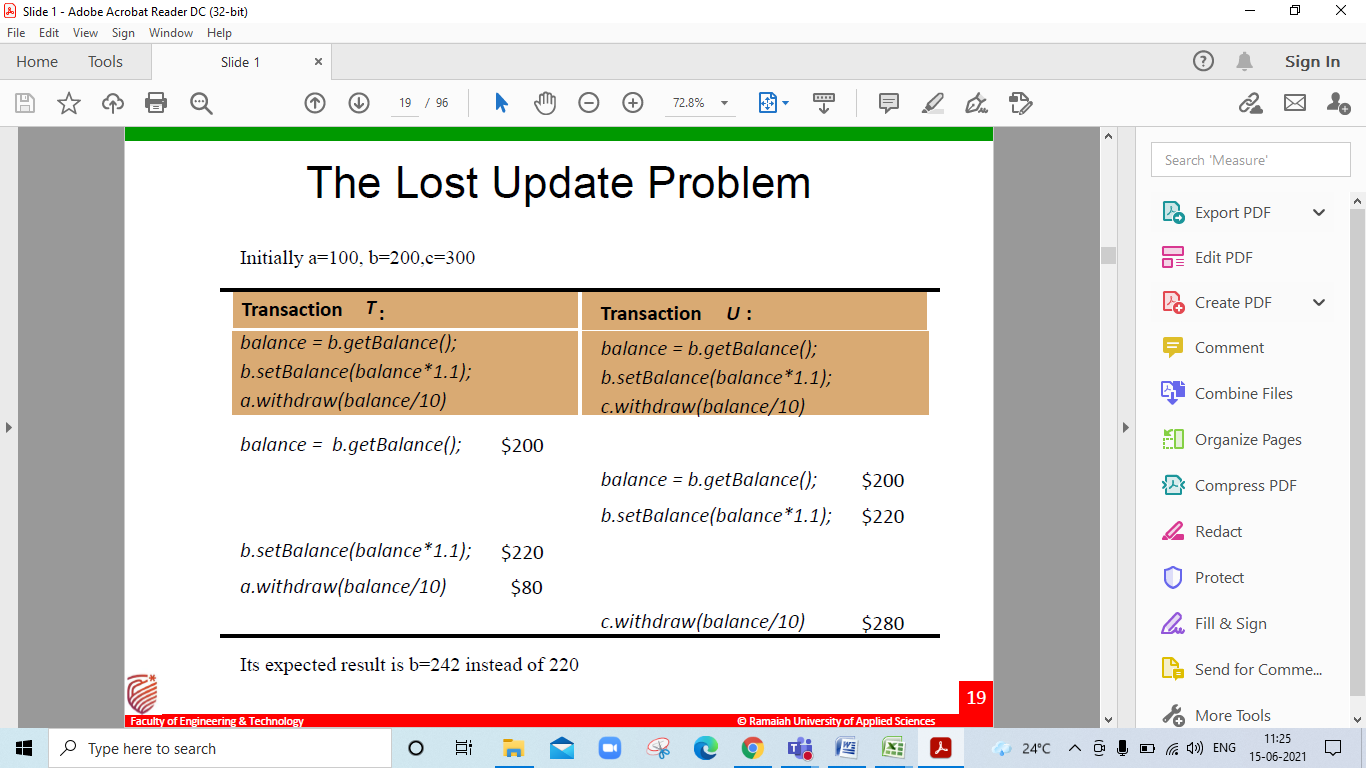
Aim and Objectives

Aim

* To develop a program to perform concurrent transactions

1. Experimental Procedure
   * 1. Analyse the problem statement
     2. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
     3. Implement the algorithm in Java language
     4. Compile the Java program
     5. Test the implemented program
     6. Document the Results
     7. Analyse and discuss the outcomes of your experiment
2. Question

Create a Multithreaded Java program with two threads handling Transactions, U and T. Transactions T and U have at-least one common object and at-least two operations. The program should ensure that the execution of the transactions is serializable (free from conflicts). Proper Concurrency control mechanism needs to be used wherever necessary.



1. **Computations/Algorithms**

* **Step 1:** Start
* **Step 2:** Create a main class Lab6\_ds to declare thread objects and invoke each thread where, each thread denotes a Transaction
* **Step 3:** Create a generic class Account that wraps an Integer as the underlying data type
* **Step 4:** Create an object of Integer type and a Lock object of ReentrantReadWriteLock in the generic class
* **Step 5:** Declare the following methods in class Account: -  
  getBalance() – Read operation to return current balance before or after another operation setBalance() – Write operation to update balance after another operation  
  deposit() – Write operation to deposit amount  
  withdraw() – Write operation to check balance and withdraw amount
* **Step 6:** Use the read lock function of the ReentrantReadWriteLock object Lock to safeguard concurrent access to all the read operations in a given transaction i.e. getBalance()
* **Step 7:** Use the exclusive write lock function of the ReentrantReadWriteLock object Lock to safeguard concurrent access to all the write operations in a given transaction i.e. setBalance(), deposit() and withdraw()
* **Step 8:** Create a class Transaction\_T that extends thread class
* **Step 9:** Class Transaction\_T takes three objects of class Account named a, b and c and the transaction ID as input parameters from the newly created thread object (of class Transaction\_T) invoked in main class i.e., any thread invocation of class Transaction\_T is a transaction to perform any of the four operations defined in Step 5
* **Step 10:** Using the class Account objects passed during thread invocation, read or write operations are performed inside class Transaction\_T in any order
* **Step 11:** Create a class Transaction\_U that extends thread class
* **Step 12:** Class Transaction\_U takes three objects of class Account named a, b and c and the transaction ID as input parameters from the newly created thread object (of class Transaction\_U) invoked in main class i.e., any thread invocation of class Transaction\_U is a transaction to perform any of the four operations defined in Step 5
* **Step 13:** Using the class Account objects passed during thread invocation, read or write operations are performed inside class Transaction\_U in any order
* **Step 14:** Declare and initialize three objects a, b and c of class Account and pass the objects as parameters of the threads created for classes Transaction\_T and Transaction\_U **Step 15:** Each thread that is created, represents a transaction. Once all threads are invoked, wait for the processes to terminate before main thread terminates
* **Step 16:** Stop

1. **Presentation of Results**

**Java Code**

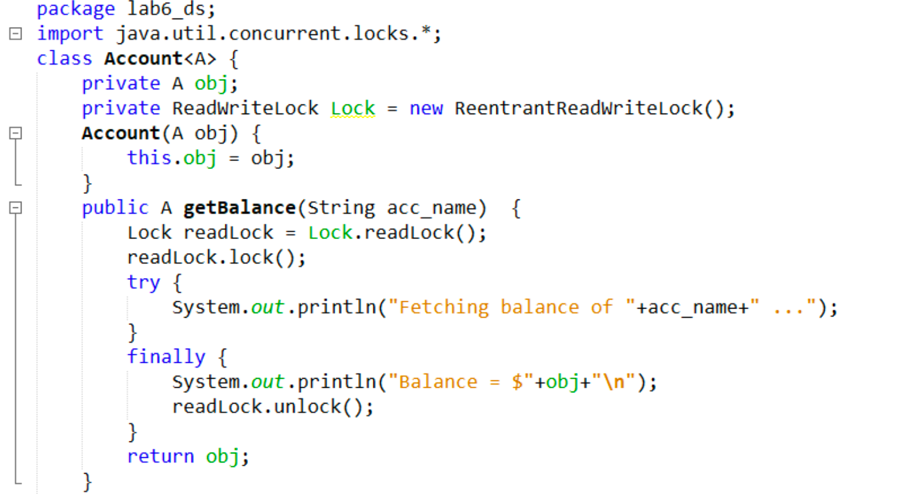
****

Figure 1 Java Program for the given problem statement

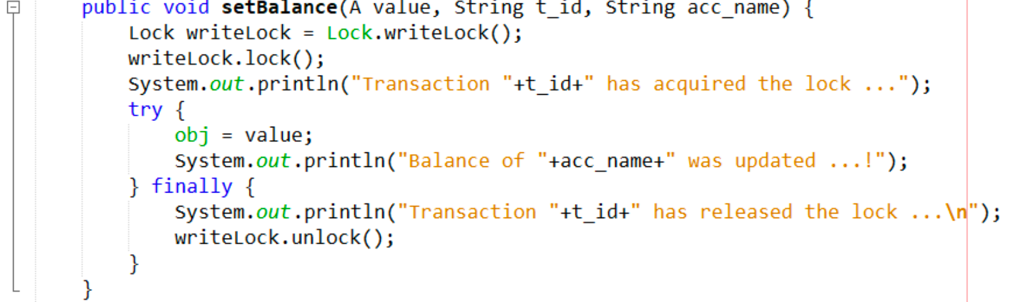
****

Figure 2 Java Program for the given problem statement (Continued)

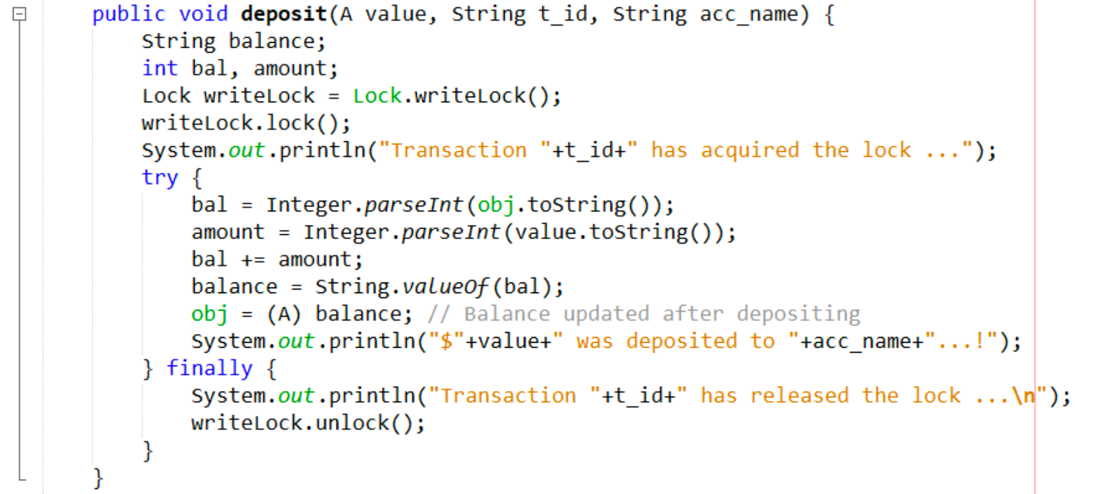
****

Figure 3 Java Program for the given problem statement (Continued)

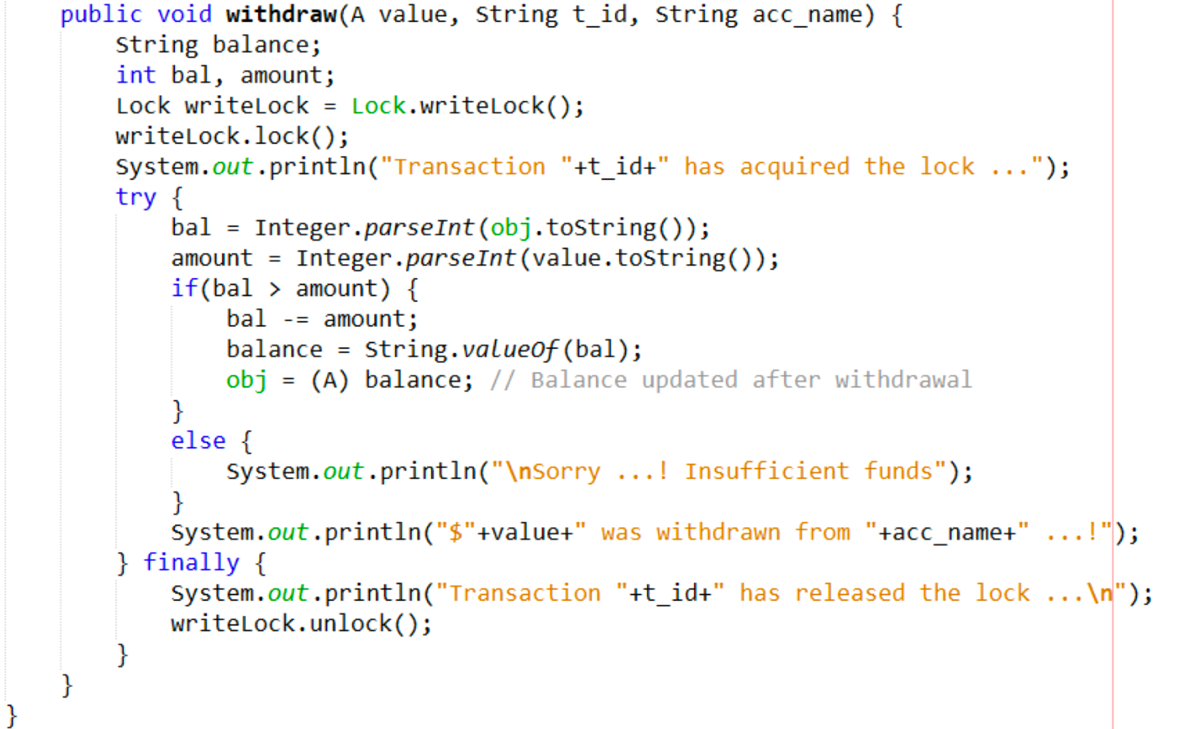
****

Figure 4 Java Program for the given problem statement (Continued)

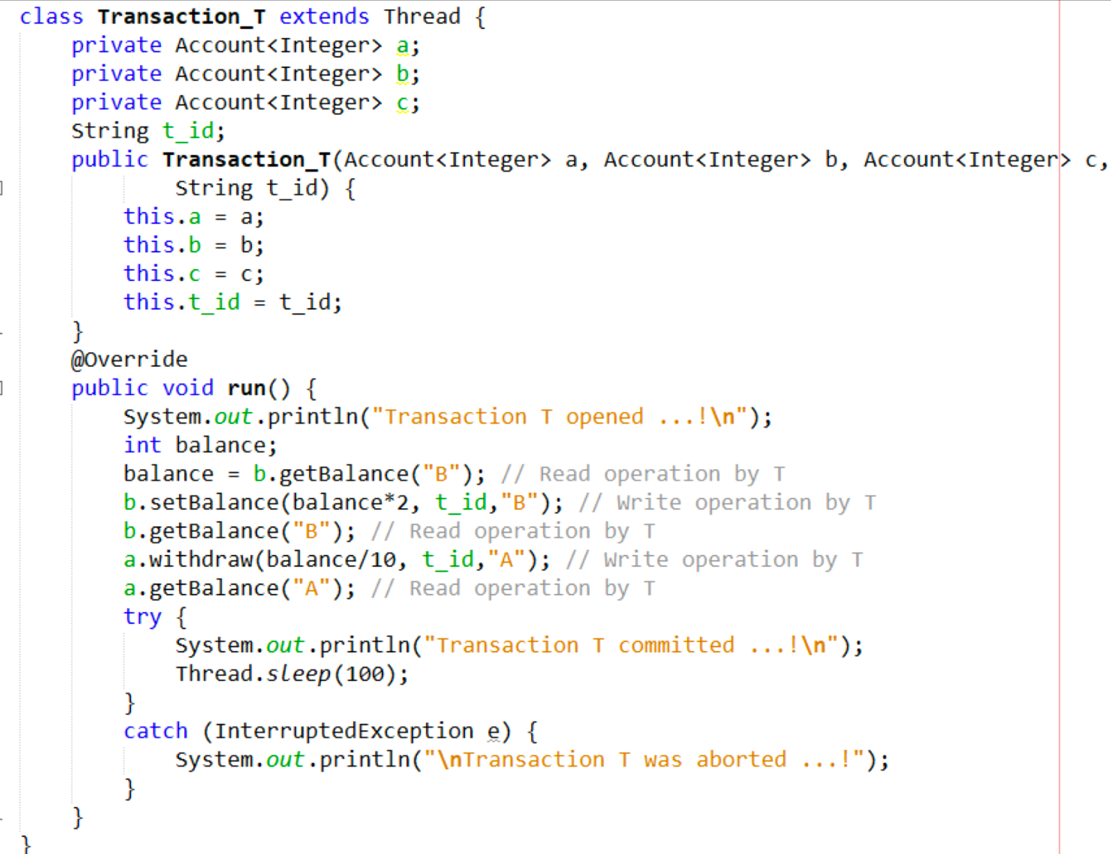
****

Figure 5 Java Program for the given problem statement (Continued)

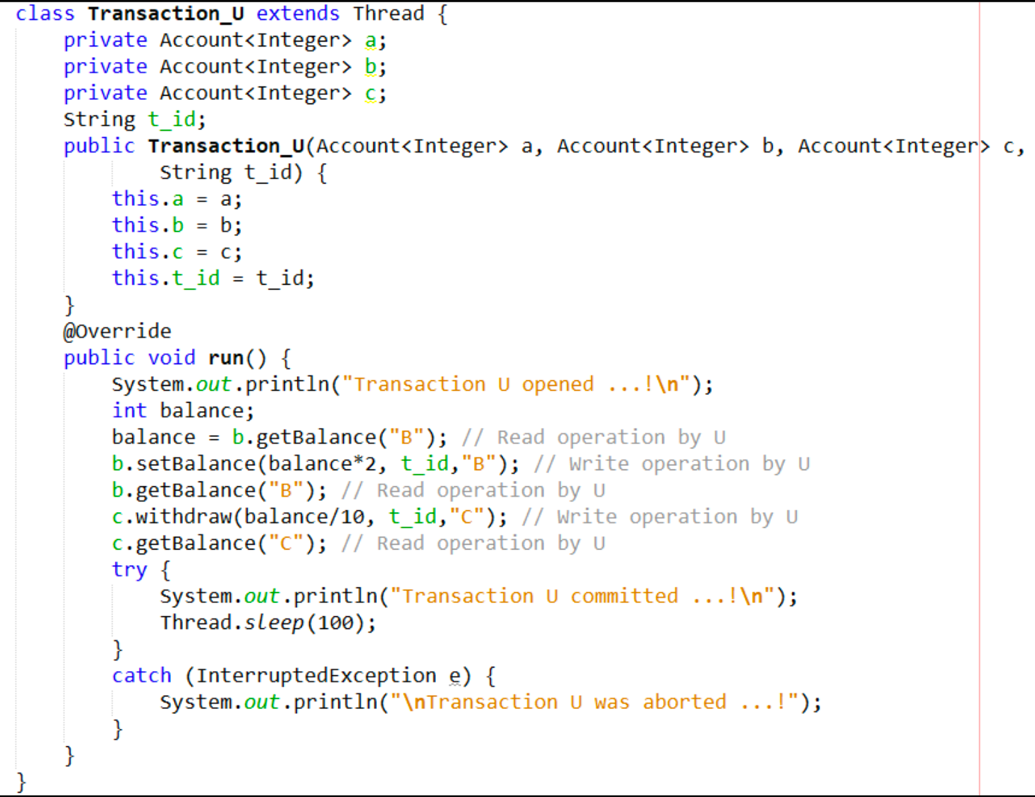
****

Figure 6 Java Program for the given problem statement (Continued)

**Java Output**

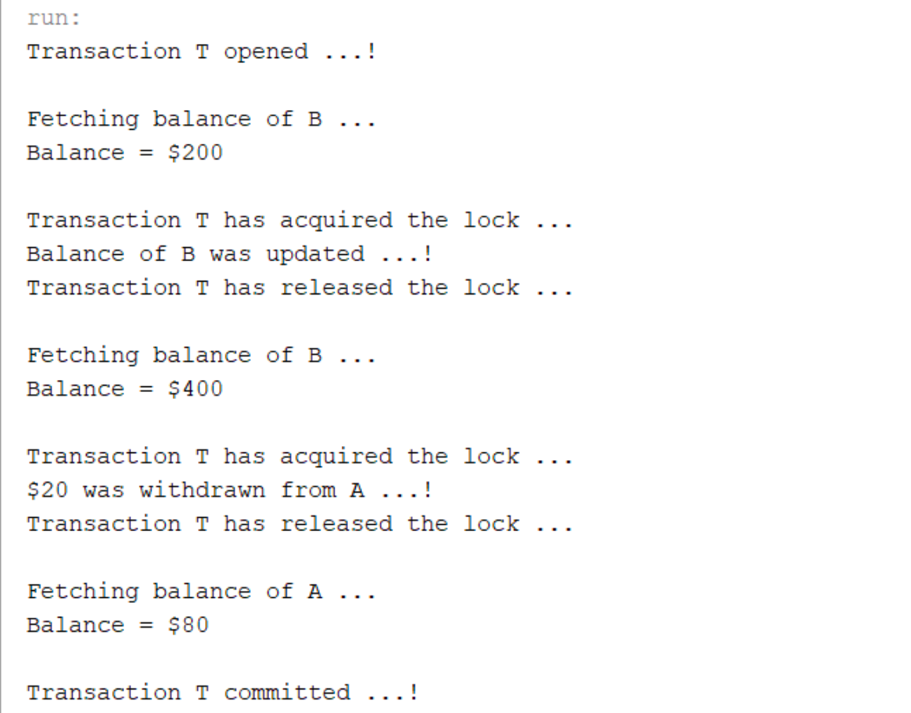
****

Figure 7 Java Program Output for the given problem statement

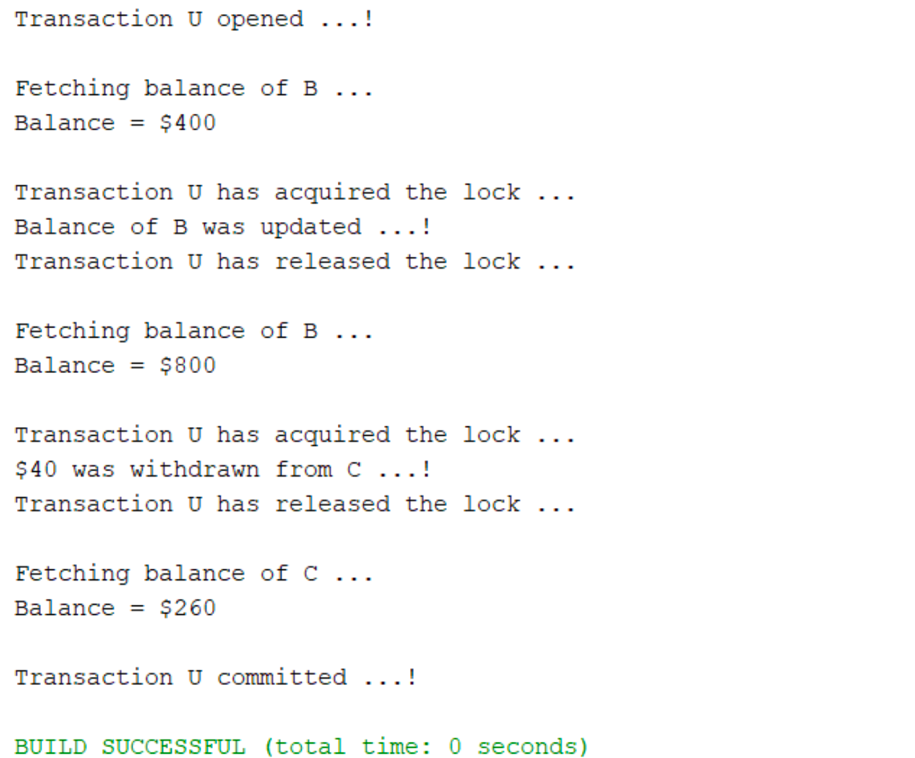
****

Figure 8 Java Program Output for the given problem statement (Continued)

1. **Analysis and Discussions**

* The variable “*a”, “b”* and *“c”* are the shared variables accessed by the two threads (transactions) concurrently and it must be synchronized by using exclusive locks to achieve serially equivalent operations.
* In any transaction-based system, thread contention is common where one thread is waiting for a lock/object that is currently being held by another thread. The waiting thread, thus cannot use that object until the other thread has unlocked that particular object.
* A server can achieve serial equivalence of transactions by serializing access to objects
* A simple example of serializing mechanism is the use of exclusive locks.
* In this locking scheme the server attempts to lock any object that is about to be used by any operation of a client’s transaction.
* If a client requests access to an object that is already locked due to another client’s transaction, the request is suspended and the client must wait until the object is unblocked.
* If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not write that object until T commits or aborts.
* To enforce this, a request for a write lock on an object is delayed by the presence of a read lock belonging to another transaction.
* If a transaction T has already performed a write operation on a particular object, then a concurrent transaction U must not read or write that object until T commits or aborts.
* To enforce this, a request for either a read lock or a write lock on an object is delayed by the presence of a write lock belonging to another transaction.