Laboratory 6

Title of the Laboratory Exercise: Transactions

1. Introduction and Purpose of Experiment

Aim and Objectives

Aim

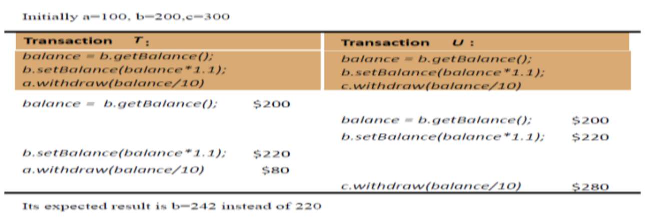
- To develop a program to perform concurrent transactions

2. Experimental Procedure

i. Analyse the problem statement
ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
iii. Implement the algorithm in Java language
iv. Compile the Java program
v. Test the implemented program
vi. Document the Results
vii. Analyse and discuss the outcomes of your experiment

3. 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.



Initially a=100, b=200,c=300

Transaction *T* **:**		**Transaction** *U* **:**	

balance = b.getBalance();
*b.setBalance(balance*1.1);*
a.withdraw(balance/10)
		balance = b.getBalance();
		*b.setBalance(balance*1.1);*
		c.withdraw(balance/10)	

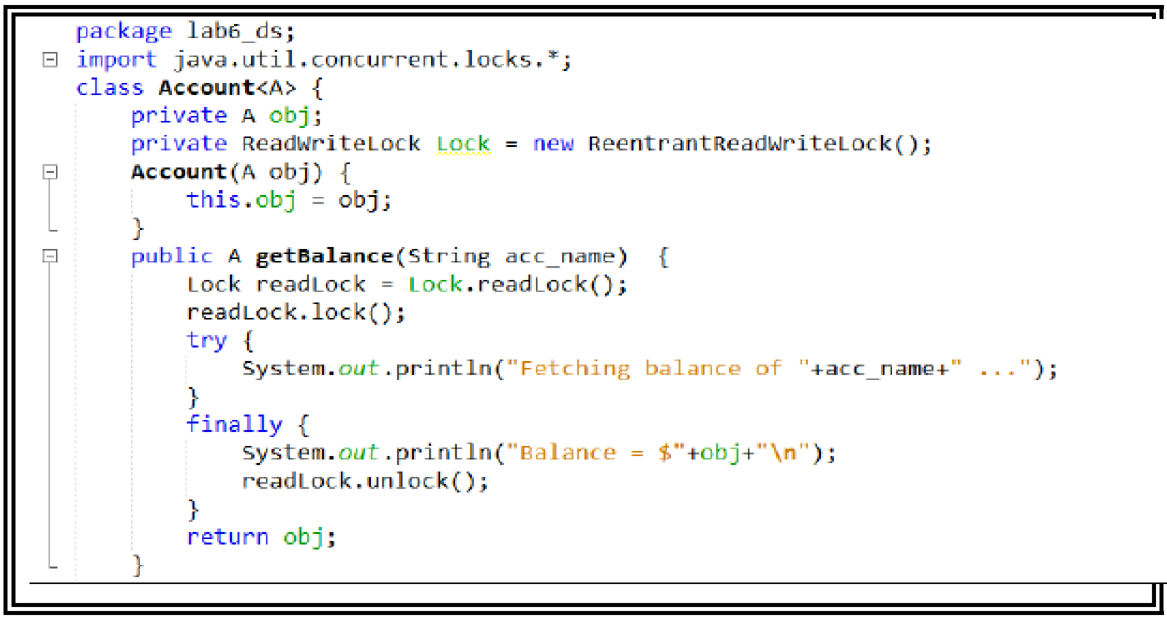
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Its expected result is b=242 instead of 220

4. Computations/Algorithms

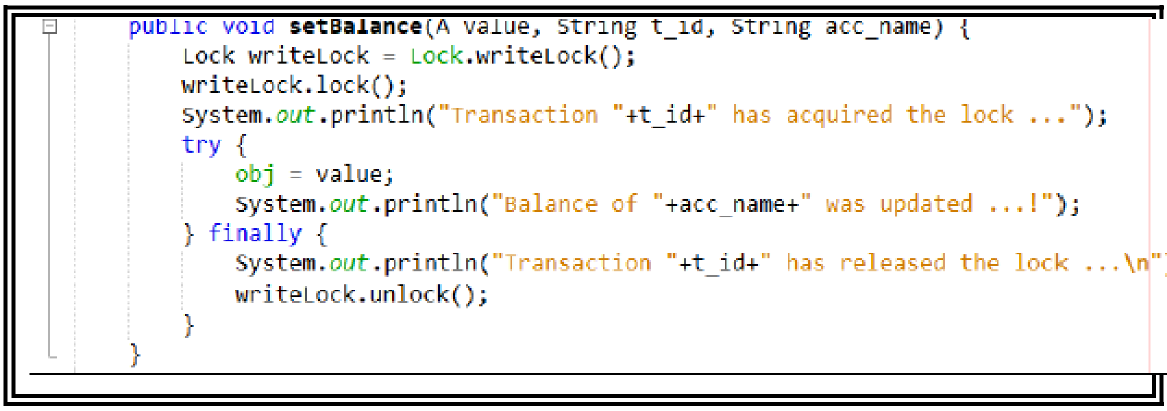
Step 1: Start
Step 2: Create a main class to declare thread objects and invoke each thread where, each thread denotes a Transaction
Step 3: Create a generic class Account that wraps an Integer as the underlying data type
Step 4: Create an object of Integer type and a Lock object of ReentrantReadWriteLock 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

5. Presentation of Results



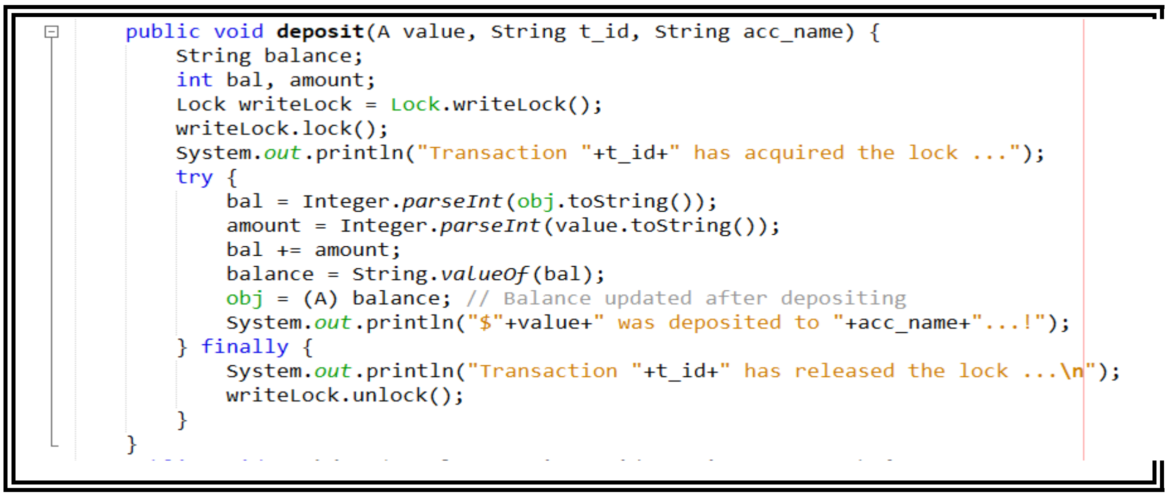
```java
package lab6_ds;
import java.util.concurrent.locks.*;
class Account<A> {
    private A obj;
    private ReadWriteLock Lock = new ReentrantReadWriteLock();
    Account(A obj) {
        this.obj = obj;
    }
    public A getBalance(String acc_name)  {
        Lock readLock = Lock.readLock();
        readLock.lock();
        try {
            System.out.println("Fetching balance of "+acc_name+" ...");
        }
        finally {
            System.out.println("Balance = $"+obj+"\n");
            readLock.unlock();
        }
        return obj;
    }
```

Fig 1 ***Account class with getBalance() - read operation***



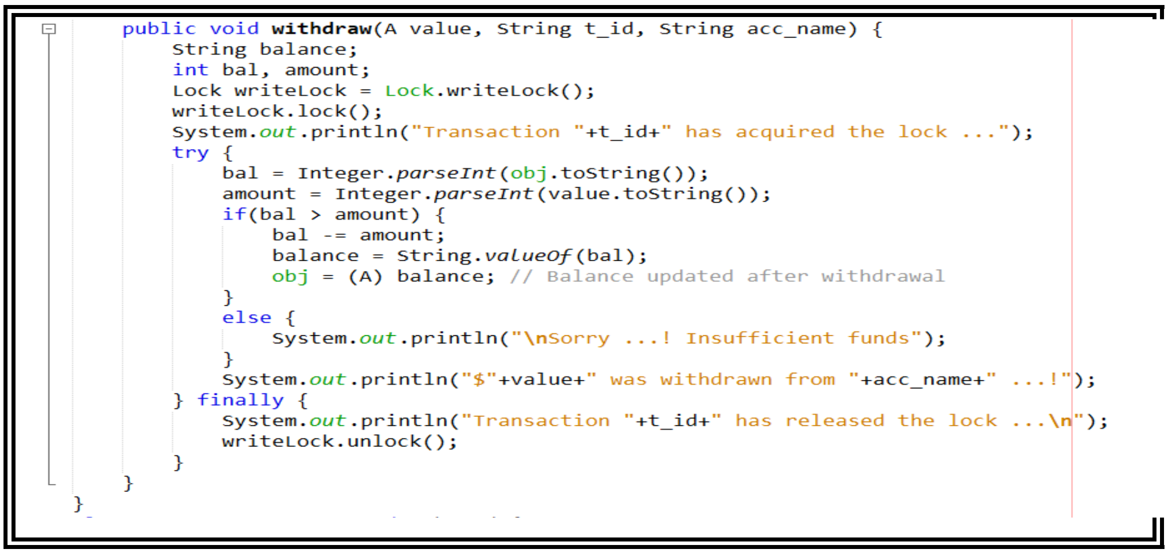
```java
    public void setBalance(A value, String t_id, String acc_name) {
        Lock writeLock = Lock.writeLock();
        writeLock.lock();
        System.out.println("Transaction "+t_id+" has acquired the lock ...");
        try {
            obj = value;
            System.out.println("Balance of "+acc_name+" was updated ...!");
        } finally {
            System.out.println("Transaction "+t_id+" has released the lock ...\n");
            writeLock.unlock();
        }
    }
```

Fig 2 ***Account class with setBalance() - write operation***



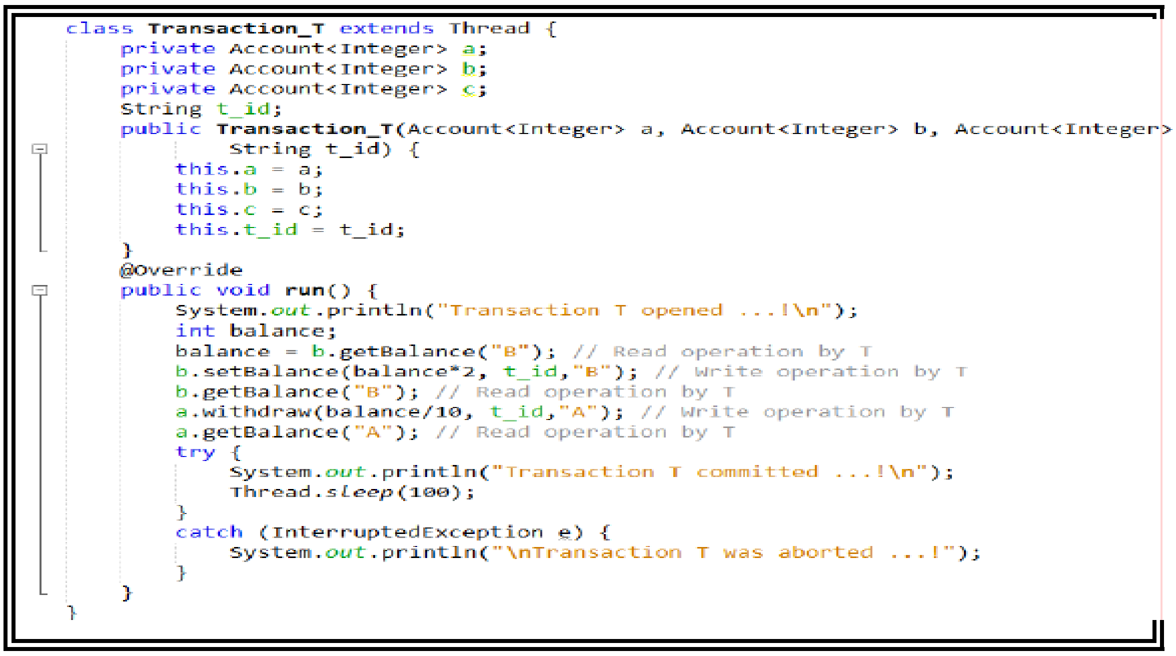
```java
    public void deposit(A value, String t_id, String acc_name) {
        String balance;
        int bal, amount;
        Lock writeLock = Lock.writeLock();
        writeLock.lock();
        System.out.println("Transaction "+t_id+" has acquired the lock ...");
        try {
            bal = Integer.parseInt(obj.toString());
            amount = Integer.parseInt(value.toString());
            bal += amount;
            balance = String.valueOf(bal);
            obj = (A) balance; // Balance updated after depositing
            System.out.println("$"+value+" was deposited to "+acc_name+"...!");
        } finally {
            System.out.println("Transaction "+t_id+" has released the lock ...\n");
            writeLock.unlock();
        }
    }
```

Fig 3 ***Account class with deposit() - write operation***



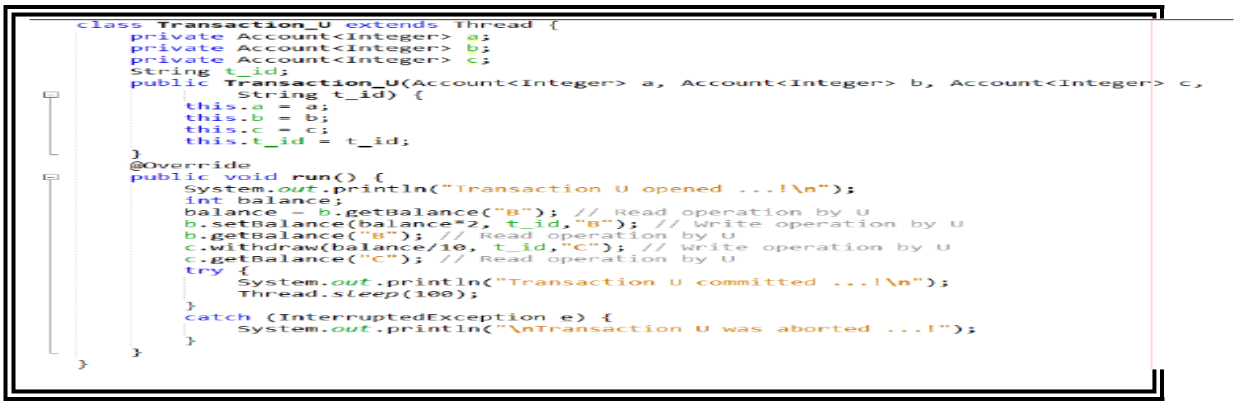
```java
    public void withdraw(A value, String t_id, String acc_name) {
        String balance;
        int bal, amount;
        Lock writeLock = Lock.writeLock();
        writeLock.lock();
        System.out.println("Transaction "+t_id+" has acquired the lock ...");
        try {
            bal = Integer.parseInt(obj.toString());
            amount = Integer.parseInt(value.toString());
            if(bal > amount) {
                bal -= amount;
                balance = String.valueOf(bal);
                obj = (A) balance; // Balance updated after withdrawal
            }
            else {
                System.out.println("\nSorry ...! Insufficient funds");
            }
            System.out.println("$"+value+" was withdrawn from "+acc_name+" ...!");
        } finally {
            System.out.println("Transaction "+t_id+" has released the lock ...\n");
            writeLock.unlock();
        }
    }
}
```

Fig 4 ***Account class with withdraw() - write operation***



```java
class Transaction_T extends Thread {
    private Account<Integer> a;
    private Account<Integer> b;
    private Account<Integer> c;
    String t_id;
    public Transaction_T(Account<Integer> a, Account<Integer> b, Account<Integer> c,
            String t_id) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.t_id = t_id;
    }
    @Override
    public void run() {
        System.out.println("Transaction T opened ...!\n");
        int balance;
        balance = b.getBalance("B"); // Read operation by T
        b.setBalance(balance*2, t_id,"B"); // Write operation by T
        b.getBalance("B"); // Read operation by T
        a.withdraw(balance/10, t_id,"A"); // Write operation by T
        a.getBalance("A"); // Read operation by T
        try {
            System.out.println("Transaction T committed ...!\n");
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            System.out.println("\nTransaction T was aborted ...!");
        }
    }
}
```

Fig 5 ***Class Transaction_T showing all read-write operations***

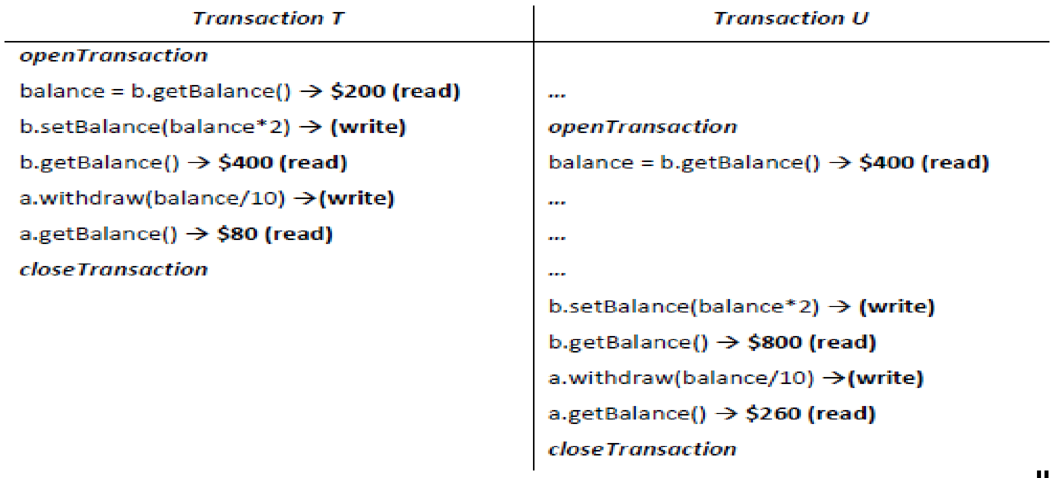


```java
class Transaction_U extends Thread {
    private Account<Integer> a;
    private Account<Integer> b;
    private Account<Integer> c;
    String t_id;
    public Transaction_U(Account<Integer> a, Account<Integer> b, Account<Integer> c,
            String t_id) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.t_id = t_id;
    }
    @Override
    public void run() {
        System.out.println("Transaction U opened ...!\n");
        int balance;
        balance = b.getBalance("B"); // Read operation by U
        b.setBalance(balance*2, t_id,"B"); // Write operation by U
        b.getBalance("B"); // Read operation by U
        c.withdraw(balance/10, t_id,"C"); // Write operation by U
        c.getBalance("C"); // Read operation by U
        try {
            System.out.println("Transaction U committed ...!\n");
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            System.out.println("\nTransaction U was aborted ...!");
        }
    }
}
```

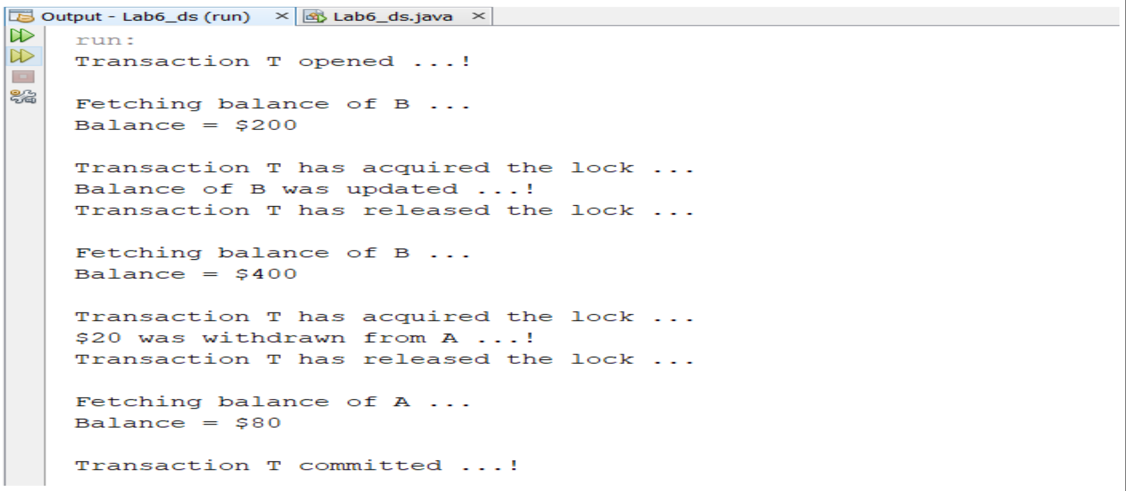
Fig 6 ***Class Transaction_U showing all read-write operations***

Output

***For* a = $100, b = $200, and c = $300 Initially**

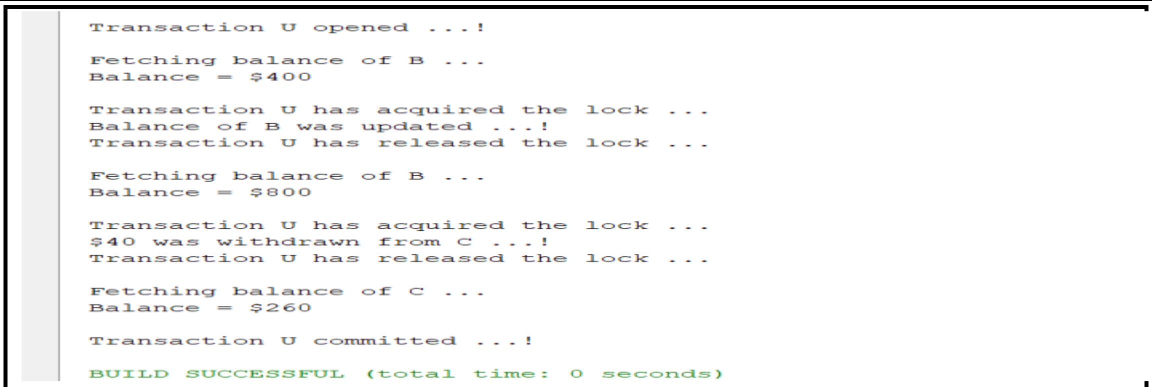


Transaction T	*Transaction U*
openTransaction	
balance = b.getBalance() → **$200 (read)**	**...**
b.setBalance(balance*2) → **(write)**	*openTransaction*
b.getBalance() → **$400 (read)**	balance = b.getBalance() → **$400 (read)**
a.withdraw(balance/10) →**(write)**	**...**
a.getBalance() → **$80 (read)**	**...**
closeTransaction	**...**
	b.setBalance(balance*2) → **(write)**
	b.getBalance() → **$800 (read)**
	a.withdraw(balance/10) →**(write)**
	a.getBalance() → **$260 (read)**
	closeTransaction



```
run:
Transaction T opened ...!

Fetching balance of B ...
Balance = $200

Transaction T has acquired the lock ...
Balance of B was updated ...!
Transaction T has released the lock ...

Fetching balance of B ...
Balance = $400

Transaction T has acquired the lock ...
$20 was withdrawn from A ...!
Transaction T has released the lock ...

Fetching balance of A ...
Balance = $80

Transaction T committed ...!
```

Fig 7 ***Output showing flow of operations in transaction T***



```
Transaction U opened ...!

Fetching balance of B ...
Balance = $400

Transaction U has acquired the lock ...
Balance of B was updated ...!
Transaction U has released the lock ...

Fetching balance of B ...
Balance = $800

Transaction U has acquired the lock ...
$40 was withdrawn from C ...!
Transaction U has released the lock ...

Fetching balance of C ...
Balance = $260

Transaction U committed ...!

BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 8 Output showing flow of operations in transaction U

6. Analysis and Discussions

• 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.

1. Limitations of Experiments

None

2. Limitations of Results

Checked only for Small Data Result May Vary if Checked for large data.

3. Learning happened

To develop a program to perform concurrent transactions

4. Recommendations

None