Project 1
Intermediate Report- Rigorous 2 Phase Locking protocol with wound-wait method for dealing with deadlock.

Team: **1001241701 Rohit Bhawal**
1001246133 Mark Ashley Fernandes

Programming Language: JAVA

Data Structures:
The following are the table structures that will be used to capture the transaction and locking information:

1. Transaction Table:- TransactionTable

Transaction id	Transaction Timestamp	Transaction State	List of items locked

2. Locking Table:- LockDetails

Item Name	Lock State	Holding Transactions ids	Waiting Transaction ids

The implementation of the simulation will make use of HashMap class of JAVA for storing and processing the transactions and the locks applied on the data items.

The HashMaps are as follows:

```
HashMap<Integer, TransactionDetails> TransactionTable = new HashMap<Integer,
TransactionDetails>();
HashMap<String, LockDetails> LockTable = new HashMap<String, LockDetails>();
```

The user define datatype **TransactionDetails and LockDetails** will consists of following declaration:

```
class TransactionDetails {
        String status; // store status like 'Active', 'Blocked', 'Aborted', 'Committed'
        int TID; // store Transaction ID
        int timestamp; // store the TimeStamp
        List<String> operationItemList = new ArrayList<String>(); // store operation with DataItem
}

class LockDetails {
        String Item; // store Data Item
        String operation; // store operation like 'RL' or 'WL'
        List<Integer> holdingTID = new ArrayList<Integer>();  //store Holding Transaction IDs
        List<Integer> waitingTID = new ArrayList<Integer>(); //store Waiting Transation IDs
}
```

Both the above classes have their respective setter and getter methods, such that retrieval and update on any element is made possible.

Implementation:

Step 1. Read from file
The program will read the Schedule from a text file line by line and will recognize the operation, transaction id, operation (read/write) and the data item.
This is done by using JAVA's split method to capture the elements of the operation as stated above.

Pseudo code:
Each line read is stored as a String s.
String operation[], transaction_id[], data_item[] splits s;

Such that – example: r1(Y); - brackets- () and semicolon are ignored.
operationFetch[]=r: stored as a String called ***operation***
idFetch []= 1: stored as an Integer called ***TID***
dataItemFetch []=Y: stored as a String called ***dataItem***
A method: performOperation(operation, TID, dataitem) which is implemented in **Step 2**.
Note: In the following pseudo code, highlighted defined methods are those that are called. Its definition in later pseudo codes.

Step 2. Identify the type of operation obtained from step 1 and perform the respective action.
Based on the switch statement, the respective methods are invoked.

Pseudo code:

```
performOperation(String operation, int TID, String dataItem){
        switch(operation)
        {
        case Begin:
                Call method: beginOperation(TID)- Refer Step 3

        case Read:
                Call method: readWriteOperation(TID, dataItem, Read)- Refer Step 4

        case Write:
                Call method: readWriteOperation(TID, dataItem, Write) Refer Step 4

        case Commit:
                Call method: commitOperation (TID)- Refer Step 5

        default:
                "invalid operation Detected"
        }
}
```

Step 3: Begin of transaction.

Class TransactionDetails uses HashMap- TransactionTable to store the following information: Transaction id, transaction timestamp, transaction status and List of items locked.

Note: In many cases, objects of Class *TransactionDetails or/*and *LockDetails* are created calling the respective parameterized constructors.

Pseudo code:

```
beginOperation(int TID)
{
        if TransactionTable.containsKey(TID)=false,
                transaction = new TransactionDetails(TID, activeStatus, timeStamp);
        else
                "Error, transaction is already started."
}

// Update the Transaction table
Class TransactionDetails
{
        TransactionDetails(int TID, String status, int timestamp)
        {
                this.TID = TID;
                this.status = status;
                this.timestamp = timestamp;
        }
}
```

Note: *Status* will be 'Active' for all new transactions. *Timestamp* will be incremented whenever a new transaction is read from input file. Initially set to 1.

Step 4: Operation detected is 'Read' or 'Write'

a. Determine whether the transaction is present in the Transaction table.
containsKey()- is used to perform this function.

b. Identify the type of lock(Read/Write) requested by current transaction.

c. Before placing a read lock, a check has to be made whether transaction state should be *Active.* *getStatus()* function of Class *TransactionDetails* is used for this purpose. Followed by a check to verify whether an Exclusive write lock on the data item is present or not.

d. If transaction is active, Check the current status of the data item by calling the method- *checkLockTable()*

e. If transaction is blocked, call method *operationItemList.add()*- add current operation with data item so that the operation can be resumed once the transaction becomes un-blocked/Active.
Other JAVA functions used.
put() and *get()* puts and returns the value of the transaction to which the specified key is mapped.

Pseudo code:

```
readWriteOperation(int TID, String dataItem, String operation)
{
        if TransactionTable.containsKey(TID)==false,
                "Transaction not started yet." And exit from method.

        If operation == "Read"                  //gets the current lock type applied on the data item
                then set lockType= readLock;
        else
                set lockType= writeLock;

        TransactionDetails transaction = TransactionTable.get(TID);
        switch(transaction.getStatus())
        {
        case activeStatus:
                Call method- checkLockTable(TID, dataItem, LockType)

        case blockStatus:
                //add this operation to a list to be resumed once transaction is unblocked
                transaction.operationItemList.add(operation + dataItem);
                TransactionTable.put(TID, transaction);

        case abortStatus:
                "Transaction is already aborted; hence operation is skipped"
        case committedStatus:
                "Transaction is already committed, error in transaction schedule order"
        }
}
```

Apply Wound-wait method for dealing with deadlock when an existing *'Write'* on data item exists and '*Read'* is requested.
getTimestamp() function of *TransactionDetails* is used to get the timestamp of the transactions that have detected a collision.

Pseudo code:

```
checkLockTable(int TID, String dataItem, String operation)
{
        LockDetails lockDetails = new LockDetails(dataItem, operation);
        TransactionDetails transaction = TransactionTable.get(TID);
        if( LockTable.containsKey(dataItem)==false)          //dataitem is not yet locked, then add
        {
                lockDetails.addHoldingTID(TID);
                transaction.addOperationItem(dataItem);
        }
        else                                  // dataitem is already locked, hence find what lock is placed
        {
                switch(operation)
                {
                case readLock:
                        //if readLock is already present, append TID to holding list on lock table
                        if(lockDetails.getOperation()==readLock)
                                then    lockDetails.addHoldingTID(TID);
                        else –data item has a write lock in Lock Table
                        {
                        If checkWoundWait(TID, Read+dataItem, lockDetails.getHoldingTID()) ==true
                        {
                                //then current transaction wounds holding transaction
                                lockDetails.setOperation(readLock);
                                lockDetails.holdingTID.clear();
                                lockDetails.addHoldingTID(TID);
                        }
                        else    //younger transaction added to waiting list
                                lockDetails.addWaitingTID(TID);

                case writeLock:
                        //if transaction wounds younger transaction, irrespective of the lock held
                        If checkWoundWait(TID, Write+dataItem, lockDetails.getHoldingTID()) ==true
                        {
                                lockDetails.setOperation(writeLock);
                                lockDetails.holdingTID.clear();
                                lockDetails.addHoldingTID(TID);
                                transaction.addOperationItem(dataItem);
                        }
                        else            //younger transaction added to waiting list
                                lockDetails.addWaitingTID(TID);
                }
```

```
        }
//Update Lock table and transaction table
LockTable.put(dataItem, lockDetails);
TransactionTable.put(TID, transaction);
}
```

checkWoundWait()- It compares the timestamp of the requested transaction with the one that has a lock on the data item.

Pseudo code:

```
Boolean checkWoundWait(int reqTID, String reqOperation, List<Integer> holdingTID)
{
        result= true;
        TransactionDetails reqTrans = TransactionTable.get(reqTID);
        //looping holding transactions from the set/list of transactions currently holding lock on item
        for(Integer holdTID: holdingTID)
        {
                TransactionDetails holdTrans = TransactionTable.get(holdTID);

                // requesting transaction is same as the transaction that has a lock
                if(reqTrans.getTimestamp() == holdTrans.getTimestamp()){
                        return result;
                }
                //Requesting transaction is older than holding transaction, abort the transaction
                if(reqTrans.getTimestamp() < holdTrans.getTimestamp())
                {
                        holdTrans.setStatus(abortStatus);
                        TransactionTable.put(holdTID, holdTrans);
                }
                else    //requesting transaction is younger than the transaction that holds the item
                {
                        reqTrans.setStatus(blockStatus);
                        reqTrans.addOperationItem(reqOperation);
                        TransactionTable.put(reqTID, reqTrans);
                        result= false;
                }
        }
return result;
}
```

Step 6: Operation detected is '*Commit*'

When current transaction encountered should release all the existing locks placed on all the data items. Furthermore, a check should be done to find out the transactions that were blocked due to the data items being locked by this transaction. Thus those transactions should *try* re-lock the data items such that there is no deadlock in the process.

Pseudo code:

```
commitOperation(int TID)
{
        If(TransactionTable.containsKey(TID)==false)
                "Transaction not started yet", return to read the next operation from file

        TransactionDetails transaction = TransactionTable.get(TID);

        switch(transaction.getStatus())
        {
        case activeStatus:
                //loop through all the operations in the stored operations list
                for operation to transaction.operationItemList
                {
                        String optSplit [] = operation.split("|");
                        releaseLock(TID, optSplit[1]);

                }
                transaction.setStatus(commitStatus);
                "Transaction Committed successfully."

        case blockStatus:
                "Transaction is already blocked and hence cannot commit"

        case abortStatus:
                "Transaction already aborted."

        case commitStatus:
                "Transaction already committed."
        }
}
```

releaseLock(TID, dataItem)- Given a transaction id, release all the data items held by it.

Pseudo code:

```
releaseLock(int TID, String dataItem)
{
        LockDetails lockDetails = LockTable.get(dataItem);

        //if transaction has already released the lock on the data item
        If lockDetails.holdingTID.contains(TID) ==false
                return;

        switch(lockDetails.getOperation())
        {
        case readLock:
                for i= to lockDetails.holdingTID.size()          //loop through lockDetails list
                {
                        If lockDetails.holdingTID.get(i) == TID
                        {
                                //release locks
                                lockDetails.holdingTID.remove(i);
                        }
                }
                If lockDetails.holdingTID.isEmpty()==true
                {
                        for  i to  lockDetails.waitingTID.size()
                        {
                                int nextTID = lockDetails.waitingTID.get(i);
                                lockDetails.waitingTID.remove(i);
                                lockType = getNextValidTransactionOperation(nextTID, dataItem);
                                        if (! lockType.isEmpty())
                                        {
                                        //transaction places lock on data item
                                        lockDetails.addHoldingTID(nextTID);
                                        lockDetails.setOperation(lockType);
                                        }
                                }
                        }
                }
        case writeLock:

                lockDetails.holdingTID.clear();
                //for the given item release write lock
                        For int i = 0 to  lockDetails.waitingTID.size()
                        {
                                int nextTID = lockDetails.waitingTID.get(i);
                                lockDetails.waitingTID.remove(i);
```

```
                        lockType = getNextValidTransactionOperation(nextTID, dataItem);
                        if (! lockType.isEmpty())
                        {
                                //Placing a lock on data item
                                lockDetails.addHoldingTID(nextTID);
                                lockDetails.setOperation(lockType);
                        }
                }
        }
}
```

getNextValidTransactionOperation()- To get the next valid operation from the waiting transaction list.

Pseudo code:

```
getNextValidTransactionOperation(int TID, String dataItem)
{
        TransactionDetails transaction = TransactionTable.get(TID);

        switch(transaction.getStatus())
        {
        case blockStatus:
                // Transaction Status changed from Blocked to Active
                transaction.setStatus(activeStatus);

                // Getting operation from Operation List of the transaction
                for i to transaction.operationItemList.size()
                        {
                        operation = transaction.operationItemList.get(i).split(delimeter);

                        //If operation is not Blank then read the operation and dataItem
                        if(!operation[0].isEmpty() && operation[1].equals(dataItem)){
                                if(operation[0].equals(Read)){
                                        lockType = readLock;
                                }
                                else{
                                        lockType = writeLock;
                                }

                // Since Operation will now get the Lock, removing the operation from Operation List
                                transaction.operationItemList.set(i, dataItem);
                                break;
                        }
                }
        }
```

performFinalCommit()- Is called towards the end, this is needed as the transactions

Pseudo code:

```
performFinalCommit(){
        Boolean Executed = Boolean.FALSE;
        Boolean [] doneTrans = new Boolean[TransactionTable.keySet().size()];
        finalTrans = Boolean.TRUE;

        for i =0 to doneTrans.length
                doneTrans[i] = Boolean.FALSE;

        while(!Executed){

                for(int TID : TransactionTable.keySet()){

                        TransactionDetails transaction = TransactionTable.get(TID);

                        If transaction.status==Active or transaction.status==Blocked
                        {
                                for(int i = 0; i < transaction.operationItemList.size(); i++){
                                        String opt = transaction.operationItemList.get(i);
                                        String [] operation = opt.split(delimeter);
                                        if(! operation[0].isEmpty()){
                                                String dataItem = "";
                                                if(operation.length >1){
                                                        dataItem = operation[1];
                                                }

                                                finalTransLockSuccess = Boolean.FALSE;

                                                performOperation(operation[0], TID, dataItem);

                                                if(! dataItem.isEmpty() && finalTransLockSuccess){
                                                        transaction.operationItemList.set(i,
delimeter+dataItem);
                                                TransactionTable.put(TID, transaction);
                                                }

                                                printTransactionTable();
                                                printLockTable();
                                        }
                                }
                        }
                        else{
                                doneTrans[TID-1] = Boolean.TRUE;
                        }
                }
```

```
                for i =0 to doneTrans.length
                {
                        If doneTrans[i]==true
                                Executed = Boolean.TRUE;
                        else
                                Executed = Boolean.FALSE;
                }
        }
}
```

The Log class handles writing to the output text file.

Pseudo code:

```
class Log{
        // Retrieve file name from Step 1

        void writeToFile(String data)
        {
                BufferedWriter outFile = null;

                try {
                        File file = new File(this.fileName);
                        outFile = new BufferedWriter(new FileWriter(file, true));
                        outFile.write(data);
                        outFile.newLine();
                        outFile.close();
                }
                catch (exception)
                        "Error"
    }
}
```