# CS44800 Project 4
# Two-Phase Locking
# Fall 2018

Due: November 29, 2018, 11:50PM
**Total Points: 9**

## Introduction

When multiple transactions are executing in a database environment simultaneously, it is important to guarantee that each transaction executes correctly without interfering with the execution of any other transaction. If two transactions are trying to simultaneously read and write on the same data item, it is possible for race conditions to occur or other concurrency synchronization issues that will result in an inconsistent database state.

One intuitive way to ensure consistency is through locking. An appropriate locking protocol such as Two-Phase Locking (2PL) ensures that no transaction can access records in a way that would result in an inconsistent state, while still allowing for a concurrent, non-serial execution of transactions. Please refer to the textbook (Fundamentals of Database Systems, 7th Edition, Elmasri Navathe, Addison-Wesley, 2015), Chapter 21, for more information about 2PL protocol or the readings in the Course Website.

In this assignment, you will be asked to implement:

1. A locking protocol using a lock table.
2. A log of all uncommitted transactions, recording operations the transaction has performed.
3. Deadlock detection and handling.

You will implement all of this in the **ExecuteSchedule()** method in the **CC.java** file. The **ExecuteSchedule()** method should return the state of the database after executing a schedule of transactions (as an array of integers) and should print the generated log to the console. You may implement whichever helper methods or additional code you need to satisfy these requirements, but you must complete this method.

The rest of this handout specifies in detail how to implement the requested functionality.

# Part 1: Locking Protocol

You will be given a schedule of operations for multiple transactions that will be executed concurrently. The syntax for this will be as follows:

`<TransactionID>:<Operation>`

The supported `<Operation>` are one of the following:
- `W(<RecordID>,<Value>)`
- `R(<RecordID>)`
- `C`

where:

`W` is the "Write" operation, which will write the given value to the given record, denoted by `RecordID`.

`R` is the "Read" operation, which will read the value from the given record.

`C` will commit the transaction, making all operations of the transaction permanent. After a commit operation finishes, you must ensure that all locks on a transaction are released.

You will be provided a List of Strings, where each String is a schedule for one transaction. All entries for the schedule of operations will be separated by a single semicolon. All data values will be integers, and the database will consist of 10 records, with record IDs ranging from 0 to 9 and initial values for each record being the same as the record ID. No insertion or deletion operations will be done, only updates.

Example:

```
T1:W(1,5);R(2);W(2,3);R(1);C
T2:R(1);W(1,2);C
```

where each line is one transaction

You must implement the 2PL locking protocol using shared and exclusive locks. You will maintain a lock table, recording the lock state of each record. The operations are as follows

- **Read**: When a read operation is encountered, attempt to acquire a **Shared Lock** on that record. A Shared Lock can be acquired only if there is not already an Exclusive Lock on that record from another transaction. If the current transaction already has either a Shared or Exclusive lock on that record, no lock needs to be acquired.
  Read the value of that record.

- **Write**: When a write operation is encountered, attempt to acquire an **Exclusive Lock** on that record. An Exclusive Lock can be acquired only if there is no other lock on that record from another transaction. If the current transaction already has an Exclusive Lock on that record, no lock needs to be acquired. If the current transaction already has a Shared Lock (and no other transaction has any other locks) you will upgrade the Shared Lock to an Exclusive lock.
  Store the new value in that record.

- **Commit**: When a commit operation is encountered, release all locks owned by that transaction.

To execute a schedule of transactions, execute operations one at a time in a round robin fashion. For example, for a schedule of three transactions, execute the first operation from Transaction 1; then the first from Transaction 2, then the first from Transaction 3, then the second from Transaction 1, and so on. Attempt to acquire a lock before executing each operation – if a lock cannot be acquired, then have that transaction wait and proceed with the next transaction. Check to see if the lock can be acquired the next time you process that transaction during the round-robin.

Example:
```
T1:W(1,5);C
T2:R(9);R(7);C
T3:R(1);C
```

The final ordering of operations in the schedule will be:
```
T1:W(1,5);T2:R(9);T1:C;T2:R(7);T3:R(1);T2:C;T3:C
```

# Part 2: System Log

You will maintain a log of uncommitted operations performed during the current run of the database transactions. **For the purposes of this assignment, this does not have to be persistent across different executions of the database as we do not have to worry about transaction recovery. You may simply store this in-memory as a List**. This log will allow transactions to be safely rolled back if they need to be aborted.

Whenever a Write operation is executed, you must write the corresponding log entry for that operation. Modify the above `Write()` and `Commit()` methods to complete this. You may create your own methods to write log operations and simply call these methods inside the `Write()` and `Commit()` methods if you wish.

When a **Write** operation occurs, store the following information:
- **Timestamp** (this can simply be a counter starting at 0, that can be incremented for each log entry)
- The **Transaction ID**
- The **Record ID**
- The **old value** stored in that record
- The **new value** stored in that record
- The **timestamp of the previous log entry for this transaction** (this will be useful for commit/rollback). For the first entry for a transaction, store **-1** in this value in the log
  If you have implemented this log as a list, note that this timestamp is equivalent to the index of the previous operation in the log in that list.

When a **Read** operation occurs, store the following information:
- **Timestamp**
- The **Transaction ID**
- The **Record ID**
- The **value read** from the database
- The **timestamp of the previous log entry for this transaction**

When a **Commit** operation occurs, store the following information:
- **Timestamp** (this can simply be a counter starting at 0, that can be incremented for each log entry)
- The **Transaction ID**
- The **timestamp of the previous log entry for this transaction**

It is also recommended that for each transaction you maintain a pointer to the most recent log entry for that transaction (again, this will be useful for commit/rollback).

Note that you should add entries to the log **only as they are successfully completed**, not when they are encountered in the schedule. If an operation must wait for a lock, then do not add it to the log until the operation actually executes.

An example log for the following schedule generated from the first example:

```
T1:W(1,5);T1:R(2);T1:W(2,3);T1:R(1);T1:C; T2:R(1);T2:W(1,2);T2:C;

W:0,T1,1,1,5,-1
R:1,T1,2,2,0
W:2,T1,2,2,3,1
R:3,T1,1,5,2
C:4,T1,3
R:5,T2,1,5,-1
W:6,T2,1,5,2,5
C:7,T2,6
```

At the end of the **ExecuteSchedule()** method, you should print the contents of the log to the console or to a file. You do not need to format your log entries in exactly the order specified here, but they must contain all the requested information. **If you format your log entries in a different order than specified here or print it to a file rather than the console, please include a README file describing the format of your log entries and where to find the log.**

This log will be used to help determine credit in the case of mistakes in the locking protocol or deadlock resolution, so it is important to ensure that it is easy for us to read.

# Part 3: Deadlock Detection/Handling

You will handle deadlocks by constructing a **wait-for graph** and detecting cycles in the graph. If a cycle is detected, you will resolve the deadlock by picking a transaction to abort, rolling back any changes that transaction has made, and release all its locks.

To construct a **wait-for graph**, create one node for each transaction. Whenever a transaction X must wait for a lock held by another transaction Y, draw a directed edge from X to Y.

In order to detect deadlocks, run a graph search algorithm such as DFS to detect if a cycle exists in the graph. If a cycle does exist, then the nodes that form that cycle are in a deadlocked state.

There are several approaches that can be taken in order to resolve deadlocks. For this project, **we will always abort the transaction with the lowest priority**. We can determine priority based on the timestamp of when a transaction entered the system – the older a transaction is, the higher its priority.

For this project, the Transaction ID can be thought of as equivalent to this transaction – a higher transaction ID means the transaction has entered the system at a later time. Therefore, **abort the transaction in the deadlock with the highest transaction ID (T1 < T2 < T3 < …)**.

In order to perform this abort, first add an Abort entry to the transaction log. This should be similar to the Commit entry, except that it should denote an Abort. Example:

    `A:7,T2,6`

would denote that at time 7 we are initiating an abort of Transaction 2.

Before resolve any deadlock, **find all the cycles in the wait-for graph**. Then, choose the conflicting transaction with the lowest priority and add the abort entry to the log (as described above). Next**, rollback all changes the transaction has made to the database state**. This can be done by tracing back through the log entries, starting from the most recent. For each Write entry encountered, restore the state of the database before that operation (using the old value stored in the log).

Note that since we only release locks once a transaction is completely finished, we will never have to worry about cascading aborts from other transactions that might have read this updated value as it is not possible for another transaction to read uncommitted data.

**Finally, remove the aborted transaction from the wait-for graph and see if a deadlock still exists. If it does, repeat the process until no deadlock remains.**

You do not have to re-execute the transaction after it has been aborted. Simply proceed with the round-robin execution as normal.
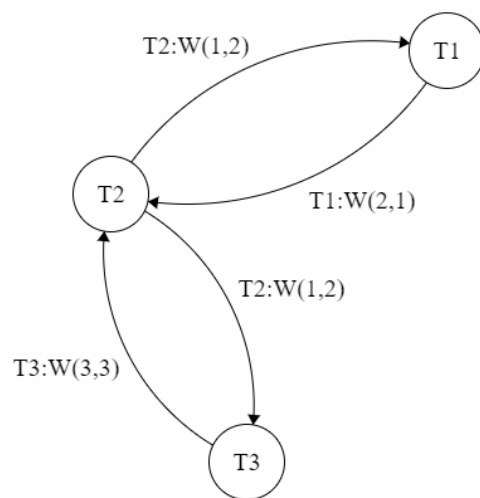
For example, assume that we have the following schedule of transactions:

```
T1: R(1);W(2,1);C
T2: R(2);R(3);W(1,2);C
T3: R(1);W(3,3);C
```

Ordering of the operations (round-robin):

```
T1:R(1);T2:R(2);T3:R(1);T1:W(2,1);T2:R(3);T3:W(3,3);T1:C;T2:W(1,2);T3:C;T2:C
```

The wait-for graph after `T2:W(1,2)` will be:



At this point, we have two cycles. The transaction with the lowest priority is T3. Hence, T3 is selected to be aborted. After rollback all operations of T3, we still have another cycle. Therefore, T2 is selected to be aborted.

## Additional Notes

Although in practice concurrency control usually implies parallel execution of transactions, please implement the entirety of this project (execution of transactions, deadlock detection, transaction logging) in a **single-threaded architecture**. This ensures that the results will be deterministic and makes it easier to code and debug.

You must be able to handle an arbitrary number of transactions in a schedule, not just two.

The main method of the project is in **Project4.java**. To test, compile all your code and then run:

```
java Project4.java
```

Test cases exist in the **Project4** class. ExpectedOutputs.txt contains the expected output of these test cases. Feel free to write your own if you feel there are additional scenarios you would like to test.

---

## What to turn in

You only need to turn in file: **CC.java.** Rename the file to **CC_your-career-login.java**.. Do not put any logic or any changes in **Project4.java**, as this will be replaced for grading.

Include a README file including any necessary information for how to read your log, and how to compile and run your program.

**The file should be submitted on Blackboard.**