A Simple Tx Manager and Deadlock detector Implementation

Submitted by: Team 15 - Deepak Thipeswamy (1001235195)

Under Supervision of: Dr. Sharma Chakravarthy and GTA, Mr. Jay Bodra

Department of Computer Science,

University of Texas at Arlington

**Acknowledgements**

My sincere thanks to Dr. Sharma Chakravarthy who gave us brilliant lectures on Transaction Management and Concurrency Control before the project and throughout and without his guidance, implementing the project would have been really difficult.

Deepak Thipeswamy (1001235195)

**Overview**

This project implements a transaction manager (as part 1) that manages concurrency control using locking. You will implement the strict two-phase locking (S2PL) protocol with shared locks for read and exclusive locks for write. The transaction manager handles locking and releasing of objects. Lock escalation (Upgrading) is not considered in this project. In part 2, you will implement the wait for graph using information from the lock table, detect deadlocks by checking for cycles in the graph, and choosing a victim transaction to abort. The transaction manager handles locking and releasing of objects. Lock escalation (Upgrading) is not considered in this project.

**Overall Status**

The Transaction Manager and Deadlock Detection is implemented **completely** in the method skeleton provided. The major components of the implementation and their implementation details are as follows:

* **Transaction Manager**

Transaction Manager is implemented based on the understanding of the hash table and the data structure of the of the transaction manager. Based on the input read from the input files, zgt\_test.C has the implementation to take it through the different methods of zgt\_tm.C. For every input like Begin, read, write there will be different threads created and will be running concurrently to access the object desired. Once the threads are created, it calls the base methods in zgt\_tx.C to do the actual operations.

Important methods are begintx, committx, aborttx, readtx, writetx, setLock. Begintx, starts the transaction. When there is a request for read/write then, the transaction manager will check whether the transaction exists and if the transaction exists and requested object is not held by any other transaction, then the transaction manager gives the access to that object by changing the value of the object number. Reduces by 1 for read and increases by 1 for write. If it is held by any other transaction then it makes the current transaction is made to WAIT (Deadlock might happen). All atomic operations are performed after locking the transaction by calling **zgt\_p(0)** and it is released using **zgt\_v(0)** after completing all operations**.**

Commit and abort does the operation as the name signifies by releasing all the resources the transaction is holding, calls a **zgt\_v** operation on the semaphores which are found by calling **zgt\_nwait** (all waiting transactions get back to race condition to get a lock on the resource the transactions wanted to acquire) and finally removes the transaction from the hashtable.

One of the important components of the transaction manager is to set the lock on any given object. Set lock basically also has a call for deadlock detection and upon solving the deadlock situation, set lock is called recursively to try and acquire the lock again. The hash table is the base for visualizing the current status of all the threads running concurrently.

* **Deadlock Detection**

Deadlock implementation is based on the instructions provided as a comment on the method signature. It begins with creating a wait graph for each of the transactions and traversing through the dependents. Along the way, we have to mark the transactions to WHITE, GREY and BLACK (level = -1, 0, 1 respectively). This can be identified be calling the “visited” method and finding the level where the transactions currently are in each of the wait graphs. If the transaction is visited more than once in a given wait for graph, then the cycle exists. The transaction to be aborted is chosen by calling the method “choosevictim” and abort the chosen victim transaction.

Traversal of the wait graph is the most important part for the deadlock detection. First, owners of the lock have to be found which are waiting. If the same transaction is visited and the location is not NULL (check levels), then the deadlock exists. Abort transaction immediately.

If that is not the case then, create a temporary node, set pointers and assign last to temp node, mark on the wtable., find the transaction in hashtable with the same tid and set the level to 1 by using the same object number as head from the first traverse iteration. Call traverse recursively by finding out from the list of transaction by checking the status as W and also with the same level as the temp node. Use the same values to assign it to temp node.

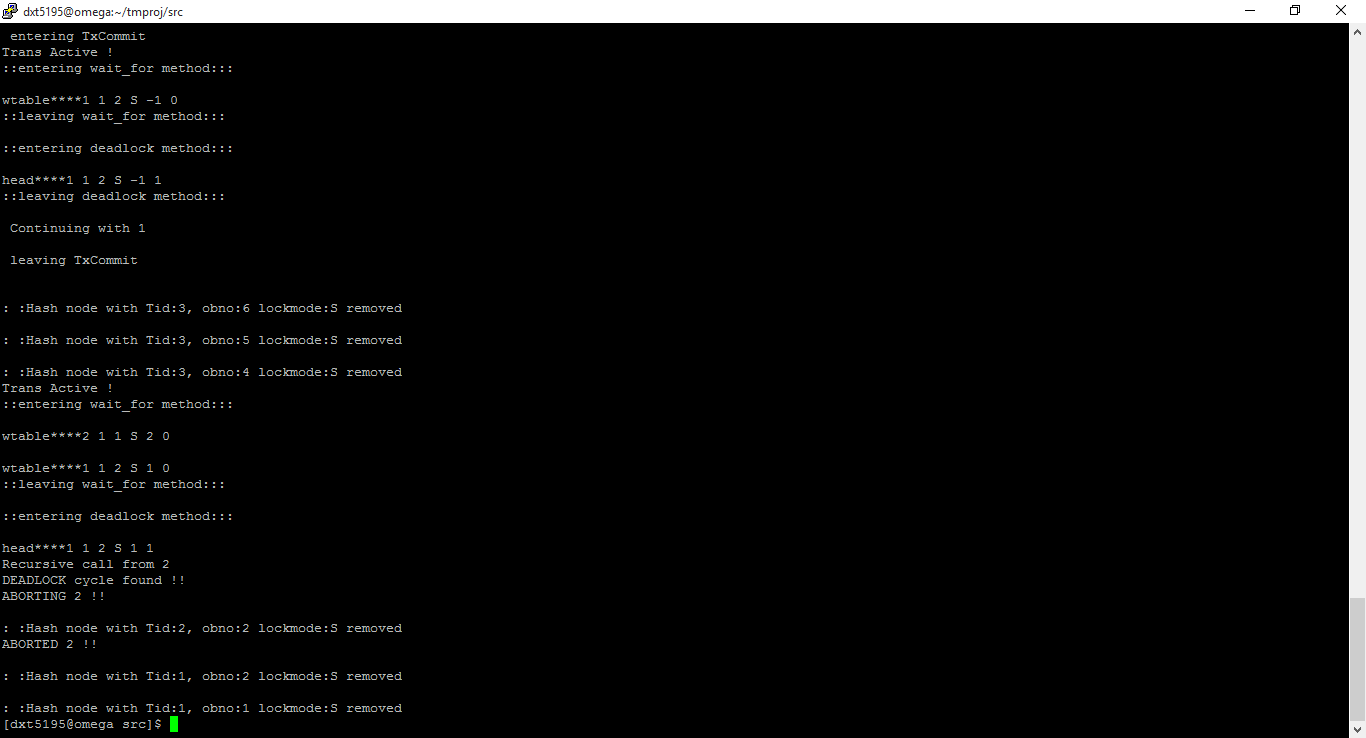
**Where I encountered difficulty**

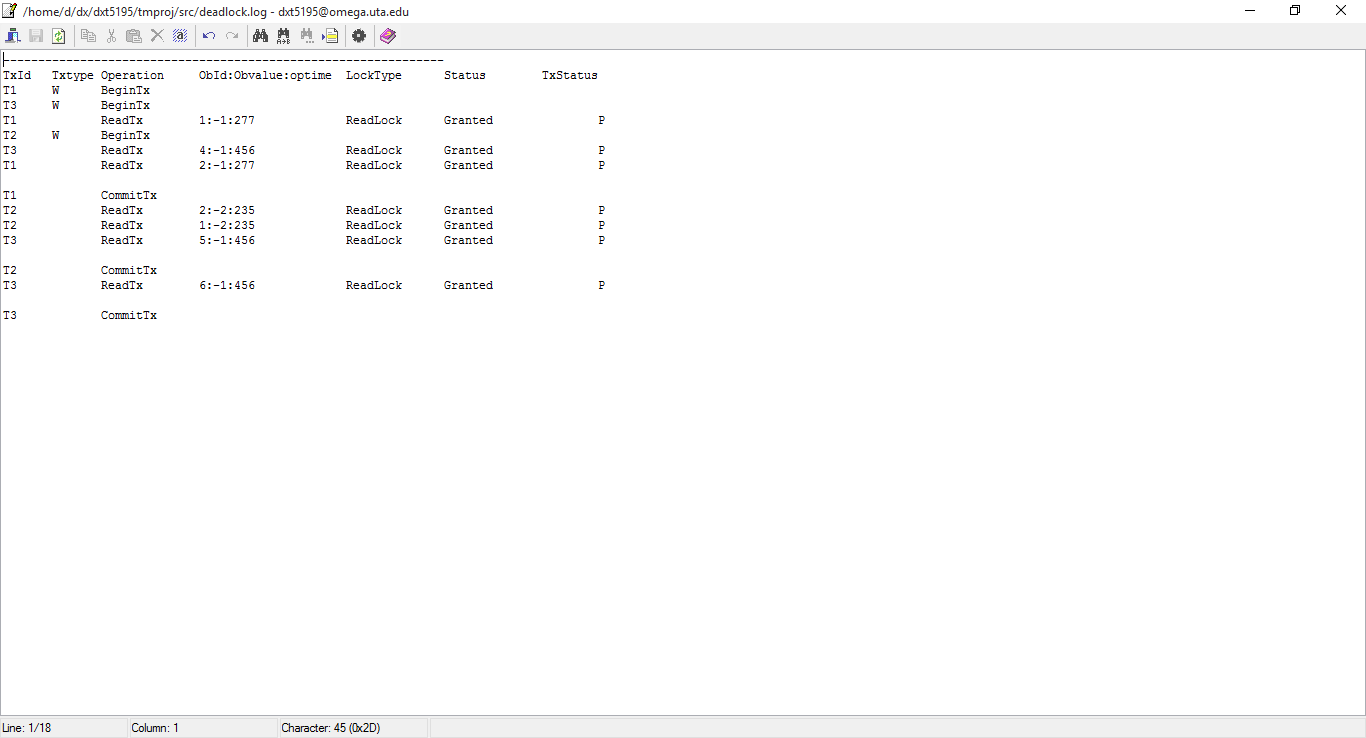
Many number of discussions with Professor helped me get through the understanding of components much faster than expected. Basically, a demo of existing code functionality and also data structures was desired.

The part where I had difficulty is not being able to debug through an IDE. That would have helped me understand the data structure in a better way and also lack of documentation on the existing code make me take a lot of time to understand functionalities only by using print statements.

**Test Cases**

All the test files provided are test against the code and below is the screenshot of deadlock scenario and also how the code resolves the deadlock.





**Division of Labor**

As there was a minimal documentation on this project and also it took a lot of time to get used to the code in C++. The skeleton had to be studied end to end to understand how to look in the perspective of a Transaction Manager. Applying the approach in comments section (before traverse method starts) and the algorithm provided in the pdf together took some time. Hence the total number of hours spent in implementation consumed: **~40hrs**.

**Logical Errors**

* Semaphore: By releasing just the transaction will not make way for the other waiting transactions. The program has to find out all the waiting transactions and in a loop, the program has to call a V operation on that semno. The reason being, that the same transaction maybe holding more than one object and also different objects. So releasing all the waiting ones will let the other transactions race for the available resources.
* Calling the deadlock code at the right moment is crucial, the transaction cannot by pushed to a Wait state and then expect the deadlock to be solved, first the current scenario has to be analyzed for a potential deadlock scenario and abort the victim transaction and then attempt to lock again in a recursive manner. Hence deadlock is called before the setLock is called recursively.
* The tricky part was to make sure that the traverse actually considers 3 different levels and also constructing the wait graph for all possible waiting transactions which might enter to a deadlock. Rearranging the pointers to make a link list act like a graph where a cycle is detected only by matching its properties.