|  |
| --- |
|  |

|  |
| --- |
| **IMPLEMENTATION OF TRANSACTION MANAGER** |
|  |
|  |
| This project implements transaction manager that manages the concurrency control using locking |
|  |
| Submitted By: |
| **Akshay Sarkar - 1001506793 Remesh Sreemoolam Venkitachalam – 1001414827** |
|  |

**Table of Contents**

Table of Contents

**Overall Status2**

Method Description ZGT\_TX2

* readTX……………………..…………………………………………………………………………………………………………………2
* writeTX………………………..………………………………………………………………………………………………………..……2
* abortTX………………………………………………………………………………………………………………………………….……2
* commitTX……………………………………………………………………………………………………………………………………3
* do\_commit\_abort………………………………………………………………………………………………………………….……3
* set\_lock………………………………………………………………………………………………………………………………………3
* perform\_readWrite ……………………………………………………………………………………………………………………4

**Project Difficulty4**

**File Description 4**

**Division of Labor4**

**Logical Errors 5**

**Implementation of a Transaction Manager**

**Overall Status**

In this project, we have asked to implement one of the lower levels of DBMS – the Transaction Manager which will be called by higher layers to handle concurrency control. For this purpose we have implemented Strict Two-Phase Locking (S2PL) protocol where an exclusive lock is assigned for write operations and a shared lock is assigned for read operations. Hence the Transaction Manager handles locking and releasing objects as and when necessary. It consists of a main thread that handles the input, initializes the necessary data structures and a pool of mutex and condition variables to ensure that operations belonging to the same transaction are executed in proper order. The main thread creates a transaction manager object and the associated hash table that acts as a lock table. Also each operation statement is converted to thread inside zgt\_tm.c, so that concurrency could be achieved.

In zgt\_tx.c file, we are writing codes for following methods:-

* void \*readTx(void \*arg) :

In this method, we are looking to acquire the shared lock and find the object that the current transaction needs. In case the transaction is aborted, no further operation is performed. Else we send the transaction to set\_lock method, and we look for the object in hash table. Finally, when the set\_lock method returns a true value, we are sure that we have the shared lock. Now we perform read operation. We also try to handle the corner cases and put the error statements on log.

* void \*writeTx(void \*arg) :

This method is similar to readTx method as above. We are acquiring the critical lock and find the object for the current transaction pointer. In case the transaction is aborted, no further operation is performed. Else we send the transaction to set\_lock method, and we look for the object in hash table. Finally, when the set\_lock method returns a true value, we are sure that we have the exclusive lock to do the write operation. We also try to handle the corner cases and put the error statements on log.

* void \*abortTx(void \*arg) :

In this method we just change the status of current transaction to abort and initiate a method do\_commit\_abort which internally handles the freeing of locks held by the transaction and remove transaction from Transaction Manager.

* void \*commitTx(void \*arg) :

Similar to abort method, in this method we just change the status of current transaction to commit and initiate a method do\_commit\_abort which internally handles the freeing of locks held by the transaction and remove transaction from Transaction Manager.

* void do\_commit\_abort() :

This method performs the operation of freeing up the lock and removing the transaction from the transaction manager. This method is used by both Abort() and Commit(). It also checks the wait queue if any other transaction needs to be invoked which is in waiting mode due to the current transaction. We retrieve the transactions that are waiting on semaphore using zgt\_nwait() and store it in a variable. Then we release all the semaphores one by one using zgt\_v() on the semaphore. This will allow those transactions to go for the released objects in case of an exclusive lock.

* int set\_lock() :

In this method, we decide when to grant lock to a transaction on a particular object.

We try to retrieve the transaction which is the object owner with the lock for the requested object and if it exists, we check further. Else, the current transaction can acquire the lock to the object. If some transaction already holds the lock, we need to check if it is the same transaction as the one that requires the lock. If it is, then it already has the lock granted to it. In case, the transaction is different, the lock can be acquired in Shared mode if and only if the transaction holding the lock is also in Shared mode. We also need to see if any other transactions are waiting on the present object to acquire it in Exclusive mode. If we overtake the Exclusive mode transaction, there could be a chance that that particular transaction waiting for Exclusive mode might have to wait infinitely, in case more and more Shared lock enquiries keep coming in. Also as the object is already held by some transaction, our current transaction needs to add its node to hashtable for the same object and point the HEAD of the transaction to this object node on the hashtable, if this is the first object it needs. If it already points to other objects, then it should iterate to the last object node for the particular transaction using NextP, and add the new object to the end of its object list.

In case, it is unable to acquire lock based on above conditions, this transaction has to wait. Its status is made to ‘W’, and it sets a semaphore on the waiting transaction,ie, if transaction 2 is waiting on transaction 1 for an object, the semaphore of transaction 1 becomes 1.

The above conditions are repeatedly checked in a loop as long as the lock is granted. When the lock is granted it returns to ReadTx or WriteTx from which it is called to complete the operation.

* void perform\_readWrite() :

Updates the objarray[] value by incrementing 1 for write and decrementing 1 for read.

**Difficulty**

Finding out the semaphore logic and flow of threads with respect to concurrency control concept was among the tedious tasks in our assignment.

Acquiring and releasing lock for maximum efficiency also needed certain consideration.

For multiple times execution the console showed no information, which made it difficult and almost impossible to debug.

Debugging code with logs overflowing in the console was a trouble due to thread concepts, it take us a while in started grabbing the flow of code.

Took time to understand that the value to be changed on read or write is in objarray[ ].

**File Description**

There were no new files added. We just needed to implement the above functions in zgt\_tm.C and zgt\_tx.C file and our goal was to implement those functions and execute all the test cases that were there in the project bundle.

**Division of Labour**

We took a couple of meetings together to understand how the transaction manager works and how we should plan to approach the problem. After the study we started looking into the functions given to us and created document future use.

Once we accustomed ourselves with the classes and multi-threaded in C++, we met to discuss

about the research and create a plan to formulate the algorithm. By coding together, we were able to help each other with the debugging of the program, and that made it easier to find flaws in the code. We coded for around almost the 10-15 Hour with total time including the preparation 30 Hour.

Akshay worked on Read and Aborts, while Remesh performed Write and Commits.

**Encountered Errors**

1. It was difficult to implement set\_lock function as it is incorporating a lot of conditions as per the algorithm and need to check state of all transaction related to that particular object before granting lock to the current transaction. Logical errors were encountered while writing the algorithm for lock acquiring, but attention was paid to the information on the document and we were able to correct any mistakes.
2. Release and holding semaphores were a little tricky. When a transaction ends we need to release semaphore so that any other transactions waiting for the object that the ending transaction holds, can now try to acquire that object. The releasing was a little confusing. But thankfully with clear explanations from TA in class, we were able to resolve that issue.
3. While working on Cygwin, placing g++ in the correct folder and updating the makefile was important. We were working on gcc and it caused troubles until we found that we were using the wrong compiler.
4. The multiple execution of the test cases shows no debug information. This is causing a block in pinpointing what might have gone wrong, or at which point execution has stopped.
5. As code is involving a lot of pointers, we found ourselves stuck at multiple places due to missing links or dead nodes.