CSE 5331

Introduction

The project was successfully COMPLETED with fully working zgt_tx.c and zgt_tm.c. When run, the code gives the desired output as required by this project. Appropriate log files such as S2T.log, C2Tsz.log, NOC2T.log, RR.log and deadlock.log are created carrying the required output. *Through this project, we can get a thorough understanding of how transactions are handled. *

1. readtx() -

Read transaction requires only a SHARED lock as no change is committed through a read action. In the event of an abort, operations are stopped. The transaction is then sent to the set_lock() method where the hash table is searched for the object. If the set_lock() method returns a true value, we can be sure that a shared lock has been obtained. The read operation can now be performed.

2. writetx() -

Write transaction requires an EXCLUSIVE lock on the object as changes might be inflicted on it and no other transaction is allowed to perform a write transaction on it. The transaction is then sent to the set_lock() method where the hash table is searched for the object. If the set_lock() method returns value as true, we can be sure that an exclusive lock has been granted and the write operation can be performed.

3. abortx() -

The status of the current transaction will be changed to abort and will be sent to the do_commit_abort() method where the freeing of locks that were held by the transaction will be removed by the transaction manager.

4. committx() -

The status of the current transaction will be changed to commit and will be sent to the do_commit_abort() method where the freeing of locks that were held by the transaction will be removed by the transaction manager.

5. Do_commit_abort() -

Locks are freed up and the transactions are removed by the transaction manager. Both abort() and commit() methods call on this method. It checks the wait queue if any other transactions are in waiting mode due to the ongoing transaction. We can obtain those waiting transactions by using the zgt_nwait() and store it within a variable. We can then release the semaphore one after the other by using the zgt_v() on the semaphore. Now, the active transactions will be allowed to go for the released objects in case of an exclusive lock.

6. set_lock() -

Granting of locks to transactions for a particular object is done in this method. If a particular object that is being requested already has a transaction that owns the object, we would have to perform further checks on it. If the object has no current owner, the current transaction requesting for that object may obtain a lock on it. If the same transaction requests a lock on an object it already owns, then the lock has already been granted to it. In the event of another transaction requesting for a shared lock on the same object, the lock shall be granted if the object is already being held by a transaction in shared lock 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 inquiries 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 the above conditions, this transaction has to wait. Its status is made to ‘W’, and it sets a semaphore on the waiting transaction. These conditions are repeatedly checked in a loop until the lock is granted and returns back to the readtx() or writetx() method that called it.

7. readwrite() -

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

File Description:

We added additional test case files to check serial schedule and interleaving schedules. Appropriate changes were made to the zgt_tx.c and zgt_tm.c files which were then run on the test-cases such as S2T.tx, C2Tsz.txt, RR.txt, deadlock.txt, NOC2T.txt that were provided to us.

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.

We individually went through the skeleton code and referred the textbook on this topic to get an accurate understanding of it. After that, we met up and discussed on how we could approach the issue and made a plan for it. We then began writing the logic behind the set_lock() method while parallelly planning out the other methods as well.

We met up thrice a week to work on the project together and made sure that our code passed the test-cases that was provided to us and the a few additional test-cases of our own to get a better understanding of the code.

Logical errors:

1. Using the wrong complier

We didn't notice that g++ is used instead of gcc. Fixing the issue on mac took time before the code could be run error free.

2. redefinition of 'semun' in zgt_semaphore.c

union semun` is defined by most platforms in their header files.(sys/sem.h)
This was causing compilation error on mac since the header file sys/sem.h already includes it.
Removing the reference to header files caused major issues so there were 2 options:

1. Check for the paltform and then use the semun
   #if defined (__FreeBSD__)
   #else

2. Use a different semun instance.

We opted for option 2 to avoid conflicts with other platforms and renamed semun to semun1 thus creating an alternate instance.

3. undefined reference to pthread_create and pthread_join (in linux distribution: Ubuntu 16.04)

This error was caused because -lpthread was used in the make file to make the executables. In linux distribution 16.04 (14.04 and above) -lpthread was replaced by -pthread to make it work. An alternative was to move all libraries behind the objects needing them but we went with first option.

4. Segmentation fault (code dumped)

We encountered this couple of times through the testing of our code. Major reason which caused this was we were trying to reference a memory that did not existed (dead nodes).

(We used valgrind tool to investigate the error and get the information for the cause.)

5. Encountered problems while releasing the semaphores when a transaction ends and waking up other transactions waiting for that object. After meeting the TA and understanding the concept we were able to resolve this.

6. We faced problem implementing the set_lock method as it involves a lot of conditions to be taken care of. We used VS Code C++ debugging tool to get the value at each step and check when the object can be acquired by other transaction.