CSE 4331/5331
Project 2
A Simple Tx Manager and deadlock detector Implementation

Sharma Chakravarthy
Information Technology Laboratory (IT Lab)
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76019
Email: sharma@cse.uta.edu
URL: http://itlab.uta.edu/sharma

Note on bb and deadline

- Deadline indicated on the project description is what you should go by
- If there is an extension, I will send an email explicitly indicating that
- Block board (bb) does not allow me to set separate deadlines for submission and with penalty
- What you see there is the deadline with penalty!

Overview

- Implementing a Tx Manager and a deadlock detector is project 2
- They are marked as step 1 and step 2 a you cannot test step 2 without finishing step 1
- If you prefer you can submit them as 2 separate submissions
- In which case, I suggest that you submit the first one by Oct 22nd.

Implement

- A transaction manager that is responsible for
  - Starting a Transaction (Tx)
  - Committing a Tx
  - Aborting a Tx
  - Performing read/write operations on items on behalf of a transaction
  - Acquiring necessary locks for performing operations (e.g., read/write)
  - Blocking transactions and continuing them when resources become available.

Basics

- You are given
  - Zgt_test.C
    - Implemented: accepts input and calls appropriate methods
  - zgt_tm (transaction manager class)
    - Partially implemented
  - zgt_tx (transaction class)
    - Need to be implemented (partially implemented)
  - Zgt_ht class // implements the lock hash table
    - Implemented, some extensions are NEEDED
  - Zgt_semaphore.C //does p and v operations
    - Implemented
- Hash table size (13, not hard-wired) and other constants are defined as well

Basics

- Input file format
  - // up to 3 words of comment
  - // only 4 tokens per line!
  - Log logfileName
  - BeginTx Txid R or W//begins a new transaction with Txid;
  - Read Txid item // Transaction Txid reads object item
  - Write Txid item //simulated as increment object value by 1
  - AbortTx Txid //aborts Txid and release all resources
  - CommitTx Txid //commits Txid and release all resources
  - item is an integer from 1 to MAX_ITEMS
  - TxId is an integer from 1 to MAX_TRANSACTIONS
  - Read and write are simulated by inc and dec operations + idling for some number of cycles to simulate computation

Input Example

```
// serial history
// 2 transactions
// same object accessed
// multiple times
Log S2T.log
BeginTx 1 W
Read  1 1
Read  1 2
Write  1 3
Write  1 4
read  1 1
write  1 2
write  1 4
write  1 4
commit 1
begintx 2 W
read  2 5
write  2 5
write  2 6
read  2 6
commit 2
```

In this project, you will not use the last input from line BeginTx

This shows 2 write txs. But it can be any combinations of read-only or read-write txs. If they are read-only Txs, then the same object can be operated by multiple read-only Txs.

Currently, the hash table has only one copy of an object. You need to extend it to have multiple copies if multiple read-only tx are operating onit.

Note that multiple write or red-only and write txs CANNOT operate on the same object at the same time.

What to implement

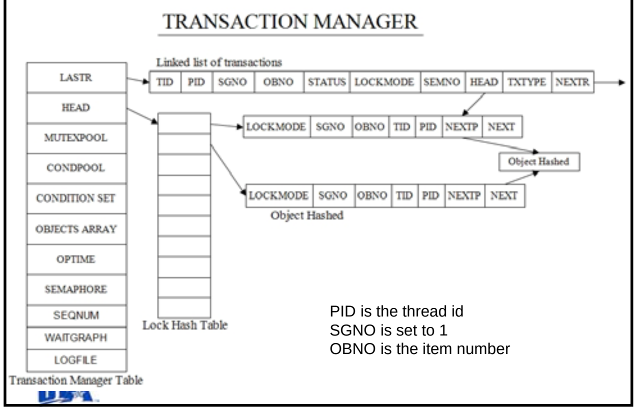
- The following five functions have to be implemented in zgt_tx.C. As needed, additional functions to support the above need to be implemented as well.
  - begintx(thrdArguments),
  - readtx(thrdArguments),
  - writetx(thrdArguments),
  - aborttx(thrdArguments),
  - committx(thrdArguments).
- The return type is void*
- Parameters need to be passed in a structure (param)
- Extensional to hash table to support multiple Read-only Txs.

Output

TxId	TxType	Operation	ObId:Obvalue:optime	LockType	Status	TxStatus
T1	W	BeginTx				
T1		ReadTx	1:-1:10	ReadLock	Granted	P
T1		ReadTx	2:-1:10	ReadLock	Granted	P
T2	W	BeginTx				
T1		WriteTx	3:1:10	WriteLock	Granted	P
T2		ReadTx	5:-1:863	ReadLock	Granted	P
T1		WriteTx	4:1:10	WriteLock	Granted	P
T2		WriteTx	5:0:863	WriteLock	Granted	P
T1		ReadTx	1:-2:10	ReadLock	Granted	P
T2		WriteTx	6:1:863	WriteLock	Granted	P
T1		WriteTx	2:0:10	WriteLock	Granted	P
T2		ReadTx	6:0:863	ReadLock	Granted	P
T1		WriteTx	4:2:10	WriteLock	Granted	P
T2		CommitTx				
T1		WriteTx	4:3:10	WriteLock	Granted	P
T1		CommitTx				

Read decrements the count and write increments the count; can check whether the computation is correct!

Overall Approach



PID is the thread id
SGNO is set to 1
OBNO is the item number

Flow and Tx states

- The main thread (in zgt_test.C) creates a transaction manager object and the needed hash table in the main or test program. There is only one transaction manager object. However, there will be one transaction object for each transaction, created by begin Tx input.
- Transaction states (reflected in the tx object)
  - TR_ACTIVE (P)
  - TR_WAIT (W)
  - TR_ABORT, (A)
  - TR_COMMIT (E)

Locking of objects

- An object is inserted into the hash table if a lock can be obtained for that object by that tx. The very presence of an object in the lock table indicates that that object is being used by a tx. Lockmode in the tx object indicates the type of lock a Tx is waiting for (S or X). TxType is used to indicate the type of the tx (R or W).
- All Txs are linked using lastr
- Head points to the hash table
- All objects within the same bucket are linked using next
- Head of Tx object points to the objects held by that tx as a list (using nextp of object)
- Semno in the tx object is used to make other txs wait for that tx on that semno (Tx k uses semno k)

Example

- For example, if Tx 1 is waiting on Tx 2 for object 6 for writing (X lock), then the tx objects will have the following information

Tid	Thrid	objno	lock	Txstatus	TxType	semno
2	2051	-1		P	W	2 //semno not -1 means someone is waiting
1	1026	6	X	W	W	-1 // -1 means is no one is waiting on this tx

Lockmode X indicates that item 6 is waiting for a W lock on T2, blank indicates initial value

Txstatus indicates that T1 is waiting and T2 is active

Txtype indicates the type of the transaction (R for readonly or W for read/write)

Deadlock

- For example, if Tx 1 is waiting on Tx 2 for object 6 for writing (X lock), then the tx objects will have the following information

Tid	Thrid	objno	lock	Txstatus	TxType	semno
2	2051	4	S	W	R	2
1	1026	6	X	W	W	1

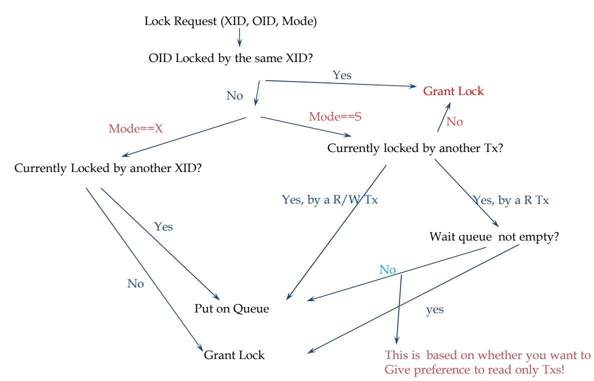
A deadlock can be formed by 2 or more Txs. Deadlock detection needs to detect these situations and abort one of the transactions participating in the deadlock to release all the locks held by that Tx. Then other Txs can proceed.

Since deadlocks can form any time a lock is requested, it needs to e checked periodically for every time a lock is requested.

Important

- An object is locked before operating on that object
- Lock table is used for that purpose
  - This is done by using one semaphore for the entire table
  - In this implementation sem (an attribute of TxMgr object) is an array of locks
  - Sem 0 is used for the lock table, sem k is used by Tx k
    - Sem 0 is initialized to 1 to allow first operation
    - Others sems are initialized to 0 as a p operation is done to make a Tx wait!
- **Hold a lock for the shortest duration**
- Never suspend or make a Tx wait while holding a lock
- Make sure all p operations have a corresponding v operation (irrespective of the conditionals and flow)

Handling a Lock Request (a la project 2)



Deadlock Detector

Basics

- You are given
  - zgt_test
    - Implemented: accepts input and calls appropriate methods
  - zgt_tm (transaction manager class)
    - Completely implemented it creates entry in the tm table and forks a thread
  - zgt_tx (transaction class)
    - Completely implemented and responsible for processing transaction.
  - zgt_ddlock //detects cycle
    - Partially implemented
  - zgt_ht class // implements the lock hash table
    - Completely Implemented
- Hash table size and other constants are defined as well

What to implement

- The following two functions have to be implemented in zgt_ddlock.C
  - Int deadlock(),
  - Int traverse(node* p) //recursive function
- The return type for deadlock is int
- You are given the data structures
- But if you want to use your own, you can do so as well
  - Need to add your own zgt_ddlock.h and zgt_ddlock.C

Node structure

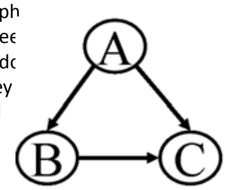
```
//defines the node for the waittable(wtable)
struct node {
long tid;
long sgno;
long obno;
char lockmode;
char Txtype; //sharma, oct 2014
int semno;
int level;
node* next;
node* next_s;
node* parent;
};
```

Wait_for class

```
//class wait_for which contains all the methods to
//detect deadlock
class wait_for {
node* head;
node* wtable;
int   found;
node* victim;
int visited(long);
node* location(long);
int traverse(node *);
node* choose_victim(node *, node *);
public:  int deadlock();
void print_waitTx();
wait_for(); ~wait_for(){};};
```


Cycle Detection

- Cycle detection on a graph is a bit different than on a tree due to the fact that a graph node can have multiple parents. On a tree the algorithm for detecting a cycle is to do depth first search, marking nodes as they are encountered. If a previously marked node is seen again, then a cycle exists.



- This won't work on a graph. The graph in figure will be falsely reported to have a cycle, since node C will be seen twice in a DFS starting at node A.

Cycle Detection

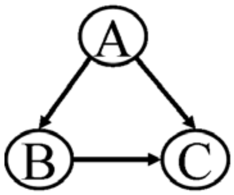
- The cycle detection algorithm for trees can easily be modified to work for graphs. The key is that in a DFS of an acyclic graph, a node whose descendants have all been visited can be seen again without implying a cycle. However, if a node is seen a second time before all of its descendants have been visited, then there must be a cycle.
- Can you see why this is?
- Suppose there is a cycle containing node A. Then this means that A must be reachable from one of its descendants. So when the DFS is visiting that descendant, it will see A again, before it has finished visiting all of A's descendants. So there is a cycle.

Cycle detection algorithm

- In order to detect cycles, you can use a modified depth first search called a colored DFS. All nodes are initially marked white. When a node is encountered, it is marked grey, and when its descendants are completely visited, it is marked black. If a grey node is ever encountered, then there is a cycle.

Cycle detection algorithm

- This is an acyclic graph
- No cycle



Algorithm

```
boolean containsCycle(Graph g):
  for each vertex v in g do:
     v.mark = WHITE;
  od;
  for each vertex v in g do:
    if v.mark == WHITE then:
      if visit(g, v) then:
        return TRUE;
      fi;
    fi;
  od;
  return FALSE;
```

```
boolean visit(Graph g, Vertex v):
  v.mark = GREY;
  for each edge (v, u) in g do:
    if u.mark == GREY then:
      return TRUE;
    else if u.mark == WHITE then:
      if visit(g, u) then:  //recursive call
        return TRUE;
      fi;
    fi;
  od;
  v.mark = BLACK;
  return FALSE;
```

This is a recursive depth first search or traversal

How do we apply this?

- Suppose, we have the following:

T1 holds item 1
T1 holds item 2
T1 hold item 3
T1 waits (on T3) for item 6

T2 holds item 4
T2 waits (on T1) for item 3

T3 holds item 5
T3 holds item 6
T3 waits (on T2) for item 4

From the has table, you should Be able to construct a wait for Graph by using the list of Waiting Txs and finding out Who is holding the items those Txs are waiting on.

After that you can use the DFS Cycle detection algorithm For detecting a cycle.

Pick any Tx as a victim. If you Want to be smart, you can Choose the one that holds Minimum number of items!

Thank You !

PTHREADS AND SEMAPHORES

PTHREADS

Pthreads

- To take full advantage of the capabilities provided by threads, a standardized programming interface was required.
- For UNIX/Linux systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
- Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads.

Thread Basics

- Multiple threads can be created within a process.
- Threads use process resources and exist within a process (different from a real DBMS)
- Scheduled by the operating system (you have some control over its scheduling, can specify FIFO, etc.)
- Run as independent entities within a process.
- If the main program blocks, all the threads will block.

Thread management

- Creating and deleting a thread

  Pthread_create(thread, attr, start_routine, arg**)**

  where
  - thread argument returns the new thread id.
  - *attr* parameter for setting thread attributes. NULL for the default values.
  - *start_routine* is the C routine that the thread will execute once it is created
  - A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.
  - If you need to pass multiple args, need to create an struct and pass that (param in our case)

Other thread functions

- **pthread_self()**
- **Attribute Set**

  pthread_attr_init(&attr)

  pthread_attr_setschedpolicy(&attr, SCHED_FIFO)
- **Exiting**

  pthread_exit(status);

  This routine terminates the calling thread and makes a status value available to any thread that calls pthread_join and specifies the terminating thread.

Example

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void *threadid) { printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL); }

 int main(){
  pthread_t thread;
  int rc, t =1;
  printf("Creating thread %d\n", t);
  rc = pthread_create(&thread, NULL, PrintHello, (void *)t);
   if (rc)
    { printf("ERROR; return code from pthread_create() is %d\n", rc);
     exit(-1); }
}
```


The University of Texas ARLINGTON

SYNCHRONIZATION PRIMITIVES

Synchronization primitives

- Semaphores
- Mutexes
- Condition variables
- We will be using all of the above and I want you to understand clearly why!

Synchronization primitives

- **Semaphores**
  - A locking mechanism
  - Any thread can acquire and release
  - Generalization of mutex.
  - A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number
  - Operations: p and v (Dijkstra)
- Think of 4 toilets with 4 keys. 4 people can be using the resource at the same time!
- We use this for the lock table

semaphores

- Array of semaphores are generated by semid = semget(key, nsems, semflg) where nsems = 0 to no_of transactions.
- Semaphore 0(SHARED_MEM_AVAIL) is for locking the transaction manager.
- Semaphores: 1 to no_of_transactions are used for threads to wait when objects are locked by other transactions.

semaphore

- semaphore creator can change its ownership or permissions using semctl(); and semaphore operations are performed via the semop() function
- Semaphore 0 is initialized to 1 i.e., holds one resource (transaction manager). Do 'p'(zgt_p) operation to obtain the resource and 'v'(zgt_v) to release the resource.
- Rest of semaphores are initialized to 0 i.e., hold no resources. Hence on the first p operation, the thread/process will wait till a v operation is done on the semaphore.

Synchronization primitives (2)

- **Mutexes:** Deals with synchronization, which is an abbreviation for "mutual exclusion“
  - A semaphore with count as 1
  - A signaling mechanism
  - There is ownership with mutex
  - Only the owner can release the lock
  - Used for exclusive access to a shared resource (critical section)
  - Operations: lock, unlock
- This is like the key to the door of the bathroom! Only one person can use at a time!

Synchronization primitives (3)

- **Condition variables (CV):** Condition variables provide yet another way for threads to synchronize.
  - While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data/condition.
  - A thread can wait on a CV and then the resource producer can signal or broadcast the variable
  - Tied to a mutex for mutual exclusion
  - Wait for event and signal or broadcast
    - Signal if any thread can proceed
    - Broadcast if you have to select a thread based on Cv value!!
    - We use this for sequencing operations of a Tx. Using conset and SEQNUM

Condition Variable

- To synchronize thread A and B
  - Declare and initialize global data/variables for synchronization. e.g:condset[tid] =0
  - Declare and initialize a condition variable object.
    - **pthread_cond_init (condition,attr)**
  - Create and initialize associated mutex.
    - **pthread_mutex_init (mutex,attr)**
  - Create threads A and B to do work.

Thread A

- Lock associated mutex
- Change the value of the variable (If condset[tid] =0, set it to –1)
- ……… operations ………….
- Set the global variable condset[tid]= 0, for thread B to continue
- **Do , pthread_cond_signal(condition) or pthread_cond_broadcast(condition)**
- Unlock mutex
- Continue

Thread B

- Lock associated mutex and check value of a variable(condset[tid]=0)
- Call pthread_cond_wait to perform a blocking wait if condset[tid]!= 0. Note that a call to pthread_cond_wait automatically and atomically unlocks the associated mutex variable so that it can be used.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex after completion of operation.
- Continue

Compilation Details

- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h
- a thread library 'pthread' has to be linked.
  ie. -lpthread