DBMS Models and implementation
Instructor: Sharma Chakravarthy
Project II: Implementation of a Transaction Manager

Made available on: 09/20/2025
Complete Project Due on: 10/23/2025 (11:59 PM)
Submit by: Canvas (1 zipped folder containing all the files/sub-folders)
Weight: 15% of total
Total Points: 100

This project implements a transaction manager 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. This is a stand-alone project using C/C++ and does not use Minibase. Portions of the skeletal code are given so you can focus on the important and interesting aspects of Tx management.

Read the project description carefully and identify what you need to do before proceeding with the project. Ask questions at the beginning of each class to get clarity. Once you had a chance to look at the description, we will provide additional details and answer questions. I will devote first few minutes at the beginning of each class for questions and clarifications on the project.

I. Problem Statement:

Your project's run time environment 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 (zgt_test.C). The main thread creates a transaction manager object (there is only one Tx mgr object) and the associated hash table that acts as a lock table. As the input is read from a file, **a separate thread** is created and started for carrying out each operation (in a real DBMS one thread or process will handle all the operations of a Tx) requested by that transaction (zgt_tm.C). The reason for using threads is to allow concurrent execution of multiple threads, where possible, by the Tx mgr logic. The thread is terminated at the end of the operation using a pthread_exit call. The main thread reads transactions (instructions from the input – see below) and invokes a method on the created transaction manager object. This method will create a new thread to perform each operation.

The thread created for each operation will execute a function (not a method, as we are using C) for that operation in zgt_tx.C. For example, the function begintx does the operation of starting a transaction. All the functions for a transaction execution (begintx, readtx, writetx, committx, and aborttx) need to be implemented or completed as part of this project. **The thread has to make sure that a previous operation by the same**

transaction has been completed before starting the current operation (otherwise, as you know, it is not a legal schedule). In order to do that, it uses a condition wait on a mutex. Each transaction has a mutex associated with it for this purpose. This is accomplished by using the SEQNUM array and condset array defined in class zgt_tm. They are initialized to zero. Each operation is assigned its position using the count variable before a thread is created. SEQNUM is used to keep track of the order of operations for each Tx. The count variable in each thread holds the current operation number. They are compared with the condset[tid] value to decide whether to wait (using cond_wait on the thread) for an operation or proceed. This allows the operations to be executed sequentially **within a Tx.** Use the cond_wait and cond_broadcast functions. The working of these functions is described in the supplementary material. If the operation is successfully completed (i.e., transaction gets a shared or exclusive lock for that operation), the appropriate parameters should be inserted in the log file. Do a flush on the logfile write operation to force write it. If there are no threads available, an error should be written in the log. You make sure all threads complete before exiting the test program.

Since each operation of each transaction is done in a separate thread, all errors and the log output need to be printed (or transmitted) at the point of occurrence. These threads cannot return any error status.

A transaction abort operation should release locks held on all objects by **that** transaction (not necessarily in any particular order). In addition, the abort operation should release **all** transactions waiting on objects held by the aborted transaction so that ***one*** of them can proceed.

A transaction commit will also release locks held on all objects by that transaction and the transactions waiting on those objects (not necessarily in any particular order). This is done by doing as many **p** operations associated with the semaphore for that tx. In a real DBMS, the data durability (or persistence) will be handled by the buffer manager and the log is written for recovery which we are not addressing in this project. However, in this simplified project, values of the data items reflect the work accomplished by a transaction.

II. Transaction manager interface

The simplified Transaction manager interface that you will implement in this assignment allows a client (a higher-level program that calls the Transaction manager) to create the data structures used by the transactions. We will assume default values for the hash table size. The header files zgt_tm.h, zgt_tx.h and zgt_ht.h describe the interface you will need to implement. The following five functions have to be implemented/completed in zgt_tx.C. As needed, additional functions to support the above need to be implemented as well.

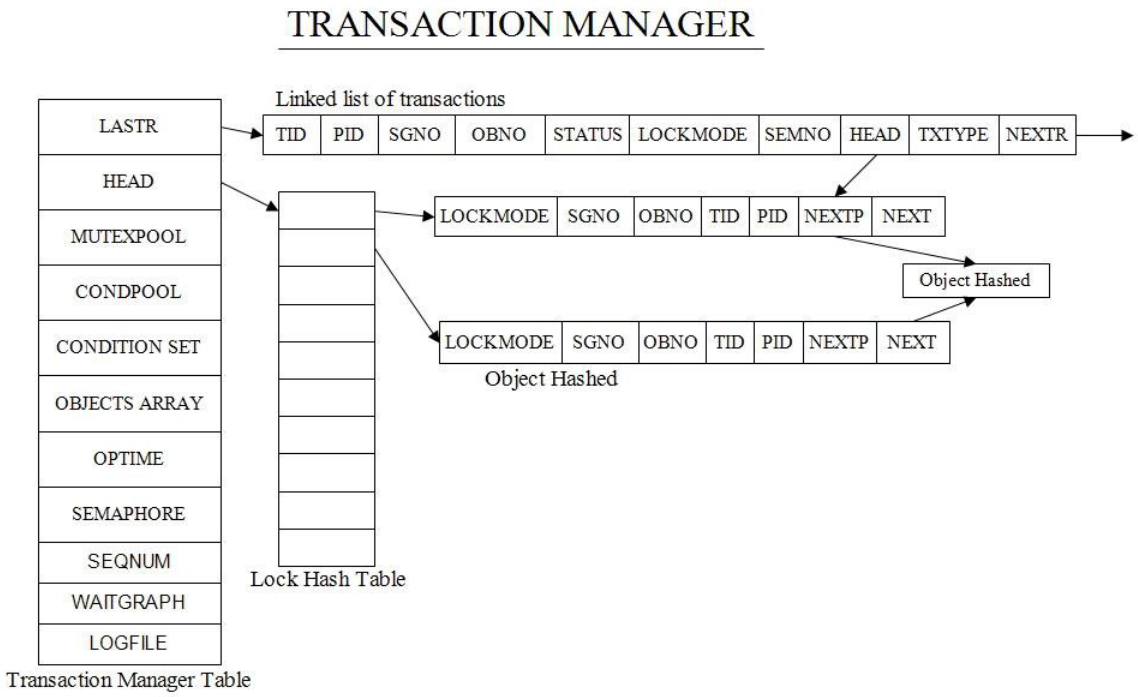
- begintx(thrdArguments)

- readtx(thrdArguments)
- writetx(thrdArguments)
- aborttx(thrdArguments)
- committx(thrdArguments)

The return types are void*. If more than one parameter needs to be passed in the thread, the required parameters of the function like transaction id, object number etc. need to be passed in a structure (param in our case). After creating a thread for each transaction operation, you are supposed to call these functions from zgt_tm.C. The transaction manager object used is the global ZGT_Sh. All the functions you need to implement are specified in the files. If you feel you need to add new functions, please discuss that with the instructor or the TA so that we can understand their need and help you implement it correctly!

III. Overall Approach

The following diagram shows the overall organization of the transaction manager data structures along with the transaction and hash table objects.



Note: PID = threadID , SGNO = 1 (always), NEXT links nodes hashed to the same bucket, NEXTP links nodes of the same transaction, i.e., all locks held by the transaction. The main thread creates a transaction manager object in the main or test program. There is only one transaction manager object. However, there will be one transaction object for each transaction. Transactions can be in one of the following states:

- TR_ACTIVE or processing (represented as "P")
- TR_WAIT (represented as "W")
- TR_ABORT (represented as "A") and
- TR_END (represented as "E"). This is the state while commit is going on.

When a transaction starts, it is set to TR_ACTIVE with obno = 0 and sgno being 1 by default. Before reading or writing, a transaction inserts the object into the hash table in the appropriate lock-mode. The presence of an object in the lock table indicates that the object is being used by some transaction (may be the same as the one requesting it).

If the object is in the hash table and is being held by the *same* transaction as the one requesting it, it gets the lock (i.e., can continue with the operation irrespective of whether it is read or write). We will not consider lock upgrades in this project. The given inputs for testing will avoid such cases. Once you get the lock on an object, you perform the operation. You hold the lock until the transaction either commits or aborts (strict two-phase locking). However, the operations of a transaction should be performed in the same order as given in the input. That is, the previous operation of the same Transaction should be completed before starting the next operation.

The operation in our case is decrementing the object value by 1 and printing a line in the log if it is a read operation and sleeping for the given time (optime). For a write operation, we increment the value by one, write this to the log and sleep for given optime. The sleep is to simulate disk read/write operations. Begin and commit operations are also written to the log. Writes are flushed to the log to ensure that it is written immediately. An array containing the optime for each transaction will be supplied. **You need to set the TEAM_NO to your team number in your zgt_tm.C file for the optime to work correctly!**

In this project, there are two types of transactions: read only (R) and read/write (W). Read only transactions do not modify any object and hence read operations from **multiple read only transactions** can proceed concurrently. If the object is held by a read/write transaction, the current transaction (whether read only or read/write) has to wait (both for read and write) for the lock to be released by that transaction (on a commit or abort). *The transaction object of the waiting transaction should indicate the object (along with lock -mode) for which it is waiting on. The transaction status of the waiting tx is changed to TR_WAIT to indicate that it is waiting for that object. The tx that holds the object currently (holding tx) is the tx on which the requesting tx is going to wait. This is done by making the requesting tx thread wait on a semaphore and that semaphore number is inserted into that holding tx object. Note that a tx can wait only on one other tx. On the other hand, a number of*

transactions may be waiting for the same (or different) object, and this is reflected in many transaction objects being in the wait state and waiting on the same semaphore. Semaphore s is used to make other transactions wait on transaction s (we have an array of semaphores and s corresponds to the index. Semaphore 0 is used as a lock for the transaction manager.

For example, if Tx 1 is waiting on Tx 2 for object 6 for writing, then the tx objects will have the following Information (Tid with number m uses the semno m in this project) in the output.

Tid	Txtype	Thrid	objno	lock	Txstatus	semno
2	R	2051	-1		P	2
1	W	1026	6	X	W	-1

In the above, tx object for tid 1 indicates that it is waiting for object 6 and the status is W (does not actually indicate which Tx it is waiting on. Here there are only 2 txs and hence it is obvious.) Tx 1 waits on semno 2 as it is waiting for Tx 2 to complete. It is important that you initialize the attributes properly to undefined values (example, -1 for objno and ' ' (blank) for lockmode, tx_type, and –1 for semno). Otherwise, you will not be able to differentiate between undefined and defined states. Semaphore number for Tx 2 is 2 (convention) and a value of semno indicates that some other Tx is waiting on this tx to release objects.

As another example, if two transactions are deadlocked, then the tx objects will have the following pattern

Tid	Txtype	Thrid	objno	lock	Tx status	semno
2	W	2051	4	S	W	2
1	W	1026	6	X	W	1

Here Tid 1 is waiting on semno 2 for the objno 6 held by Tid 2. Similarly, Tid 2 is waiting on semno 1 for objno 4 held by Tid 1. Hence, they are deadlocked. When Tid 1 aborts/commits, it releases all the threads waiting on its semno (1 in this case). As you can see both semaphores 1 and 2 (associated with Tx1 and Tx2 are positive indicating one is waiting on the other.

In strict two-phase locking, each transaction holds all objects until the end (commit or abort) of the transaction. Hence, even after an individual operation is over, the object is in the lock table indicating that it is held by that transaction.

Note that if the input file produces a deadlock, your program will hang. It needs to be killed and semaphores released explicitly from command prompt using *ipcs* and *ipcrm* shell commands (see below). Even if the program does not hang, you may have to remove the semaphores explicitly from command prompt using ipcs and ipcrm shell commands. We have provided a script *ipcs_cleanup.sh* for that purpose. It will remove

all the shared resources that you might not have cleaned up!

When a transaction commits, all the objects held by that transaction are released (using the head pointer of the transaction object). Transactions waiting for these objects are allowed to continue (by performing v operations on the appropriate semaphore) and the transaction object itself is deleted. If many transactions are ready for continuation, **only one of them** will be able to get the lock (in case of conflicts) and the first one that gets the lock will proceed (race conditions are normal in multi-threaded programs and makes it difficult to reproduce an error and hence debug!.) The rest will go to wait mode again.

When a transaction aborts, the status of that transaction is set to TR_ABORT. Again, all the locks held by that transaction is released and waiting transactions enabled. Since the aborted transaction can be waiting for another resource (in case of a deadlock), it is important to check when the thread is released, whether the transaction status has changed to TR_ABORT. In this case you do not proceed with the operation.

It is important that all of the transaction operations (insert/delete into the hash table and the transaction list) are protected. The semaphore sem<SHARED_MEM_AVAIL> is used for that purpose. It is very important that a transaction does not go into the wait state holding a lock. In this case the program will hang. **You should acquire the semaphore as late as possible and release it as early as you can (thereby reducing the length of the critical section and the duration for which you hold the semaphore or lock on the main data structure)**. Note that this semaphore is different from the semaphore on which you make a transaction wait. This semaphore is for managing the transaction manager object as a critical section.

NOTE 1: In order that semaphores used by each team does not conflict with other team's semaphores if running on the same machine (we are using shared memory semaphores), we need to make sure that the key for initializing the semaphores is set differently for each team. Hence each team needs to change the value of x in line (in file zgt_tm.C)

```
#define TEAM_NO x          //team number to be substituted for x
                           // your team number is known to you already
```

NOTE 2: Various print_xx methods are under the debug flags. TM_DEBUG, HT_DEBUG, and TX_DEBUG (names are self-explanatory) print details in the corresponding classes. Disable these flags in the Makefile by putting a # in front of them. The order of these flags does not matter. For example,

DEBUG_FLAGS = -DHT_DEBUG # -DTX_DEBUG -DTM_DEBUG

will enable the first flag and disables the other two. All or none can be disabled/enabled by moving the # sign. Once you change the Makefile, you have to make clean first and then make again.

NOTE 3: In order to make sure your program is correct, it needs to be executed multiple (read a large number of) times so that it takes all possible paths and handles all possible race conditions. In order to do that we have included a *tmtest* script that accepts a number (number of times to be run) and a test file name (without the extension) and executes the test as many times. Please use that to thoroughly test the correctness of your implementation. Running each test case 5000 to 10000 times is a must for testing the correctness of your implementation thoroughly. More times and under different conditions are even better.

NOTE 4: Initially and before your program executes correctly, you will be acquiring and may not be releasing semaphores correctly. If this continues, you may run out of shared memory and/or semaphores (or may not be able to acquire them). The following commands will be useful for checking and removing semaphores that have not been released.

```
>ipcs          //will list all shared memory and semaphores that you have acquired
>ipcrm –s semid //will remove semaphores with given id (obtained using the ipcs
                 command)
```

Note 5: We have included a *ipcs_cleanup.sh* script for removing semaphores and other shared memory resources held by you. You can use it separately to cleanup all shared memory resources held by you. It is also included in the tmtest to clean up resources at the end of your run.

Additional help on threads and semaphores:

Semaphore: http://www.cs.cf.ac.uk/Dave/C/node26.html
Posix Threads: http://www.thefreecountry.com/developercity/freeclibraries.shtml#threads

III. What you are asked to do:

In this assignment you will implement a simple transaction manager. The transaction manager 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.

1. Complete the implementation of the `zgt_tx` class given to you

2. Complete the implementation of the `zgt_tm` class given to you

3. Compile and run using the test data files given to you. We may add/extend the

test cases for evaluation. You are welcome to create your own additional test cases and use them. Please follow the format indicated.

Here is an example of input (file: S2T.txt) and expected output:

```
// 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
```

The log file (S2T.log) should contain the following output (your program produces 1 line per input command) and terminate properly.

Logfile S2T.log

TxId	Txtype	Operation	ObId:Obvalue:optime	LockType	Status	TxStatus
T1	W	BeginTx				
T1		ReadTx	1:-1:277	ReadLock	Granted	P
T1		ReadTx	2:-1:277	ReadLock	Granted	P
T1		WriteTx	3:1:277	WriteLock	Granted	P
T1		WriteTx	4:1:277	WriteLock	Granted	P
T1		ReadTx	1:-2:277	ReadLock	Granted	P
T1		WriteTx	2:0:277	WriteLock	Granted	P
T1		WriteTx	4:2:277	WriteLock	Granted	P
T1		WriteTx	4:3:277	WriteLock	Granted	P
T1		CommitTx				
T2	W	BeginTx				
T2		ReadTx	5:-1:235	ReadLock	Granted	P
T2		WriteTx	5:0:235	WriteLock	Granted	P
T2		WriteTx	6:1:235	WriteLock	Granted	P
T2		ReadTx	6:0:235	ReadLock	Granted	P
T2		CommitTx				

Note that the order of lines may be different for each run. You need to make sure the order of operations WITHIN each Tx is done correctly!

IV. Getting Started

Copy the zip file and unzip it in your project directory. It is easier to use makefiles and the make command for compiling and executing your code and testing it. If you are using WSL on windows, the Linux version should already have makefile installed. If you are using Linux, make is already there. You can also install Cygwin on windows to get into a linux-like command prompt. You also need the GCC (g++) compiler and the runtime environment. Alternatively, you can use Omega for compiling and executing this project. The instructions are the same! **We know for sure the code given runs on Omega. If you use any other option mentioned above, you need to figure it out yourself.**

After you copy the code, the code should compile without any errors when you execute the command. You need to change one line (TXMGR=..) in the makefile to set the directory into which you copied your files. The g++ is set for Omega. You need to change the GCCPATH=/usr to correctly indicate where the g++ compiler is. Do not delete any files you copy!!

For checking/executing *during the coding phase*

```
Change the active directory to the src folder (cd src)
>make clean
>make
>./zgt_test ../test-files/<test_file_name>
```

When I executed make my WSL (with ubuntu version 20.04 using g++ version 9.4), I got a bunch of warnings which seems fine. ONE sample is below:

```
zgt_ht.C: In member function 'void zgt_ht::print_ht()':
zgt_ht.C:152:11: warning: format '%d' expects argument of type
'int', but argument 2 has type 'long int' [-Wformat=]
  152 |  printf("%d %d %c ->", hlink->tid, hlink->obno, hlink-
>lockmode);
      |       ~^        ~~~~~~~~~~
      |        |           |
      |        int         long int
      |       %ld
```

When I ran
>./zgt_test ../test-files/Mult_ROTxs.txt I got the following output.

```
Initializing the TM

leaving TM initialization
```

```
// Multiple RO Txs test case
// Multiple RO Txs test case
// read only transactions
// read only transactions
log Multi_ROTxs.log
Log file name:Multi_ROTxs.log

entering openlog

Given log file name: (null)
leaving openlog
// op   Tx#     type
// op   Tx#     type
// op   Tx#     Obj
// op   Tx#     Obj
BeginTx 1 R
BeginTx : 1

TxType : R


creating BeginTx thread for Tx: 1

finished creating BeginTx thread for Tx: 1
Read 1 1
Read : 1 : 1


creating TxRead thread for Tx: 1

exiting TxRead thread create for Tx: 1
Read 1 2
Read : 1 : 2


creating TxRead thread for Tx: 1

exiting TxRead thread create for Tx: 1
Read 1 3
Read : 1 : 3


creating TxRead thread for Tx: 1

exiting TxRead thread create for Tx: 1
Read 1 8
Read : 1 : 8
```

```
creating TxRead thread for Tx: 1

exiting TxRead thread create for Tx: 1
BeginTx 2 R
BeginTx : 2

TxType : R


creating BeginTx thread for Tx: 2

finished creating BeginTx thread for Tx: 2
Read 2 1
Read : 2 : 1


creating TxRead thread for Tx: 2

exiting TxRead thread create for Tx: 2
Read 2 8
Read : 2 : 8


creating TxRead thread for Tx: 2

exiting TxRead thread create for Tx: 2
Read 2 5
Read : 2 : 5


creating TxRead thread for Tx: 2

exiting TxRead thread create for Tx: 2
BeginTx 3 R
BeginTx : 3

TxType : R


creating BeginTx thread for Tx: 3

finished creating BeginTx thread for Tx: 3
Read 3 1
Read : 3 : 1


creating TxRead thread for Tx: 3

exiting TxRead thread create for Tx: 3
```

```
Read 3 5
Read : 3 : 5


creating TxRead thread for Tx: 3

exiting TxRead thread create for Tx: 3
Read 3 3
Read : 3 : 3


creating TxRead thread for Tx: 3

exiting TxRead thread create for Tx: 3
read 3 7
Read : 3 : 7


creating TxRead thread for Tx: 3

exiting TxRead thread create for Tx: 3
Commit 2
Commit : 3

commit 3
Commit : 2

Commit 1
Commit : 3

end all
Release all resources and exit:


Entering End of schedule thread with thrNum: 17
Wait for threads and cleanup
Thread 0 completed with ret value: 0
Thread 1 completed with ret value: 0
Thread 2 completed with ret value: 0
Thread 3 completed with ret value: 0
Thread 4 completed with ret value: 0
Thread 5 completed with ret value: 0
Thread 6 completed with ret value: 0
Thread 7 completed with ret value: 0
Thread 8 completed with ret value: 0
Thread 9 completed with ret value: 0
Thread 10 completed with ret value: 0
Thread 11 completed with ret value: 0
Thread 12 completed with ret value: 0
```

```
Thread 13 completed with ret value: 0
Thread 14 completed with ret value: 3
Thread 15 completed with ret value: 3
Thread 16 completed with ret value: 3
ALL threads finished their work
Releasing mutexes and condpool
Releasing all semaphores
endTm completed

Finished end of schedule thread: endTm
```

For testing at the *end of the coding and initial testing phase when you think your code is working correctly and robust*

Change the active directory to the **src** folder (cd src)
>make clean //for deleting .o and other files
>make zgt_test //for making the executable
>tmtest n test_file1 //for running your program n times using test_file1.txt
//initially use 1 for n; once working test up to 50000 times
//2nd param is input file with .txt extension
// and will create a new log file specified in the input file.
// log file will include the output
// it will also redirect the stdout to test_file1.out

V. Project Report

Please include (at least) the following sections in a **REPORT.{doc}** file that you will turn in along with your code:

- **Each input and its output (.log file contents) as shown in this document for S2T.txt**
- **Also include all .log files in the .zip uploaded for each test case with proper naming. For example, if the input file name is Multi_ROTxs.txt, the log file name should be MultiROTxs.log**
- **A spread sheet that shows maximum times you were able to run each test case correctly. Follow the format given in test-robustness.xlsx in test-files dir. The TA will ask you to run a couple from this list during the demo**
- **Overall Status**
  Give a *brief* overview of how you implemented the major components. If you were unable to finish any portion of the project, please give details about what is completed and your understanding of what is not. (This information is useful when determining partial credit.)
- **Where you encountered difficulty**
  Please indicate what aspects of the transaction manager (e.g., semaphore logic, thread logic, flow) that was difficult. This will help us understand what additional information we can provide in subsequent offerings.
- **File Descriptions**
  List any new files you have created and *briefly* explain their major functions and/or

data structures. If you have added additional test cases, please summarize them.

- **Division of Labor**
  Describe how you divided the work, i.e., which group member did what. Please also include how much time each of you spent on this project. (This has no impact on your grade whatsoever; we will only use this as feedback in planning future projects -- so be honest!)
- **Logical errors and how you handled them**
  List at least 3 logical errors you encountered during the implementation of the project. Pick those that challenged you. This will provide us some insights into how we can improve the description and forewarn students for future assignments.

  And finally, any suggestions you may have on this project.

VI. What to Submit

- After you are satisfied that your code does exactly what the project requires, you may turn it in for grading. Please submit your project report and the TxMgr package. We will ignore source code in any other directories.
- You will turn in one zipped file containing you source code as well as the report
- All of the above files should be placed in a single zipped folder named as - '5331_**Proj2Fall2025_team_<teamNo>**'. **Only one zipped folder should be uploaded using CANVAS.**
- You can submit your zip file at most 3 times. The latest one (based on timestamp) will be used for grading. So, be careful in what you turn in and when!
- **Only one person per group should turn in the zip file!**
- To discourage late submissions, a penalty of 20% per day (no partial penalty) will be imposed. This means that if your submission is delayed by more than 5 days, do not even bother submitting. You certainly do not want this delay to hurt your next project!

VII. Coding Style:

Be sure to observe the following standard naming conventions and style. These will be used across all projects for this course; hence it is necessary that you understand and follow them correctly. You can look this up on the web. Remember the following:

a. Class names begin with an upper-case letter, as do any subsequent words in the class name.
b. Method names begin with a lower-case letter, and any subsequent words in the method name begin with an upper-case letter.
c. Class, instance and local variables begin with a lower-case letter, and any subsequent words in the name of that variable begin with an upper-case letter.
d. No hardwiring of constants. Constants should be declared using all upper-case identifiers with _ as separators.
e. All user prompts (if any) must be clear and understandable.
f. Give meaningful names for classes, methods, and variables even if they seem to

be long. The point is that the names should be easy to understand for a new person looking at your code.

g. Your program is properly indented to make it understandable. Proper matching of if … then … else and other control structures is important and should be easily understandable.

h. Do not put multiple statements in a single line.

In addition, ensure that your code is properly documented in terms of comments and other forms of documentation.

VIII. Grading Scheme for the Complete Project:

The project will be graded using the following rubric:

1.	Correctness of the TxMgr code:	
	a. Correctness of Subroutines	***30***
	i. readTx()	5
	ii. writeTx()	15
	iii. commitTx()	5
	iv. abortTx()	5
	b. Correctness of Output Log Files (For all inputs given including spread sheet)	***30***
2.	Documentation and commenting of the code: (including coding style)	**10**
3.	Answering questions during demo	**30**
	TOTAL	**100**

5 days after the due date, the submission will not make any sense as there is a penalty of 20% per day (no partial penalties within a day!).

Your program must be executable (without any modification by us or you during the demo) from the WSL Linux command prompt or Cygwin command prompt (either on UTA's Omega server or a PC or laptop). So, please test it for that before submitting it. However, the source code files can be created and/or edited on any editor that produces an ASCII text file. As I mentioned in the class, an IDE is not necessary for this and subsequent projects. If you decide to use it, please learn it on your own and make sure your code compiles and executes with appropriate package information.

IX. Interview with the team members

If we suspect that the code is copied or plagiarized from anywhere, we reserve the right to ask the team members to explain the code and the output. If it is determined that the code is copied or if the team members are unable to explain what they have done, we intend to take full action as dictated by the academic dishonesty procedures of UTA (that I am bound by.)

Just want to remind students that we use a sophisticated tool to check for similarity of code. Also, we input previous project implementations at UTA and code from Internet/Github. All the best!