Project 1 Report

Overall Status: Complete, nothing left unimplemented

Implemented Correctly: Everything Assigned

A thread is created for each operation of each transaction.
These threads are run in the correct order by having a decrementing mutex value match their respective negative indices.
When a thread is created, it checks if it is its turn based on the condset for its transaction. If it is not its turn, it gives up the lock and waits for the next time it is awakened and checks again. When it gains the lock and keeps it, it continues to its specific operation.

For a begin operation, it creates the tx, adds it to the beginning of the transaction manager in a lock of the tm. It then logs the operation to the log file.
For a read operation, it attempts to gain a lock based on the flowchart in the lecture with the type based on the transaction type. Once it gains the lock, it decrements obval by 4 and logs the operation to the log file. It then waits optime microseconds.
For a write operation, it attempts to gain an exclusive lock based on the flowchart in the lecture. Once it gains the lock, it increments obval by 7 and logs the operation to the log file. It then waits optime microseconds.
For an abort operation, it sets the status of the transaction to aborted, logs the operation to the log file, frees all locks, wakes all any threads waiting on its semaphore if there are any, and removes the transaction.
For a commit operation, it sets the status of the transaction to committed, logs the operation to the log file, frees all locks, wakes all any threads waiting on its semaphore if there are any, and removes the transaction.
Finally, the thread decrements the condset, wakes all of the other threads in its transaction to check if it is their turn, and exits.

When each thread does this individually, the collective performs the expected actions.

Difficulties Encountered

1: Issues in skeleton code

The provided skeleton code had multiple spacing, output, log output, log name, ... provided in it. I will list the changes made here, none make any impact other than making stdout outputs more readable for us:

- avoided printing duplicating information when read from a file.
- fixed openlog() so logfilename is printed correctly.
- changed spacing of print statements such that data relating to one operation is grouped together. (removing duplicate \n)
- changed the order of functions in zgt_tx to make the code more readable and to outline our work
- made commit print out correct transaction id. Only effected print, not execution

These changes were all in zgt_test.c, so they are not part of the submitted code, but they were helpful to enhancing readability of the printed information.

There are a lot of memory leaks in the part of the code we are not to change and many of them cannot be freed by us.

We hypothesize that the uncaught failure to malloc space that is cause by the above memory leaks is the primary source of confounding hanging on rare occasions.

As a last bit of complaint about the skeleton code. It is not C++, it is a C/C++ amalgamation.
This is not uncommon in projects that started as C and became C++, but it makes reading the code significantly more difficult.

2: Interacting with the skeleton was far harder than the logic of the implementation

The logic of each transaction function is relatively simple given the information available in the PowerPoints and lectures.
The vast majority of the challenge of this project was figuring out how to use semaphores for locking, where to find stored information (specifically by way of code, not necessarily the data structures), etc. as opposed to when those things need to be done.

If the intent of the project was for that to be the real challenge, we understand, however it was incredibly frustrating.

File Descriptions

No new files were created.

No new functions were created.

Division of Labor

Landon Moon

- 2/29/2024 - 1 hour - Learning the requirements
- 3/07/2024 - 3 hours - Learning the skeleton
- 3/07/2024 - 3 hours - Fixing and cleaning up provided code
- 3/13/2024 - 1 hour - Made transaction operations run in order
- 3/13/2024 - 2 hours - Found where object values are located. read code
- 3/14/2024 - 2 hours - Completed perform_..._operation()
- 3/15/2024 - 2 hours - First implementation of gaining read locks
- 3/15/2024 - 1 hour - refactor. possibly done
- 3/16/2024 - 3 hours - memory leak investigation
- 3/17/2024 - 1 hour - Debugging and testing on Omega
  Jacob Holz
- 2/29/2024 - 1 hour - Learning the requirements
- 3/07/2024 - 3 hours - Learning the skeleton
- 3/14/2024 - 2 hours - Learning the new code and planning next steps (didn't know the next time I looked, the code would be done)
- 3/15/2024 - 3 hour - Debugging and Testing on Omega
- 3/15/2024 - 2 hours - Report Writing
- 3/16/2024 - 4 hour - Debugging and Testing on Omega (stubborn hanging bug)
- 3/17/2024 - 1 hour - Finishing Report

Logical errors and how you handled them

Many logical errors were avoided before they cropped up through the use of careful planning, and reviewing line by line.

Logical Error 1: Reading also decreases the value?

Before we ran tests for the first time, we figured that if the read operation is decrementing the value of the object (and the example logs show that it does), it need to have an exclusive lock to avoid race conditions. This feels very odd as it introduces deadlocks into cases that should logically be free of them if reads were not exclusive. We then noticed that the lock given to a transaction is based on the type of the transaction not the type of the operation, so we didn't have to worry about it being exclusive as it was required to be shared in some cases even though that allows very rare race conditions.

Logical Error 2: Forgot to allow an exclusive lock to succeed when the Tx already has the exclusive lock

We just had to have it check whether the tid of the lock entry was the same as the requesting transaction instead of just returning false when there is already an exclusive lock. We also noticed that the same would normally go for a shared lock, but since a transaction can only ever do one kind of lock in this project, that is not necessarily.

Logical Error 3: Library memory leak instability

The provided code and library from the skeleton code NEVER frees any dynamically allocated memory. This causes the operating system to mistake memory values if executions of the program are done back-to-back. Time must be given for the operating system to clean up the programs unfreed memory. This was tested extensivly and was determined through multiple means:

- tid values passed to a given thread could be non-existent in the txt file. Causing an operation of an invalid tid to try to gain an invalid mutex lock. This was confirmed multiple times and causes the thread with the invalid tid to hang the entire program. The int returned from string2int is incorrect.
- malloc calls fail more often directly after a previous execution. The skeleton code rarely tests for a successful malloc call, instead it segfaults due to trying to access a null pointer. The dynamic memory allocation is located in the library and is not our responsibility to rewrite the entire framework from scratch. This occurs primarily during param creation. Again this was tested by printing out the ptr value from malloc and it is occasionally null which will always cause a segfault if not checked for. Additional checks were put in place to mitigate this.

Logical error 4: remove_tx() never removed a transaction

a bug was found in remove_tx() which caused it to never remove a transaction. This was due to both pointers always pointing to the same transaction. When the transaction found nextr it is set to nextr which does nothing. Additionally, the special case for the first transaction was not originally considered by the skeleton code. It would have required double pointers with the original method.

Logical error 5: general logical errors with the skeleton code

There are still a bunch of other smaller logical errors that aren't worthy of an entire section. This section is a general complaint with the state of the skeleton code at the beginning of the project. Many C++ coding

standards could have prevented the vast majority of bugs in the provided library. Due to not rewriting every line provided in the skeleton code, there are bound to be errors beyond our control.