CS 6210: Advanced Operating Systems Apr 5, 2023

Project 3 Report

Instructor: Ada Gavrilovska *Aditya Vikram*

1 Abstract

This report provides an overview of the design and implementation of a file system (GTFS) library, and the test suite to verify its functionalities. The file system uses in-memory and disk logs to provide persistence and crash recovery.

2 Introduction

The project sets out to create a file system as presented in the LRVM paper ([1]), with the goals of crash recovery and persistence. To do so, files get mapped to virtual memory segments when opened. Each file update is treated as a virtual memory write, and is first persisted to a log file before the actual file is updated. This write-ahead log is the key to providing crash recovery. Once the system is up again, we can read the log files again and replay the committed transactions to the actual files on disk to recover.

The figure below shows an overview of the lifecycle of a file opened and operated upon using GTFS APIs.

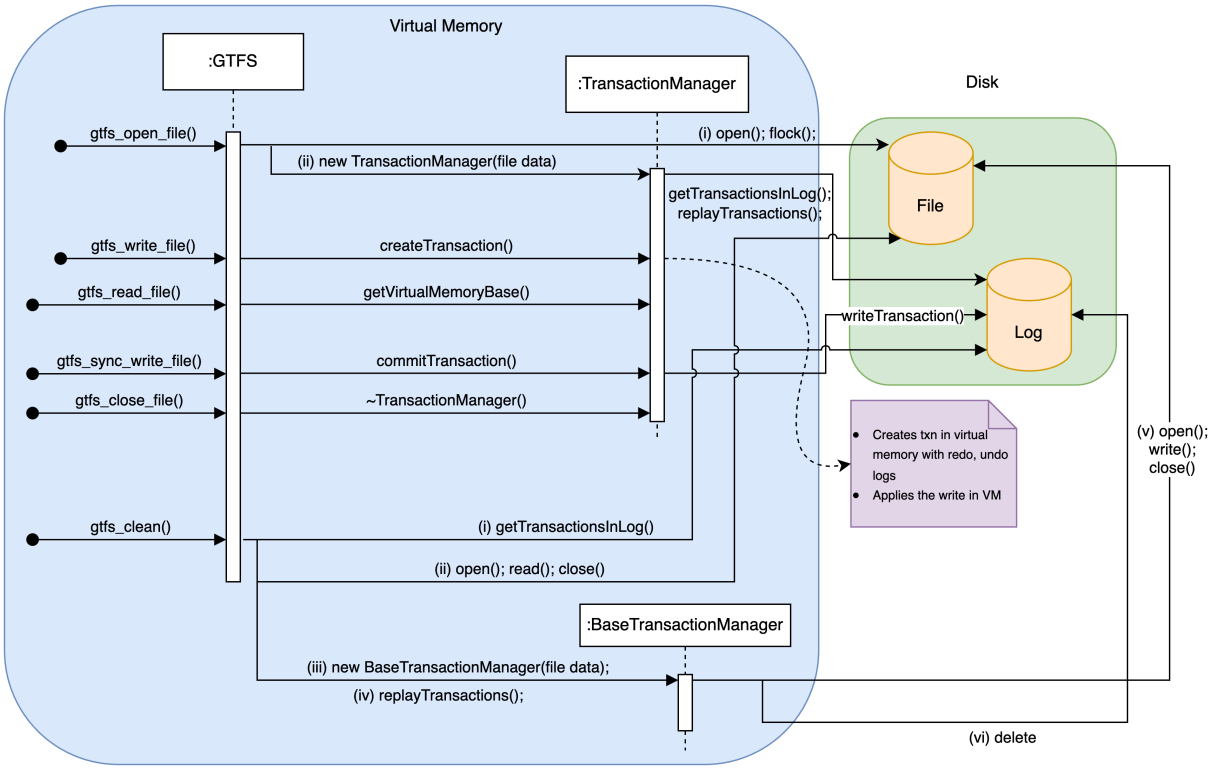

Figure 1: A sequence diagram for a normal GTFS usage
(GTFS is not an object in the actual implementation, it is meant to represent the GTFS API layer here)

The design is described in the next section.

3 Design

Each write in the file system is captured as a transaction in the logs (either in-memory or on disk, when synced). A `Transaction` consists of the following data:

- `transactionId`: id to identify the transaction, incremented as a counter per TransactionManager (one per file)
- `offset`: start offset of the write from the base of the virtual memory segment (file start)
- `oldData` (UNDO): data previously present in the segment where the write is going to occur, used for aborts
- `newData` (REDO): new data that is copied in the segment after the write, used for applying logs to the actual file

3.a BaseTransactionManager

This is a generic transaction manager that gets constructed with a virtual memory segment that it needs to manage, and has the following functions:

- `createTransaction(offset, length, newData)`: initializes a transaction with the required data and pushes the transaction onto its `uncommittedTransactions` vector. This also writes newData at offset, while extending the VM segment it manages, if required.
- `abortTransaction(transactionId)`: Finds the transaction by `transactionId` in `uncommittedTransactions`, copies its `oldData` back into the memory offset of the VM segment and erases that transaction.
- `replayTransactions(transactions)`: Iterates through each transaction and applies its newData (REDO data) to the corresponding offsets in the managed VM segment, while extending the segment whenever required.
- `getVMBase()`: Returns the base of the managed VM segment

3.b TransactionManager

This is a subclass of `BaseTransactionManager` that uses log files to commit the transactions to disk. It is constructed with a file path and its data, and has the following additional functions:

- `commitTransaction(transactionId, bytes)`: writes a transaction present in `uncommittedTransactions` onto the log file on disk, using `LogManager::writeTransaction()`. Note that we don't need the oldData on disk since committed writes don't have to be aborted, so we only write newData in the log file. If bytes param is specified, then this also resizes the transaction newData to `bytes` bytes.
- `getLogFilePath()`: Utility function to get the path of the log file created for the filename, which is just `.log` appended to the filename.

3.c LogManager

This is a utility class used to read/write transactions from/to log files on disk. Static function `writeTransaction(logFilePath, transaction)` appends a given transaction to the log file at the given path, and `getTransactionsInLog(logFilePath)` reads each transaction present in the given log file and returns a vector of transactions.

Now we can move on to describing the GTFS API surface itself.

3.d GTFS APIs

Please refer back to Figure 1 to look at the sequence diagram to supplement the overview of the API implementation provided below:

- `gtfs_init(directory, verbose)`: If the directory doesn't exist, it is created in this function. Then a `gtfs_t*` is dynamically allocated and also stored in a global map with the directory path as key, so that subsequent calls to `gtfs_init()` with same directory name return the cached pointer.
- `gtfs_open_file(gtfs, filename, file_length)`:
  - If the file doesn't exist, it is created with the given `file_length`. If the file exists and its size exceeds the given `file_length`, a null pointer is returned to indicate error since truncating the file would lose data. If the existing file's size is smaller than the given `file_length`, then it is extended using the `std::filesystem::resize_file()` function.
  - Now that file exists in the required state, we open the file and ensure that `flock()` (with non blocking flag to prevent deadlock) returns success so that this is the only file that is open. Then we read the file and create a new `TransactionManager` with the read buffer.
  - We get all transactions from the log file (if exists) for the filename using `LogManager::getTransactionsInLog()` and replay the transactions on the created `transactionManager` to apply the synced writes from previous opens of the file.
  - Finally we set the transaction manager and other properties on a dynamically allocated `file_t*` and return that.
- `gtfs_read_file(gtfs, fl, offset, length)`: This just checks that file is open and copies `length` bytes starting from `offset` bytes ahead of `fl->transactionManager.getVMBase()` since transaction manager contains the most updated representation of a file's data in the file system, including writes synced from other processes before the file was opened, plus all writes since current open.
- `gtfs_write_file(gtfs, fl, offset, length)`: Checks that file is open and calls `fl->transactionManager.createTransaction()` to create the corresponding transaction in memory. This updates the in-memory segment of the file and prepares a transaction for commit/abort later. We attach the returned `transactionId` on the dynamically allocated `write_t*` along with other properties, and return it.
- `gtfs_sync_write_file(write_id)`: Checks that the file was open (because closed files lose all unsynced writes in our system), and calls `fl->transactionManager.commitTransaction()` with the transactionId to write the corresponding transaction from memory to the disk log.
- `gtfs_abort_write_file(write_id)`: Checks that the file was open (closed files lose all unsynced writes), and calls `fl->transactionManager.abortTransaction()` with the transactionId to apply the undo data from the corresponding transaction in memory, and erase that transaction from memory.
- `gtfs_close_file(gtfs, fl)`: Closes the file handle, which also drops the lock held by `flock()` used in `gtfs_open_file()` (so that other processes can open the file). Also destroys the corresponding transactionManager by setting it to nullptr.
- `gtfs_remove_file(gtfs, fl)`: Checks that the file is not open currently (only for the current gtfs instance, doesn't check other processes), and deletes the file corresponding to `fl->filename` in the directory, along with any log file it would've created.
- `gtfs_clean(gtfs)`: Iterates through each log file in the directory and performs the following operations (also shown in order in the sequence diagram in Figure 1):
  - reads all the transactions from the log file: these are transactions that haven't been applied to the original file yet
  - opens the original file for that log file and reads its data

  - creates a `BaseTransactionManager` object (since we don't need to commit any data) with the read original file buffer
  - replays all transactions on the `transactionManager`: this applies all the log file transactions to the managed VM segment
  - opens the original file again and overwrites it with the VM segment managed by `transactionManager`
  - deletes the log file from disk

- (Additional credit) `gtfs_sync_write_file_n_bytes(write_id, bytes)`: Same as `gtfs_sync_write_file()` implementation above except that it uses the `bytes` param while committing. The main work to truncate bytes is done in `TransactionManager::commitTransaction()` which was discussed previously in 3.b.

- (Additional credit) `gtfs_clean_n_bytes(gtfs, bytes)`: Similar to `gtfs_clean()`, this function also iterates through each log in the directory and performs the same steps except the first step is modified to:

  - reads all the transactions from the log file. Iterates through all transactions and truncates at the `bytes` bytes. If there is any partial transaction, it is also discarded.

  So, only the transactions having redo data completely within the first `bytes` bytes get applied to the original disk, and the log files get deleted.

4 Additional Tests

In addition to tests already provided, I added tests to make the test suite as comprehensive as I could within the time bounds. In the process, I added negative (invalid input) test cases for the APIs, added cases in which the order of operations wouldn't be what we expect and tests for the additional APIs which simulate crashes. Following are the tests I've added in `test.cpp`:

1. `test_write_partial_sync_read()`: tests that if the first process only syncs first n bytes of a write with more bytes, then another process that reads the file can only read the first n bytes.

2. `test_truncate_log_partial()`: tests that if the first process calls `gtfs_clean_n_bytes` in a log with more bytes, then another process that reads the file can only read the transactions that fit into the first n bytes. To simulate this, I synced two transactions of length 20 bytes each, and then cleaned only 30 bytes from the first process. I verified that the second process only reads the first write fully, and the second write isn't reflected at all.

3. `test_remove_file()`: tests that a file can be removed

4. `test_remove_synced_file()`: tests that removing a file with synced writes also removes the corresponding log file.

5. `test_remove_open_file()`: tests that removing an open file fails.

6. `test_read_write_closed_file()`: tests that reading from or writing to a closed `file_t*` handle fails (returns error status or nullptr).

7. `test_read_unwritten_data()`: tests that reading data from a file segment that is previously unwritten should return a pointer to the empty string.

8. `test_open_already_open_file()`: tests that opening a file that is already open in another process returns a NULL `file_t*` handle.

9. `test_sync_write_more_bytes_than_written()`: tests that calling `gtfs_sync_write_file_n_bytes()` with more bytes than was originally written should result in an error return code, since there aren't enough bytes to write.

10. `test_sync_invalid_write()`: tests that syncing (calling `gtfs_sync_write_file()`) a write that doesn't exist should fail.
11. `test_sync_closed_write()`: tests that syncing a write for which the file was closed, should fail.
12. `test_sync_aborted_write()`: tests that syncing a write which was aborted should fail, since aborted writes get erased by design.
13. `test_abort_invalid_write()`: tests that aborting a write that doesn't exist should fail.
14. `test_abort_closed_write()`: tests that aborting a write for which the file was closed, should fail.
15. `test_abort_aborted_write()`: tests that aborting a write which was already aborted should fail, since aborted writes get erased by design.
16. `test_abort_synced_write()`: tests that aborting a write which was synced should fail, since synced writes get flushed from memory by design.
17. `test_sync_synced_write()`: tests that syncing a write which was already synced should fail, since synced writes get flushed from memory by design.

5 Implementation comments

C++ Filesystem library was very useful in implementing many of the functions and test cases. It provides a modern cross-platform interface for iterating through files, directories, reading file sizes and resizing files, which makes it convenient. The seamless integration with fstream operations is a plus point. Since I developed on my personal machine with an updated g++ compiler that contains the STL version (and not the experimental) of filesystem APIs, I referenced StackOverflow ([2]) to include filesystem in a cross-compiler fashion so that it works universally for both my machine and the VM Cluster.

For all the buffers, it turned out to be easiest to use a `vector<char>` to represent them in memory since vectors provide amortized O(1) time resizing when extending buffers, and the size accounting is provided out of the box. When writing these vectors to file, I first write their size, followed by a whitespace and then `size` bytes of characters. Same format is followed when reading the vectors from logs.

6 Conclusion

I discussed my design and implementation of the GT file system. In the current implementation, the system provides **data persistence** for all synced writes. Whether the writes happen to the original file instantly or not is determined by the frequency with which the client calls `gtfs_clean()`, but any reads using the `gtfs_read_file()` API will always reflect the latest state with all synced writes. This is because whenever a file is opened, we also apply all the transactions in the corresponding log to our VM buffer.

With regards to **crash recovery**, the system guarantees that if a crash occurs before a sync, then the writes wouldn't be reflected at all. If a sync succeeds before the crash, then the logs must have the data, and we can recover easily in the next open. However, as demonstrated by `gtfs_sync_write_file_n_bytes()`, if a crash occurs while writing data to the log file, then partial redo log would be written and our system doesn't recover well from this scenario. Similarly for the clean scenario, if the clean succeeds before the crash, then the updates from log files would have been applied to the original files on disk. If the crash happens before a clean call, then the logs would still hold the transactions. So we can clean again once the system is back up again and update the original files. If the crash occurs midway however, only the transactions that had been processed in the log file would have been written to the original file, and rest of the log file would be dropped, leading to loss of data. This is emulated in `gtfs_clean_n_bytes()` and the current system doesn't provide a good way to recover from this.

Performance of GTFS depends somewhat on the client app. In the critical path, the client can choose to not sync many writes, because each sync goes to disk to write the transactions to disk. Clean is another I/O operation and should be done periodically when the system load is low, so that it doesn't affect overall

throughput. Other than that, the only major overhead of GTFS lies in the `gtfs_open_file()` function because the file buffer has to be read in memory, and we also have to read the corresponding log file to apply all previous transactions. This is a one time overhead though, and the client gets crash recovery and persistence guarantees by paying for this cost.

Making the test suite comprehensive helped me to handle the edge cases better and even helped in finding a few bugs in the logic of the `_n_bytes` functions that simulate crashes. As future work, the aim would be to get 100% code coverage on the code and handle more combinations of operations to ensure the stability of this GTFS implementation.

References

[1] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight recoverable virtual memory," *SIGOPS Oper. Syst. Rev.*, vol. 27, p. 146–160, dec 1993.

[2] "How to determine whether to use <filesystem> or <experimental/filesystem>." https://stackoverflow.com/a/53365539.