CS 4210/6210 Project 3 - GTFileSystem

Due: Thursday April 4 11:59pm

1. Goals

In this project you will create a wrapper of a flat file system that offers persistence and crash recovery guarantees through the use of recoverable virtual memory. You will need to implement specific API calls and pass certain test cases. You'll be given a code skeleton.

2. File System Description

The file system keeps track of changes made in files created under one directory via the use of *disk and in-memory logs*, as described in the LRVM paper (see Section 9 Additional Readings). Files are mapped to *virtual memory segments*. Multiple processes can have access to the directory, but concurrent operation to the files of the directory should not be permitted by the file system. The file system does not need to support multiple or nested directories, you can assume all files will be under a single directory. The file system internally can use existing file operations, e.g. open, mmap, close, fsync etc, but needs to wrap them so as to provide persistence and crash recovery via the recoverable virtual memory mechanisms. More details will be introduced in the description of the API calls.

Finally, to avoid any misunderstanding, you're not asked to implement a "log-structured file system", rather a file system that leverages virtual memory to speed up file operations and provide crash recovery through the use of in-memory and on-disk logs.

3. API calls

You're given a code skeleton that includes the declaration of certain data structures and API calls that you need to implement. You're also given a test suite that utilizes the API.

Data Structures

`gtfs_t:` Represents a GTFileSystem instance.
`file_t:` Represents a file.
`write_t:` Represents a write operation.
`do_verbose:` Flag that allows detailed printing of the file system's operations.

File System Operations:

`gtfs_t` **`gtfs_init`**`(string directory, int verbose_flag)`: Creates a new GTFS instance under the specified directory and initializes any necessary metadata for proper file system operation. If directory is already initialized then returns its existing representation. On success returns the directory representation else NULL.

int **gtfs_clean**(gtfs_t* gtfs): This operation reduces the length of the file system logs by applying and persisting any necessary changes to the existing files of the directory. On success returns 0 else -1.

File Operations:

file_t* **gtfs_open_file**(gtfs_t* gtfs, string filename, int file_length):
File name can be of length up to MAX_FILENAME_LEN and there can be MAX_NUM_FILES_PER_DIR under the directory. A file with the specific filename under the specified directory can only be opened by one process at a time. If a file with this name does not exist inside the directory, then create the file and the *corresponding memory segment* that has length equal to file_length. If a file does exist but has smaller length than file_length, then extend its length appropriately. If the specified file_length is smaller than the existing file length, then this operation should not be permitted, because it will lead to data loss. If the file exists, then the most up-to-date version of the file contents need to be available. On success returns the new or existing file representation else NULL.

int **gtfs_close_file**(gtfs_t* gtfs, file_t* fl): Closes the file and accordingly updates the file system metadata. On success returns 0 else 1.

int **gtfs_remove_file**(gtfs_t* gtfs, file_t* fl): Removes the file and accordingly updates the file system metadata. This operation cannot be performed on a currently open file. On success returns 0 else -1.

char* **gtfs_read_file**(gtfs_t* gtfs, file_t* fl, int offset, int length): This function should return the most recent data of an open file starting from offset up to offset + length into the data buffer. On success it returns a pointer to the data read else NULL.

write_t* **gtfs_write_file**(gtfs_t* gtfs, file_t* fl, int offset, int length, const char* data): This function initiates a write of the data string into an open file starting from the specified offset and accordingly updates any in-memory related metadata, so as to be able to restore the unmodified file content from a potential crash. On success it returns a pointer to the write operation, so that it can later be committed or aborted, else returns NULL.

int **gtfs_sync_write_file**(write_t* write_id): This function finalizes the write operation initiated by the corresponding gtfs_write_file() function call. This function should persist to disk enough information so that the data can be successfully restored on a program crash. On success returns the number of bytes written, else -1.

int **gtfs_abort_write_file**(write_t* write_id): Restore the file contents before the gtfs_write_file() function call. On success returns 0 else -1.

4. Code Implementation Summary (50%)

We've provided a code skeleton that you need to fill in with the aforementioned functionality and all necessary checks.

- `gtfs.hpp`: Includes the declaration of the above API calls and any additional functions.
- `gtfs.cpp`: Includes the implementation of the API and any additional functions.
- `Makefile`: do not modify
- `tests/Makefile`: do not modify
- `tests/test.cpp`: Includes existing tests.
- `./run.sh`: do not modify

Instructions:

- Do not create additional `.{hpp,cpp}` files, rather put any additional functions and data structures inside `gtfs.{hpp,cpp}.`
- The code is already a mix of C/C++, feel free to use any C++ data structures, as long as your code can compile as-is in the class servers.
- You can use global data structures but the code needs to be thread-safe, since multiple processes can access the same directory.
- For testing purposes adhere to the return values of the existing APIs.
- You cannot modify the given APIs.
- Assume that the GTFS directory is created under the folder `tests/.` Thus all created files will be located there. You can assume that files read/written will be text files.
- If you choose to implement more tests, do so in the existing test file `test.cpp` and follow the PASS/FAIL mechanism used.

5. Test cases (30%)

We've provided a script that compiles and runs your code for the given tests. For debug purposes run `./run.sh 1,` where 1 is a flag for verbose operation that is expected to print all actions taken in the file system. For testing purposes run `./run.sh 0,` so that you can test against the expected outputs.

The given tests only test the minimal functionality of the file system and are there to help you understand how the API comes together. Your code needs to pass all these tests to receive 15% points. TAs will run additional tests (15%) to check the full expected functionality, e.g. checks for possible errors given the described GTFileSystem semantics, crash recovery, multi-threaded operation. For example, if you want to emulate a crash, you can use the abort() linux call inside a forked process function and test if recovery from crash is successful from the main process (similar to existing test).

You're allowed to **work in pairs** in order to come up with more tests. Any particularly interesting tests or testing methodology may be awarded with **5% extra credit.**

6. Writeup (20%)

Your report needs to address the following:

- Description of your logging design. What data do you choose to log, when and how are logs utilized across the given APIs.
- How does your file system provide
  a. data persistence
  b. crash recovery
  c. good performance when reading/writing files.
- Describe any additional data structures used to store the file system's metadata. How do you ensure they maintain consistent when accessed by multiple child processes/threads?
- High level implementation details of each given API call in bullet points.
- If your code doesn't pass one of the existing tests, why do you think that happens?
- Did you implement any additional tests? If so, describe them. If you worked in pairs for additional testing, then document the name and contribution of each collaborator.

7. Collaboration

This is an individual project. You're allowed to collaborate in pairs in order to develop together additional test cases. You can discuss solutions approaches with other students or on piazza, but absolutely no code sharing/copying is allowed. Your code will be extensively tested for potential plagiarism with any other student or with existing online solutions of related past semester projects.

8. Deliverables

Follow these strict guidelines for the project submission:
Compress and submit the following inside
`cs6210-project03-<GTusername>`**`.tar.gz`**

- `report-project03-<GTusername>.pdf`
- `test_team.txt`
- `gtfs.{hpp,cpp}`
- `Makefile`
- `run.sh`
- `tests/Makefile`
- `tests/test.cpp`

Do not include any object or library files. Your code should compile and run on the provided class servers. The `test_team.txt` should include your name and GT ID, together with your testing collaborator, if any.

9. Additional Readings

Refer to the following papers, which you can find in `canvas/files/papers/`:

- M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. 1994. **Lightweight recoverable virtual memory.** ACM Trans. Comput. Syst. 12, 1 (February 1994), 33-57. DOI=http://dx.doi.org/10.1145/174613.174615
- Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., & Steere, D.C. (1990). **Coda: A Highly Available File System for a Distributed Workstation Environment.** *IEEE Trans. Computers, 39*, 447-459.

10. Considerations

You're encouraged to fork the existing github repo that includes the skeleton code, but please do so in a private repo. Frequently commit your code under version control, do not rely on the availability of the class servers to access your codebase. Finally, kindly consider keeping your code in a private repo after finishing up with the class (e.g. available upon request) so as to help us reduce future plagiarism cases.