Project 3: GT File System

Due Date - Monday, Apr 3, 2023, AoE

0. Reminder

- Start to work this project as early as possible.
- Write code by yourself and be honest. You can form a team of up to 2 people for this project.
- Ask questions through Piazza or come to office hours.
- Test program on VM cluster before making compression and submission .

1. Introduction

To get full score, students will need to finish the program and a complete report, and pass the test cases.

The implementation details are listed at section 2; the information of test cases is listed at section 3; the requirment of report is listed at section 4.

This project has two **additional credit** parts:

- Additional API call implementation (10 points)
- Creative test cases (5 points)

Therefore, you can get at most **115** points. More details can be found at section 5.

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. 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.

Additional Readings

1. M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler, Lightweight recoverable virtual memory. SOSP'93 (1993).
2. M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, Coda: A Highly Available File System for a Distributed Workstation Environment, IEEE Transactions on Computers (1990).

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`: Presents the GT File System (GTFS) instance.
- `file_t`: Presents the file.
- `write_t`: Presents the write operation.
- `do_verbose`: Presents the flag that allows detailed printing of the file system's operations.

File System Operations

```
gtfs_t gtfs_init(string directory, int verbose_flag)
```

This function 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)
```

The length of the name of file can be 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 a pointer to the most recent data of an open file starting from `offset` up to `offset+length`. If there was no string written at that offset, the function should return a pointer to the empty string `""`.

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 enough information to disk 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`.

2. Implementation

We've provided a code skeleton that you need to fill in with the aforementioned functionality and all necessary checks. There are following files in directory `./gtfs`:

- `src/gtfs.hpp`: Includes the declaration of the above API calls and any additional functions.
- `src/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 implement more tests, do add them in the existing test file `test.cpp` and follow the **PASS/FAIL** mechanism used.

3. Test Cases

We've provided a script that compiles and runs your code for the given tests. For debug purposes, you can follow below commands:

```
// Enable verbose mode, print all actions taken in the file system
(./gtfs) $ ./run.sh 1

// Disable verbose mode, have a clear screen
(./gtfs) $ ./run.sh 0
```

The given tests only test the minimal functionality of the file system and help you understand how the API comes together. Your code needs to pass all these tests to receive points. TAs will run additional tests to check the full expected functionality, e.g. checks for possible errors given the described GT File System semantics, crash recovery, multi-threaded operation. For one specific case, 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).

GT File System semantics,
crash recovery
Multi–threaded operation

4. Report

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
  - data persistence
  - crash recovery
  - 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.

5. Additional Credit

There are two challenging parts for you to get more credits.

Additional API Calls (10 points)

Students who implement following two APIs and pass the related hidden test cases can get additional credit points.

```
int gtfs_clean_n_bytes(gtfs_t *gtfs, int bytes)
```

Similar to `gtfs_clean`. However, this function will clean only given number of bytes instead of the entire length of redo record. This is used to simulate a partial write of redo log to the original file before an abort or crash.

On success returns `0` else `-1`.

```
int gtfs_sync_write_file_n_bytes(write_t* write_id, int bytes)
```

Similar to `gtfs_sync_write_file`. However, this function will sync only the given number of bytes instead of the entire length of the write. This is used to simulate a partial write to redo log before an abort or crash.

On success returns `0` else `-1`.

Creative test cases (5 points)

Instead of just testing the correctness of your GTFS, do you have any idea of how it work with special file access situation?

Implement new tests in `test.cpp`, and document why this is interesting and whether your GTFS work as expects.