Project 3: GT FileSystem

Authors: Eva Grace Bennett, Gopesh Singal

Design Justification

When creating the system, the logging mechanism was based upon the `write_t` structure; whenever a process attempts to write to a file it has opened, a `write_t` object is created, serving as an in-memory log of the write attempt. This `write_t` object holds the data it wrote to the file, as well as the data it overwrote. In doing so, if the write was chosen to be aborted, then the changes made to the in-memory representation of the file, the `file_t` object, can be undone and reverted to just before the write operation was applied. When performing `gtfs_sync_write_file`, the `write_t` is then applied to the disk memory, being applied to the log tracking the write changes and to the file in question. This is to align with the specifications outlined in note @232 on Piazza, where under Method 2 it is stated we may "sync all corresponding pending in-memory logs by moving them to on-disk logs and applying them to the file" for `gtfs_sync_write_file`.

Data Persistence

When `gtfs_sync_write_file` is called by a process, the `write_t` structure passed in is applied to the on-disk logs and file, ensuring that even if the process were to crash, the data is now recoverable from the disk memory. The process would simply need to call `gtfs_open_file` once more to receive the persisted changes. In a sense, for a change written to persist, the process needs to call `gtfs_sync_write_file` or `gtfs_clean` to update the disk memory.

Crash Recovery

Similarly to what is discussed in **Data Persistence**, the system handles crash recovery through syncing the written data to the disk memory when either `gtfs_sync_write_file` or `gtfs_clean` are called. When these methods are called, then the disk version of the file is updated with the `write_t` object, therefore meaning that in the case of a crash, the process could open the file again and read whatever information was synced to the disk. However, if the information was not synced to the disk then the information would be lost, as in the implementation it was designed such that a user must sync to the disk to ensure that the information leaves the process memory. This is in-line with Method 2 from note @232 on Piazza.

Performance for Reading / Writing

For both reading and writing, only the specified length are being read or written into the in-memory `file_t`. This ensures good performance in that there is no need to call on the disk memory's representation of the file. Instead, the process is relying on the in-memory localization of the file that it has opened. Because read and write are relatively simple in the implementation, with the majority of the run-time being the creation of a `write_t` object or the use of `memcpy`, these methods remain relatively quick.

Additional Data Structures

Each `file_t` object keeps a vector of `write_t` objects applied to it. Because the `file_t` object is maintained locally by each process through the usage of `mmap`, this ensures that there is no synchronization errors between processes. The `file_t` object manages its own writes and these writes only come into play for synchronization when they are used in `gtfs_sync_write_file`. This vector serves as the in-memory log for the files.

Implementation Details

The following are the implementation details for each of the API calls of the GT FileSystem.

- `gtfs_init`: A `gtfs` object is initialized with the directory name, as well as a map to store the files that the directory will hold. If the directory does not already exist, it will be created.
- `gtfs_clean`: The pending `write_t` objects for each `file_t` object are synced, writing them to the disk memory. The memory logs and the disk logs are then wiped, cleaning the system.
- `gtfs_open_file`: The system checks if the file is available for opening and, if so, allows the process to open the file. A `file_t` object is created if it does not already exist.
- `gtfs_close_file`: The system closes the file if it is open, unlocking the file. Every pending `write_t` on the file are discarded since they were not synced before the close was called.
- `gtfs_remove_file`: The system removes the file from the directory and the `file_t` is removed from the map of the `gtfs`. Because you cannot call this method on an opened file, the effects of `gtfs_close_file` will also be present.
- `gtfs_read_file`: The process can read the open file, reading it from in-memory. This ensures that it will see the changes made from any writes, even if these writes have not been synced to the disk.
- `gtfs_write_file`: The process writes data to the file, creating a `write_t` object. This data written is only changed in-memory for now. The file on the disk remains unchanged.
- `gtfs_sync_write_file`: The information from a `write_t` object are applied to the disk, updating the disk log and the file in the disk. This means that the file is changed on the disk itself, and will reflect such changes whenever it is opened from now on.
- `gtfs_abort_write_file`: The `write_t` object is cancelled, and the data it overwrote is restored in memory. The object is deleted. Nothing is changed on disk since nothing was written to the disk yet for the `write_t` if it is able to be aborted.

Testing

Several test cases were added to check the basic correctness of our design and to check several edge cases. Test cases 1-3 are the basic cases provided with the project. The remaining test cases are custom cases. Each test outputs pass/fail and are described below.

- Test 4: Only synced writes should be saved to disk in the case of a crash. The `crash_writer` makes two writes to the file but only syncs one. Next `crash_reader` checks that the on-disk file only contains the synced write to PASS.
- Test 5: An existing file can only be opened if the new file length is as large as the existing file. A file of length 100 bytes is opened and reopened but with a length of 50 bytes. A nullptr for the file object should be returned to PASS.
- Test 6: When an existing file is opened with a file length larger than the current length, the existing file needs to be extended. This test opens a file of size 100 bytes and then reopens it with size 200 bytes. A helper api function, `gtfs_get_file_length`, returns the length of a file. The new file length should be 200 bytes to PASS.
- Test 7: Only one process should be allowed to open a file at a time. This test forks a child and then calls an opener function which attempts to open the same file. The parent and child should attempt to open the file at the same time, but the child will be unsuccessful since the parent does not close the file. The file returned by `gtfs_open_file` should be a nullptr to PASS.
- Test 8: The most recent version of a file should be read. This test makes two writes to the file and then reads the file. First **Hello** is written and then **World** writes over it. `gtfs_read_file` should return **World** for the test case to PASS.
- Test 9: If a process attempts to read a file at an offset that is longer than the existing file, an empty string, "", should be returned by `gtfs_read_file`. This test case attempts to read a file of length 100 bytes at offset 200 bytes. `gtfs_read_file` should output "" causing the test to PASS.