Project 3

Algorithm Description

We work with three structures:

- A buffer (acts as a recoverable memory segment or RVM)
- A log file (acts as a disk log)
- A source file

The files are created inside the specified directory. The log file differs from the original file in that ".log" is appended to the end.

During the file's opening, we transfer the entire contents of the file to the mapped_data buffer (recoverable memory segment) and then work with it to read and write data.

Before any data modification is done, the original data from the mapped_data buffer is transferred to the undo_data buffer. Thus, the initial contents of the file can be restored in the event of a failure.

After that, you can modify the data in the mapped_data buffer using the gtfs_write_file function. Any change (transaction) can finish successfully (gtfs_sync_write_file), then the contents from the mapped_data buffer are transferred to the log file, or a problem may occur, and the transaction will be discarded (gtfs_abort_write_file); in that case, the value from the undo_data buffer will be returned to the recoverable memory segment. The log file records the modification's length, the offset, and the data, which was received by calling the "gtfs_write_file" function. The log file is needed to ensure crash recovery, as it stores all the data of successful transactions. Log file's data is transferred to the source file during the call of gtfs_close_file, gtfs_clean, and gtfs_open_file (if suddenly a failure occurred after the transaction was completed successfully, but the source file was not closed because of a crash). After the data is transferred from the log file to the original file, the log file will be deleted. The recoverable memory segment is cleared when the file is closed.

In case we want to undo all changes for the last transaction, we can call the gtfs_remove_file function, which will delete the log file without transferring the data to the original file.

Each modification has its unique identifier, which is assigned during the gtfs_write_file function and is used for further work during the call gtfs_sync_write_file and gtfs_abort_write_file functions. To correlate the file and the transaction identifier, we use map <tr identifier, pointer to the required file structure>.

Mutexes were used to synchronize streams. We also introduce a flag «is_used» to check that only one process uses a file at a time.

Since we are working not with an actual file on disk but with its mapping into virtual memory, the read/write speed will be comparable to local machine’s RAM's rate, and will be faster. Also, all changes occur in virtual memory and do not physically modify the file. If changing the data ended successfully, only after that we transfer the data to a physical file. This ensures persistence.

Data structures

Gtfs - contains the name of the directory, the initialization flag (checks if the selected directory is initialized), and an array with pointers to all open files in the chosen directory (files_db). This array is used to avoid multiple threads working with the same file.

File - contains the name of the file, file's lenght, the path to the log file (log_filename), the path to the source file (home_filename), pointer to the buffer used as a virtual memory segment (mapped_data), the flag to check if the file is being used, the flag to indicate that data from the file has been transferred to the virtual memory segment, the initialization flag, a buffer for reading data, and a structure for the transaction queue.

Write - contains the file name, offset value, file's size, temp buffer for crash recovery (undo_data), the transaction's unique id, and flag to check if the file is being used.

typedef map <string, shared_ptr <file_t>> filesdb_t - used to find a match between pointers to files being used and their names.

typedef map <const string, shared_ptr <gtfs>> gtfsdb_t - used to find a pointer to a GTFS instance by directory name.

extern gtfsdb_t g_gtfsdb is an array of initialized directories.

map extern <uint64_t, file_t *> g_trans_map - used to find a match between a transaction id and the file we opened.

Libraries used:

#include <map>. Maps are associative containers that store elements in a combination of key values and mapped values that follow a specific order. No two mapped values can have the same key values.

#include <list>. Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions. Used to work with trans_queue.

#include <vector>. Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Used to work with some buffers .

API Description

gtfs_t * gtfs _ init (string directory, int verbose_flag)

The function first checks if the chosen directory exists in gtfs. If the previous check is valid, it returns a pointer to it. If not, it creates a directory and fills in the data of the gtfs_t structure. In other words, it changes the initialization flag, assigns the gtfs-> dirname variable to the name of the created directory, and preserves the correspondence between the directory name and gtfs in g_gtfsdb. Returns NULL if an error occurs at any stage.

int gtfs_ clean (gtfs_t * gtfs)

The function is used to transfer all successful changes from the log file to the source file. To do this, we go through the entire array of opened files (files_db) for a given gtfs_t and use the gtfs_truncate_log function for all found files. On success returns 0 else -1.

file_t * gtfs_open_file (gtfs_t * gtfs, string filename, int file_length)

The function first checks if a file with the given name is open by another thread. For this, it traverses the entire array of opened files (files_db). If it finds it, it returns an error since the file can only be opened by one process at a time.

Next, it checks if the log file for the given file exists (log_file_exist = true). If the check is valid, we transfer all successful log file changes to source files.

Next, we check if the original file exists (home_file_exist = true), then open it and assign its actual size to the file_size variable using the lseek (fd, 0, SEEK_END) function. Following, we compare the size of file_length and file_size. If file_size> file_length, then output an error because cut actions will lead to data loss. Otherwise, we expand the size of the original file to the desired one. Allocate memory for the mapped_data buffer and transfer the contents of the file to it.

If a file with the given name does not exist, then create it and allocate memory for the mapped_data buffer of "file_length" size.

Finally, we change the values of the file structure fl-> is_mapped = true (data from the file has been transferred to the virtual memory segment), fl-> is_used (the file is in use), and add the file pointer to the array of opened files.

On success returns the new or existing file representation else NULL.

int gtfs_close_file (gtfs_t * gtfs, file_t * fl)

The function closes all files and transfers successful changes using the gtfs_close_file_and_ clean( ) function , and also clears the array of opened files. On success returns 0 else -1.

int gtfs_remove_file (gtfs_t * gtfs, file_t * fl)

The function checks if the file is used if (!fl -> is_used), if not used, it deletes the original file and the log, otherwise it gives an error. On success returns 0 else -1.

char * gtfs_read_file (gtfs_t * gtfs, file_t * fl, int offset, int length)

The function returns a pointer to data from an open file, starting from offset up to offset + length. To do this, we check if the file is open, and if so, copy the data from the mapped_data buffer to read_buffer using the memcpy function. On success it returns a pointer to the read_buffer else NULL.

write_t * gtfs_write_file (gtfs_t * gtfs, file_t * fl, int offset, int length, const char * data)

The function fills in the data of the write_t structure. First, we copy the data from the data buffer (actually from the mapped_data buffer, since write_id-> data = (char *) fl-> mapped_data) into the undo_data buffer, in case the processing will be interrupted. Then it writes the information of the write_t structure to the transaction queue to keep correspondence between the transaction id and the file. Then we make changes to the data buffer starting at offset. 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)

The function finds a pointer to the required file structure by transaction id. After that, we check if the log file exists; if not, then we create it. Next, we write data from the virtual memory segment (in our case, the data buffer) to the log file. This action ensures that the original file can be restored if a crash happens. On success returns the number of bytes written, else -1.

int gtfs_abort_write_file (write_t * write_id)

The function is used to restore the content of the virtual memory segment to the state before the gtfs_write_file () function call, for this we copy data from the undo_data buffer to the data buffer from a specified offset. On success returns 0 else -1.

bool gtfs_truncate_log (string home_file_name, string log_file_name)

The function checks if the source file and the log file exist. Then we check if there are any changes in the log file that have not yet been transferred to the original file. If they exist, we transfer the changes to the original file. After successful transfer, we delete the log file . On success returns true else false.

int gtfs_close_file_and_ clean (file_t * fl)

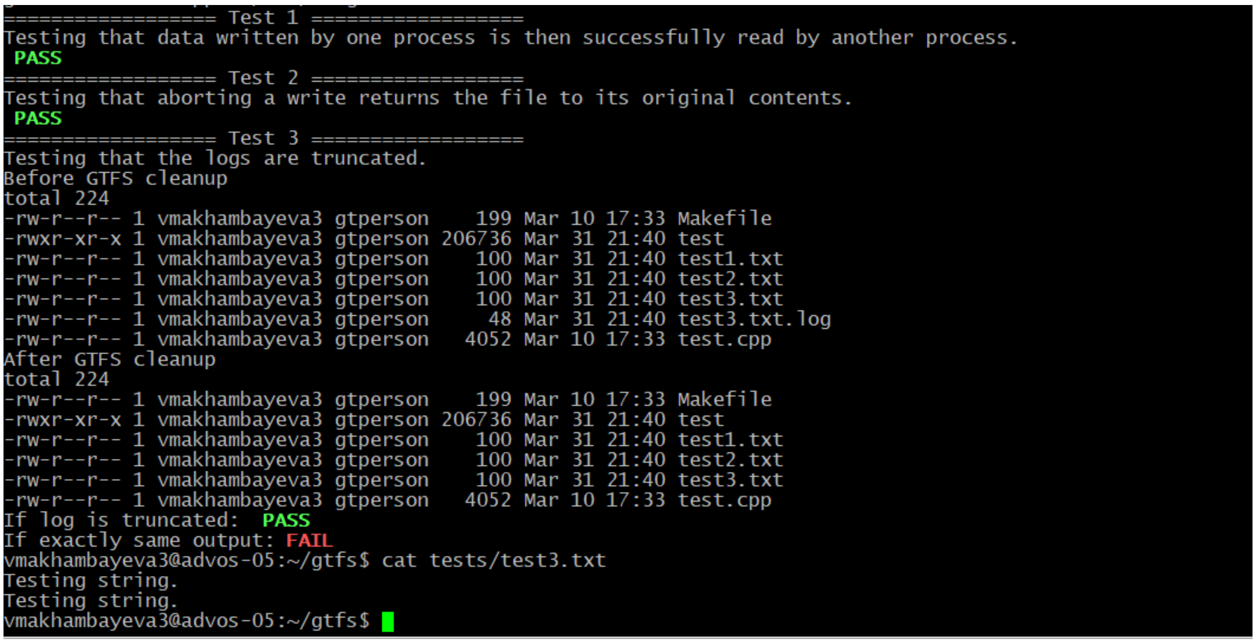
The function transfers data from the log file to the source file using the gtfs_truncate_log function and clears the transaction queue and virtual memory segment.

int gtfs_sync_write_file_n_ bytes (write_t * write_id, int bytes)

The function is identical to gtfs_sync_write_file. The only difference is that the entered number of bytes will be written to the log file. However, if the bytes variable's value is greater than the length of the changes made due to gtfs_write_file, only the modified data will be transferred to the log file.

Tests

We can see that all tests are passed.



```
================== Test 1 ==================
Testing that data written by one process is then successfully read by another process.
 PASS
================== Test 2 ==================
Testing that aborting a write returns the file to its original contents.
 PASS
================== Test 3 ==================
Testing that the logs are truncated.
Before GTFS cleanup
total 224
-rw-r--r-- 1 vmakhambayeva3 gtperson    199 Mar 10 17:33 Makefile
-rwxr-xr-x 1 vmakhambayeva3 gtperson 206736 Mar 31 21:40 test
-rw-r--r-- 1 vmakhambayeva3 gtperson    100 Mar 31 21:40 test1.txt
-rw-r--r-- 1 vmakhambayeva3 gtperson    100 Mar 31 21:40 test2.txt
-rw-r--r-- 1 vmakhambayeva3 gtperson    100 Mar 31 21:40 test3.txt
-rw-r--r-- 1 vmakhambayeva3 gtperson     48 Mar 31 21:40 test3.txt.log
-rw-r--r-- 1 vmakhambayeva3 gtperson   4052 Mar 10 17:33 test.cpp
After GTFS cleanup
total 224
-rw-r--r-- 1 vmakhambayeva3 gtperson    199 Mar 10 17:33 Makefile
-rwxr-xr-x 1 vmakhambayeva3 gtperson 206736 Mar 31 21:40 test
-rw-r--r-- 1 vmakhambayeva3 gtperson    100 Mar 31 21:40 test1.txt
-rw-r--r-- 1 vmakhambayeva3 gtperson    100 Mar 31 21:40 test2.txt
-rw-r--r-- 1 vmakhambayeva3 gtperson    100 Mar 31 21:40 test3.txt
-rw-r--r-- 1 vmakhambayeva3 gtperson   4052 Mar 10 17:33 test.cpp
If log is truncated:  PASS
If exactly same output: FAIL
vmakhambayeva3@advos-05:~/gtfs$ cat tests/test3.txt
Testing string.
Testing string.
vmakhambayeva3@advos-05:~/gtfs$
```