CS 6210 Advanced Operating Systems
ASSIGNMENT 3: Inter-Process Communication Services

NAME: NIHAR MEHTA
GT USERNAME: nmehta80

Collaboration on testcases with:
NAME : KHUSHHALL CHANDRA MAHAJAN
GT USERNAME: kmahajan32

A. Design and Implementation

The design involves a file system which can open, read, write, close and remove files. Two kinds of logs are used in this file system: in-memory logs and on-disk logs. The log structure is as follows:

The in-memory logs include an in-memory undo log which is a copy of the original file chunk before any modification in memory and an in-memory redo log which is the modified chunk of the file. A chunk is defined with a filename, offset, length and data. It includes file content starting from the offset to offset + length. On the other hand, the on-disk logs only consist of redo-logs persisted to the disk. Each file has one on-disk log named as filename.bak. The file system is able to perform log truncation and provides crash recovery as well as efficiency in file reading and writing using logs. The implementation details are described below:

1. **gtfs_init:** This function creates an instance of the GT File System. It includes the name of the directory where the files are stored. A map is created that shall map the names of an open file to addresses of the memory mappings of the files. This map shall help to access the files quickly if there are in memory.
2. **gtfs_open_file:** This function opens a file if it exists, else creates a new file and opens it. The open function used is thus given read, write and create access. Next, the file size is computed after the file is opened. If the size is smaller than the file length specified, the file is extended using lseek upto the file length. The file is memory mapped to a contiguous virtual memory with a size equal to file length. Next, the filename and the address of the virtual memory are saved in the GTFS Map. This function returns a file instance, which has a filename, file length and the address of the virtual memory as its data members.
3. **gtfs_read_file:** Since the read operation can be performed only after the open file operation, the file must be in the GTFS Map. If there is a filename key in the Map, then get the address in the virtual memory to which the file is mapped to and read the

data of given length from the offset to the virtual memory address. If filename key is not present in the Map, then this operation cannot be permitted.

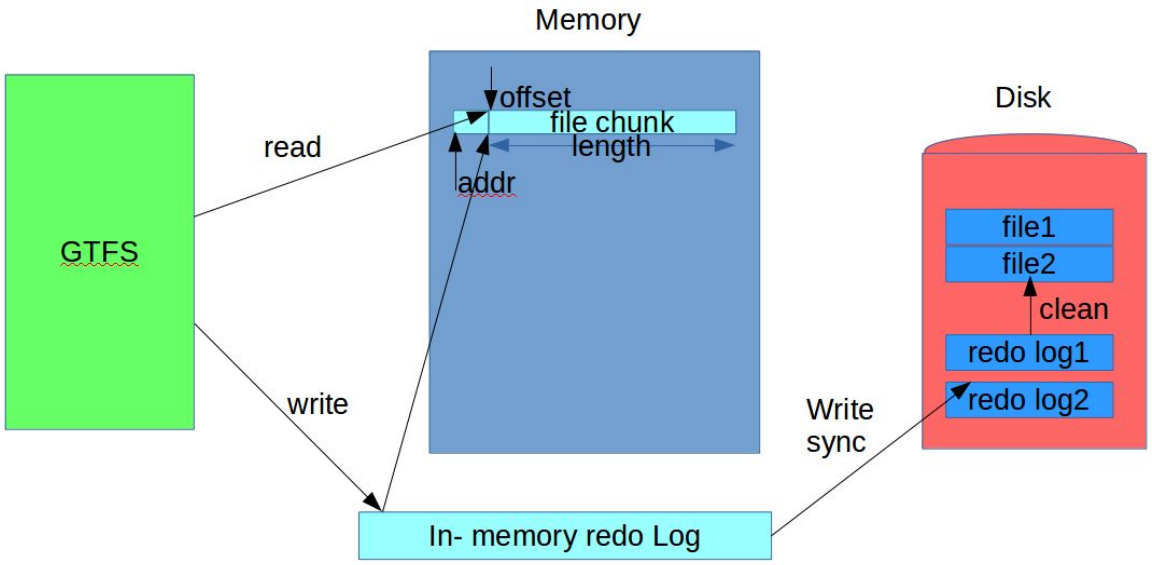
4. **gtfs_write_data:** This function writes the data in the file chunk in the virtual memory and in the in memory redo log but at the same time saves an undo instance of the original file chunk content. The write_id object that it returns consists of filename, offset, length, data, original_data (Undo log) and the address of the virtual memory. It first creates the original data copy as an undo log in theo write_id object. Then it copies the data to be written in virtual memory from offset to offset + length. Finally it copies the data in the write id object and returns it. Note that this function has made no changes in the disk logs or in the actual file. All changes exist in the virtual memory and the write_id object. A read after this function will read the modified content as the virtual memory has the modified content. To undo the modified content, a gtfs_abort_write_file function needs to be called which can restore the data back to its original state.
5. **gtfs_sync_write_file:** This function publishes the changes made in memory to the disk. It does not write the chunk to file directly. Its job is to save the in memory redo log to the on disk redo log. It uses the filename to check if an on disk redo log file exists. If it exists, then this function appends the chunk data to the on disk log file. If it doesn't, it creates a new on disk file named filename.bak and writes the chunk to this new file. These on disk redo logs accumulate as time proceeds. In this file system, we have chosen to apply these chunks to the actual file when the file is closed or when the logs are cleaned. Once the changes are successfully applied, it is safe to remove the log files. This is called log truncation. Once this function has been called, a crash in the system can still recover the file from the saved disk logs. It appends the chunk details in the on disk log file in the following format:

   <filename>
   <offset>
   <length>
   <data>
   @@@###$$$ (separator)

   The sync_write_file_function also writes these content to the in-memory file as the data may have been modified by previous unsynced writes.
6. **gtfs_abort_write_file:** This function is responsible for cancelling a write operation and thus reverts the data in the chunk in memory to the original version before the write operation. If in case there are any unsynced write operations, the in-memory file will be modified after they are synced (persisted to disk).
7. **gtfs_close_file:** This function closes the open file and truncates its logs. It first checks if there is any on-disk log file corresponding to the file to closed. If the on-disk log file exists, it reads the file one by one and starts applying the modifications to chunks sequentially. This is done by memory mapping the file with MAP_SHARED parameter so that changes made in the memory are flushed to the disk. The separator helps in differentiating the chunks. Finally it unmaps this shared version of the file and unmaps the original file too from the memory. It removes the filename key from the GTFS Map. Thus everything related to the file is detached and the file now has the persisted information and the on-disk logs of the file are cleaned.

8. **gtfs_remove_file:** This function removes the file and its on-disk redo log file from the file system. It removes the filename key from the GTFS Map. It also unmaps the file from the virtual memory.
9. **gtfs_clean:** This function truncates all the log files in the disk. It basically loops through all the keys in the GTFS Map. These are the open files and their logs have not been cleaned and applied to the original files. Thus it opens all the log files and applies them to the files similar to the gtfs_close_file procedure. Only difference is that it does not unmap the original file.

The overall model can be summarized by this figure:



B. Changes in the data structures

1. The gtfs instance has a directory and a GTFS map that maps a filename of an open file to the address of the memory mapped file.
2. The file data structure has filename, file length and the starting address of the memory mapped file. The starting address helps to directly copy the contents from the offset in the virtual memory when the offset, length and the content are given. This has been used in the virtual memory
3. The write data structure has filename, offset, length, data, original_data and starting address of the file in memory. The original_data acts as an in-memory undo log and is used to retrieve the unmodified data whenever an abort operation is performed. The data acts as an in-memory redo log. This consists of modified data and when synced, this is pushed to the on-disk redo log.

C. Persistence, Protection and Performance

1. Persistence: The GT file system provides reliable persistence of data with the help of in-memory undo logs and on-disk redo logs. The in-memory undo logs are useful to

abort a write operation and thus bring the system back to the unmodified state. When the writes are synced, the in-memory redo chunk logs are copied to the on-disk logs. These on-disk logs are persisted versions of in-memory logs. These are truncated and applied to the actual files at a later stage. But the presence of logs ensure that the data persistence is achieved without actually opening the file again and again.

2. Protection: Crash recovery ensures the protection of the data. If there were no logs, a crash in the system could result in potential loss of data of an unclosed file. But here even if there is a crash in the system, the on-disk redo logs still exist and hence gt_clean can be performed so that all the logs are applied to the files and everything is up to date with limited loss of data (only unsynced writes are lost).

3. Performance: Performance boost is the USP of this file system. The files are opened in memory once and then every read/write operation happens in memory and the latencies involved with the disk are eliminated. The log chunks ensure that only the area to be modified is saved and not the entire file. The chunks are also stored locally and even when they are synced only the chunk is transferred to the disk. This ensures space and time optimization and hence better performance. The synced logs are applied to the disk at a later stage when more CPU cycles are available and thus ensures less contention with the I/O operations. This improves overall performance.

D. Description of Test cases

1. test_write_read()
   Call the writer and once writing is complete, call the reader. The writer first initializes the gtfs instance. Then it opens the file, writes to the file, syncs the write and closes the file. The reader initializes a new gtfs instance, opens the file, reads the file and checks if the data read is same as the data written and finally closes the file. After closing the file after write sync, the file is changed and modifications are reflected in it. This shows that open, read, write and close are correctly working.

2. test_abort_write()
   Here, two functions are tested. One is the gtfs_sync_write_file function and the other is the gtfs_abort_write_file function. It syncs write 1 and aborts write 2 and reads the file after the abort. The file reflects the write1 modifications but write2 chunk remains unmodified. This shows that sync and abort are working correctly.

3. test_truncate_log()
   This test cleans the logs and applies them to the corresponding files. It initializes a gtfs instance, opens a file, performs write 1, write 1 sync, write 2, write 2 sync and then cleans the logs. It is observed that the log corresponding to this test and this file is removed and the file on disk reflects the changes incorporated by writes 1 and 2. This shows that the clean function is working correctly.
   Thus the code works on all the provided tests. Additional test cases are described below:

4. test_unsynced_write_read()
   This test checks if an unsynced write modifies the content in memory and an abort reverts it back to the original content. For this test, a new gtfs instance is created, opens the file, performs write 1, write 1 sync, read 1, write 2 read 2, write 2 abort, read 3 and closes the file. The read 1 incorporates changes by write 1, read 2 incorporates changes by write 2 and read 3 reverts back to write 1. This shows that write only affects local memory and does not change any on-disk contents.

5. test_abort_sync()
   This test reverts the previous operations. It aborts the first write and syncs the second write. Also, it performs both the writes first and then aborts and syncs the writes. After write 1 is aborted, the in memory file has the original data before write 1 and after write 2 is synced, the both in memory and on-disk versions of the file reflect the changes by write 2. This is expected behavior as we assume that the changes the final file should follow the order in which the writes are synced. Thus if write 2 is synced before write 1, the file should have write 2 modifications first and write 1 modifications later.

6. test_close_before_open()
   In this test case, we try to close the file before opening it. This throws an error by returning -1 from gtfs_close_file()

7. test_remove_before_create()
   In this test case, we try to remove the file before creating it. This throws an error by returning -1 from gtfs_remove_file() saying that unable to remove the file which doesn't exist.

8. test_filename_too_long()
   When the filename is too long, the file should not be created. gtfs_open_file returns a NULL file saying that the filename size exceeds maximum file length

9. test_crash()
   Here, a crash is simulated by aborting a process before completion. This is done with a writer function which aborts before closing the file. Thus the writes are synced to the redo logs on disk. But these logs haven't been yet applied to the files on disk. Thus when the crash happens and we read the file then the file shows empty. But then we close the file which results in the cleaning of the logs of the file. Now when we open the file again and read its contents, we find that the persisted data that was synced to the disk has been recovered and it reflects in the file contents. Thus we have shown that the GTFS system has crash recovery functionality.

E. Conclusion

Thus GTFS is an efficient file system that focuses on persistence and crash recovery and boosts performance of I/O and log cleaning operations.