Operating System: Project 3

Instructed by *Wei Xu*

Due on Dec 13, 2020

Data Structures

Our system consists of data structures corresponding to physical and logical entities, which keeps complete information to restore a consistent state of LFS. To make read and write easier, they are all structs and arrays (instead of classes). Detailed declarations can all be found in header file `utility.h`:

- The disk file (representing hard disks) is usually read in blocks and written in segments. For clarity, `block` is an `char[1024]` array. For parameters we may simply use `char*` or even `void*`.
  - *File data blocks* are struct-less, which means they are plainly bytes without extra information.
  - *Directory data blocks* consist of 16 directory entries in type `dir_entry`. Each entry contains a `filename` field as `char[60]`, and a `i_number` field as `int`. The maximum length of name can be modified, as long as the total size of a `directory` is 1024 B.
  - *Inode blocks* have fixed internal structure defined as `struct inode` (of length 1024 B exactly).
  - *Indirect blocks* are not separately structured for the sake of convenience. Rather, we borrow inodes to represent indirect blocks (with mode set to `-1`). Note that these "fake" inodes are also recorded in local `inode_map`s and the global `inode_table` (see below), but direct access to them will return an error. ***Note:*** *this design not only makes the path-traversing procedure easier, but also reduces the update cost, since we only have to update a few inodes rather than the complete chain.*
- Segment is a logical unit for efficient write-backs and metadata management. Since we do not explicitly read segments out, we only have to maintain a `segment_buffer` in memory. Segments consist of consecutive blocks: the first 1008 of them are normal blocks (declared above), while the last 16 of them store segment metadata, including *inode map*, *segment summary* and *segment metadata*.
  - *Inode map* is an array consisting of inode map entries (`struct imap_entry`). An inode map entry consists of `i_number` and `inode_block` fields, representing a map from inode number to its block number. There may be several entries with identical inode number even in the same segment, but the last one always stands for the latest version, and is cached in a linear table `inode_table`.
  - *Segment summary* is an array consisting of summary entries (`struct summary_entry`). An inode map entry consists of `i_number` and `direct_index` fields, where the latter stands for the offset of the data block within the file, represented using the index in `index[]` (write `-1` for inodes).
  - *Segment metadata* is an array consisting of other metadata (up to 256 bytes that remains free). Currently we maintain `updata_time` (last write-back time) and `cur_block` (next block number).
- Following the 100 segments, we store some metadata for the whole file system.
  - *LFS "superblock"* (as `struct superblock`) keeps basic properties of the file system. In our project these are all constants, so we only write them once, and never directly read them.
  - *LFS checkpoints* (as `struct checkpoint`) keep periodical state snapshots (most global variables in memory) of the file system. The two checkpoint fields will be alternatingly written to ensure completeness of the snapshot. Checkpoints are read only at initialization state (for crash recoveries).

We also declare several constants to describe the properties of the system. Theoretically, any *coherent* set of constants should work fine with the system, but we only test with the constants provided in the code.

Code

Functions in the system are classified into different abstract levels for clarity and robustness. Header files `*.h` contain functions that can be called from other files, while source files `*.cpp` contain their implementation and some auxiliary files. Non-interface functions are equipped with explanatory comments.

- `utility.*` contains all global data structure declarations, constants, global variables (line 173-184), debug switches (190-204) and behaviour flags (206-207, 213-215) (in header file). It also provides *lowest-level I/O interfaces* (a collection of "read"s and "write"s) and *lock interfaces* (not implemented yet).
- `path.*` contains functions regarding path resolution and path traversal (`locate()`).
- `blockio.*` contains *higher-level I/O interfaces* that handle segment "read"s and "write"s. These functions automatically locate the data (in memory / on disk), append blocks at the end of the segment, update segment metadata, write-back full segments, and generate checkpoints.
- `file.*`, `dir.*`, `buffer.*`, `metadata.*`, `stats.*`, `perm.*` and `system.*` implement necessary FUSE interfaces. `index.*` manages these interfaces, and `main.cpp` passes these interfaces to FUSE.
- `logger.*` and `print.*` provide output interface and pretty-print functions for all data structures.

Details for Requirements 7-10

- *Requirement 7:* `chmod` and `chown` by modifying metadata in inodes.
- *Requirement 8:* A global lock is declaired in `lfs/utility.cpp`. For simplicity in project 3, all functions require this lock when start and return this lock before return.
- *Requirement 9:* We implemented a segment buffer, and flush it whenever `sync` is called.
- *Requirement 10:* `init` function in `lfs/system.cpp` scans the permanent storage file on disk, and restores the file system before crash. It only takes into account segments which have been successfully written to disk, which can be determined by extra timestamps recorded in each segment. Inode maps can be restored by reading segment summary and imap table in each segment.

Tests

Open terminal in `proj3` directory, then run `bash test.sh`. This will execute all five tests. Refer to `test.sh` for compile options if you want to run each test separately.

`testfile.cpp`

A test for operations `open`, `create` and `write`. This test creates a file with 100 characters. Copy the binary file to an empty directory and execute `./testfile`.

If the file system functions properly, there should be no errors.

`testfile2.cpp`

A test for operations `create`, `write` and `mkdir`, an implicit requirement is thread-safety. This test creates 1000 directories, each with a file inside. Copy the binary file to an empty directory and run `./testfile2`.

If the file system functions properly, there should be no errors.

`testmkdir.cpp`

A stress test for block-segment management and operation `mkdir`. This test creates directories named $0, 1, 2, \ldots, n-1$. Copy the binary file to an empty directory and execute `./testmkdir <n>`.

If the file system functions properly, there should be no errors.

Tests continued on next page...

`testrmdir.cpp`

A stress test for block-segment management and operation `rmdir`. This test creates a tree structure of n directories first, then keeps removing a random directory until all directories are deleted. Copy the binary file to an empty directory and execute `./testrmdir <n>`.

If the file system functions properly, `testrmdir` should not exit due to assertion failure.

`testconcurrency.cpp`

A stress test for block-segment management and thread-safety. This test invokes n threads. Each thread creates m directories, each with a file inside. Copy the binary file to an empty directory and execute `./testrmdir <n> <m>`.

If the file system functions properly, there should be exactly $n \times m$ directories.

Command-line Tests

Open a shell in directory `lfs`, and execute `./fuse disk100Mi` first. Then you are free to try any of the following command-line tests. To deal with file name conflicts between tests, you may directly use `rm -rf *` to wipe LFS. These tests are based on Linux shell commands, so the correct results can be obtained by trying on a real Linux system (however, updates for `atime` may be slightly different).

Note: *due to the implementation of FUSE, commands are executed under the permission of* `others`*. This should be dealt with caution when analyzing the results of the following tests.*

Test for permission control Run through the following commands to test permission control of files and directories. Use `chmod` to change permission. Directory should contain some files initially.

(1) Files. Under permission `774`, file is readable but not writable; under permission `776`, file is both readable and writable. It is trickier to test for `772` (file is writable but not readable), and you have to write a simple C++ program. ***Note:*** *file permission is* `664` *by default, so we manually run* `chmod 666` *below.*

(2) Directories. Under permission `774`, `772` and `771`, the directory (`a`) can only be read (e.g. `ls a`), write (e.g. `touch a/f.txt`) and accessed (e.g. `cd a`), respectively. Permissions are composable.

Note: *you may disable permission by flags* `ENABLE_PERMISSION` *(for internal control by internal "*`if`*"s) and* `ENABLE_ACCESS_PERM` *(for external permission queries through* `access`*), since they follow different mechanisms. You may refer to the manual below for details.*

Test for timestamps Run through the following commands in the first column of the table.

Commands	`stat ?`	no flags	`nodiratime`	`nodiratime & relatime`
`mkdir a`	`a`	`a`, `m`, `c` are initialized to the same.		
`touch a/x.txt`	`a`	`a`, `m`, `c`	`m`, `c`	`a`, `m`, `c`
`ls a`	`a`	`a`, `c`	—	—
`mv a b`	`b`	`c`	`c`	`c`
`ls b`	`b`	`a`, `c`	—	`a`, `c`
`chmod 666 b/x.txt`	`x.txt`	`c`	`c`	`c`
`echo "abc" >> b/x.txt`	`x.txt`	`a`, `m`, `c`	`a`, `m`, `c`	`a`, `m`, `c`
`cat b/x.txt`	`x.txt`	`a`, `c`	`a`, `c`	—
`mv b/x.txt b/y.txt`	`y.txt`	`c`	`c`	`c`
`cat b/y.txt`	`y.txt`	`a`, `c`	`a`, `c`	`a`, `c`

Note: *we implement different* `atime` *policy as Linux does. You may turn on* `nodiratime` *by setting* `FUNC_ATIME_DIR`*, and turn on* `relatime` *by setting* `FUNC_ATIME_REL`*. You may refer to the manual below.*

Test for common commands `ln`, `mv` and `cp` Run through the following commands.

- `touch a; chmod 664 a; echo "abc" >> a; ln a b`: use `cat` and `ls` to verify that they are identical. ***Note:*** *`stat` may return different inode numbers, but this seems to be a FUSE bug. By opening debug switch `DEBUG_METADATA_INODE`, you may verify they are actually the same.*
- `mv b c; echo "def" >> c`: after renaming the link, `ls` will return `a, c`, and both will contain "def".
- `touch d; chmod 666 d; cp c d; echo "ghi" >> d`: by copying a hard link, a new file completely irrelevant of the linked file is created. Only file `d` contains "ghi", while files `a` and `c` remain the same.

Test for crash recoveries The FUSE background console can be called out by adding `-f` argument after mount path (e.g., `./fuse disk100Mi -f`). The system can be crashed by sending `Ctrl-C` to the console, or use debug tools like `gdb`. ***Note:*** *to make it even harder, you may avoid generating checkpoints on exit (by commenting line 232 out in `system.cpp`). Our system survives crashes without checkpoints.*

Manual

- To compile, execute `scons` in `lfs/` directory. If any issue happens, you may need to use Ubuntu 20.04 and install `scons`. The environment we use is equipped with `scons 3.1.2` and `gcc 9.3.0`.
- To mount file system, either execute `bash buildfs.sh` in `proj3/` directory, or manually execute (this also applies to the "echo file system" in task 1) `./fuse <mount directory> <options>`. For example, to show debug and error messages in background console, you should append option `-f`.
- Line 190-204 of `lfs/utility.h` contain several debug switches. To make them work, add `-f` first.
  - To enable "echo" in LFS, toggle on `DEBUG_PRINT_COMMAND` (*on by default*).
  - To print the procedure of name resolution (in `locate()`), toggle on `DEBUG_LOCATE_REPORT`.
  - To print checkpoints after each update, toggle on `DEBUG_CKPT_REPORT` (*on by default*).
- Line 206-207, 213-215 of `lfs/utility.h` provide some flags for modifying system behaviour.
  - `FUNC_ATIME_DIR` is a flag for directory `atime` updates. When it is turned on, access timestamps of all files *along the path* will be updated. Note that this will be very space-consuming.
  - `FUNC_ATIME_REL` is a flag for "relative" `atime` updates. When it is turned on, access timestamps of all files and directories will be updated only if (1) `atime` is earlier than `mtime` or `ctime`, or (2) it has been a long time since last update (longer than `FUNC_ATIME_REL_THRES`.
  - `ENABLE_ACCESS_PERM` is a flag for external permission queries. When it is turned on, the `o_access()` interface will report true permissions of the request. ***Note:*** *all Linux commands use `o_access()` to request for file permission. Therefore, when you use it for a test, the internal permission control does not really obtain an opportunity to work, although it prints debug messages.*
  - `ENABLE_PERMISSION` is a flag for internal permission control.
  - Flags for `atime` are *off by default*, while flags for permission are *on by default*.
- Since we provide a full set of overloaded pretty-print functions (in `print.*`), we do not provide an explicit `block_dump` class (which is equally hard to be called from outside). If you want to print anything out for checking, just add appropriate print functions in `o_init` (probably after a `Ctrl-C` crash).

Limitations

We have not implemented garbage collection, so when the file system is full, we can not even delete files. This deficiency will be settled in Project 4.