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` array. For parameters we may simply use `char*` or even `void*`.
  - *Data blocks* are struct-less, which means they are plainly bytes without extra information.
  - *Inode blocks* have fixed internal structure defined as `struct inode` (of length 1024 B exactly).
-

We also declare a sequence of constants.

Code

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 execute `./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.

`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.*

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

Command-line Tests continued on next page...

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.

Manual

- To compile, execute `scons` in `lfs/` directory. If any issue happens, you may need to use Ubuntu 20.04 and install scons.
- 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>`.
- Line 190-204 of `lfs/utility.h` contain several debug options. To enable "echo" in log-structured file system (should add `-f` to options), toggle on `DEBUG_PRINT_COMMAND`.
- Line 213 of `lfs/utility.h` is a switch for directory access time updates. When `FUNC_ATIME_DIR` is 1, access timestamps of all files along the path will be updated.

Limitations

We have not implemented garbage collection, so when the file system is full, we can not even delete files.