Rutgers Operating Systems CS416

P4: Tiny File System with FUSE Library

By Thomas Milan Cuba, Ryan Lee Callahan
tmc231 rlc192
May 6, 2019

Objective

To simulate a functional file system infrastructure using a flat file as a “disk”, define our own versions of all standard file and directory operations, and support singly direct inodes for large files in the file system

Result

Absolute success! Note: Please be sure to check out “Usage” directly below for instructions on how to run our code as well as changes we made to base files.

Usage

- To start, set the macro “BLOCK_SIZE” in “block.h” to whatever you wish. **Make sure that “MAX_DNUM” and “MAX_INUM” in “tfs.h” are each no more than 8 times “BLOCK_SIZE”, as our file system’s maximum bitmap size is one block**. We unfortunately did not have the time (or willpower) to implement bitmaps that span multiple blocks.

- Open a command line and type “make” to compile the code. **We have edited the makefile to also create directories “/tmp/tmc231_rlc192” and “/tmp/tmc231_rlc192/mountdir”. Please make sure to use the latter name for any test case files.**
- Mount the filesystem with the command “./tfs -s /tmp/tmc231_rlc192/mountdir”.
- Run any tests you like. Note that our “benchmark” directory contains both “simple_test.c” and “test_case.c”, unmodified (we think), and the makefile in “benchmark” has been updated to compile and clean up both.
- When finished, unmount the filesystem with the command “fusermount -u /tmp/tmc231_rlc192/mountdir”. **Do this before typing “make clean” into the command line so any files do not get deleted!** (We don’t think Linux would let you, anyways; when we tried, we got something along the lines of “Device or resource busy”).
- Finally, type “make clean”. This will remove the directories “make” created along with their contents, in addition to the default settings.

Planning & Creation Process

It's easy to say that without the stub files, we (and likely many other people) would have been toast on this project. Following the instructions and the recommended guides, along with assorted searching online, the functions were all very straightforward and easy to implement. The excellent state of what was already given to us before we even began the project left us plenty of time to clean up and streamline our code, as well as polish off a few small details.

Essential Data Structures

sb: Our in-memory superblock (and our only in-memory data structure), for referencing the disk properties.

superblock, inode, dirent, bitmap_t: No need to explain these, as you know as much, if not more, about them than we do.

Division of Work

The workflow on this project was incredibly straightforward. Thomas implemented the driver functions (anything not directly called by the FUSE library, such as dir_find()) while Ryan implemented the FUSE system functions. Then, both of us meticulously debugged the project and agreed on how to pretty up our code.

Difficulties

This project went much more smoothly than either of us would have hoped. Regardless, there were obviously a few bumps along the road.

- Issues with checks and return values. So many of the functions either return 0 or some negative number, and sometimes one overlooked check resulted in us having to search for a needle in a haystack.
- Casting and memory copy errors. Since sometimes large char[]s are cast as smaller structs, there were some errors with reading incorrect data.
- Overwrite protection. Many of our older implementations were unstable and prone to overwrite the superblock.

Key Design Choices

- By globally calculating how many inodes and how many dirents can be stored in one block, we have made it so that BLOCK_SIZE needn't be a power of two. Whatever leftover space there is in each block is ignored. This leftover space, while it is taken up, does not contribute to the noted size of a file or directory; this is a purposeful design choice, whether it be correct or not.
- Because of how we parse dirents and both direct and indirect pointers, data blocks must be wiped to consist exclusively of the null terminator '\0' (ASCII 0) when freeing files or directories. This introduces an overhead that perhaps can be avoided with good use of some sort of

valid bit, but we figured this was outside the scope of this project.

- We tried our best to ensure that every function called directly by FUSE returned the correct Linux file system error codes. We specifically made use of ENOENT (for when inputted paths are not valid), ENOTDIR (for when directory operations are called on a file), EISDIR (for when file operations are called on a directory) and ENFILE (for when our file system is full).
- Our documentation is great (because it was 100% provided to us)!

Performance

Every function works as expected, with one extremely minor hiccup. Sometimes, when doing the “large file write test” in “test_case.c”, it would take a fairly long time (upwards of 25 seconds). We do not believe that it is a fault or inefficiency of our implementation, as most of the time the tests all happen very quickly (<1 second for all ten tests in “test_case.c”), but rather hardware performance. Other than this anomaly, performance is as you would expect from these functions.

Final Note

Thanks to the CS416 staff (Yujie, Baber and Professor Kannan) for an excellent final project and a great semester. This was a very fast paced and well taught class, and I feel like the projects did a great job of reflecting OS design on a micro scale. Take care and keep on coding! :)