CS416 Project 4: Tiny File System using FUSE Library

Aditya Verma (175007700)
Joshua Siegel (169007779)

May 2, 2019

Abstract

We implemented a simple file system called Tiny File System, built on top of the FUSE library. File systems provide the file abstraction. Standing between the user and the disk, they organize data and indexing information on the disk so that metadata like filename, size, and permissions can be mapped to a series of disk blocks that correspond to a file. File systems also handle the movement of data in to and out of the series of disk blocks that represent a file in order to support the abstractions of reading and writing from/to a file.

1 Introduction

This implementation of a Tiny File System works in the following way:

1. The user mounts our implementation of the Tiny File System in some directory.

2. An abstraction for the 'DISK' is used to simulate persistent memory. We call this a 'DISKFILE'. A given file system's DISKFILE has its unique identifying 'MAGIC_NUM' flag.

3. The user can then cd into the directory where the TFS is mounted and carry out basic UNIX terminal commands related to files and directories.

4. The user can also use system calls in their own user code to run programs that use the TFS.

The Tiny File System (TFS) carries out operations on the simulated disk (diskfile) in granularity of "BLOCK_SIZE". Any writes, reads, superblock updates, metadata updates, etc. are done by reading and writing chunks of "BLOCK_SIZE".

To keep track of the status (allocated or free) of blocks and inodes in the disk, we maintain 2 bitmaps. These are the "inode_bitmap" and the "data_bitmap". The bitmap is essentially an array of 1's and 0's. Once the bitmap is created, all indices are initialized to 0, which represents the unallocated or 'empty' state. When a block is written to, it's corresponding bit in the bitmap is set to 1.

As with any file system, the structure of the TFS is crucial to it's implementation. Therefore, to understand our implementation, it's structure needs to be understood. All the blocks in our disk are divided into groups in the following way:

1. **Block 0:** Block 0 defines the TFS. It holds the 'superblock', which contains information about the rest of the TFS.
2. **Block 1:** The Inode Bitmap is stored here.
3. **Block 2:** The Data Bitmap is stored here.
4. **Block 3 to Block (MAX_INUM/BLOCK_SIZE):** These blocks are used to store the inodes of the TFS. Inode 0 is reserved for the root directory.
5. **Block (MAX_INUM/BLOCK_SIZE+1) onwards:** These are the data blocks, which will store all the actual data for files and directories.

The inodes and data blocks are used to hold information about files and directories. The file system uses these structures to then carry out operations on these persistent memory blocks.

1.0.1 Assumptions

For the main codebase mentioned above, we made the following assumptions:

1. The largest file that is supported by the Tiny File System is **BLOCK_SIZE*(16+8*(BLOCK_SIZE/sizeof(int)))**.

2. The total number of entries (sub-directories+files) in a directory is **16*(BLOCK_SIZE/sizeof(struct dirent))**.

3. The directory that the user mounts on must be empty before mounting TFS.

2 Implementation

For this project, our primary goal was to create an efficient and correct file system that would manage persistent memory.

2.1 Design Specific Choices

We briefly mention below some design choices, some of which were at the direction of the CS416 staff.

1. **tfs_unlink()**: Here, we first decrement the amount of links present in the inode. If the number of links is now 0, we remove the inode, the data in the file, clear all the bitmaps, and remove the file from the directory containing it.

2. **tfs_rmdir()**: We only remove the directory specified if all of it's entries are empty. Meaning if there are any files or sub-directories within the directory, we DO NOT remove it, and instead operation not permitted is printed to the terminal. rm -r can be used to remove everything within the directory as well as the directory itself.

3. **dir_add()**: We decided when adding a dirent to the directory's data blocks, that adding to an empty slot in an allocated data block is better than allocating a new data block. So while going through the directory's dirents, if we notice an empty slot, we mark it to be added to, otherwise we allocate a new data block and add the dirent to the first entry in it.

4. **tfs_write() and tfs_read()**: We first calculate the number of blocks the offset is. If the offset is greater than 16 blocks, then we start reading/writing to indirect blocks. Otherwise we start from the offset and go until the 16th block, and read/write indirect blocks if there is still data to be read/written. For write, we check if the direct/indirect pointer is -1, and if it is we allocate a new data block to it, and if it's and indirect data block, initialize it to all -1. For indirect data blocks, we iterate through all BLOCK_SIZE/sizeof(int) direct pointers in the data block, and if they are -1 we allocate a new data block to point to.

5. **Unallocated Pointers**: We set any direct or indirect pointer to -1 to indicate that it points to garbage. If the pointer is not -1, we know it points to actual data blocks.

6. **Bitmaps**: We decided to read the bitmaps in from the disk and modify them and write them back all within the same function. We did this because if the function crashed at any point the bitmaps on disk wouldn't be accurate.

3 Testing and Results

Note: Testing was done on the following machines:

- kill.cs.rutgers.edu

We tested that our TFS implementation works for the following operations:

- Creating files and sub-directories.
- Writing to files. Both large writes and small writes were tested. We were able to successfully create a file, do a large write (or a bunch of large writes) to it, delete said file, and repeat this process about 2000 times.
- UNIX Terminal commands such as cat, ls, mkdir, rmdir, rm, touch were tested in the mounted directory and it's sub-directories.
- simple_test.c
- test_cases.c

We also created a few bash scripts that can be used to do a quick test of our implementation. These will be included in the submission file.

3.1 Timing and Disk Usage Analysis

We carried out analysis on the time efficiency of our file system. These analyses were carried out on the benchmarks provided to us. Each benchmark was run 5 times, to obtain an averaged result, as seen below:

1. **simple_test.c:** The average time this benchmark took during testing was: (0.563 + 0.551 + 0.466 + 0.434 + 0.525)/5 seconds = **0.508 seconds**. The breakdown of the disk blocks in use at the end of the benchmark is the following:
    - SuperBlock: 1 block
    - Inode Bitmap: 1 block
    - Data Bitmap: 1 block

- Inodes: 102 inodes or ceil(102/BLOCK_SIZE) blocks = 7 blocks
- Data Blocks: 8 blocks

Therefore, the total number of blocks in use after running this benchmark = 1 + 1 + 1 + 7 + 8 = 18 blocks.

The amount of data blocks makes sense since there are 8 data blocks for directory entries, since there is 1 directory entry block for the files directory, and 7 directory entry blocks for the 100 sub-directories. The amount of inodes make sense since there is 1 inode for the root, 1 inode for files, and 100 inodes for the sub-directories.

2. **test_cases.c:** The average time this benchmark took during testing was: (0.401 + 1.061 + 0.425 + 0.959 + 6.316)/5 seconds = **1.832 seconds**. The breakdown of the disk blocks in use at the end of the benchmark is the following:

   - SuperBlock: 1 block
   - Inode Bitmap: 1 block
   - Data Bitmap: 1 block
   - Inodes: 103 inodes or ceil(103/BLOCK_SIZE) blocks = 7 blocks
   - Data Blocks: 2058 blocks

   Therefore, the total number of blocks in use after running this benchmark = 1 + 1 + 1 + 7 + 2058 = 2068 blocks.

   The amount of data blocks makes sense since there are 8 data blocks for directory entries, 2 data blocks allocated for indirect pointers, and 2048 data blocks for actual data. The amount of inodes make sense since there is 1 inode for the root, 1 inode for the largefile, 1 inode for files, and 100 inodes for the sub-directories.

4 Conclusion

We successfully implemented our own Tiny File System with a simulated disk. We learned more about the inner workings of file systems and what happens behind the scenes of persistent memory, system calls and terminal commands, as well as how mounting different file systems work together. Overall this project taught us a lot about file systems and we had a lot of fun doing it.