RU File System using FUSE

Project 4

Authors: Krishna Sathyamurthy (ks2025), Abhinav Bharadwaj Sarathy (ab2812)

Project 4

Contents

1 Introduction

For this project, we'll be implementing a RU File System (RUFS) on top of the FUSE library. We are given with the following -

```
// A supreme structure which handles all the underlying data
struct superblock;

// Used to store all the directory entries in the file system
struct dirent;

// Used to store the meta data of files/directories in the system
struct inode;
```

Before jumping to the implementation, let's brief what we'll be doing here. First thing which is essential for a file system is the disk. We'll be seeing how we have implemented the disk as blocks, how the blocks are managed efficiently. Further, we'll see how files and directories are implemented to manage. And lastly how FUSE handlers are operated to support system calls.

2 Implementation of File System

2.1 Disk

The Disk implementation is the foundation to our file system. We have considered the disk to be a flat file of DISK_SIZE (32MB) bytes. Further we have partitioned the disks as blocks of BLOCK_SIZE (4KB). By doing this, it becomes fairly simple to manage the entities in the file system.

2.1.1 dev_init

This function takes the diskfile_path as input; which is assigned in the main function of 'rufs.c'. It creates a flat file of size DISK_SIZE bytes (32MB), which acts as our disk for the file system. In case of failure in opening the diskfile, we exit the program with error code EXIT_FAILURE.

2.1.2 dev_open

This function takes the char pointer diskfile_path as input and opens the diskfile, everytime the file system is mounted. It returns -1 when disk open fails.

2.1.3 dev_close

This function closes the diskfile when the file system is unmounted.

2.1.4 bio_read

This function reads entries from the block and its inputs are const int block_num and void *buf. Based on the block_num, this method identifies the block which is requried, and calls pread function to read the data from the block in diskfile into buf. pread returns a status, which can be utilized to identify, if the block is empty or block read has failed. In case of empty, we return 0; if failed, we return -1.

2.1.5 bio_write

This function inputs are const int block_num and void *buf. Based on the block_num, this method identifies the block which is requried, and calls pwrite function to write the data from buf into the block in the diskfile. We return -1, if the program encounters a write error.

2.2 Inode

Inodes are crucial, as they function as the blueprint of the file system. All the data related file and directory(see next section) entries will be stored. We utilize the *struct inode* to handle the metadata of the entities. These would be really helpful in establishing any sort of relation between the directories, and upholds the tree-structure of the directories and files by maintaining a mapping between them.

2.2.1 readi

This function functions similar to the *bio_read()*, it takes in the inode number *uint16_t ino* and a inode pointer. It further uses the s_block to identify the block in the disk, where the inode data is present. It finds the offset address within the block. Using the block, we call the *bio_read()*. We set the inode pointer to the result we fetched from the block read.

2.2.2 writei

The inode counterpart of blocks, it takes in the inode number *uint16_t ino* and a inode pointer and finds the block in the disk similar to *readi()*. Then we read the current data in the block, update the temporary node with the data from inode and do a block write, thereby updating the inode value.

2.3 Bitmap

Bitmaps are essential in both, inodes and blocks. The availability of Inodes and Blocks are resolved using bitmaps. There are five methods that rely primarily on these bitmap

2.3.1 get_bitmap

Takes the bitmap(ino/blk bitmap) and the inode/block number as inputs, estimates the char bit to get based on the given number.

2.3.2 set_bitmap

Takes the bitmap(ino/blk bitmap) and the inode/block number as inputs, estimates the char bit to set based on the given number.

2.3.3 unset_bitmap

Takes the bitmap(ino/blk bitmap) and the inode/block number as inputs, estimates the char bit to unset based on the given number.

2.3.4 get_avail_ino

The function traverses the entire bitmap, from 0 to max_inum. For every char byte, we use built in ctz to compute the trailing zero's position. If it is available, indicating there are free inode(s). In that case, we'll use the position which was returned by that function to update the bitmap inode and return this new position. If no inode is free, we return -1.

2.3.5 get_avail_blkno

The function is the block counterpart for *get_avail_ino()*, very similarly traverses the entire bitmap, from 0 to max_dnum. For every char byte, we use built in ctz to compute the trailing zero's position. If it is available, indicating there are free block(s). In that case, we'll use the position which was returned by that function to update the bitmap block and return this new position. If no block is free, we return -1.

2.4 Files and Directories

2.4.1 dir_find

This helper function will help us identify if the given folder/file name exists on any block/dirent of the given parent directory. The parent details, i.e., inode is retrieved using the inode number.

2.4.2 dir_add

This method is similar to earlier, except that if this function finds the same file name, it exists. Otherwise, it will try to find an empty block and add the entry to our parent node.

2.4.3 get_node_by_path

Also called the namei operation, this takes the path name and iterates through the path till we reach the end, or throws an ENOENT error. We use string tokens on the path, and split the names of directories/file using a delimiter ('/'). Using the inode number, we use dir_find to iterate through all the entries of the given directory and comparing that against the current token, i.e., path. Finally if everything results in a success, we load the inode from memory using the final inode number.

2.4.4 dir_remove

The function is responsible for removing the inode entry from the parent inode. Subsequently, all the block related details will be updated, i.e, this will also free any unused memory blocks if the current block of memory is empty.(*bio_write()*).

2.5 FUSE handlers

2.5.1 rufs_init

This function is invoked when the file system is mounted. We allocate memory for the superblocks and bitmaps which are necessary. If a diskfile already exists, we read the contents of this file and update all our primary data structures. Otherwise, this will call *rufs_mkfs()*.

2.5.2 rufs_mkfs

This function is the setup of our file system. We call the *dev_init()* to create a new diskfile. Later, we write all the information related to superblock on the disk. This is followed by initializing and writing the bitmap information onto the disk. Finally, we create the root directory and update the corresponding bitmap and inode.

2.5.3 rufs_destroy

This function stores the last known instance of all the data onto the disk, and de-allocates the data structures which are in memory. This includes the bitmaps, superblocks as well as calling *dev_close()* to close the diskfile.

2.5.4 rufs_getattr

This function takes the input path, finds the inode from the path using *get_node_by_path()*. If there's no inode found, we return the error code ENOENT(No such file or directory). Else, we update the stbuf with the stat of the directory/file inode.

2.5.5 rufs_opendir

This function returns 0 if it finds an entry in it's tree, otherwise it returns -1.

2.5.6 rufs_readdir

This function takes in path as an input, and finds the inode from the path using *get_node_by_path()*. Once found, it reads all the dir_inode's data block and identifies the directory entry. It copies each directory entry into filler.

2.5.7 rufs_mkdir

This function takes in path, and uses dirname() and basename() to identify the parent and base names. First, it checks if parent exits, and return ENOTDIR if not found. Later, it attempts to add a new directory, by checking for free inodes. This is followed by a call to dir_add() to add directory entry of target directory to parent directory where it updates the parent directory inode. Finally, the inode is updated in the disk.

2.5.8 rufs_create

This function is similar to *rufs_mkdir()*, i.e, it attempts to create a new dirent entry, which is followed by adding a new file entry onto the disk.

2.5.9 rufs_open

This function takes the path, tries to find the inode entry from the path using *get_node_by_path()*, returns 0 if found, -1 if not.

2.5.10 rufs_read

This function takes the path, gets the inode entry from the path using *get_node_by_path()*. Based on the size and offset, it identifies the data block and reads it into the read buffer from the disk. We have ensured we only starting reading the data from the offset into the buffer. Finally, we will return the total size that has been written onto the buffer.

2.5.11 rufs_write

This function takes the path, gets the inode entry from the path using *get_node_by_path()*. Based on the size and offset, it identifies the data block. We then write from the buffer to the disk. We have ensured we only start writing the data from the offset to disk. We then update the inode info of the base. Finally, we will return the total size that has been written into the disk.

2.5.12 rufs_rmdir

This function has been implemented similar to the linux implementation of rmdir. The folder will not be removed if there is any data inside the directory.

This function takes in path, and uses dirname() and basename() to identify the parent and base names. First, it checks if dir_inode exists for the parent and base, return ENOTDIR if not found. And then it unset the data block bitmap and inode bitmap in the disk. Also we reset the block and the inode to 0 in the disk. At the end, we call the *dir_remove()* to remove the directory entry of the base in its parent directory.

2.5.13 rufs_unlink

Very much similar to what *rufs_rmdir()* does, expect this method is called to delete files.

3 Benchmark testing

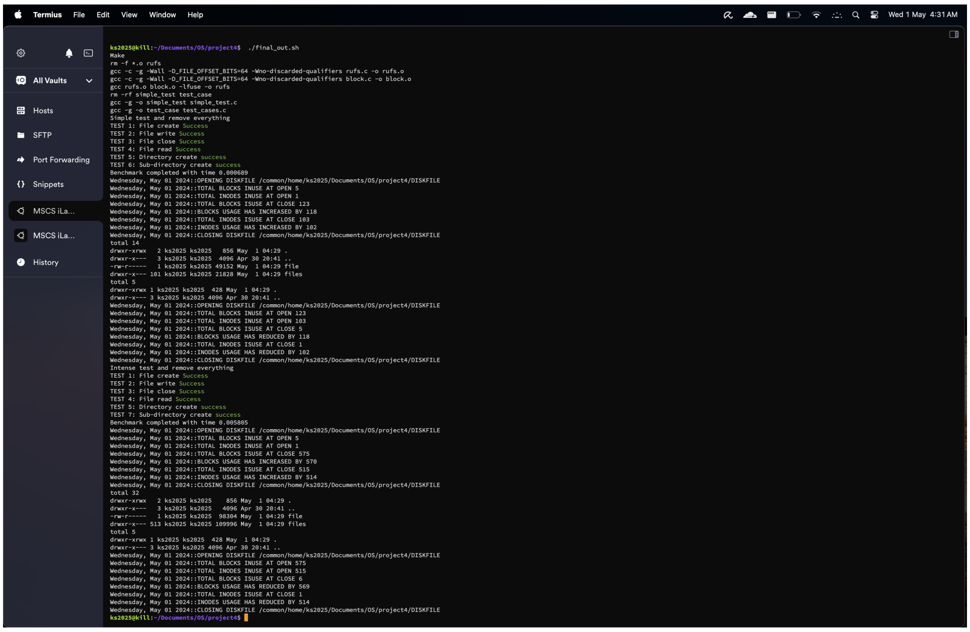

Figure 1: simple_test, test_cases output

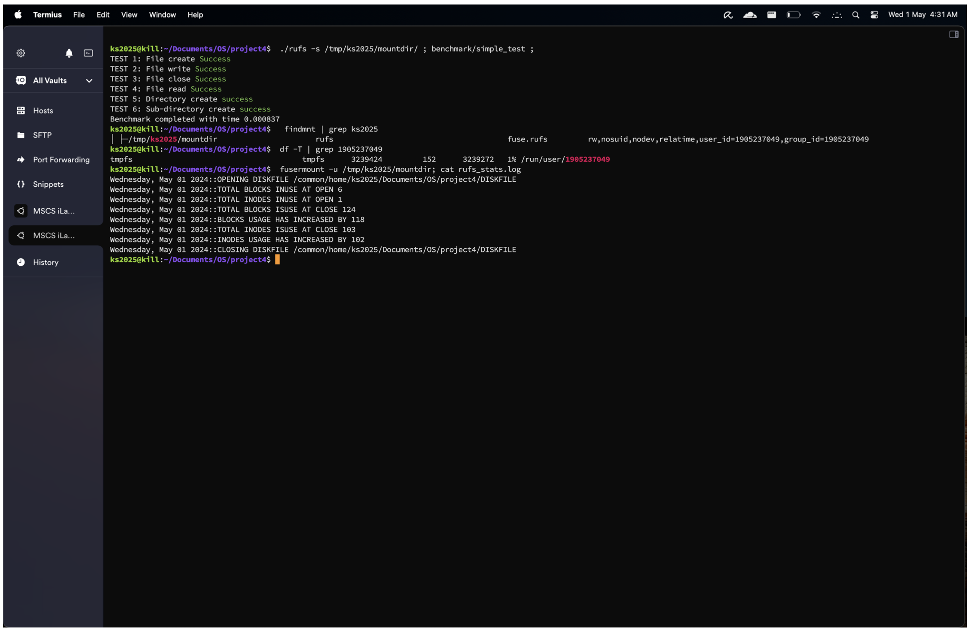

Figure 2: simple_test case with block count

figure 1) tests both the cases. In our implementation, we accounted for "indirect_ptrs" and increased the threshold of test_cases. simple_test took 0.000689 seconds to execute.
figure 2) shows the number of blocks that are being used by our disk file. We identified that the mount directory is using 152 blocks, when we run the command $df - T < id >$. But, when we print the same data onto our log file, we notice that it is only using 124 blocks instead. Please note, that these blocks also include the super block, and the number of blocks occupied by the inodes.

4 Challenges faced while implementing

4.1 Bitmaps

Instead of using bitmaps like we did earlier, we wanted to take a different route, and explored ways to reduce the total iterations made. We settled on $\_\_builtin\_ctz$ for calculating the available bits for a given char, and $\_\_builtin\_popcount$ to get the total bits used by that char.

4.2 Indirect pointers

The major roadblock I faced was identifying which block was freed and which wasn't. This was especially troublesome in $mkdir$ & $rmdir$. Debugging was not easy, and the n number of print statements didn't help either. After going through the code n number of times, removing the direct pointers entirely, I could pin point the cause of the issue and fix it eventually. Similarly, identifying how to offset the read and write buffer was challenging since it was not as "direct" like earlier.