CS 518 - Project 4: RU File System using FUSE

Submitted by Anmol Arora (aa2640) and Raunak Negi (rn444)
Code run on cd.cs.rutgers.edu | ilab2.rutgers.edu

Introduction:
In this project, we are tasked with designing a user-level file system using the FUSE (Filesystem in Userspace) library. Our objective is to develop the RU File System (RUFS), which will function at the user level. RUFS will act as an intermediary between the user and the disk, organizing data and indexing information on the disk.

Data Structures:

1. **struct superblock**: This structure represents the 'superblock' in our file system, a crucial component that contains metadata about the file system itself. The *magic_num* is a unique identifier typically used to validate the file system type. The *max_inum* and *max_dnum* fields specify the maximum numbers of inodes and data blocks, respectively, indicating the capacity of the file system. *i_bitmap_blk* and *d_bitmap_blk* mark the starting blocks of the inode and data block bitmaps, which are used to track the allocation status of inodes and data blocks. Finally, *i_start_blk* and *d_start_blk* indicate the starting blocks of the inode region and data block region, where the actual data is stored.

2. **struct inode**: This structure defines an 'inode', which is a fundamental concept in file systems used to represent files and directories. The *ino* field is the inode number, serving as a **unique identifier**. *valid* indicates whether the inode is in use. The *size* field represents the size of the file or directory. type specifies the type of file (regular file, directory, etc.), and link counts the number of hard links to the inode. The *direct_ptr* array holds direct pointers to data blocks, while *indirect_ptr* contains indirect pointers, allowing the file system to address more data blocks. *vstat* is a structure (typically defined in **sys/stat.h**) that stores various file metadata like *modification times*, *permissions*, etc.

3. **struct dirent**: This structure is used for directory entries within the file system. Each *dirent* represents an individual file or subdirectory within a directory. The *ino* field contains the inode number of the entry, which links to the actual file or directory. *valid* indicates if the directory entry is in use. *name* stores the name of the file or directory, and *len* gives the length of this name. This structure is

essential for navigating through directories and accessing files within the file system.

4. typedef unsigned char* **bitmap_buffer[BLOCK_SIZE]**: is a type definition for a pointer to an array of *unsigned char*, used for representing a *bitmap* in a file system, which efficiently tracks the allocation and availability of resources.(**NOTE 3** and **NOTE 4** in the writeup).

Part 1: Bitmap

- **get_avail_ino()**: This function locates an available inode in a file system. It reads the inode bitmap from the disk, searches for a free inode, updates the bitmap to mark the inode as used, and writes the updated bitmap back to the disk. The function returns the inode number if available, or an error code if not.

- **get_avail_blkno()**: This function identifies an available data block in the file system. It reads the data block bitmap from the disk and searches for an unused block. If a free block is found, it updates the bitmap to indicate this block is now used and writes the modified bitmap back to the disk. The function returns the block number if a free block is found; otherwise, it returns -ENOSPC if no blocks are available, or a negative return code if reading or writing to the disk fails.

Part 2: Inode

- **readi()**: It calculates the block number and offset where the target inode is located based on the inode number (ino) provided. It reads the block containing the inode data into a static buffer (db_buffer) using the bio_read function, checking for errors (retstat < 0). If the block read is successful, it copies the inode data from db_buffer to the provided struct inode. If there is an error during the block read, it returns the error code. The function returns 0 on success or an error code on failure, indicating whether the inode was successfully read or not.

- **writei()**: It calculates the block number and offset for the target inode based on the inode number (ino) and uses a static buffer (db_buffer) for read and write operations. It reads the block containing the inode data from disk into db_buffer using the bio_read function, checking for errors (retstat < 0). The function then updates the inode data in the buffer with the provided struct inode and writes the modified buffer back to the disk using bio_write. It checks for errors during the write operation and, if an error occurs, returns the error code.The function returns

0 on success, indicating that the inode was successfully updated and written to disk, or it returns an error code if any step encounters an issue during execution.

Part 3: Directory and Namei

- **dir_find()**: The function begins by calling readi() to retrieve the inode associated with the provided inode number (ino) and stores it in a static cur_inode structure. It checks for errors and returns an error code if the read operation fails.It iterates through the direct pointers of the current directory inode, attempting to find the data block containing directory entries. For each data block, it reads the block's content into a static buffer (db_buffer) using bio_read, checking for errors during the read operation.It then searches within each directory entry in the data block to find a match with the provided filename (fname). If a match is found, it copies the directory entry information into the provided dirent structure and returns 0, indicating success.If no matching entry is found after searching all data blocks, the function returns -ENOENT, indicating that the requested entry does not exist in the directory.

- **dir_add()**: This function adds a new directory entry to a specified directory inode. It initializes a new directory entry structure with the provided file inode number, filename, and name length. The function checks for the existence of a directory entry with the same name and returns an error if a match is found, ensuring no duplicate entries are added. It then searches for an available slot within the directory's data blocks, allocating a new block if needed, and writes the new directory entry. After successfully adding the entry, it updates the directory inode with the modified data block information and writes it back to disk. The function returns 0 on success or various error codes, such as -ENOSPC if there are no available slots for the new entry or an error code from the write operation if writing the directory inode fails.

- **get_node_by_path()**:This function in the provided code is designed to retrieve the inode associated with a given path in a file system. It uses an **iterative** approach rather than a recursive one. The function tokenizes the input path based on the delimiter '/', and iterates through each token, searching for directory entries using dir_find. If a directory entry is found, it updates the inode number for the next iteration. The function handles cases such as empty paths or the root directory. It ultimately reads the inode corresponding to the last token in the path and returns it in the inode structure. If any errors occur during the process, appropriate error messages are displayed, and error codes are returned. It's important to note that this implementation is not recursive; it iterates through the path components sequentially.

Part 4: FUSE-based File System Handlers

# Initialisation

- **rufs_mkfs():** The function begins by calling dev_init to create and initialize a disk file. It then initializes the superblock (super_block) and writes it to the disk using bio_write. The function also sets up inode and data block bitmaps, marking blocks occupied by superblock, inode bitmaps, and data block bitmaps as in use, and the rest as available. It marks the root directory's inode as used and initializes its attributes. The inode and data block bitmaps, along with the root directory's inode, are written to the disk. Finally, it frees allocated memory for bitmaps and buffers, completing the file system initialization. The function returns 0 upon successful completion.

- **rufs_init():** function is responsible for initializing RUFS by either creating a new file system or loading an existing one. It first attempts to open the specified disk file. If the file is not found, it calls rufs_mkfs to create a new file system, initializing the superblock and necessary data structures. If rufs_mkfs fails, it exits with an error. If the file is found, it initializes in-memory data structures and reads the superblock, inode bitmap, and data block bitmap from the disk. The function then allocates memory for these data structures, copying data from disk into them. Finally, it frees the temporary buffers and returns NULL. This function ensures the proper initialization of RUFS based on the presence of the disk file, allowing it to be used for subsequent file system operations.

- **rufs_getattr():** retrieves file attributes for a specified file or directory identified by its path. It initializes a local struct inode variable, calls the get_node_by_path function to obtain the corresponding inode information, and copies the retrieved file attributes to a provided struct stat. If the inode retrieval is successful, the function returns 0; otherwise, it returns a negative error code.

# Directory Operations

- **rufs_opendir():** It calls the get_node_by_path function to obtain the inode information for the directory, checks for potential buffer overflow, and then returns the result of the inode lookup. If successful, it provides the directory's inode information for subsequent operations.

- **rufs_readdir():** function reads the contents of a directory specified by the given path and populates the provided buffer with directory entries. It retrieves the directory's inode, checks if it is a valid directory, and iterates through the directory's data blocks to fill the buffer with valid entries using the FUSE filler function. The function returns 0 on success and appropriate error codes otherwise.

- **get_parent_path():**function extracts the parent directory from a given path, modifying the provided parent string to contain the result. It does so by locating the last '/' in the path and cutting off the part after it

- **get_target_name():** function isolates the target directory or file name from a given path, copying it into the provided target string. It accomplishes this by finding the last '/' in the path and copying the portion after it.

- **rufs_create_or_mkdir():** function is a versatile implementation for creating entries in the RUFS file system, accepting a path, mode, and a flag indicating whether to create a directory or a regular file. It performs common tasks such as path parsing, parent directory lookup, inode allocation, directory entry addition, and inode initialization, providing a unified approach for both directory and file creation. The crucial divergence between rufs_mkdir() and rufs_create() lies in the initialization values of the inode before writing to the disk and the rest of the work flow is the same for both the cases.

- **rufs_mkdir():** function specializes in creating directories in RUFS. It calls rufs_create_or_mkdir with the directory flag set to 1, ensuring the necessary steps are taken to create a new directory, including updating the inode information and writing it to disk.

# File Operations

- **rufs_create():**function creates a new file at the specified path, separating the parent directory and target file name, obtaining an available inode, adding the new file entry to the parent directory, initializing the new file's inode, and writing it to disk.

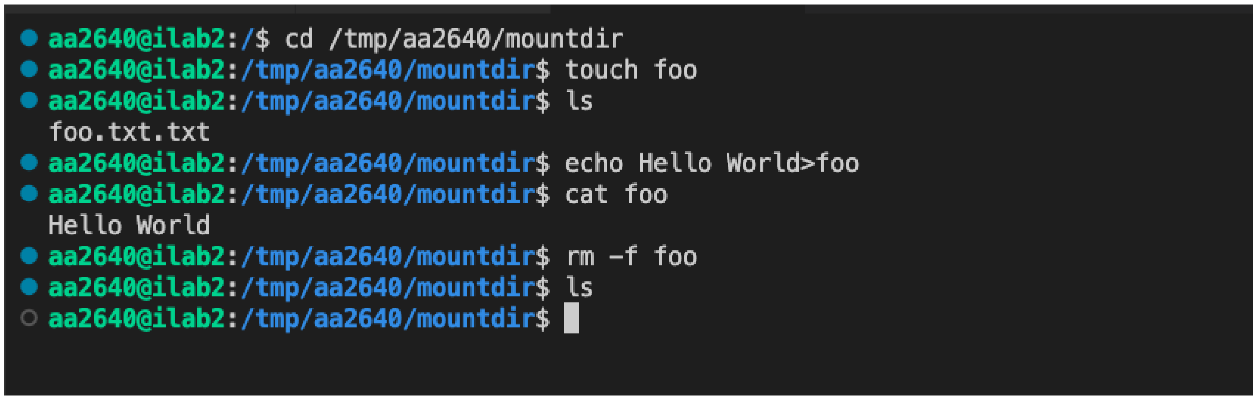
- **rufs_open():**function verifies the validity of a file specified by its path, ensuring it is a valid file type with a valid inode number.

- **rufs_read():** function reads data from a file specified by its path, identifying the relevant data blocks based on size and offset, copying the appropriate data to the buffer, and updating the file's access time and inode information.

- **rufs_write():** function writes data to a file specified by its path, determining the appropriate data blocks based on size and offset, reading existing data blocks from disk, updating the file's inode information, and writing the modified data blocks back to disk.

Part 5: Testing and Analysis

- Perform some commands (ls, touch, cat, mkdir, rmdir, rm . . .).



```
aa2640@ilab2:/$ cd /tmp/aa2640/mountdir
aa2640@ilab2:/tmp/aa2640/mountdir$ touch foo
aa2640@ilab2:/tmp/aa2640/mountdir$ ls
foo.txt.txt
aa2640@ilab2:/tmp/aa2640/mountdir$ echo Hello World>foo
aa2640@ilab2:/tmp/aa2640/mountdir$ cat foo
Hello World
aa2640@ilab2:/tmp/aa2640/mountdir$ rm -f foo
aa2640@ilab2:/tmp/aa2640/mountdir$ ls
aa2640@ilab2:/tmp/aa2640/mountdir$ 
```

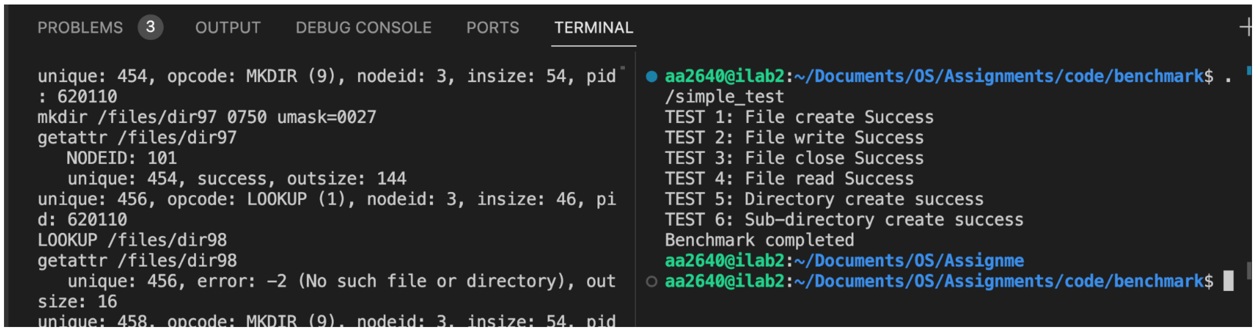
- Running the Sample Benchmarks

BLOCK_SIZE = 4096 Bytes
INODE_SIZE = 256 Bytes
DIRENT_SIZE = 256 Bytes

No. of inodes in a Data Block = BLOCK_SIZE/INODE_SIZE = 16
No. of Directory entries in a Data Block = BLOCK_SIZE/DIRENT_SIZE = 16
Total Inodes = MAX_INUM 1024
Total Blocks = MAX_DNUM 16384

SuperBlock = 1 Block
Indoes_bitmap = 1 Block
Data_block_bitmap = 1 Block

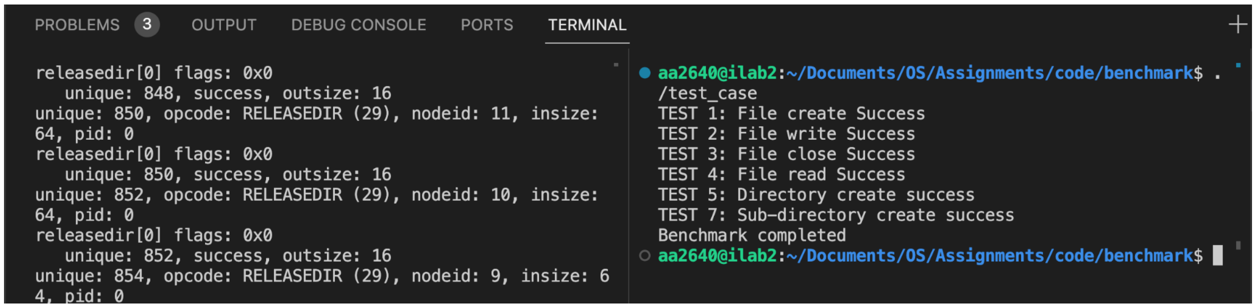
Blocks Assigned for inodes = 1024*256 Bytes / 4096 Bytes = 64 Blocks that store inodes
Remaining Data Blocks = 16384 - 67 = 16317 Blocks
# simple_test



```
unique: 454, opcode: MKDIR (9), nodeid: 3, insize: 54, pid
: 620110
mkdir /files/dir97 0750 umask=0027
getattr /files/dir97
   NODEID: 101
   unique: 454, success, outsize: 144
unique: 456, opcode: LOOKUP (1), nodeid: 3, insize: 46, pi
d: 620110
LOOKUP /files/dir98
getattr /files/dir98
   unique: 456, error: -2 (No such file or directory), out
size: 16
unique: 458, opcode: MKDIR (9), nodeid: 3, insize: 54, pid
```

```
aa2640@ilab2:~/Documents/OS/Assignments/code/benchmark$ .
/simple_test
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: Directory create success
TEST 6: Sub-directory create success
Benchmark completed
aa2640@ilab2:~/Documents/OS/Assignme
aa2640@ilab2:~/Documents/OS/Assignments/code/benchmark$
```

# test_cases



```
releasedir[0] flags: 0x0
   unique: 848, success, outsize: 16
unique: 850, opcode: RELEASEDIR (29), nodeid: 11, insize:
64, pid: 0
releasedir[0] flags: 0x0
   unique: 850, success, outsize: 16
unique: 852, opcode: RELEASEDIR (29), nodeid: 10, insize:
64, pid: 0
releasedir[0] flags: 0x0
   unique: 852, success, outsize: 16
unique: 854, opcode: RELEASEDIR (29), nodeid: 9, insize: 6
4, pid: 0
```

```
aa2640@ilab2:~/Documents/OS/Assignments/code/benchmark$ .
/test_case
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: Directory create success
TEST 7: Sub-directory create success
Benchmark completed
aa2640@ilab2:~/Documents/OS/Assignments/code/benchmark$
```

All test cases running **successfully**.

Part 6: Problems Faced

1. We were unable to implement the **rufs_rmdir()** functionality when directories contain subdirectories. Test cases 5 and 6 are omitted when a directory lacks subdirectories. Instead, an additional test case, Test Case 8, is introduced, focusing on deleting a directory without subdirectories. This test case is executed immediately after Test Case 4, which creates a directory. Consequently, to execute the test cases consecutively without unmounting FUSE, there is no need to unmount; the test cases can be run directly in sequence on the mounted file system.
2. The primary difficulty with subdirectories was attributed to the **recursive** process involved in deleting directories and the subsequent unsettlement of corresponding inodes and data blocks.
3. We couldn't implement the support for larger files which would require indirection.

References:

- https://libfuse.github.io/doxygen/index.html

- http://www.maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/
- https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/
- https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html
- Piazza discussions
- Chapter 39 and 40 of Operating System: 3 easy pieces