CS 416 Project 4: RU File System using FUSE

Instructor: Professor Srinivas Narayana

Students Name: Pavitra Patel (php51), Kush Patel (kp1085)

Q. How did we approach the project?

1. First, we read the chapter "A Very Simple File System Implementation" from the text.
2. As mentioned in the project description, we started by implementing rufs_mkfs, rufs_init, and rufs_destory.
3. Then, we implemented all the helper functions such as get_avail_ino, get_avail_blkno, writei, and readi.
4. Then, we moved on to implement directory functions get_node_by_path, dir_find, and dir_add.
5. We looked at the benchmarks and identified that we would need rufs_getattr and rufs_create to pass the first test case. So, we started implementing rufs_getattr and rufs_create. After our first test case passed, we implemented rufs functions in the order of benchmark test cases. Namely, the following order was rufs_open, rufs_write, rufs_read, rufs_mkdir, rufs_readdir etc.
6. Our goal was to complete the entire project, so we implemented indirect pointers from first. We moved on to implement extra credit functions dir_remove, rufs_rmdir, rufs_unlink, etc.
7. Once we had the code ready, we tested our code and fixed bugs for several test/edge cases which will be included in the later section.

Q. Brief description of the functions implemented:

`int get_avail_ino()`

- This function returns an available inode number within the inode bitmap. It first traverses the bitmap, starting from index 0 which represents inode 0, until it finds an unused inode or reaches the maximum allowed inode number. Upon discovering an available inode, the function updates the bitmap at that index using the set_bitmap method, writes the modified bitmap to the disk using bio_write method, and then returns the obtained inode number. In cases where no unused inodes are found within the specified range, the function returns -1.

`int get_avail_blkno()`

- This function works the same way as get_avail_ino. The only change is that we will use datablock_bitmap instead of inode_bitmap here. The first three blocks are reserved for superblock, inode bitmap, and data block bitmap respectively. Inode table starts from 4th block. Considering MAX_INUM = 1024, we would need 64 blocks for storing inodes. So, our data blocks for data start at index 67. If the first available index in datablock_bitmap is 0, then we return (67+0)th block from this function.

`int readi(uint16_t ino, struct inode *inode)`

- The readi function reads an inode (file or directory metadata) from disk into memory. It involves several steps:
  - First, we determine the on-disk block number where the target inode is stored based on ino.
  - We read the corresponding block from disk into a temporary buffer (temp_block).
  - Then, we calculate the offset within the block where the target inode is located.

  - Using memcpy, we copy the inode data from the temporary buffer into the provided struct inode in memory.
  - The function returns 0 upon successful execution and -1 if there's an issue executing bio_read function.

`int writei(uint16_t ino, struct inode *inode)`

- The writei function writes an inode (file or directory metadata) from memory to disk. It involves the following steps:
  - First, we determine the on-disk block number where the target inode is stored based on ino.
  - Then, we read the corresponding block from disk into a temporary buffer (temp_block) to ensure that existing data in the block is not overwritten.
  - Then, we calculate the offset within the block where the target inode is located.
  - Using memcpy, we copy the inode data from the provided struct inode in memory into the calculated offset within the temporary buffer.
  - Then, we write the modified block back to disk using the bio_write function.
  - The function returns 0 upon successful execution and -1 if there's an issue executing bio_read or bio_write functions.

`int dir_find(uint16_t ino, const char *fname, size_t name_len, struct dirent *dirent)`

- This function finds the directory entry that we are looking for and copies the metadata from the disk to the in memory struct dirent *dirent from parameter of the function.
  - Using readi function, first we read the inode number of the current directory.
  - First, we iterate through all allocated direct pointer data blocks to see if the directory entry we are looking for is present or not. Here, we read the block using bio_read to a temporary buffer temp_block, cast it to (struct dirent *) and then access each dirent in block based on indexes to see if entry is present. We use parameters fname and name_len, and valid and len field of dirent entries at each index in block to see if we can find the desired entry. If the entry is found, using memcpy we copy the contents of dirent from disk to in-memory structure dirent which is passed as parameter.
  - Similarly, we handle indirect pointers as well. We iterate though the allocated indirect pointers, read the indirect pointer data blocks, read the int entries in the indirect pointer data block (int entries correspond to an additional block where dirents are stored), and then read the data blocks corresponding to the int entries. Now, we can read dirents similar to what we did in direct pointers and copy contents of dirent entry that we are looking for from disk to in-memory dirent structure passed as parameter.
  - If the entry was found we return 0, otherwise we return -1.

`int dir_add(struct inode dir_inode, uint16_t f_ino, const char *fname, size_t name_len)`

- This function adds a directory entry with fields f_ino, fname, and name_len to a data block of specified dir_inode.
  - First, we use dir_find to see if the directory entry with name “fname” already exists or not. If it does, we return -1.
  - Secondly, we handle the case of direct pointers. In other words, if the direct pointer data blocks of dir_inode are not full, we will add directory entry here. We start by iterating over data blocks of direct pointers and handle two cases:

        - If data block is not allocated for a direct pointer or if a data block is full for a direct pointer, we would need to allocate data block to the next available direct pointer.
        - If the block already exists, we iterate over the block to see if there is any free space so that we can add our directory entry.
    - If all direct pointer data blocks are full, we will store our directory entry in one of the data blocks of indirect pointers. Here, we start by iterating over allocated data blocks for indirect pointers and handle following cases:
        - If none of the indirect pointers have been used yet, we allocate a data block for first indirect pointer else we iterate over integer entries in the data block. These integer entries are data blocks itself, so we iterate over dirent entries in this data block to see if there is any free space available to store our new entry.
        - If the most recent used integer entry data block is full, we allocate a new data block, store it's integer entry in indirect pointer data block and write dirent entry in the new allocated block.
        - If dirent entry can't be added to current indirect pointer, we move over to check the next most indirect pointer. If the next most indirect pointer hasn't been allocated yet, we allocate a block for indirect pointer, we allocate a block for first integer entry that will be stored in indirect pointer data block and write dirent entry to the start of the block of the integer entry data block.
    - In each of the cases of direct and indirect pointer mentioned above, we update the size of the file by sizeof(struct dirent) units. We also update the time in the vstat field of the dir_inode.
    - We return 0 if dirent entry was added successfully, else we return -1.

```
int dir_remove(struct inode dir_inode, const char *fname, size_t name_len)
```

- The dir_remove function is removes a directory entry specified by the filename (fname) and its length (name_len) from the given directory inode (dir_inode). The function performs the following steps:
    - Read and Check Direct Pointers:
        - Iterates over the direct pointers of dir_inode.
        - For each allocated direct pointer, reads the associated data block and checks each directory entry.
        - If a matching entry is found, updates it by marking it as invalid, clears related information, and writes the block back to disk.
    - Read and Check Indirect Pointers:
        - Iterates over the indirect pointers of dir_inode.
        - For each allocated indirect pointer, reads the associated data block containing integer entries (pointers to data blocks).
        - For each data block pointed to, reads the block and checks each directory entry.
        - If a matching entry is found, updates it similarly to the direct pointers, and writes the block back to disk.
    - Bitmap and Disk Operations:
        - We unset the corresponding inode in the bitmap to mark it as available.
        - Handled both direct and indirect pointers, ensuring that the correct data block is updated and written back to disk.
    - The function returns 0 upon successful removal of the directory entry.

```
int get_node_by_path(const char *path, uint16_t ino, struct inode *inode)
```

- The get_node_by_path function returns the inode corresponding to a specified file path in a file system. Here are the high-level steps:
    - We start by iterating from the root inode identified by the provided inode number (ino).
    - We tokenize the given file path using '/' as the delimiter to navigate through directory levels.
    - For each token/directory name, we search for the corresponding directory entry within the current directory's inode.
    - Read the inode of the directory entry found in the previous step to access the information about the next level of the directory hierarchy.
    - Continues this process for each token in the path until reaching the end of the path.
    - Copy the final inode corresponding to the specified path into the output parameter (inode).
    - The function returns 0 upon successful retrieval of the inode for the specified path.

```
int rufs_mkfs()
```
- dev_init is called to initialize the disk file specified by diskfile_path.
- Creates and writes superblock information, including magic number, maximum inode and data block numbers, inode and data block bitmaps' block numbers, and block numbers for inode and data block storage/start.
- Initialize inode and data block bitmaps, setting all bits to zero initially.
- Sets the bitmap for the root directory's inode.
- Create and initialize the inode for the root directory, setting its attributes and vstat structure.
- Initialize time-related information for the root directory inode.
- Write the initialized root inode to the disk.
- The function returns 0 upon successful completion.

```
static void *rufs_init(struct fuse_conn_info *conn)
```
- Check if the disk file exists by attempting to open it using dev_open.
- If the disk file is not found, it calls rufs_mkfs to create and initialize the file system.
- If the disk file is found, it initializes and mallocs in-memory data structures such as superblock, inode bitmap, and data block bitmap from the disk.
- Reads the superblock, inode bitmap, and data block bitmap from the disk into the allocated memory.
- Returns NULL on success.

```
static void rufs_destroy(void *userdata)
```
- The rufs_destroy function is responsible for the cleanup and destruction of in-memory data structures:
    - Write the superblock, inode bitmap, and data block bitmap to the disk using bio_write
    - Deallocate memory for in-memory data structures, including the inode bitmap, data block bitmap, superblock, and temporary block buffer.
    - Close the disk file using dev_close.

```
static int rufs_getattr(const char *path, struct stat *stbuf)
```
- The rufs_getattr function retrieves file attributes for the specified path and populates the provided struct stat (stbuf) with the obtained information:
    - Call the get_node_by_path function to retrieve the inode associated with the given path. If the operation fails (e.g., file not found), it returns an appropriate error code (-ENOENT).

  - Populate the struct stat (stbuf) with file attributes obtained from the retrieved inode.
  - Set file mode, link count, user ID, group ID, size, access time, and modification time based on the inode's corresponding information.
  - Differentiates between directories and regular files and sets the appropriate file type in the mode field.
  - We map the file system's internal inode information to the standard struct stat structure used by FUSE to represent file attributes.

```
static int rufs_opendir(const char *path, struct fuse_file_info *fi)
```

- The rufs_opendir function opens a directory identified by the specified path:
  - Call the get_node_by_path function to retrieve the inode associated with the provided path. If the operation fails (e.g., file not found), it returns -ENOENT.
  - If the inode is not valid (e.g., not a directory), it returns -ENOENT.
  - Returns 0 on success

```
static int rufs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset,
struct fuse_file_info *fi)
```

- The rufs_readdir function is responsible for reading the directory entries of a specified path and filling them into the provided buffer using the filler function:
  - Call the get_node_by_path function to get the inode associated with the provided path. If the operation fails (e.g., file not found), it returns -ENOENT.
  - Iterate through the direct pointers of the target inode and read directory entries from their respective data blocks. For each valid directory entry, use the filler function to add the entry's name to the buffer.
  - Like direct pointers, iterates through the indirect pointers of the target inode and reads directory entries from their respective data blocks. For each valid directory entry, use the filler function to add the entry's name to the buffer.
  - Returns 0 on success.

```
static int rufs_mkdir(const char *path, mode_t mode)
```

- The rufs_mkdir function is responsible for creating a new directory in the file system:
  - Use dirname() and basename() to separate the parent directory path and the target directory name from the provided path.
  - Call the get_node_by_path function to obtain the inode of the parent directory. If the operation fails (e.g., parent directory not found), it returns -ENOENT.
  - Call the get_avail_ino function to get an available inode number for the new directory. If the operation fails (e.g., out of memory), it returns -ENOMEM.
  - Call the dir_add function to add a directory entry for the new directory in the parent directory. If the operation fails (e.g., input/output error), it returns -EIO.
  - Initialize the inode structure for the new directory, setting attributes such as type, link count, and permissions. Update the inode stat structure with relevant information, such as access and modification times.
  - Call the writei function to write the inode of the new directory to the disk. If the operation fails (e.g., input/output error), it returns -EIO.
  - Returns 0 on success.

`static int rufs_rmdir(const char *path)`

- The rufs_rmdir function is designed to remove a directory from the file system. Here's a brief overview of its key steps:
    - It extracts the parent directory path and the target directory name from the provided path using dirname() and basename() functions.
    - Use the get_node_by_path function to obtain the inode of the target directory. Returns an error if the target directory is not found.
    - Verifies whether the target directory is empty by examining its size. If not empty, it returns -ENOTEMPTY.
    - Use unset_bitmap to clear the inode bitmap entry for the target directory.
    - Retrieve the inode of the parent directory using get_node_by_path.
    - Call dir_remove to delete the directory entry for the target directory from its parent directory.
    - Update access and modification times of the parent directory inode.
    - Decrease the size of the parent directory by the size of a directory entry.
    - Write the modified parent directory inode to the disk using writei.
    - Returns 0 on success.

`static int rufs_create(const char *path, mode_t mode, struct fuse_file_info *fi)`

- The rufs_create function is responsible for creating a new regular file in the file system:
    - Use dirname() and basename() to separate the parent directory path and the target file name.
    - Call get_node_by_path() to get the inode of the parent directory. Return an error if the parent directory is not found.
    - Call get_avail_ino() to obtain an available inode number for the new file.
    - Call dir_add() to add a directory entry for the new file to the parent directory.
    - Update the inode for the new file with relevant information such as inode number, validity, size, type (regular file), permissions, and time stamps.
    - Initialize direct and indirect pointers with -1 in the inode.
    - Update the inode stat structure with appropriate values.
    - Call writei() to write the updated inode to the disk.
    - Return 0 on success.

`static int rufs_open(const char *path, struct fuse_file_info *fi)`

- The rufs_open function is responsible for opening a file in the file system:
    - Call get_node_by_path() to obtain the inode of the specified file from the given path.
    - If the inode is not found or not valid, return an appropriate error code (-ENOENT) indicating "No such file or directory."
    - If the file inode is valid, return 0, indicating success.

`static int rufs_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)`

- The rufs_read function is responsible for reading data from a file in the file system:
    - Call get_node_by_path() to obtain the inode of the specified file from the given path.
    - Determine the starting block and the location within the block based on the provided offset.

- Read data blocks from disk based on the size and offset, copying the appropriate amount of data to the buffer.
    - For direct pointers, read from the direct data blocks.
    - For indirect pointers, read from the data blocks pointed to by the indirect pointers.
- Update the inode information, specifically updating the access and modification times.
- Write the modified inode back to the disk.
- Return the amount of bytes copied to the buffer.

```
static int rufs_write(const char *path, const char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi)
```

- The rufs_write function is responsible for writing data to a file in the file system:
    - Call get_node_by_path() to obtain the inode of the specified file from the given path.
    - Determine the starting block and the location within the block based on the provided offset.
    - Read existing data blocks from disk based on the size and offset.
    - Write the correct amount of data from the buffer to the disk.
        - For direct pointers, write data directly to the direct data blocks.
        - For indirect pointers, write data to the data blocks pointed to by the indirect pointers.
    - Update the inode information:
    - Update access and modification times.
    - Adjust the file size based on the new data.
    - Write the modified inode back to the disk.
    - Return the amount of bytes written to disk.

```
static int rufs_unlink(const char *path)
```

- The rufs_unlink function is responsible for deleting a file from the file system:
    - Extract the parent directory path and the target file name from the provided path using dirname() and basename() functions.
    - Use get_node_by_path() to obtain the inode of the target file. Returns an error if the target file is not found.
    - Clear the data block bitmap of the target file:
        - For direct pointers, clear the corresponding entries in the data block bitmap.
        - For indirect pointers, clear the entries in the data block bitmap pointed to by the indirect pointers.
    - Clear the inode bitmap and its data block.
    - Use get_node_by_path() to obtain the inode of the parent directory.
    - Call dir_remove() to delete the file entry from its parent directory.
    - Update the access and modification times of the parent directory inode.
    - Decrease the size of the parent directory by the size of a directory entry.
    - Write the modified parent directory inode to the disk using writei.
    - Return 0 on success.

Q. Details on the total number of blocks used when running sample benchmark and the time to run the benchmark.

Benchmarks for:

Note: The number of blocks reported doesn't include blocks for superblock, inode bitmap, datablock bitmap, and inode/itable blocks.

1. (******* TESTS DIRECT POINTERS *********)

```
#define N_FILES 100
#define ITERS 16
```

```
./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
Elapsed time: 0.42761099 sec
Number of blocks used: 23
```

```
./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
Elapsed time: 0.83553100 sec
Number of block used: 23
```

2. (********** TESTS INDIRECT POINTERS ************)

```
#define N_FILES 1000
#define ITERS 160
```

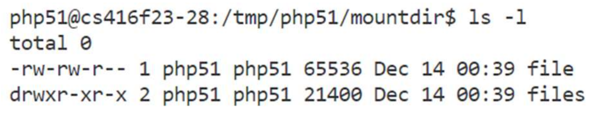
```
./simple_test

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

Elapsed time: 8.91044426 sec

Number of block used: 226
```

```
./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

Elapsed time: 12.52751064 sec

Number of block used: 226
```

Additional benchmarks:

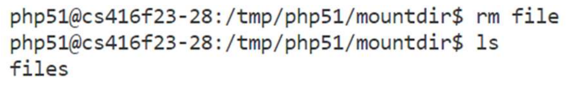
1. If you do some work on disk, turn the program off, and re-run the program without deleting the DISKFILE, all the prior work is retained.

2. The command “ls -l” works: (The size is displayed in bytes)



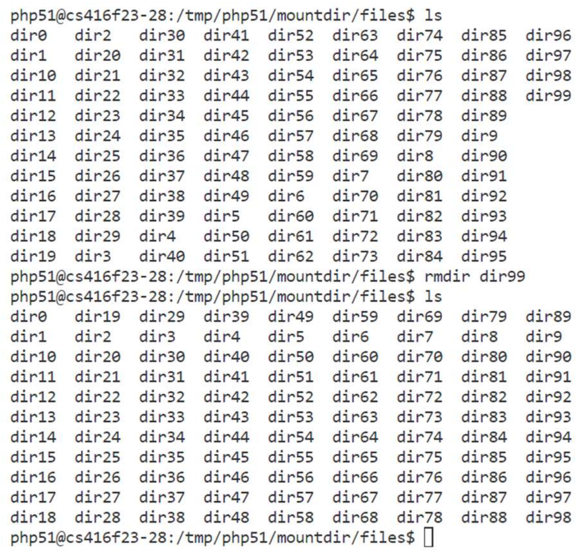
```
php51@cs416f23-28:/tmp/php51/mountdir$ ls -l
total 0
-rw-rw-r-- 1 php51 php51 65536 Dec 14 00:39 file
drwxr-xr-x 2 php51 php51 21400 Dec 14 00:39 files
```

3. The “rm” command works:



```
php51@cs416f23-28:/tmp/php51/mountdir$ rm file
php51@cs416f23-28:/tmp/php51/mountdir$ ls
files
```

4. The “rmdir” command works:



```
php51@cs416f23-28:/tmp/php51/mountdir/files$ ls
dir0   dir2   dir30  dir41  dir52  dir63  dir74  dir85  dir96
dir1   dir20  dir31  dir42  dir53  dir64  dir75  dir86  dir97
dir10  dir21  dir32  dir43  dir54  dir65  dir76  dir87  dir98
dir11  dir22  dir33  dir44  dir55  dir66  dir77  dir88  dir99
dir12  dir23  dir34  dir45  dir56  dir67  dir78  dir89
dir13  dir24  dir35  dir46  dir57  dir68  dir79  dir9
dir14  dir25  dir36  dir47  dir58  dir69  dir8   dir90
dir15  dir26  dir37  dir48  dir59  dir7   dir80  dir91
dir16  dir27  dir38  dir49  dir6   dir70  dir81  dir92
dir17  dir28  dir39  dir5   dir60  dir71  dir82  dir93
dir18  dir29  dir4   dir50  dir61  dir72  dir83  dir94
dir19  dir3   dir40  dir51  dir62  dir73  dir84  dir95
php51@cs416f23-28:/tmp/php51/mountdir/files$ rmdir dir99
php51@cs416f23-28:/tmp/php51/mountdir/files$ ls
dir0   dir19  dir29  dir39  dir49  dir59  dir69  dir79  dir89
dir1   dir2   dir3   dir4   dir5   dir6   dir7   dir8   dir9
dir10  dir20  dir30  dir40  dir50  dir60  dir70  dir80  dir90
dir11  dir21  dir31  dir41  dir51  dir61  dir71  dir81  dir91
dir12  dir22  dir32  dir42  dir52  dir62  dir72  dir82  dir92
dir13  dir23  dir33  dir43  dir53  dir63  dir73  dir83  dir93
dir14  dir24  dir34  dir44  dir54  dir64  dir74  dir84  dir94
dir15  dir25  dir35  dir45  dir55  dir65  dir75  dir85  dir95
dir16  dir26  dir36  dir46  dir56  dir66  dir76  dir86  dir96
dir17  dir27  dir37  dir47  dir57  dir67  dir77  dir87  dir97
dir18  dir28  dir38  dir48  dir58  dir68  dir78  dir88  dir98
php51@cs416f23-28:/tmp/php51/mountdir/files$
```

5. All the commands mentioned below works and have been tested:
    a. “cd”
    b. “touch a.txt”
    c. “echo “hello” >> a.txt”
    d. “mkdir”
    e. “more a.txt”

Q. Any difficulties and issues you faced when completing the project?

- Found very difficult to pass the first test case as had no experience using FUSE. After the first test case passed, faced no difficulties.

Q. If you had issues you were not able to resolve, describe what you think went wrong and how you think you could have solved them.

- No issues faced. Implemented complete extra credit part.

Q. Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

- Read Textbook
- Read documentation.
- Consistently, consulted TA's to understand project requirements