CS 518: Project 4 Report

Tarini Divgi (td508) Pruthviraj Patil (pp865)

iLab Machine – kill.cs.rutgers.edu

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

We determined the number of blocks utilized during the sample benchmark by tracking the frequency of get_avail_blk_no function calls. For the simple test.c, the data blocks used were tallied to be 168, which was achieved by incrementing a counter value within get_avail_blk_no each time an API request was made. The total execution time for simple test.c was 0.000122 seconds. Similarly, for test cases.c, the number of data blocks used was computed to be 2783, and the total execution time was 0.000112 seconds.

```
pp865@kill:~/Documents/OS_518/pp865_td508/File-system/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
Execution time = 0.000112
TEST 6: Sub-directory create success
Benchmark completed
```

```
pp865@kill:~/Documents/OS_518/pp865_td508/File-system/benchmark$ ./test_case
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
Execution time = 0.000122
mkdir: File exists
```

2. **Code Implementation** :-

int get_avail_ino()

We have this function that helps us get the available inode number from the bitmap. To do this, we first need to read the inode bitmap from the disk. Then, we go through the inode bitmap one by one to find an available slot. If we come across a bit that is equal to zero, that means we've found a free inode number, and we return it. Once we've found the inode number, we update the inode bitmap and save the changes to the disk. However, if we don't find any available inode number during the traversal, we return -1.

int get_avail_blkno()

The purpose of this function is to retrieve the available block number from the data bitmap. Initially, the function reads the data block bitmap from the disk. Then, the function traverses the

data block bitmap to locate an available slot. If we find a zero bit, that means there is a free data block and the index is returned. Subsequently, the data block bitmap is updated and the changes are written back to the disk. However, if the traversal fails to locate any available data block, the function returns -1.

int readi(uint16_t ino, struct inode *inode)

This function retrieves the inode's on-disk block number. Next, the offset of the inode in the on-disk block is retrieved, and the block is read from the disk using bioread. The retrieved data is stored into a buffer of block size. Then, the buffer data is copied into the inode structure using memcopy. The buffer is freed, and the function returns 0. At this point, the inode structure holds all the relevant data for the specified inode number.

int writei(uint16_t ino, struct inode *inode)

Here using the inode number, we get the block number where this inode resides on the disk. After getting the block number, we go to the particular block in the inode region. Then we get the offset in that block. We then create a buffer and read the block into that buffer. Then we copy the inode structure to that buffer. At last we write the buffer back into the disk and free it.

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

This function is used for directory operations. Firstly, it calls readi() to retrieve the inode of the current directory using the inode number. After obtaining the inode, the function retrieves the data blocks of the current directory. Once the data blocks are retrieved, the function reads each of them and checks each directory entry. During this process, if the name matches, the corresponding directory entry is copied to the dirent structure. For each directory entry, the name is compared with the fname provided. If there is a match, the file is found, the directory entry is copied to the passed dirent structure parameter.

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

This function is used to add a new directory entry to a given data block.If the data block has enough space for the new directory entry, then the function adds the directory entry to the data block. However, if the given data block is already full, a new data block is assigned and the directory entry is added to it. Initially, the function reads the directory inode's data block and checks each directory entry within it. It then checks whether the filename or directory name provided is already in use in other directory entries. If it is already in use, the function returns without adding anything. However, if it is not found, the directory entry is added to the directory's inode data block and written to disk.

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

We have used recursion to follow the pathname until a terminal point is found. A return value of -1 indicates an invalid path or file that does not exist. When a valid path is provided, we obtain the inode number and populate the inode structure. Finally, we free the dir_ent and return 0.

FUSE File Operations

static void *rufs_init(struct fuse_conn_info *conn)

The rufs_init API is used to initialize our file system. It starts by calling the dev_open function to check if any file is already opened. If not, it calls rufs_mkfs to create a new file system. Memory is allocated for the super block, inode bit map, and data bit map and the super block is copied to memory

rufs_mkfs()

The rufs_mkfs function is called by rufs_init and is responsible for creating the file system. It starts by initializing the disk file using the dev_init function. Then it populates the superblock structure with information about the number of inodes and data blocks, as well as the starting blocks for the inode bitmap, data bitmap, inode block, and data block. After filling all this information, the bitmaps are initialized to 0. The root inode is allocated the first bit in the inode bitmap and data bitmap, and these bitmaps are written to the disk. The root inode information is populated and written back to the disk, and an entry for "." is added to the root.

static void rufs_destroy(void *userdata)

The purpose of this API is to free the memory that has been allocated for the super block, inode bit map, and data bit map. After freeing the memory, the API closes the file by calling the dev_close function.

static int rufs_getattr(const char *path, struct stat *stbuf)

The purpose of this API is to populate the attributes of the stat structure that is associated with the inode. To achieve this, we first obtain the inode by calling the get_node_by_path API with the provided path. If the path is invalid, we return an error . The time attribute is set using the time function, while the uid and gid attributes are populated with the results of calling the getuid and getgid APIs, respectively. The mode attribute is set depending on whether the inode represents a file or directory, and the link count is updated to 2.

static int rufs_opendir(const char *path, struct fuse_file_info *fi)
The purpose of this API is to determine whether a given path exists in the file system or not. To achieve this, we make use of the get_node_by_path API, which takes the path as input and returns the corresponding inode number. If the inode number returned is invalid, we return -1 indicating that the path does not exist. Otherwise, we return 1 indicating that the path exists in the file system.

static int rufs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler,
off_t offset, struct fuse_file_info *fi)
The purpose of this API is to populate a buffer with all the directory entries. The first step is to obtain the inode of the base directory of the provided path by calling the get_node_by_path API. Next, we retrieve the data block to iterate through. We read each data block from the disk and then iterate through all the directory entries of the data block. If we come across a valid entry, we copy the name into the buffer.

static int rufs_mkdir(const char *path, mode_t mode)
This API is responsible for creating a new directory, which involves several steps. Firstly, we extract the base name and directory name from the path provided as input. Then, using the directory name, we retrieve the inode of the parent directory by calling the get_node_by_path API. Once we have the parent inode, we allocate a free inode to be used for the new directory. Then, we add the new directory to the parent directory by calling the dir_add API. After that, we initialize the fields of the newly allocated inode, populate them with appropriate values and write them back to the disk.

static int rufs_create(const char *path, mode_t mode, struct fuse_file_info *fi)
The purpose of this API is to create a new file. To achieve this, it takes in a path and then extracts the base name and directory name from it. Using the directory name, we retrieve the inode of the parent directory by calling the get_node_by_path API. Once we have the parent directory inode, we use the get_avail_inode API to find a free inode from the inode bit map. After that, we add the allocated inode for the new file to the parent directory inode structure. Lastly, we populate all the necessary fields of the inode structure for the new file and write this inode information back to the disk.

static int rufs_open(const char *path, struct fuse_file_info *fi)
The purpose of this API is to access a file specified by a path. It achieves this by calling the get_node_by_path API on the given path. If the path is not valid, the API returns -1. Otherwise, it returns 1, indicating that the file can be accessed.

static int rufs_read(const char *path, char *buffer,size_t size, off_t offset, struct fuse_file_info *fi)
The purpose of this API is to read data from a file. Initially, we use the get_node_by_path API to retrieve the inode of the file that contains the data blocks to be read. Then we perform some validation checks on the size and offset, ensuring that the sum of size and offset is not greater than the file size. Once these checks are completed, we proceed with the actual reading process.
To read the data blocks, we copy the data block-wise from the disk into the buffer. This is accomplished using the memcpy and bio_read functions. We continue this process as long as size is greater than zero, and decrement the size accordingly after each block is read.

static int rufs_write(const char *path, const char *buffer,size_t size, off_t offset, struct fuse_file_info *fi)
The purpose of this API is to write data into a file. The first step is to call get_node_by_path to obtain the inode where the data blocks will be written. Then, we calculate the total number of data blocks required to write the amount of data passed and perform some size-related sanity checks. Next, we write the data into the buffer and call bio_write to write it back onto the disk. This operation is performed using a while loop until the size is greater than 0. Finally, we update the inode information, including the modified time, and write this information back onto the disk.

Difficulties faced :-

- We faced many issues during compilation. We were constantly getting errors due to mount directory, connection etc
- It was really hard to debug the code