**CS 518: Project 4 Report**

Abhilash Kolluri (ak2048) & Gona Srikar (gs943)

iLab Machine – kill.cs.rutgers.edu

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

By finding the number of times get\_avail\_blk\_no is called, we calculated the number of blocks used while we had run the sample benchmark.

**Simple test.c**

• Number of data blocks used: 122 ( this was computed by incrementing a counter value in get\_avail\_blkno. Since, a data block is allocated whenever a request is made to API. )

• Total time taken for simple test.c: 13 milliseconds

**Test cases.c**

• Number of data blocks used: 2171

• Total time taken for test cases.c: 96.7 milliseconds

In both the test cases all the files got created successfully.

1. **Code Implementation** :-

**Helper functions we used**:-

2.1. **format\_block\_to\_dirrents** **()**:-

In this function we allocate memory for the array of directories that are to be copied to the block number. We first initialize the array to 0. We then create a temporary buffer where we copy to the buffer first and then copy that to the disk. We then copy array of directories to buffer and then finally copy the buffer to the disk. Then at last we free the array.

2.2.**write\_bit\_map()** :-

We use this function to write bitmap to the disk. First we create the buffer and allocate memory equal to the size of the block to it. We then check the function parameter i.e., whether it is a inode bitmap or data bitmap. If it is a inode bitmap we copy the inode bitmap which is of size(MAX\_INUM/8) to the buffer. Then we write it to the disk by using bio\_write(). If it is a data bitmap we copy the data bitmap which is of size(MAX\_DNUM/8) to the buffer. Then we write it to the disk by using bio\_write().. Once the bitmap is written to the disk, we free the buffer.

* 1. **name\_exists()** :-

This helper function takes the inode of directory that is of interest and searches whether the given file is present in this directory or not. It returns 0 if file already exits, else returns 1. First we calculate the number of datablocks used by that directory and iterate through them. We then allocate the buffer. After that, we read the corresponding data block from disc and store it in buffer. Then we iterate through the dirents from that block. First we check if the directory entry is valid. Then we check if the file name matches with any file present in directory. If we found that type of file, we break from the for loops and return the value. Atlast we free the buffer.

**Functions provided along with the code :-**

* 1. **get\_avail\_no()** :-

In this function we get the available inode number from the bitmap. We first Read inode bitmap from disk, then we Traverse inode bitmap to find an available slot, Atlast, we Update inode bitmap and write to disk. While traversing through the inode bitmap if we found any bit == 0 , we found the free inode number, We then return that. If we didn’t find any available inode, we return -1.

* 1. **get\_avail\_blkno()** :-

In this function we get the available block number from the data bitmap. Read data block bitmap from disk , then we Traverse that data block bitmap to find an available slot and Update data block bitmap and write to disk. While traversing through the data bitmap if we found any bit == 0 then there is a free data block in the file system, As, we found the free datablock number, We then return that. We return -1, If we didn’t find any free data block.

* 1. **readi()** :-

We use this function for the inode operations. We take two function parameters – inode number and the structure of inode. Firstly, we get the inode's on-disk block number. The, we get offset of the inode in the inode on-disk block, then we read the block from disk using bioread and store it into a buffer, by assigning the buffer a size which is equal to the block size. Then by using memcopy we copy that buffer into inode structure. Now the inode structure has all the data of that particular inode number. At last we free the buffer and return 0.

* 1. **writei()**:-

We use this function to write into inode, The function parameters include the inode number and inode structure. Firstly, by 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.

* 1. **dir\_find()** :-

We use this function for directory operations. We first call readi() to get the inode using the inode number of current directory. Then from the founded inode we get the data blocks of current directory. When we get the data blocks, we read all of them and check each directory entry. If the name matches, we copy the corresponding directory entry to the dirent structure. The sequence of operations to check the directory entries include the following, iterating through each of the data block, Then for each data block we check all the directory entries present in it and compare its name with the fname, if the name matches, file is found and we copy that directory entry to passed structure parameter dirent and return.

* 1. **dir\_add()** :-

In this function, we add new directory entry to the given data block. There are two scenarios, if the data block has space for the new directory entry, then we add the directory entry, if the given data block is full we assign new data block and add the directory entry to it. Firstly, we read directory inode’s data block and check each directory entry of that. Then we check if filename or directory name given is already used in other directory entries. If it is used, we return without adding anything. If it is not found, we add directory entry to directory’s inode data block and write that into disk.

2.10**-dir\_remove()** :-

This function is used to remove the directory entries. We get directory inode, file name as the function parameters. By using the inode given , we get all the data blocks pointed by it. Then by iterating through each data block, we find all the directory entries. We check if the name present in the directory entries matches with the given name (function parameter – file name). If the name matches, file is found and we make that entry as invalid. Now we update the block and return. If we didn’t find, we just return without updating anything.

2.11-**get\_node\_from\_path()** :-

In this function, we get the node for the given path. We use recursion, to do this. First, we resolve the path name, walk through the path recursively to find the inode. We could either implement this in iterative way or recursive way. If the given path is invalid or the file doesn’t exist, we return -1. If it is a valid path, we get the inode number and we populate the inode structure using this inode number. Then, we free the dir\_ent and return 0;

**FUSE File Operations**

* **rufs\_mkfs** : This API is responsible for making the file system which is called by the rufs\_init. At first, we call dev\_init to initialize the disk file. The next step would be to populate the superblock structure. The superblock which was allocated memory in the rufs\_init will be populated here. The available information of Number of inodes, number of data blocks is filled in super block. Computation of start block of inode bit map, data bit map , inode block and data block all are populated. Once we fill all this information, we initialize he bit maps to 0. After this we allocate the first bit in the inode bitmap and data bitmap to the root and write these bitmaps to the disk. Finally, we populate the root inode information and write it back to the disk and an entry of ‘.’ is added for the root.
* **rufs\_init** : This is the init API for our file system. It basically calls dev\_open to see if we have a file that is opened. If not, we call the rufs\_mkfs. Also, all memory required for super block, inode bit map and data bit map is allocated. Also, for sake of reference and simplicity in other parts of code super block is copied from to memory at this point.
* **rufs\_destroy** : This API deallocated the memory that has been allocated for super block, inode bit map and data bit map. Finally, dev\_close is closed to close the file.
* **rufs\_getattr** : This API is responsible for filling all the attributes of the structure stat that is part of the inode. Initially, inode structure is retrieved from the path provided by calling the get\_node\_by\_path API. If it’s not a valid path, we return the error(-ENOENT). Attribute of time is populated by using time call. The uid and gid are populated by calling the getuid and getgid API’s. Finally, the mode is set based on whether it’s a file or directory and link count is updated to 2.
* **rufs\_opendir** : This API is very straightforward one. We call the get\_node\_by\_path API by passing the path that has been sent as a parameter. If path is not valid then we return -1, else we return 1 implying the path exists.
* **rufs\_readdir** : This API is used to fill the buffer with all the directory entries using the filler API. We call the get\_node\_by\_path API on the path passed to the API and get the inode of the base directory of the path. Once we get the inode we retrieve the data block to iterate. We read each data block from the disk and iteral through all the directory entries of the data block. If it’s a valid entry, we copy the name using the filler to the buffer.
* **rufs\_mkdir** : In this API we make the director(mkdir). The base name and directory name are computed from the path that is provided. Then we call the get\_node\_by\_path to get the inode by passing the directory name, this will be our parent inode. We then get a free inode so as to map it to the new directory we are adding. We then call dir\_add to add the target directory to the parent directory. We now allocate a new inode for the target directory and populate the fields on the inode structure and write the information back to the disk. Finally, ‘.’ and ‘..’ are also added to the new target directory inode.
* **rufs\_rmdir** : This API is used to remove the directory. At first, the base name and directory name are separated from the path. Get the target inode by calling get\_node\_by\_path API. Now, for the target inode size, each data block is formatted by format\_block\_to\_dirents API and the corresponding data block number is unset the data block bitmap. Also, the inode bitmap is unset for the inode of the target directory. Now, after unsetting both the data bit map and inode bit map, it is written back to the disk. Now again, one final thing to do is call the get\_node\_by\_path on the parent directory and get its inode and do a dir\_remove to remove the target reference from the parent inode.
* **rufs\_create** : This API is used when creating a file. It takes the path and computes base name and directory name from the path. Now we get get\_node\_by\_path to get the parent directory inode. Next we find a free inode from the inode bit map by calling get\_avail\_inode. Next we add the allocated inode for the file to the paren directory inode structure. Finally, we populate all the members of the inode structure for the file and write back this inode information of the file back to the disk.
* **rufs\_open** : This API is used for accessing a file. We simply call get\_node\_by\_path API on the path. We return -1 if it’s a invalid path else return 1.
* **rufs\_read** : This API is used to perform the read operation. Firstly, get\_node\_by\_path is called to get the inode which has all the data blocks to read from and write back into the buffer. We then perform some sanity checks wrt size, ie size + offset should be less than the size passed. Once we do this, we start copying while size > 0 and do a block wise copy form the disk. This is achieved using the memcpy and bio\_read.
* **rufs\_write** : This API is used to used to perform write operation.Firstly, get\_node\_by\_path is called to get the inode where the data blocks will be written that will copied from the buffer. Next we compute the total data blocks needed to write the amount of data passed and do some sanity checks wrt size. Next, we write the data into the buffer and call the bio\_write to write back into the disk. This is performed using a while loop unit the size is greater than 0. Finally, we update the inode information along with modified time and write this information back into the disk.
* **rufs\_unlink** : This API is responsible for removing a file from its parent directory. It takes the path as parameter and computes directory and base name from the path. We then call get\_node\_by\_path to get the inode of the target file. Using this inode we unset all the data blocks of data bit map and finally unset the inode in inode bit map as well. Now get the inode of the parent directory and do an dir\_remove of the target ie base name from the target inode.

**Difficulties faced :-**

* Since this was the first time on FUSE, mounting and un mounting and making changes to code and getting them reflected was initially a challenge.
* A lot of debugging was performed using print statements as operating GDB with fuse was difficult. This led to tedious and time taking work of adding prints. Finally, all prints are removed from the code while submitting.
* To conceptualize every update, we do in the memory has to be written back to disk took some time. This led to initial errors as we were not updating the inode bit maps and data bit maps properly to disk.