Rutgers, The State University Of New Jersey

DEPARTMENT OF COMPUTER SCIENCE

RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

CS518 - Operating System Design

Project 4 : RU File System using FUSE

JANKI TRIVEDI(jdt111)
SACHIT PATEL(sp2135)

iLab machine : kill.cs.rutgers.edu

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

Simple test.c

- Number of data blocks used: 122
- Total time taken for simple test.c: 808.588 milliseconds

Test cases.c

- Number of data blocks used: 2193
- Total time taken for test cases.c: 446.541000 milliseconds

We have attached the screenshots at the end for reference. The number of blocks were calculated based on the number of times get_avail_blkno was called.

2 How did we implemented the method the code

2.1 get_avail_ino

This function is used to get the available inode number from bitmap. Firstly the inode bitmap is read from disk, then we traverse the inode bitmap to find an available spot and the inode bitmap is updated. After updating the inode bitmap is written to disk. Index of the bit is returned if inode is found and -1 if no inode was remaining.

2.2 get_avail_blkno

This function is used to get the available data block number from bitmap. Firstly the data bitmap is read from disk, then we traverse the data bitmap to find an available spot and the data bitmap is updated. After updating the data bitmap is written to disk. Index of the bit is returned if data block is found and -1 if no data block was remaining.

2.3 readi

In this function, using the ino passed as argument, that particular inode from disk is read to inode structure in memory. Firstly obtain the on disk block number of inode by dividing the inode number by number of inodes in block.Using the block number, we reach to the particular block of that inode in inode region by adding the block number to the inode start block. Next we calculated the offset of inode. A fetchedBlock named buffer is taken in memory and the particular inode block is read from disk into this buffer. A pointer of inode structure is then pointed to this buffer. By adding offset to it we reach to the particular location where the inode is present in block. From that location inode structure is copied using memcpy into inode structure in memory.

2.4 writei

In this function, we write the in memory inode structure to the inode present in our disk. This function is similar to readi but here we do write operations instead of read. Similar to readi we first get the block number where inode resides and go to the particular block in inode region. After calculating the offset we then read the block of inode and go to the exact location by adding offset to pointer. After that in memory inode structure is copied to that location using memcpy and that updated buffer in memory is then written to the inode in disk using bio write.

Directory Operations

2.5 dir_find

This function is used to find the file or sub directory in the current directory. If file or sub directory is found its structure is copied to the in memory directory entry(dirent) structure. Firstly readi is used to get the inode using ino of current directory. From inode we get the total data blocks it points to using its size. Using direct_ptr in the inode structure we compute the data block. We copy that data block from disk to buffer in memory using bio read. After that in buffer we traverse through the directory entries in it and check its validity and compare its name with the fname passed in argument. If the name matches we copy that particular directory entry to the in memory dirent structure using memcpy. 1 is returned as an indication of successful finding file/sub directory and copying to dirent structure or else return 0.

2.6 dir_add

This function is reponsible for adding new directory entry with given inode number and name in the current directory's data blocks. We first read the

dir_inode's data block and check each directory entry of dir_inode using fileExists function. If name already exists return -1. If it does not exist then add directory entry in dir_inode's data block and write the block back to disk. Firstly we get the total data blocks using the size parameter. Each data block is accessed using the direct pointer in inode structure. Read that data block from disk into buffer. In buffer check for an empty directory entry and update that entry by filling the parameters and make its validity 1 indicating this directory entry is taken. Now write this buffer in memory bakc to disk. If by traversing all data blocks we did not find an empty entry then create a new block and format it to structure of directory entries. As new block has been added, update the direct pointer and size of the inode structure. Write that new inode back to disk using writei. Now read the new block from disk using bio read, add new directory entry by setting the parameters and write that updated block back to disk using bio write.

2.7 dir_remove

This function is used to remove directory entry from current directory. Firstly get the total data blocks of dir_inode. For each data block we first read that block from disk into buffer in memory using bio read.In this buffer we traverse through every directory entry and compare its name with fname. If name is matched then we change its validity to 0 and write that new buffer in memory to that data block in disk. 1 is return if successfully copied new buffer to disk, if file is not found then 0 is returned.

2.8 get_node_by_path

In this function, we follow pathname until the end point is found. Firstly we obtain 2 duplicate paths to get the base file and parent directory using basename and dirname functions respectively. Then we check if base file is root or not. If it is then read 0 to inode and return 0. Here to find the inode of the final file we have implemented the recursive way. An inode structure and directory entry structure is allocated in memory and passed inode is passed along with parentDir again in get_node_by_path function. Using this we recursively traverse until we reach the root. Then using dir_find we check if the file isn present in parent directory. If 0 is returned from dir_find it means that file is not present so -1 is returned. Then we get the inode structure of the file using its inode number by readi. This procedure is done recursively and then in the end we get inode structure of end file in memory.

FUSE File Operations:

2.9 rufs_mkfs

This function is responsible for making the file system. We first call dev_init to initialize the disk file. Now we have to fill the superblock structure with information. First we fill the magic num, max inum and max dnum parameters.Then we find the starting block of inode bitmap. After that we calculate the inodeBitmap blocks required by diving maximum number of inodes to block size. In order to find the start block of data bitmap block we add the inodeBitmap blocks to start block of it. Similarly, we calculate the number of data bitmap blocks required. To get starting block of inode region we add number of data bitmap blocks to starting block of it. We then calculate number of blocks required by inode region and get starting of block of dta block region. After that we initialize the inode and data bitmap and write them to disk. Then update the inode and data bitmap for root directory and format the data block of superblock to directory entries. Then we update the inode for root directory and write that updated inode back to disk using bio read, memcpy and bio write.Finally, we added the dirent for '.', and return 0.

2.10 rufs_init

This function is initialization function of rufs where we open the disk and read a superblock into memory. If disk file is not found then call rufs_mkfs otherwise initialize in memory data structures and read superblock from disk. We then set super block pointer, inode and data bitmaps. We then read the inode and data bitmaps from disk to buffer using bio read and copy it to the inode and data bitmaps using memcpy.

2.11 rufs_destroy

Here we deallocate the in memory data structures and close the diskfile.

2.12 rufs_getattr

This function is called when accessing file or directory and returns the stats of files. First we get the inode by get_inode_by_path. It is stored in in-memory inodeStr. Then we fill the attribute of file into stbuf from inodeStr. We check whether its file or directory and fill the stats accordingly.

2.13 rufs_readdir

This function is used to read the directory. We first call get node by path to get the inode from path, and checked to make sure our path value was valid. The next step is to read the directory entries from its data blocks and copy them to filler. We get the data block using direct pointer in the inode structure. Next we read the data block from disk into memory to traverse through its directory entries. If the valid parameter is 1 of directory entry we copy it into the filler.

2.14 rufs_mkdir

This function is used to create a directory. Firstly use the dirname and basename to separate parent directory and target directory. Then we obtain the inode structure if parent directroy in inodeStr using getnodebypath. Then we call the geta_avail_ino to get an available inode number. Dir_add is called to add directory entry of target directory to parent directory. Next we updated inode for target directory and call writei to write inode to disk while updating the link count. After then create new inode structure for base directory and set all its values. Create two new directory entries . and .. and add using dir_add. Unlock the mutex lock and return 0 in the end.

2.15 rufs_rmdir

This function is used to remove the directory. Firstly use the dirname and basename to separate the parent directory and target directory. Then get_node_by_path is called to get the inode of target directory. Next formatting all the data blocks pointed by target directory using conversionToDirEntries and free those blocks in data bitmap using unset_bitmap. Next is to clear the inode bitmap of target directory and write the updated bitmaps to disk. Inode structure of parent directory is then taken into memory using get_node_by_path. Dir_remove is called to remove the directory entry of target directory from parent directory.

2.16 rufs_create

This function is called to create a file. Use dirname and basename to separate parent directory path and target file name. Next get_node_by_path is used to get inode of parent directory. Get_avail_ino is then called to get the available inode number.The dir_add to add the directory entry of target file to parent directory. Inode is updated for target file by setting the parameters. The new updated inode is then written to the disk using writei.

2.17 rufs_open

Here in this function we want to access the file. Get_node_by_path is called to get inode from path. Validity is checked, if the check variable is -1, it indicates the path is invalid. Otherwise 1 is returned and the inode is successfully read.

2.18 rufs_read

This function is the read operation's call handler. We first call get inode by path to get inode from the path and check that the file size is less than the offset and size passed as argument. After that the data blocks of path from disk are read making sure that difference of file size and offset is less than the size passed. Next we copy the correct amount of data from offset to buffer using memcpy and bioread.

2.19 rufs_write

This function is the write call handler. It is similar to rufs_read, here instead of reading we are writing to the disk. Firslt getnodebypath is called to get inode from out path.Based on size and offset, the data blocks it points to are read from disk. Here difference of total number of blocks and current number of blocks is considered, if more than zero extra blocks should be added, which is performed by adding getting available block number to starting block of data region. If size of inode structure is 0 or less then, mutex is unlocked and 0 is returned. After this correct amount of data is written from offset to disk. Finally, we update the inode information along with modified time and write this information back into the disk.

2.20 rufs_unlink

This function is used when removing a file(rm command). FIrstly dirname and basename is used to separate parent directory and target file name. Then call the get_node_by_path to get inode of target file. Clear the data and inode bitmaps of target file using unset_bitmap and bitmapToDisk functions. After that call get_node_by_path again to get the inode of parent directory. In the end dir_remove is called to remove directory entry of target file in its parent directory.

Helper functions and their purpose:

2.21 conversionToDirEntries

In this function the block was formatted and it consisted an array of directory entries. Firstly we allocated memory for an array consisting of directory entries. After that memset is used to fill the directory entries array with number of directory entries in block. Each directory entry was initialized and validated. Buffer was allocated in memory and array of directory entries was copied into it using memcpy. In the end bio write is used to write that buffer into the disk for that particular block number.

2.22 bitMapToDisk

As the name function suggests, it is used to write the inode and data bitmaps to disk. Here character is passed as function argument which indicates which bitmap is need to be written in the disk. The globally declared bitmaps are first copied to buffer in memory and then using bio write the buffer is written into the bitmap block in disk.

2.23 fileExists

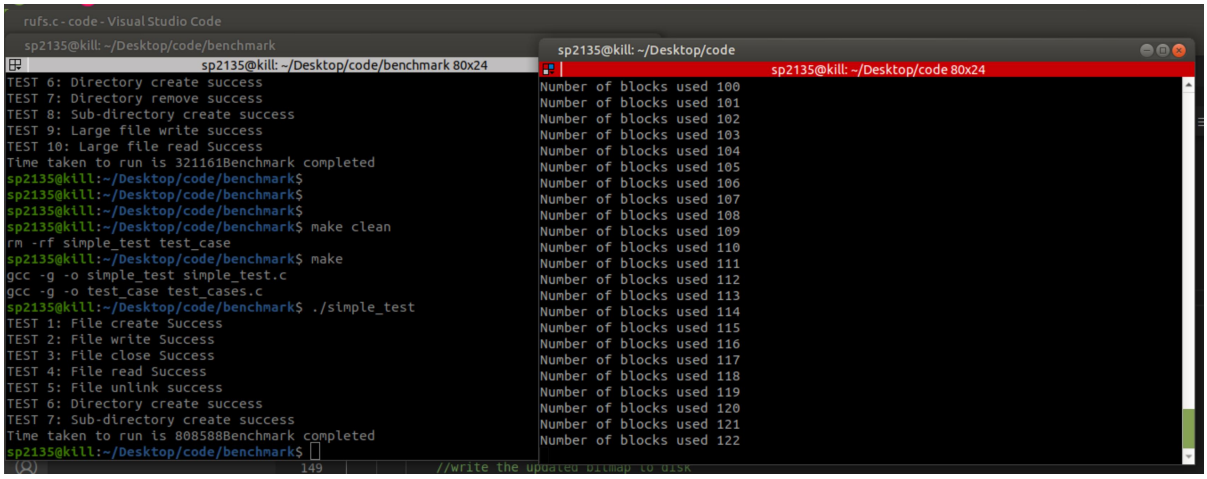
This function is used to check if the file name is already present in current directory. Total data blocks are calculated using the size of current directory. For each data block, buffer is taken in memory and the content of that particular data block are read from disk using bio read. In buffer each directory entry is checked whether its valid by taking structure of directory entry. If its valid, the file name is compared with the fname passed in function using string comparison and if its 0 then it means file has been found. The same proedure is followed for all the data blocks, where their directory entries are checked and compared with fname to determine if the file exists. If value of found remains 1 it means, no such file of similar name exists.

3 Additional steps we should follow to compile code

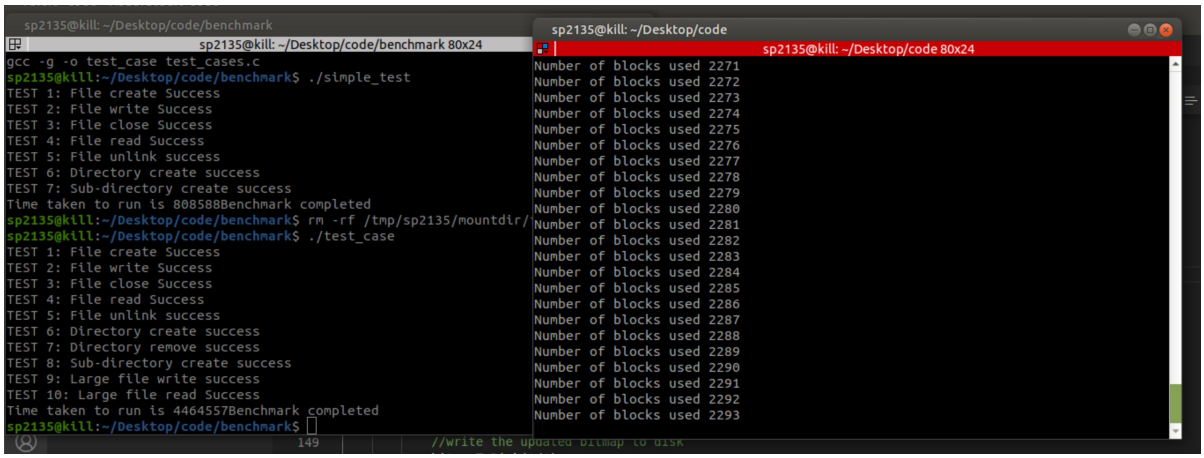
4 Any difficulties or issues faced while completing project

There were some difficulties faced while implementing the project

- The initial small challenge was mounting and unmounting RUFS and getting the changes done to code relfected. The debugging was performed was tedious and time taking as it is tough to use GDB with fuse and so print statements were used a lot. Also handling of inode and data bitmap and data blocks is very important for this project. Initially we faced some issues while implementing the concept, as we need to read from disk in memory and again write it back to disk.



Simple_test.c



Test_cases.c