CS 416: Project 4 Report

Vaishnavi Manthena (vm504) & Sanjana Pendharkar (sp1631)

May 2021

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

We calculated number of data blocks by recording the number of times get_avail_blockno was used.
Simple_test.c

- Number of data blocks used: 123
- Total time taken for simple_test.c: 856.238000 milliseconds

Test_cases.c

- Number of data blocks used: 2172
- Total time taken for test_cases.c: 81.541000 milliseconds

All of our files resulted in success for both tests.

2 How did we implement the methods

Helper functions and their purpose:

2.1 blockToDirents

In blockToDirents, we formatted our block with the block number, and our block held an array of dirents. The first thing we did was to create an array that held the number of dirents per block for our dirent structure. We then used memset, which is used to fill a block of memory with a particular value, to fill up the dirents array with the number of dirents per block. We then iterated through the number of dirents per block and set our valid bit int the array to 0, indicating that the entry in the array was initialized, and validated. We then created our buffer, and copied the number of dirents per block into the array. We then used bio_write to write out blockno into the buffer, which meant that our block would now be in the disk.

2.2 writeBitMap

In writeBitMap, we wrote our bitmaps to the disk by creating a buffer of block size. We set our char, c, to have two values: 'i' (for the inodebitmap) or 'd' (for the data bitmap). If c has value i, this means that we need to write the inodebitmap to our disk; if c has the value d, this means that we need to write the databitmap to the disk.

2.3 checkIfNameExists

In checkIfNameExists, we checked if the name could be found in our current directory, and returned 0 if the file already existed (meaning that we don't have to create a new file with the same name and details in our current directory) and 1 if it didn't exist (so we need to make the file in the directory). We set numOfDataBlocks to the current directory's size. We then looped through the data blocks, and set the variable dataBlock to the current directory's direct pointer at the value i (which was given in our loop). We then used bio read to read through the data block's contents into a buffer, so that we could step into its dirents. We set our buffer, and did bio_read, so that our dataBlock could be entered into the buffer. Next, we set a dirent structure, ptr, to be the buffer's starting location. As we looped through the dirents of this block, we set the valid bit for ptr+j to 1, meaning that it can be used (which is the current dirent that we are looking at, as we are going through all of the dirents). We then used a string comparison to find the file that corresponded with the currname, and incremented so that we could go to the next data block.

Functions provided originally:

2.4 get_avail_ino

In get available inode, we first read the inode bitmap from disk (we did this directly from memory, as was shown in the FAQs), then we traverse the inode bitmap to find an available slot and then we update the inode bitmap and write it to disk. We set a variable i to 0 and loop through it until we reach the max inum (which is the total size for the inum). Then, we set a bit to the bitmap of the inode at i. If our bit is equal to 0, this means we have not set or written it to the disk, so we do that by calling set_bitmap and writeBitMap. Then, we return the index. If we have gotten out of the loop, then we know there is no available inode, so we return -1, which indicates that there is no inode remaining.

2.5 get_avail_blkno

In get available blockno, we want to get the available data block number from the bitmap. We want to return the data block number. We will return -1 if there is no data block number remaining. For this function, we first read the data block bitmap from the disk directly from memory, then we traverse the

data block bitmap to find an available slot and finally, we updated the data block bitmap and write it to disk. The structure of this function is very similar to get available inode, except this time, we are looping through the max data num (which is the total size for the data block). If we notice that the bit is equal to 0 while traversing our max data number which is the size of the data block, then we set and write it, and return its index. If we know there is no available data block after going through the loop and never returning, then we return -1.

Inode operations

2.6 readi

In readi, we are trying to read the inode on the disk to a struct inode that will be in the memory. We first get the inode's on disk block number by dividing the ino by the number of inodes per block. Then, we know we need to go to the block's number of the inode area. To do this, we add the starting point of the inode block number to our current block number, which will give us the specific block we are looking for. We then set the offset of the inode on-disk block by using modulo function. Then we read the block from disk and copy it into the inode structure. We do this by setting the retrievedBlock's block size, and reading the block to Retrieve into our retrieved block using bio_read. Then, we create a ptr inode and add the offset to the retrieved block we found, and copy the ptr into our inode structure. We then return 0 to indicate that our operation is successful.

2.7 writei

In writei, we want to write the in-memory inode structure to the disk inode in our disk. The structure of this function is similar to readi, except this time, we are using the write operations, not the read operations. We first get the block number in which the inode resides on the disk. Then, we find the blockToRetrive by adding the start of the block number to the block number we found, which will directly place us at the block number that we want. Then, we set the offset with modulo. Finally, we write the inode to disk by reading the retrived block, and copying the ptr inode structure into the inode on disk. Finally, we call bio_write to put our retrieved block into the block to retrieve.

Directory Operations

2.8 dir_find

The dir_find function tries to find the file or sub directory of choice in the current directory. If this exists, then it puts this file into the struct dirent. This function returns a 1 if there is a file name and copies it to dirent, or it returns 0. First, we call readi to get the inode using ino, which is the inode number of current directory. We read our inode into the currDirectory, which means

that our currDirectory has all of the inode information that is necessary for our directory. We then get the data block of the current directory from inode, and then read the directory's data block and check each directory entry. If the name of the directory matches, then we copy the directory entry to the dirent struct. We did this by looping through the datablocks, and finding the data block of choice at the current directory's direct ptr at point i. Then, we used bio read to read the data block into a buffer so that we could look through its dirents using a ptr to the dirent struct. We then looped through the dirent sof this block, set the valid bit to 1, used a string comparison to find the file, and copied it into the dirent struct from the ptr+j dirent of choice.

2.9 dir_add

In this function, we want to add a new directory entry with the given inode number and name it in the current directory's data blocks. We first read the dir_inode's data block and check each directory entry of dir_inode by using check if name exists function. Then, we check if the directory name (fname) has already been used in other entries; since we used the check if name exists function, this is easy because if the check returns 0, this means that our file already exists. Next, we add the directory entry in dir_inode's data block and write it to disk. We first allocate a new data block for this directory if it does not exist, then we update directory inode and write directory entry. We set checkValid to 0, and the number of data blocks to the directory's inode size. We then looped through the data blocks, and found the data block of choice using the direct_ptr at i. Then, we read this data block into a buffer, so that we could see its dirents. We looped through the dirents of the entrie block, and checked to make sure if they were valid. If they were not, this means that the file did not exsit, and we wrote the directory entry at this location with the file contents of ptr+j (the block of choice). We then wrote the buffer back to the disk, and incremented to make sure our loop would exit.

Next, we checked if checkValid was equal to 0, which meant that a new block was required for this directory. We first got the next available block, and converted this block into an array of dirents. Then, we updated the direct_ptr and size of this directory's inode by reading the disk inode struct into our memory inode inode struct. We then set the newdirInode to size+1, find the block number by adding the inode's direct ptr to the number of data blocks. Then, we write the inode of the dir_inode into the newdirInode by taking the in memory inode struct and putting it into the disk inode struct. This concludes our update procedure. Next, we set our buffer, and add the new bock number into our buffer to create a location dirent. We set everything for the location dirent so that it becomes valid, because we are creating a new file to our directory. Next, we use bio_write to write the block number from disk to in memory, and return 0, because we have concluded our operation.

2.10 dir_remove

In dir_remove, we want to remove a directory entry from a directory (fname); we will return 1 if fname has been removed successfully, and 0 if there is no fname. We first read the dir_inode's data block and check each directory entry of dir_inode, then we check if fname exists, and then if it exists, we remove it from the dir_inode's data block and write it to disk. We first set the number of data blocks as the dir_inode's size to make sure that we are accounting for all data blocks. Then, we loop through the number of data blocks, find our data block of choice at value i, and then, use bio_read to read the data block into a buffer. Next, we loop through the dirents of this block, and use a string comparison to find the file in our directory, if it exists. Once we have found it, we set its valid bit to 0 and write it to disk. If, in the case that we do not find the file of choice with name fname, we return 0.

2.11 get_node_by_path

In get node by path, we want to follow a pathname until a terminal point is found. We first resolve the path name, walk through the path and find its inode; we did this procedure recursively. We first create duplicates of our path, by setting path1 and path2 variables to a duplicate pointer string of our path. We then set the baseName of our path1 to baseName and the parentName of our path2 to parentName (using dirname). In the base case, we check if the path is just the root or not. If it is, we read 0 into the inode, and return 0. Next, we want to address the recursive case. We first set our dirent variable into the struct dirent, and our recursiveInode into a struct inode. Then, we set an int a to our get node by path's returning value. In this recursive call, what happens is that our function keeps splitting up the parent name and the recursiveInode until it finds the int of choice. If our path is invalid, we just free the dirent and recursiveInode structs and return -1. Next, we use an integer b to see the input of dir find to make sure if the file exists or not based on if the path is valid or not. If b returns 0, we know that the path is not valid, so we return -1. Next, we set the inodeNum to the dirent's ino, and read the inodenum to the inode on disk.

FUSE File Operations:

2.12 tfs_mkfs

In tfs_mkfs, we are making the file system. We first call dev_init to find the diskfile path, and then write superblock information by setting the magic_num, max_inum and max_dnum to our given parameters. Next, we find the starting block of inode bitmap, which will give us the number of blocks required to store the superblock struct. We then increment if the block size goes over 1, which is found by checking modulo. Next, we find the calculatoion or the number of blocks required by inode bitmap, and increment if we need another block be-

cause there is still inode data remaining. Next, we set the data bitmap block to the sum of the inode bitmap blocks and the number of inode bitmap blocks that we originally set. Once again, we check to make sure that the number of data bitmap blocks fits in our current bitmap blocks size and if it doesn't, we increment by 1. We do the same for the inode bitmap blocks too. Then, we write our superblock's pointer into the our superblock, and write 0 into superblock's pointer. Next, we initialize inode bitmap, write the inodebitmap, and do the same for databitmap.

Next, we update bitmap information for the root directory. The root directory will always get the first inode number. We reserved the first data block for the root directory. We then formatted the data block 0 into an array of dirents. Then, we updated the inode for the root directory, and set all of its value accordingly. Next, we did a bio_read and inserted the value of the inode start ptr into our buffer. We then used bio_write to write our inode start ptr into the on disk buffer. Finally, we added the dirent for '.', and returned 0.

2.13 tfs_init

In tfs_init, we want to initalize the TFS by opening a disk, and reading our superblock into memory, and checking to make sure if the flat file exists or not. To do this, we first checked if the disk file could be found by using dev_open. If not found we use mkfs for setting up our inode bitmap and data bitmap. If the disk file is found, we will initialize in memory data structures and read the superblock from disk. We set our super block pointer, inode bitmap and data bitmaps. Then, we read the inode bitmap from our disk using bio_read and use memcpy to put the contents of inodebitmap into buffer. Then, we read the data bitmap from the disk, and follow the same procedure of memcpy as above.

2.14 tfs_destroy

This is the function that occurs when our file system is no longer mounted, so we want to erase all of the data. First, we de-allocate in memory data structures like sbPointer, inode bit map and data bitmap. Then, we close the disk file.

2.15 tfs_getattr

This function is called when we are trying to access a file or directory, and it returns the status of your files. In this function, we first called get node by path to get the inode from our path, and checked if our path was valid or not (if it was not valid, then we used a special code -ENOENT which means no such file or directory exists). Next, we filled in the attributes of our file into stbuf from inode, using S_IFREG (type constant of a regular file) and S_IFDIR (type constant of a regular directory). We also set the link number of our stbuf to 2, and used the stat struct to find uid and gid. Then, we freed pathsInode and returned 0 because our function was completed.

2.16 tfs_opendir

In tfs_opendir, we want to access a directory (cd command). We first called get node by path to get the inode from the path. If this was not found, then we returned -1, which meant that our path was not valid. We returned 0 if we were able to find the file.

2.17 tfs_readdir

In tfs_readdir, we wanted to read the directory (ls command). We first called get node by path to get the inode from path, and checked to make sure our path value was valid (if it wasn't, we returned -1). Next, we read the directory entries from its data blocks and copied them to our buffer. We set the number of data blocks to the pathsInode's size, and looped through them. We used bio_read to thread our data block into a buffer, so that we could look through its dirents. Next, we looped through the dirents of our datablock of choice, and filled in our buffer with the valid datablock's name.

2.18 tfs_mkdir

In tfs_mkdir, we wanted to create a directory (mkdir command). We first used dirname and basename to separate parent directory path and target directory name. We then called get_node_by_path to get the inode of the parent directory, and checked to make sure it was valid (if it wasn't, we returned 0). Next, we caled the get avail ino to get an available inode number (if there wasn't, this meant that there was no on memory space for the file), and dir_add to add the directory entry of the target directory to the parent directory, once again, checking to make sure if it already existed). Next, we updated inode for target directory and called writei to write the inode to disk. We first updated the directory name inode's link count, and then created a new inode struct for the base directory. We set all of the values of our newInode, and added the creation time. Next, we created two new directory entries for '.' (ptr to itself) and '..' (ptr to parent) for the base file, and returned 0.

2.19 tfs_rmdir

In tfs_rmdir, we wanted to remove the directory (rmdir command). We first used dirname and basename to separate the parent directory path and the target directory name. Similar to tfs_mkdir, we duplicated our path strings into path1 and path2, and then found the directoryName and baseName of the paths. Then, we called get_node_by_path to get the inode for the target directory (if the path is not valid, then we return -1). Then, we traversed through all directory entries of the target directory. We first looped through the data blocks, and bio_read this data block into a buffer, so that we could look through its dirents. Then, we looped through the dirents of this block, found the dirent which was valid. If we noticed, in our string comparison, that the name was '.'

or '..', we continued, because this meant that we had not accessed the actual dirent contents of our given file. If we found the specific dirent corresponding to our file, then we freed the buffer, paths inode and mutex unlock and returned ENOTEMPTY, which meant that our directory was not empty.

After this, we cleared our data block bitmap of the target directory. We first went to the direct_ptr and traversed through all block numbers, and freed these blocks in the data bitmap. We then cleared the inode bitmap and its datablocks, after writing them to disk. Then, we called get node by path to get the inode of the parent directory, and then did dir remove to remove the directory entry of the target directory.

2.20 tfs_create

In tfs_create, we want to create a file (touch command). We first use the dirname and base name to separate parent directory path and target file name. Then, we call get node by path to get the inode of the parent directory, and return 0 if this path is not valid. Then, we call get available inode to find an available inode number for our file (if it returns -1, this means there is no space for the file). Next, we call dir_add to add the directory entry of the target file to the parent directory (if this returns -1, it means that the directory entry already exists). Then, we update the inode for the target file by setting the values and malloc for a newInode that will contain our data. After calling writei to write the inode to disk, we free the pathsInode and newInode because it has already been accounted for.

2.21 tfs_open

In tfs_open, we want to access the file. We first call get node by path to get the inode from path, if the check returns -1, this means that the path is not valid. Otherwise, we will return 1 and successfully read the inode.

2.22 tfs_read

This functions is the read operation's call handler. We first call get inode by path to get the inode from our path, and then we check to make sure that the filesize that we have found is less than the offset and size (if it isn't, then we have to return 0). Next, we read the path's data blocks from disk, checking to make sure that the file size minus the offset is less than size. Next, we copy the correct amount of data from offset to buffer by using memcpy and bio_read. We make sure that all size is accounted for by adjusting the blockIndex and blockToStart as necessary.

2.23 tfs_write

In tfs_write, we follow the same pattern as is in tfs_read, the only difference is that we are writing to disk in this function, not reading from disk. We first call get node by path to get the inode from our path (if the path is not valid, we return -1). Next, based on the size and offset, we read the file path's data blocks from disk. We get the extra data blocks needed, total file size, and number of blocks total needed for this operation. Then, we account for the difference in the number of blocks in total and the current number of blocks we need to fulfill our operation (this is the space that is left over). If our diff is greater than 0, this means that we need to add extra blocks, which we do by adding a variable of getting available block number into our data block starting position. If the pathsInode size is equal to 0, we return 0. If the pathsInode's size is less than blockIndex + 1, we return 0 as well.

Next, we want to write the correct amount of data from offset to disk. We do this by going through a while loop in which size is greater than 0; in this loop, we find the remaining amount of block size required and use bio_read to go through the block to start. If the size is less than the remaining number of blocks, we use memcpy to copy the current location into the buffer, and do a bio_write to put it into the disk. If the size is greater than the remaining, this means that we need more blocks. Therefore, we use bio_write, add to our buffer if there are reamining blocks. If the pathsInode's size is less than the blockIndex+1, then we write the pathsInode's inode into our pathsInode, and reduce the size by size. Finally, we update the inode info and write it to disk using writei, along with updating the inode's modified time in stat.

2.24 tfs_unlink

tfs_unlink is called when removing a file (rm command). We first use dir name and base name to separate the parent directory path and the target file name by setting path1 and path2 variables that are duplicates of the path. Then, we call the get node by path to get the inode for the target file (if the path is not valid, we return 0). Next, we clear the data block bitmap of the target file by unsetting the bitmap of our data index. After that, we clear inode bitmap and its data block, and set the targetsInode's valid to 0 and write it into our disk. We then call get node by path to get the inode of our parent directory, and dir remove to remove the directory entry of the target file in its parent directory.

3 Additional steps we should follow to compile code

4 Any difficulties or issues faced while completing project

Some difficulties we faced while completing the project are listed in depth below:

- Debugging was a challenge, because there were a lot of pointer mainpulations. We had to make sure that we wouldn't run into segfaults by checking that our pointers did point to a valid item in memory.
- There were race conditions at times that could not be avoided as well, so we had to account for multiple threads to enter and exit our code using mutex. Often, we would think that there would be a specific output, but this would not be matched with our expectations because of multiple threads going through the code at once.
- It was also a challenge to learn how to use GDB with FUSE, so we ended up debugging by inserting print statements all over our code. This was a hassle, as we had to make sure that we inserted the line numbers in our print statements and their location to find a point in our file system that wasn't working as expected.