CS416 Project 4

Christopher Yong (cy287), Joshua Ross (jjr276)

1 Modifications

block.c and **block.h**, we changed the location of the DISK_SIZE macro from block.c to block.h so we could access it in tfs.c.

2 Benchmark Block Information

Total blocks used: 8192 or 32 MiB divided by BLOCK SIZE (4096).
Note: during the tfs_mkfs or the first time creating the disk, we assume there is enough blocks for the super block, inode bitmap, data bitmap, and inode region and **atleast** one block for the data portion.
To calculate the total number of data blocks that can be allocated in the data bitmap, we take the number of blocks we can partition the disk space into and then subtract the number of blocks used for the superblock, inode bitmap, data bitmap, and inode region.
For instance, in the default parameters given to us (4096 BLOCK SIZE, 32 MiB DISK SIZE, and max inodes is 1024, and max data blocks is 16384), there will be 67 blocks used solely for the superblock, inode bitmap, data bitmap, and inode region which leaves 8192 - 67 = 8125 blocks available for the data region. Then we compare this number to the MAX_DNUM macro and take whenever is smaller and in the default case, $8125 < 16384$ therefore we set max datablock index to be 8124 and maximum number of data blocks for the data region that can be used to 8125.
Note I noticed bio_write used pwrite so the disk file could grow beyond the DISKFILE macro but I do not believe that was the intended behavior according to piazza@183_f1.

3 Implementation with EXTRA CREDIT

We followed the comments outlined in the functions to implement the tfs functions and added some safety checks and updating of the links and size of inodes.

get_avail_ino

Note, we keep a global char array of our bitmaps and a global superblock struct that is read in from the disk during tfs_init.
Find a spot in the inode bit map where there is a free inode and return the inode number.

Step 1) Calculate the maximum char (byte) we should iterate too via ceil($\frac{superBlock.max\_inum+1}{8.0}$)

Step 2) For each char in the inode bit map (that is less than the max char calculated above), check if the char has a free page via 0xFF mask and see if it does not equal 0xFF. (that means one of the bits in the char is 0 and there is a free inode)

Step 3) for each bit of the free char, check if it is free and if so, create mask to set it to 1 by creating a bitMask (1 << bitIndex) and bitwise OR with the char and then write the bitmap to the disk and return the inode number associated with the bit via (charIndex * 8.0) + bitIndex.

get_avail_blkno

Note, we keep a global char array of our bitmaps and a global superblock struct that is read in from the disk during tfs_init.
Find a spot in the data bit map where there is a free data block and return the index.

Step 1) Calculate the maximum char (byte) we should iterate too via ceil($\frac{superBlock.max\_dnum+1}{8.0}$)

Step 2) For each char in the data bit map (that is less than the max char calculated above), check if the char has a free page via 0xFF mask and see if it does not equal 0xFF. (that means one of the bits in the char is 0 and there is a free data block)

Step 3) for each bit of the free char, check if it is free and if so, create mask to set it to 1 by creating a bitMask (1 << bitIndex) and bitwise OR with the char and then write the bitmap to the disk and return the data block number associated with the bit via superBlock.d_start_blk + ((charIndex * 8.0) + bitIndex).

readi

General idea is to create a buffer, read in the block index and then memcpy the inode in the buffer into the struct inode.

Step 1) Calculate the block index of where the inode lies via $superBlock.i\_start\_blk + \frac{\frac{ino}{BLOCK\_SIZE}}{sizeof(structinode)}$

Step 2) create a buffer size of BLOCK_SIZE

Step 3) read in the block from the disk into the buffer

Step 4) memcpy where destination is the inode struct and the source is (buffer + sizeof(struct inode) * (ino % $\frac{BLOCK\_SIZE}{sizeof(structinode)}$) and bytes to copy is sizeof(struct inode)

writei

General idea is the same as readi except we just write the inode struct into the buffer and then write the block back into disk.

Step 1) Calculate the block index of where the inode lies via $superBlock.i\_start\_blk + \frac{ino}{\frac{BLOCK\_SIZE}{sizeof(structinode)}}$

Step 2) create a buffer size of BLOCK_SIZE

Step 3) read in the block from the disk into the buffer

Step 4) memcpy where destination is the (buffer + sizeof(struct inode) * (ino % $\frac{BLOCK\_SIZE}{sizeof(structinode)}$) and the source is inode struct and bytes to copy is sizeof(struct inode)

Step 5) write the buffer back into the disk at the block index calculated in step 1

dir_find

General idea is to read in every direct ptr and indirect ptr that is not 0 and traverse through each direct block by dirent struct and see if it is valid and the fname and len match.

Step 1) read in the inode struct of the ino passed in via readi

Step 2) Create a buffer of size BLOCK_SIZE.

Step 3) For each direct pointer in the inode struct

(a) if direct pointer is 0, continue to next direct ptr

(b) if direct pointer is not 0, read in the block of the direct pointer via bio_read and then search through the read in block by struct dirent by casting the read in block to (struct dirent*), comparing if the valid bit is 1 and name and length matches. If we find a match, memcpy the found dirent into the dirent struct and return 1.

Step 4) At this point, all direct pointers have been searched and still has not been found. Therefore we now search for each indirect pointer in the inode struct.

(a) if indirect pointer is 0, continue to next indirect pointer

(b) if indirect pointer is not 0, read in the block pointed by the indirect pointer and then traverse the read in indirect block by int (since each direct pointer will be int) and check if the direct pointer inside the indirect block is not 0, and if it is not 0, perform step 3b

Step 5) At this point we searched all allocated indirect and directed blocks and could not find the entry thus we just return -1

dir_add

General idea is traverse the direct pointers and indirect pointers in order and for each one that is allocated, read till reach the direct block and search if there is a non valid dirent struct that can be inserted and if the direct or indirect pointer are not allocated, allocate a data block and insert the dirent into it.

Step 1) Create a dirent struct called toInsertEntry and fill the fields with the necessary info

Step 2) create a buffer of BLOCK_SIZE

Step 3) For each direct pointer in the inode struct

(a) if direct pointer is 0, allocate a new data block for the direct pointer and memset the buffer to '0' and then memcpy the toInsertEntry dirent struct into the first index of the buffer and bio_write the buffer to the new data block index. Update the directory inode's size by struct dirent and the vstat size by BLOCK_SIZE and increase vstat st_blocks by 1 and then writei the inode to update it on disk and return 1. (Note, we could not get a new data block, return -1)

(b) if the direct pointer is not 0, read in the block pointed by the direct pointer and then traverse by struct dirent and check if the valid bit is 0, if it is, memcpy the toInsertEntry into that spot in the read in block and then update the size field of the inode by sizeof(struct dirent) and writei the directory inode to disk and return 1.
If the valid bit is 1, just continue to next dirent struct in the buffer.

Step 4) At this point, found no spot in direct pointers, now we search the indirect pointers.

(a) if indirect pointer is 0, allocate a new data block for the indirect pointer and a new data block for the 0th index of the indirect block. (If we could not allocate 2 data blocks, return -1 and free the data block if one was allocated for the indirect pointer but not the direct block) Memset the buffer to 0 and write the direct block index to the buffer at index 0 and then bio write to disk. Afterwards

memset the buffer to 0 again and then memcpy the toInsertEntry to the buffer at index 0 and write to disk again. Update the inode information, size field will be increment by sizeof(struct dirent), vstat size by BLOCK_SIZE * 2 and st_blocks by 2 and update directory inode to disk via writei and return 1.

(b) if indirect pointer is not 0, read in the indirect block pointed by the indirect pointer and then traverse by int (since direct block indexes are int) and perform step 3 for each int in the indirect block.

Step 5) At this point we searched all allocated indirect and directed blocks and could not find a spot thus we just return -1

dir_remove

General idea is to traverse the direct pointers and indirect pointers in order and for each one that is allocated, read till reach the direct block and search if there is a valid entry with name and len desired and if so, set the valid bit to 0 and write to disk. (Very similar to find except we just invalid the valid bit and write to disk afterwards)

Step 1) create buffer of size BLOCK_SIZE

Step 2) for each direct pointer

(a) if direct pointer is 0, continue to next direct pointer

(b) if direct pointer is not 0, read the block in pointed by the direct pointer and then traverse by struct dirent and if it matches the fname,length, and is valid, set the valid bit to 0 and write the block to the disk. Update the directory inode size field by decrementing it by sizeof(struct dirent) and then writei and return 1.

Step 3) for each indirect pointer

(a) if indirect pointer is 0, continue to next indirect pointer

(b) if indirect pointer is not 0, read in the indirect block pointed by the indirect pointer and then traverse the read in indirect block by int (since each direct pointer will be int) and check if the direct pointer inside the indirect block is not 0, and if it is not 0, perform step 2b

Step 4) At this point we searched all allocated indirect and directed blocks and could not find the entry thus return -1. Note it could technically be 1 too since if it does not exist in the inode struct then obviously it can be considered removed too.

get_node_by_path

General idea is to record the characters till we reach a '/' and treat it as a delimiter where we perform a dir_find to find it and then continue till the path is '
0', reading in the next inode of the next directory.

Step 1) read in the inode struct of the ino passed in

Step 2) create a path buffer of size 256 (chosen for file name being maxed 255 and + 1 for null terminator) and a pathIndex variable to 0 and index to 1

Step 3) see if path[index] == '\0' and if so, we are just searching for the root inode therefore we just return 1

Step 4) while path[index] != '\0'

(a) if path[index] == '/', dir_find the current pathBuffer and if it exists then readi the ino of the directory entry ino and then reset the pathBuffer and pathBufferIndex to 0 and '
0'. (Note if dir_find returns -1, return -1)

(b) if path[index] != '/', then add the current character in the path to the pathBuffer via pathBuffer[pathBufferIndex] = path[index] and then pathBufferIndex++

(c) regardless if path[index] == '/' or != '/' we have to do: index++ to get next character of the path.

Step 5) At this point, the directory entry we want is the basename of the path and thus we have to do one more dir_find and readi to get the specified inode (note if dir_find returns -1, return -1).

Step 6) return 1

tfs_mkfs

General idea is to call dev_init to initialize the disk and then set the necessary info of the superblock and bitmap (ensuring the bits of the bitmap are consistent with the maximum number of data blocks and inodes)
Note we keep a global superblock struct and bitmaps.

Step 1) call dev_init()

Step 2) set superblock.magic_num to the MAX_NUM macro

Step 3) set superBlock.max_inum to MAX_INUM - 1

Step 4) set superBlock.i_bitmap_blk = 1

Step 5) set superBlock.d_bitmap_blk = 2

Step 6) set superBlock.i_start_blk = 3

Step 7) set superBlock.d_start_blk = 3 + ceil($\frac{MAX\_INUM}{\frac{BLOCK\_SIZE}{sizeof(structinode)}}$)

Step 8) set superBlock.max_dnum = ($\frac{DISK\_SIZE}{BLOCK\_SIZE} - superBlock.d\_start\_blk$) < MAX_DNUM ? ($\frac{DISK\_SIZE}{BLOCK\_SIZE} - superBlock.d\_start\_blk$)$-1 : MAX\_DNUM -$ 1

Step 9) create a buffer size of BLOCK_SIZE and copy the superblock into it and then bio_write to block index 0

Step 10) Check if the number of inodes is a multiple of 8 and if not, set the necessary bits in the last char to 1's to indicate those bits should not be used to give inodes since that exceeds the max inode number

Step 11) Check if the number of usable data bit map (superBlock.max_dnum + 1) is a multiple of 8 and if not, set the necessary bits in the last char to 1's to indicate those bits should not be used to give data blocks since that exceeds the max data block number

Step 12) create the root inode and call get_avail_ino() to retrieve an ino and set the necessary fields and then dir_add the "." and ".." entry. (Note we do not need to explicitly call bio_write for this since dir_add and get_avail_ino will do it for us to update the bitmaps and write the inode to the disk)

Step 13) return 0

tfs_init

General idea is to call disk open and see if it returns a non-error and if so just read in the superblock and bitmap info otherwise call tfs_mkfs

Step 1) lock global mutex

Step 2) call dev_open and check if it is -1, if it is -1 then call tfs_mkfs

Step 3) if dev_open was successful, read in the superblock and bitmaps

Step 4) unlock global mutex

Step 5) return null

tfs_destroy

General idea is to close the disk file

Step 1) lock global mutex

Step 2) bio write the bitmaps

Step 3) call dev_close

Step 4) unlock global mutex

tfs_getattr

General idea is to call get node by path and if not error, set the returned inode struct to stbuf

Step 1) lock globalmutex

Step 2) create struct inode called ino

Step 3) call get node by path and if error, unlock global mutex and return -ENOENT

Step 4) if no error, set stbuf to inode.vstat

Step 5) unlock globalmutex

tfs_opendir

General idea is to call get node by path and check the return type

Step 1) lock global mutex

Step 2) call get node by path and if error, unlock global mutex and return -ENOENT

Step 3) check if returned inode is type DIRECTORY_TYPE and if not, unlock global mutex and return -ENOTDIR

Step 4) update the access time of the inode (update meaning write to disk as well)

Step 5) unlock global mutex

Step 6) return 0

tfs_readdir

General idea is same as dir_find where traverse through all direct and indirect pointers except we just fill in the buffer with anything that is valid.

Step 1) lock global mutex

Step 2) for each direct pointer

(a) if direct pointer is 0, continue to next direct pointer

(b) if direct pointer is not 0, read in the block pointed by the direct pointer and traverse by struct dirent and for each one that is valid, write to the buffer

Step 3) for each indirect pointer

(a) if indirect pointer is 0, continue to next indirect pointer

(b) if indirect pointer is not 0, read in the indirect block pointed by the indirect pointer and then traverse by int (since each direct block index is int) and perform step 2

Step 4) unlock global mutex

tfs_mkdir

General idea is to retrieve parent directory inode and then add the new directory as a new entry in the parent directory.
Note I excluded some error checking that we do in the code for the steps below (e.g. if the sub directory could not allocate "."), we revert the changes in parent directory, unlinking and unlocking the mutex and then return -EDQUOT)

Step 1) separate the basename and parent directory path

Step 2) lock global mutex

Step 3) retrieve the parent directory inode via get node by path (if get node by path returns -1, unlock global mutex and return -ENOENT)

Step 4) retrieve a new ino via get avail ino and if it is -1 meaning it could not find a new inode, unlock global mutex and return -EDQUOT

Step 5) add the sub directory to the parent directory's entries

Step 6) add the "." and ".." to sub directory

Step 7) update the parent directory and sub directory inodes with the necessary info (e.g. + 1 link for parent directory, 2 links for sub directory) (update meaning write to disk as well)

Step 8) unlock global mutex

Step 9) return 0

tfs_rmdir

General idea is same as mkdir where we retrieve the parent directory but instead we perform a dir_remove.

Step 1) lock global mutex

Step 2) retrieve the inode of the removed directory via get node by path

Step 3) check if the being removed directory inode size is 2 * sizeof(struct dirent) to see if it is empty and if it is not, unlock global mutex and return -ENOTEMPTY

Step 4) retrieve the parent directory inode and then perform a dir_remove for the directory to be removed

Step 5) free the inode of the removed directory inode struct and the data blocks

(a) toggle the bit of the inode bitmap of the ino the inode struct uses

(b) for each direct pointer, if it is not 0, toggle the bit in the data bitmap

(c) for each indirect pointer, read in the indirect block and traverse via int and see if it is not 0 and if it is not 0, toggle the bit in the data bit map and then after traversing through the whole indirect block by ints, toggle the indirect block index in the data bitmap

(d) write the bitmaps to disk

Step 6) update the parent directory inode

Step 7) unlock global mutex

Step 8) return 0

tfs_create

General idea is to get an ino for the file and then just create/set a dirent struct inside one of the data blocks of the parent directory.
Note we excluded some error checking from the steps (we do check if the dir_add failed to add because maybe not enough data blocks or something, then we unallocate the file ino and return -EDQUOT)

Step 1) lock global mutex

Step 2) get parent directory inode via get node by path (if get node by path returns -1, unlock global mutex and return -ENOENT)

Step 3) get an ino from get avail ino (and if get avail ino returns -1 then unlock global mutex and return -EDQUOT)

Step 4) dir_add the new file name in the parent directory

Step 5) update the directory and file inode (update meaning write to disk as well)

Step 6) unlock global mutex

Step 7) return 0

tfs_open

General idea is to get node by path to see if it exists or not.

Step 1) lock global mutex

Step 2) call get node by path and see if it returns -1 and if it does, unlock global mutex and return -ENOENT

Step 3) unlock global mutex

Step 4) return 0

tfs_read

General idea is to convert the offset to block number and then go through each block, reading either BLOCK_SIZE (might be less than BLOCK_SIZE for first block since offset could start in middle of a block) or size whether is smaller till size is 0 or encountered unallocated data.

Step 1) lock global mutex

Step 2) retrieve the file inode by get node by path (if get node by path returns -1, unlock global mutex and return -ENOENT)

Step 3) check if offset $\geq$ file size and if so return 0

Step 4) find blockIndex to start at via offset / BLOCK_SIZE

Step 5) find the number of bytes that is able to be read in the first block via size $\leq$ (BLOCK_SIZE - (offset % BLOCK_SIZE)) ? size : (BLOCK_SIZE - (offset % BLOCK_SIZE)) and set it equal to a variable called bytesToCopyInBlock

Step 6) create two buffers of size BLOCK_SIZE for the indirect and direct blocks and a variable called previousBlockIndex = 0 and a variable called bytesCopied = 0

Step 7) while size > 0

(a) if blockIndex < 16

i. if file inode's direct pointer == 0 then break otherwise bio_read in the direct block into the direct buffer

(b) if blockIndex $\geq$ 16

i. if (blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)} \geq 8$ then break (this means no more indirect blocks to read and thus we are done with reading)

ii. if file inode indirect pointer[(blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$] == 0 then break (this means indirect block has not been allocated and thus we are done reading)

iii. if previousBlockIndex == 0 || (blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$!= (previousBlockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$ then bio_read the indirect block from the indirect pointer specified by (blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$ into indirect block buffer. (Basically this line sees if we have to read in a new indirect block or not)

iv. Check if the direct block within the indirect block is allocated and if not, break and if so read it in into the direct block buffer

(c) At this point, the direct block buffer has the contents we would like to read. Therefore we do a memcpy where source is the direct buffer + (offset % BLOCK_SIZE) and destination is buffer + bytesCopied and the size is bytesToCopyInBlock

(d) set offset to 0

(e) update bytesCopied by bytesToCopyInBlock

(f) decrease size by bytesToCopyInBlock

(g) update bytesToCopyInBlock to size < BLOCK_SIZE ? size : BLOCK_SIZE

(h) set previousBlockIndex to blockIndex

(i) increment the blockIndex

Step 8) update time access of file inode (writing to disk too)

Step 9) unlock global lock

Step 10) return bytesCopied

tfs_write

General idea is the same as tfs_read except whenever you reach an unallocated block, allocate a new data block for it.

Step 1) lock global mutex

Step 2) retrieve the file inode by get node by path (if get node by path returns -1, unlock global mutex and return -ENOENT)

Step 3) check if offset > file size and if so return -ESPIPE

Step 4) find blockIndex to start at via offset / BLOCK_SIZE

Step 5) find the number of bytes that is able to be read in the first block via size ≤ (BLOCK_SIZE - (offset % BLOCK_SIZE)) ? size : (BLOCK_SIZE - (offset % BLOCK_SIZE)) and set it equal to a variable called bytesToCopyInBlock

Step 6) create two buffers of size BLOCK_SIZE for the indirect and direct blocks and a variable called previousBlockIndex = 0 and a variable called bytesWritten = 0 and create a variable called originalOffset = offset

Step 7) while size > 0

(a) if blockIndex < 16

i. if file inode's direct pointer == 0 then allocate a new data block in that index and memset the direct block to 0 otherwise bio_read in the direct block into the direct buffer

(b) if blockIndex $\geq$ 16

i. if (blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)} \geq 8$ then break (this means no more indirect blocks to read and thus we are done with reading)

ii. if file inode indirect pointer[(blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$] == 0 then allocate a new data block and memset indirect block to 0 otherwise:

A. if previousBlockIndex == 0 || (blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$!= (previousBlockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$ then bio_read the indirect block from the indirect pointer specified by (blockIndex - 16) / $\frac{BLOCK\_SIZE}{sizeof(int)}$ into indirect block buffer. (Basically this line sees if we have to read in a new indirect block or not)

iii. Check if the direct block within the indirect block is allocated and if not, allocate a new data block and memset the direct buffer to 0 otherwise if it is allocated, read it in into the direct block buffer

(c) At this point, the direct block buffer has the contents we would like to write. Therefore we do a memcpy where source is the buffer + bytesCopied and destination is direct buffer + (offset % BLOCK_SIZE) and the size is bytesToCopyInBlock

(d) bio write the direct block buffer to the disk

(e) set offset to 0

(f) update bytesWritten by bytesToCopyInBlock

(g) decrease size by bytesToCopyInBlock

(h) update bytesToCopyInBlock to size < BLOCK_SIZE ? size : BLOCK_SIZE

(i) set previousBlockIndex to blockIndex

(j) increment the blockIndex

Step 8) update the file size by size += bytesWritten $\leq$ (file size - original offset) ? 0 : bytesWritten - (file size - original offset)

Step 9) update time access of file inode (writing to disk too)

Step 10) unlock global lock

Step 11) return bytesWritten

tfs_unlink

General idea find the parent directory and perform a dir_remove and then free the file inode and the data blocks allocated inside the inode.
Note we excluded some error checking that is in our code in the steps below.

Step 1) lock global mutex

Step 2) get file inode by get node by path

Step 3) get parent directory inode by get node by path

Step 4) perform dir_remove with parent directory and the file

Step 5) free the file inode ino and the data blocks

(a) toggle the bit of the inode bitmap of the ino the inode struct uses

(b) for each direct pointer, if it is not 0, toggle the bit in the data bitmap

(c) for each indirect pointer, read in the indirect block and traverse via int and see if it is not 0 and if it is not 0, toggle the bit in the data bit map and then after traversing through the whole indirect block by ints, toggle the indirect block index in the data bitmap

(d) write the bitmaps to disk

Step 6) unlock global mutex

Step 7) return 0

4 Benchmark Results

4.1 simple_test

```
cy287@cp:~/CS416/Project 4/code/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: File unlink success
TEST 6: Directory create success
TEST 7: Sub-directory create success
running time: 631 micro-seconds
Benchmark completed
```

Total Blocks Used
blocks for data + blocks for inodes + blocks for others = 107 + 7 + 3 = 117 blocks.
Required blocks for data

- 1 data block for root directory entries

- 100 data blocks for the 100 sub directories in files
- 6 data blocks to store the mappings of the 100 sub directories in the files directory

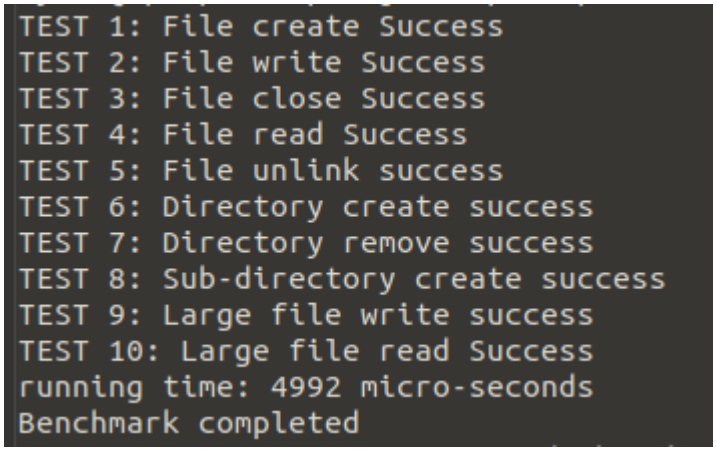
Total Blocks = 1 + 100 + 6 = **107** blocks used for data.
Required blocks for inodes

- 100 inodes required for sub directories in files
- 1 inode for the files directory
- 1 inode for root directory

Inode used = 100 + 1 + 1 = 102 inodes and each block can hold 16 inodes therefore we require **7** blocks for the inode blocks allocated in the inode region.
Required blocks for others
We used 3 blocks for the inode bitmap, data bitmap, and superblock
Run time: 631 microseconds
Note sometimes we get 20-50 microseconds too.

```
cy287@cp:~/CS416/Project 4/code/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: File unlink success
TEST 6: Directory create success
TEST 7: Sub-directory create success
running time: 23 micro-seconds
Benchmark completed
```

4.2 test_case



Total Blocks Used
blocks for data + blocks for inodes + blocks for others = 2157 + 7 + 3 = 2167 blocks.
Required blocks for data

- 1 data block for root directory entries
- 100 data blocks for the 100 sub directories in files
- 6 data blocks to store the mappings of the 100 sub directories in files directory
- 2050 data blocks for the large file (8388608 bytes), require 2048 direct blocks + 2 indirect blocks

Total Blocks: 1 + 100 + 6 + 2050 = **2157** blocks required for the data.
Required blocks for inodes

- 100 inodes for the sub directories in files
- 1 inode for file directory
- 1 inode for the root directory
- 1 inode for the large file

Inode used: 100 + 1 + 1 + 1 = 103 inodes used and each block can hold 16 inodes therefore we require **7** blocks for the inode blocks allocated in the inode region.
Required blocks for others
We used 3 blocks for the inode bitmap, data bitmap, and superblock.
Run time: 4992 microseconds
Note sometimes I am getting 150 to 160 microseconds for this too.

```
cy287@cp:~/CS416/Project 4/code/benchmark$ ./test_case
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: File unlink success
TEST 6: Directory create success
TEST 7: Directory remove success
TEST 8: Sub-directory create success
TEST 9: Large file write success
TEST 10: Large file read Success
running time: 290 micro-seconds
Benchmark completed
```

4.3 Full Capacity test_case

For this testcase, we modified test_case.c and tested the maximum number of inodes and all data blocks filled in the disk.

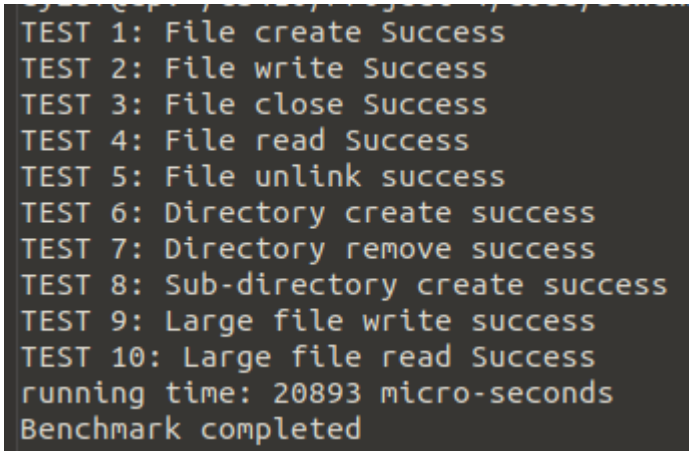
Modifications

```
5 #define N_FILES 1021
6 #define BLOCKSIZE 4096
7 #define FSPATHLEN 256
8 #define ITERS 16
9 #define ITERS_LARGE 7041
0 #define FILEPERM 0666
1 #define DIRPERM 0755
```

LS

```
total 3556
drwxr-xr-x    3 cy287 allusers     4096 Apr 29 19:52 .
drwx------    3 cy287 allusers     4096 Apr 22 15:56 ..
drwxr-xr-x 1023 cy287 allusers   225280 Apr 29 19:52 files
-rwxr-xr-x    1 cy287 allusers 28839936 Apr 29 19:52 largefile
```

Results



Total Blocks Used
blocks for data + blocks for inodes + blocks for others = 8125 + 64 + 3 = 8192 blocks.
Required blocks for data

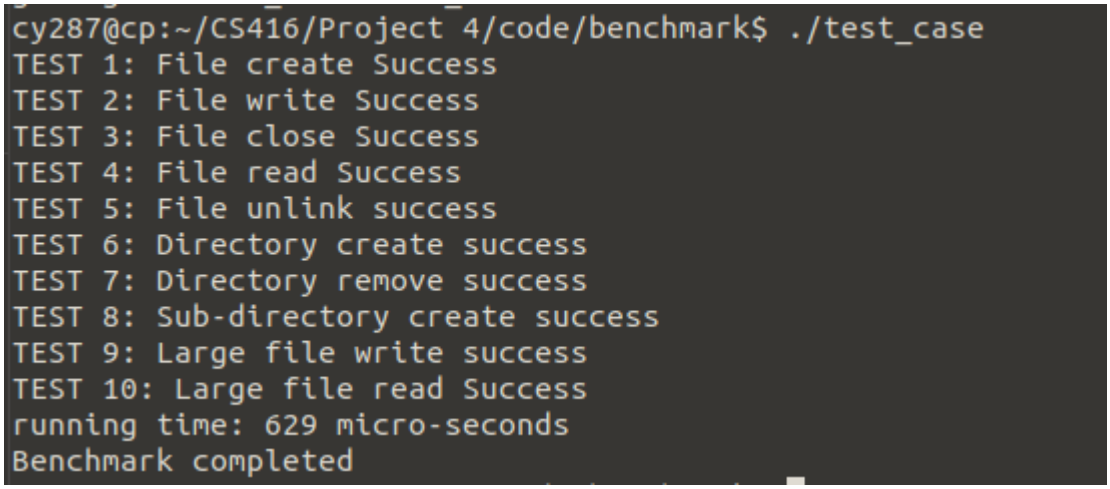
- 1 data block for root directory entries
- 1021 data blocks for the 1021 sub directories in files
- 55 data blocks to store the mappings of the 1021 sub directories in files directory, requiring 54 direct blocks + 1 indirect block
- 7048 data blocks for the 28839936 byte file, requires 7041 direct blocks + 7 indirect blocks

Total blocks: 1 + 1021 + 55 + 7048 = **8125** blocks used for the data.
Required blocks for inodes

- 1021 inodes for sub directories in files
- 1 inode for the files directory
- 1 inode for the root directory
- 1 inode for large file

Inodes used: 1021 + 1 + 1 + 1 = 1024 inodes. Every block can hold max 16 inodes thus we require **64** blocks for the inode blocks allocated in the inode region.
Required blocks for others
We used 3 blocks for the inode bitmap, data bitmap, and superblock.
Verifying the results
MAX DISK SIZE = 32 MiB therefore $\frac{32MiB}{BLOCK\_SIZE}$ = maximum number of

blocks = $\frac{(32*1024*1024)}{4096}$ = 8192 blocks which is the total number of blocks allocated and therefore this checks out.
Run time: 20893 microseconds
Sometimes I am getting 400 to 600 microseconds.



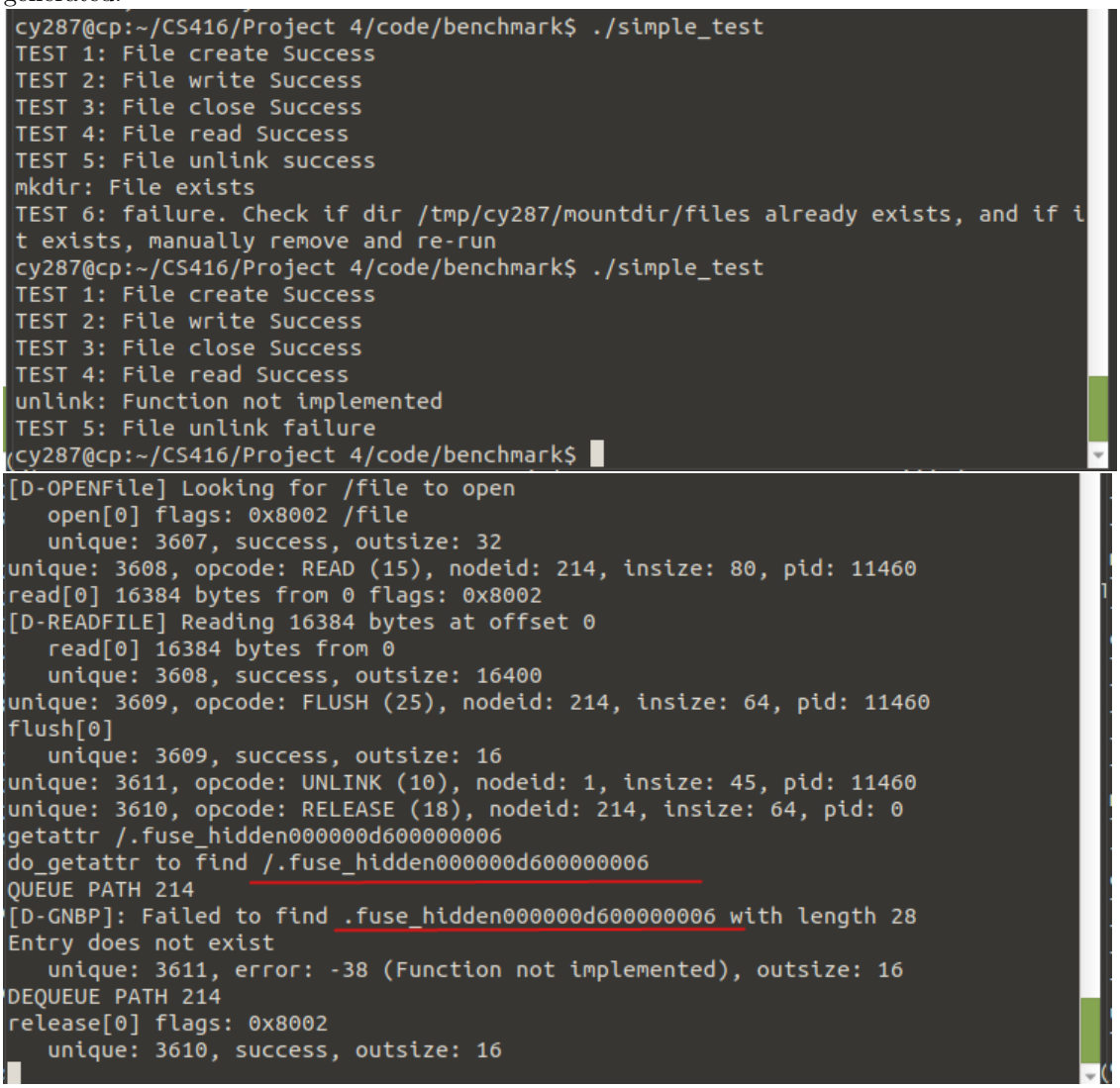
5 Additional Steps

We do not have any additional steps required to compile our code. (We used pthread library to add locks however I do not think we need to have -pthread to use it or if we do, it is not giving us an error when we compile right now for the benchmarks)

6 Difficulties & Issues

There is a rare race condition? bug with our program that occurs when we are not using the $-s$ flag even though all our of tfs functions that we have to implement (except tfs mkfs) have a global mutex that we lock in the beginning of the tfs function and unlocks just before we return as seen in the implementation section of our report and in the code.
We are not sure how to fix it since it occurs very rarely and when it does, it is because FUSE is attempting to delete some fuse file but it could not and thus the removal of the file fails. If you run the test case again afterwards, it succeeds. Note, we were not able to get this bug when tfs was run with $-s$ however since this bug is not easily reproducible, this could just us getting lucky with it not occurring.
Here is the output and the gbd output of when this occurs. **NOTE** Ignore the failed to remove the directory (we were just spamming simple test to get this

bug and it would occur even when if we removed the 100 directories simple test generated.



Showing if we run it again, it succeeds now.

```
t exists, manually remove and re-run
cy287@cp:~/CS416/Project 4/code/benchmark$ ./simple_test
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
unlink: Function not implemented
TEST 5: File unlink failure
cy287@cp:~/CS416/Project 4/code/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: File unlink success
mkdir: File exists
TEST 6: failure. Check if dir /tmp/cy287/mountdir/files already exists, and if i
t exists, manually remove and re-run
cy287@cp:~/CS416/Project 4/code/benchmark$
```

```
File  Edit  View  Search  Terminal  Help
   read[0] 16384 bytes from 0
   unique: 3635, success, outsize: 16400
unique: 3636, opcode: FLUSH (25), nodeid: 214, insize: 64, pid: 24932
flush[0]
   unique: 3636, success, outsize: 16
unique: 3637, opcode: RELEASE (18), nodeid: 214, insize: 64, pid: 0
unique: 3638, opcode: UNLINK (10), nodeid: 1, insize: 45, pid: 24932
QUEUE PATH 214 (w)
release[0] flags: 0x8002
   unique: 3637, success, outsize: 16
DEQUEUE PATH 214 (w)
unlink /file
[TFS-UNLINK]: Unlinking /file
   unique: 3638, success, outsize: 16
unique: 3639, opcode: FORGET (2), nodeid: 214, insize: 48, pid: 0
FORGET 214/2
DELETE: 214
unique: 3640, opcode: LOOKUP (1), nodeid: 1, insize: 46, pid: 24932
LOOKUP /files
getattr /files
do_getattr to find /files
   NODEID: 3
   unique: 3640, success, outsize: 144
```

Successful Unlink

In addition, we have some implementation specific stuff listed in section 8 which could change the expected output.

7 Extra Credit

See section 3

8 Our Implementation

Data bitmap

In our implementation, the data bitmap represents the number of blocks in the disk that are not used by the super block, bitmap blocks, and inode region (aka inode table).
For instance, in our implementation given 32 MiB, 1024 inodes, and 4096 Block Sizes, the inode region takes 64 blocks and the super and bitmaps take up 3 blocks therefore the data bitmap will only allow 8192 - 67 or 8125 data blocks to be represented in our data bitmap with the first block being the 67th block and last being 8124th block (assuming indexed 0).

Write and Read Offsets

In our write, we only allow append mode and not truncate meaning every write will either append to the end or just overwrite written data already at some offset. (Thus ECHO will not work correctly). Note according to Piazza @200 this is ok. In addition, our write does not allow data to be written at offsets greater than the file size as we deemed this to be invalid operation so technically this is our implementation way and not an issue but leaving it here in case you guys consider it one. To solve this, we would just have to allocate the bytes between the file size and the offset however we did not implement this because according to Piazza @200, this is fine to consider it an error.

Directory and File Sizes

For directories, the size of each directory is in multiples of blocks allocated and it includes both the direct and indirect blocks used for this directory. (stat to see how many blocks are allocated to the directory)
For files, the size of each file is the number of written bytes to the files (note overwriting old written bytes with new written bytes will not increase the file size), to see the number of blocks allocated to the file, check the st_blocks of the file (use stat). This will be the number of indirect and direct blocks allocated to the file.