Tiny File System using FUSE

Names:

Christopher Naporlee - cmn134 Section 3
Michael Nelli - mrn73 Section 3

Bitmap Functions:

1. get_avail_ino()
    a. Reads in the inode bitmap from disk
        i. Location of the bitmap is determined from superblock
    b. Traverses through bitmap, looking for an open (0) bit
        i. If found, sets that bit to closed (1)
        ii. Writes the updated bitmap back to disk
    c. Returns open inode. If no open inode returns -1
2. get_avail_blkno()
    a. Performs the same logic as get_avail_ino(), but instead the writes and reads are to the data block bitmap
    b. Also, the index we retrieve from the bitmap will be added to the start of the data block.
        i. Example, if we get index 2 from the bitmap, and our data blocks start at 3, return 5.

Inode Functions:

1. readi()
    a. Based on the passed in inode number, can calculate the block that contains our inode and the offset
        i. Inode_block = inode_start_block + (ino * sizeof(struct inode) / BLOCK_SIZE
        ii. Offset = ino % (num inodes in one block)
    b. Read in the block that contains our inode
    c. Once we read in the block that contains our inode, and the offset, we can simply index the buffer and retrieve the relevant information
2. writei()
    a. Based on the passed in inode number, can calculate the block that contains our inode and the offset
        i. Inode_block = inode_start_block + (ino * sizeof(struct inode) / BLOCK_SIZE
        ii. Offset = ino % (num inodes in one block)
    b. Read in the block that contains our inode
    c. Once we read in the block that contains our inode, and the offset, we can simply index the buffer and write to that location our inode data.
    d. Then, write that block back to disk to update inode on disk

Directory and Namei functions:

1. dir_find()
    a. Read in the inode of the directory using readi()
    b. Allocate a buffer for the directory’s entries
    c. For each valid data block of the directory inode
        i. Read the data block into memory
            1. For each entry in the data block, check if it is valid and the string matches the requested file name
                a. If found, pass back entry information and return
    d. If requested entry not found, return no entry.
2. dir_add()
    a. Uses dir_find() to see if the entry name already exists within the directory
        i. If it does, returns exists error
    b. Allocate buffer for directory’s entries
    c. For each valid data block of the directory inode
        i. Read the data block into memory
            1. For each entry in the data block check if it is not valid
                a. If we found a non valid entry, we can use that to make a new directory entry.
                b. Write updated inode to disk
                c. Write updated directory entries to disk

3. dir_remove()
    a. Allocate a buffer for the directory’s entries
    b. For each valid data block of the directory inode
        i. Read the data block into memory
            1. For each entry in the data block check if it is valid and the entry name matches the requested file name
                a. If both true, delete the entry, update directory inode and write updated entries to disk

4. get_node_by_path()
    a. Given a path, separates the words
        i. /path/to/file becomes “path” “to” “file”
    b. For each word, look up the word using dir_find() and store the possibly found entry in a direct struct.
        i. Using the dirent struct, we can keep walking down the path until the last word (entry) is found or not.

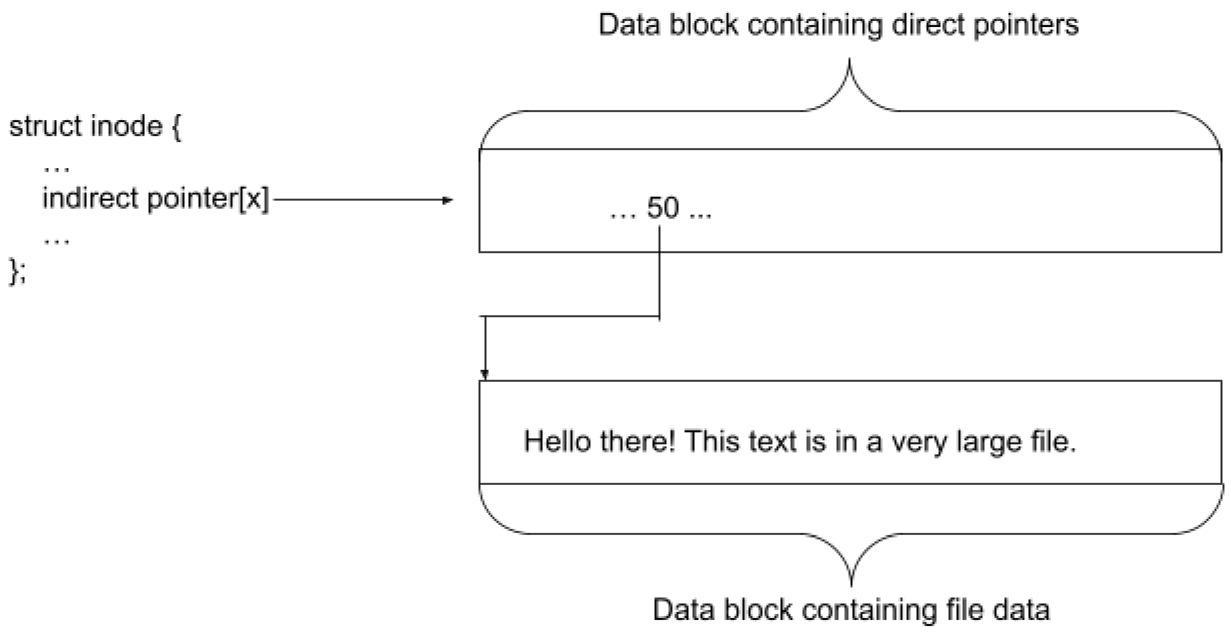
FUSE-based File System Handlers:

1. tfs_init()
    a. Checks if disk file is valid, if not calls tfs_mkfs()
        i. tfs_mkfs() does the important work of setting up the disk
        ii. Sets up superblock values such as location of inode and block bitmaps, and where the data blocks start
        iii. Initializes root directory and writes initial entries (“.” and “..”) to disk
2. tfs_destroy()
    a. Deallocates all malloc’d data structures
        i. superblock
        ii. inode bitmap
        iii. block bitmap
    b. Closes the disk
3. tfs_getattr()
    a. Uses get_node_by_path() to retrieve inode information of a given path
    b. If get_node_by_path() succeeded, retrieves stat information from inode
4. tfs_opendir()
    a. Calls get_node_by_path() to retrieve inode from a given path
    b. If not found returns error, else 0
5. tfs_readdir()
    a. Reads in inode information of path using get_node_by_path
    b. For each valid data block pointer
        i. Read data block pointer into memory
        ii. For each entry within data block
            1. If the entry is marked as valid, use fuse’s filler() function to read the entry’s name into the buffer
6. tfs_mkdir()
    a. Calls get_node_by_path() to grab the inode of the parent directory
    b. Retrieves an open inode index
    c. Adds the new directory to the parent directory using dir_add()
    d. Once added, updates the new directories inode values and stats
    e. Initializes the first directory entries (“.” and “..”)
    f. Writes the new directory to the inode entries and writes the new directory’s entries to the data block
7. tfs_rmdir()
    a. Calls get_node_by_path() to ensure the directory exists and get the inode
    b. Ensures that the directory is empty before it starts delete process
    c. Clears the block bitmap that the direct pointers index
    d. Clears the inode index and writes the update inode bitmap to disk
    e. Writes the updated inode information to the inode block
    f. Calls get_node_by_path to retrieve the inode information of the housing parent directory
    g. Uses dir_remove to remove the directory from the parent directory

8. tfs_create()
    a. Calls get_node_by_path() to grab the inode of parent directory that we will modify
    b. Grab the next open inode index for our new file
    c. Add the target file name to the parent directory using dir_add()
    d. Initialize new inode and stat values for our new file
    e. Write the new inode information to disk
9. tfs_open()
    a. Calls get_node_by_path() to retrieve inode from a given path.
    b. If not found returns error, else 0
10. tfs_read()
    a. First fetch the inode of the target file using get_node_by_path()
    b. Then get the relevant block to read from by the offset / BLOCK_SIZE
        i. If offset / BLOCK_SIZE is greater than the amount of direct pointers, we use indirect pointers instead (see section on supporting large files)
    c. Read in the relevant data block from the disk to memory
    d. Begin to read the block keeping 3 things in mind
        i. We don't read more than requested
        ii. We don’t attempt to read more than the number of bytes in the file
        iii. We don’t read more than BLOCK_SIZE when we fetch a block
    e. Return number bytes successfully read
11. tfs_write()
    a. First, fetch the inode of the target using get_node_by_path()
    b. Then, get the relevant block to write to using offset / BLOCK_SIZE
        i. If offset / BLOCK_SIZE is greater than the amount of direct pointers, we use indirect pointers instead (see section on supporting large files)
    c. Read in the relevant data block from disk to memory
    d. Start writing to the data block keeping 2 things in mind
        i. We don’t write more than requested
        ii. We don’t write more than BLOCK_SIZE
            1. If we need to write more than BLOCK_SIZE write the dirty data block back to disk and fetch the next one
    e. If we added more bytes to the file, update the size
    f. If we wrote at least one byte, update the modification time
    g. Write the updated inode information to disk
    h. Return number of bytes successfully written
12. tfs_unlink()
    a. retrieves the inode of the target
    b. Releases the data blocks of the target
        i. Unsets the bitmap of the direct pointers
        ii. Reads in each data block pointed to by indirect pointers
            1. Iterates down the data block, unsetting the bitmaps from the found data indexes in the data block
    c. Uses dir_remove() to remove the parent from the parent directory

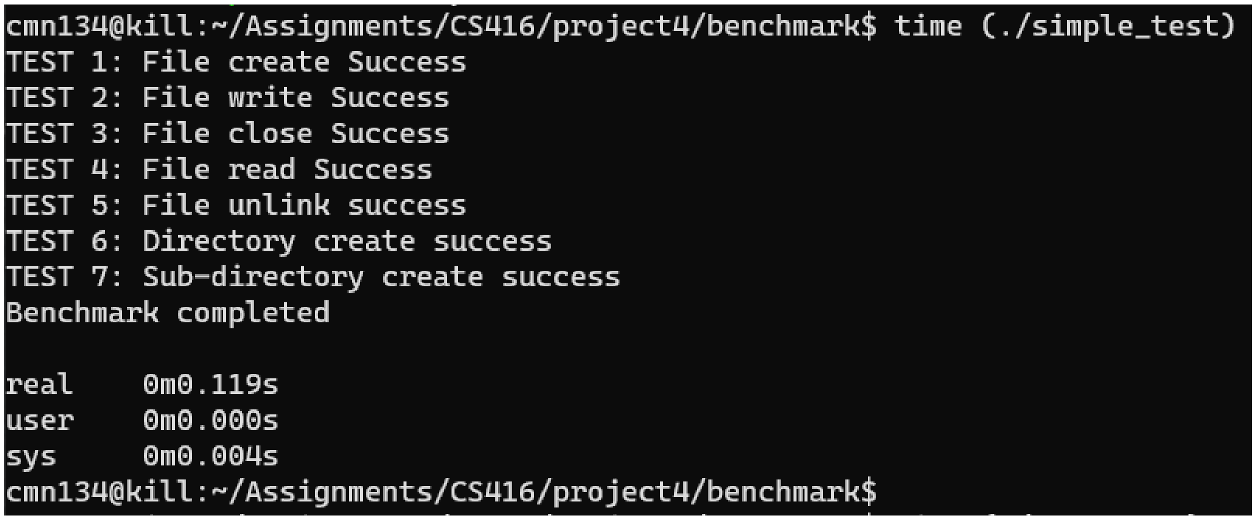
Supporting Large Files:

- To support large files, tfs_read and tfs_write use internal helper functions to determine the location of a data block given an offset
- If the offset / BLOCK_SIZE fits within the range of direct pointers, then we can just directly read the data block index from array
- Otherwise:
  - First, we must read in the data block that the indirect pointer points to
    - If we are writing and the pointer is 0, we allocate a new data block using get_avail_blkno
  - Once we have that data block, we have an array of direct pointers of size (BLOCK_SIZE / sizeof(int))
  - Then we can calculate the offset of the index within the data block
    - Once again, if the index within the data block is 0, we need to allocate another new data block. This one will be used to actually store file data
    - This indirection can be thought of as an int **, as we dereference a pointer to get another pointer.
  - We can read in the data block given by the block number in the direct pointer data block
  - This final data block we read is what we will use to actually read and write data to.
- In short, the indirect pointers point to a block containing direct pointers. Alongside file offsets, we can calculate which pointer in the data block is the correct one to use.
- Visual of indirect pointers being used:


Data block containing direct pointers
struct inode {
…
indirect pointer[x]
…
};
… 50 ...
Hello there! This text is in a very large file.
Data block containing file data

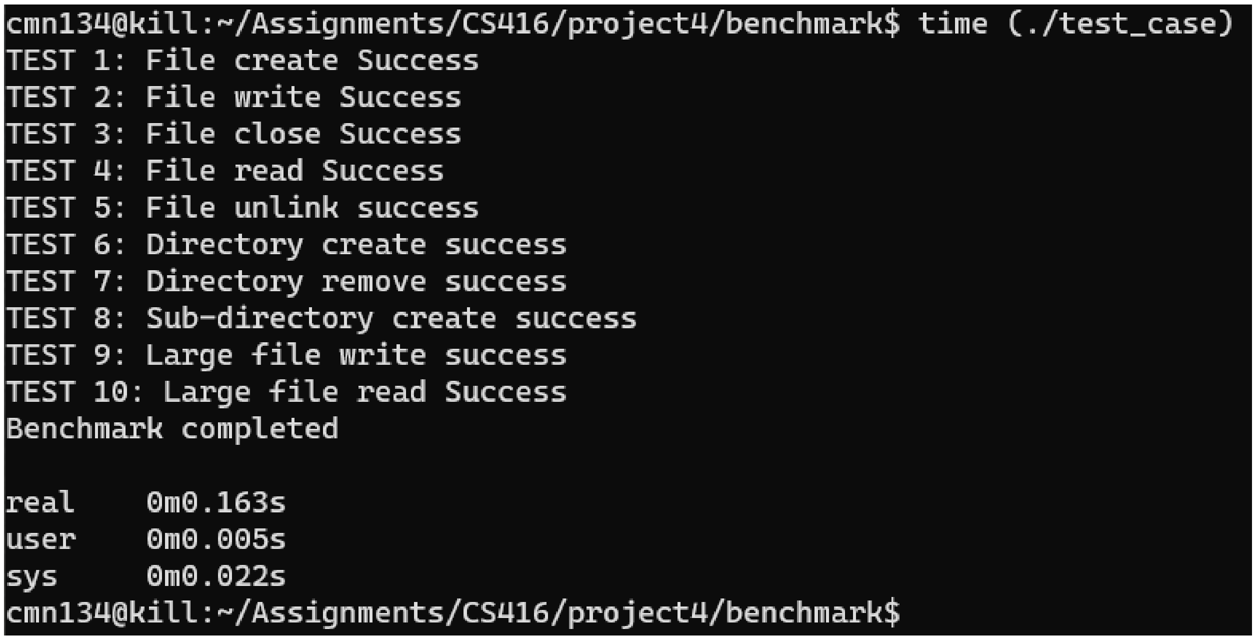
Benchmark Results (Tested on kill.cs.rutgers.edu):

*File System is mounted to /tmp/cmn134/mountdir in both tests



```
cmn134@kill:~/Assignments/CS416/project4/benchmark$ time (./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
Benchmark completed

real    0m0.119s
user    0m0.000s
sys     0m0.004s
cmn134@kill:~/Assignments/CS416/project4/benchmark$
```



```
cmn134@kill:~/Assignments/CS416/project4/benchmark$ time (./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
Benchmark completed

real    0m0.163s
user    0m0.005s
sys     0m0.022s
cmn134@kill:~/Assignments/CS416/project4/benchmark$
```

Difficulties & Issues:

- One of the first issues we faced is getting used to not being able to access data as byte addressable. Having to find the data from the disk, read it into a buffer, and index that buffer was a learning curve that took a day or two to get used to.
- Another issue is error handling. As we were writing functions, we started to think of all the possible cases that could result in an error, what to pass back as a return value, and what to clean up before returning.
- An issue faced all throughout implementation was getting used to all the debug messages that FUSE would print using the “-d” flag. Took a bit of time to actually follow what each message of the debug meant, and how we used it to ensure the filesystem was working properly.