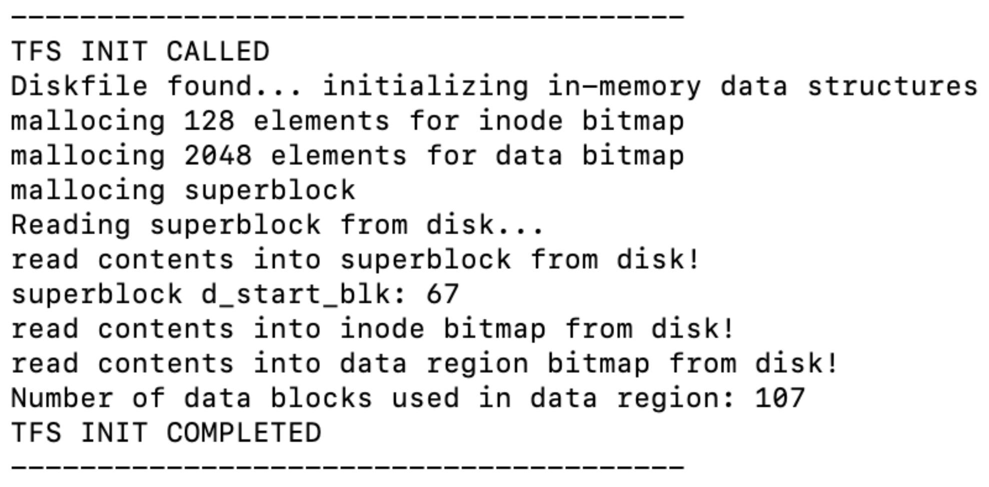
Name: James Wo NETID: jlw373
Name: Nathan Yu NETID: nty4

Benchmark Details

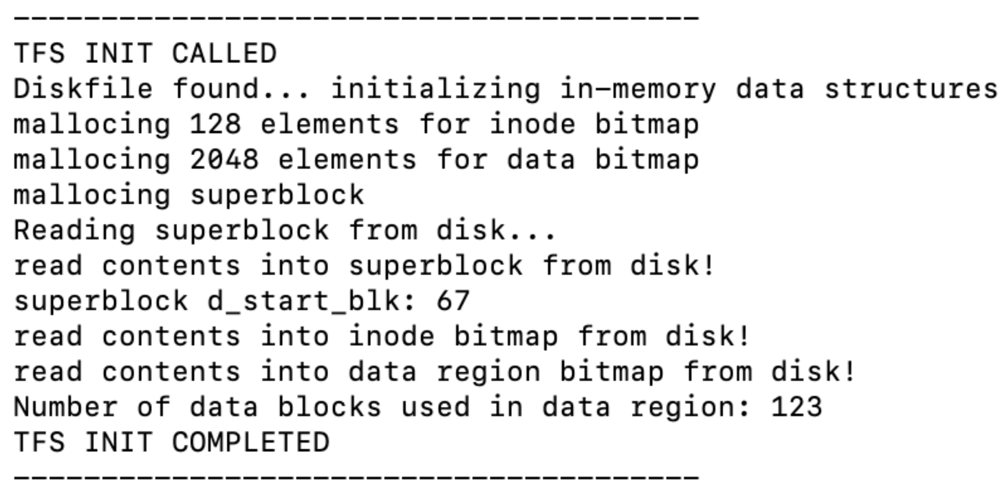
Number of Blocks

The total number of blocks used when running simple_test.c came out to be 107 datablocks (when only looking at the data region- i.e. excluding the superblock, two bitmaps, and inode table). The superblock and two bitmaps take up two blocks respectively- and the inode table's total number of blocks depends on what MAX_INUM is set to.



```
---------------------------------------
TFS INIT CALLED
Diskfile found... initializing in-memory data structures
mallocing 128 elements for inode bitmap
mallocing 2048 elements for data bitmap
mallocing superblock
Reading superblock from disk...
read contents into superblock from disk!
superblock d_start_blk: 67
read contents into inode bitmap from disk!
read contents into data region bitmap from disk!
Number of data blocks used in data region: 107
TFS INIT COMPLETED
---------------------------------------
```

When running test_case.c, it came out to be 123 data blocks.

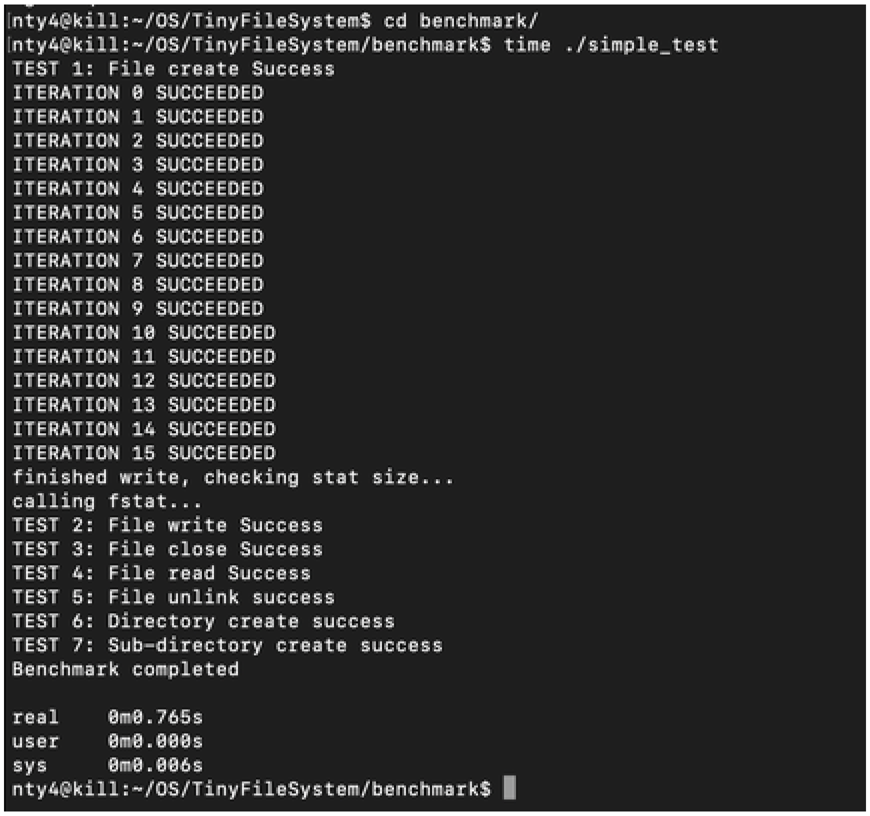


```
---------------------------------------
TFS INIT CALLED
Diskfile found... initializing in-memory data structures
mallocing 128 elements for inode bitmap
mallocing 2048 elements for data bitmap
mallocing superblock
Reading superblock from disk...
read contents into superblock from disk!
superblock d_start_blk: 67
read contents into inode bitmap from disk!
read contents into data region bitmap from disk!
Number of data blocks used in data region: 123
TFS INIT COMPLETED
---------------------------------------
```

<u>It is worth noting that we did not implement the extra credit functionality so our test case 9 fails (but every other test case passes).</u>

Time to Run Benchmark

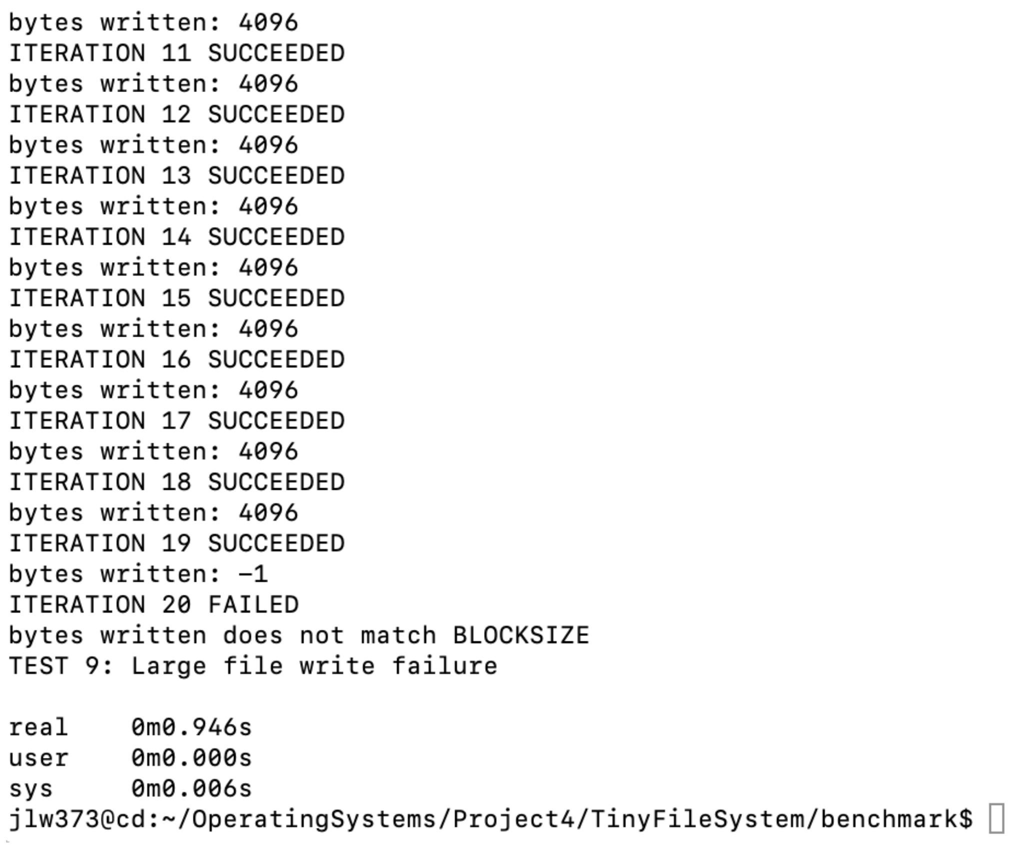
It is worth noting that our print statements most likely inflated our times
For simple_test: 0.765s



```
nty4@kill:~/OS/TinyFileSystem$ cd benchmark/
nty4@kill:~/OS/TinyFileSystem/benchmark$ time ./simple_test
TEST 1: File create Success
ITERATION 0 SUCCEEDED
ITERATION 1 SUCCEEDED
ITERATION 2 SUCCEEDED
ITERATION 3 SUCCEEDED
ITERATION 4 SUCCEEDED
ITERATION 5 SUCCEEDED
ITERATION 6 SUCCEEDED
ITERATION 7 SUCCEEDED
ITERATION 8 SUCCEEDED
ITERATION 9 SUCCEEDED
ITERATION 10 SUCCEEDED
ITERATION 11 SUCCEEDED
ITERATION 12 SUCCEEDED
ITERATION 13 SUCCEEDED
ITERATION 14 SUCCEEDED
ITERATION 15 SUCCEEDED
finished write, checking stat size...
calling fstat...
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	0m0.765s
user	0m0.000s
sys	0m0.006s
nty4@kill:~/OS/TinyFileSystem/benchmark$
```

For test_case: 0.946s



```
bytes written: 4096
ITERATION 11 SUCCEEDED
bytes written: 4096
ITERATION 12 SUCCEEDED
bytes written: 4096
ITERATION 13 SUCCEEDED
bytes written: 4096
ITERATION 14 SUCCEEDED
bytes written: 4096
ITERATION 15 SUCCEEDED
bytes written: 4096
ITERATION 16 SUCCEEDED
bytes written: 4096
ITERATION 17 SUCCEEDED
bytes written: 4096
ITERATION 18 SUCCEEDED
bytes written: 4096
ITERATION 19 SUCCEEDED
bytes written: -1
ITERATION 20 FAILED
bytes written does not match BLOCKSIZE
TEST 9: Large file write failure

real    0m0.946s
user    0m0.000s
sys     0m0.006s
jlw373@cd:~/OperatingSystems/Project4/TinyFileSystem/benchmark$ 
```

Code Implementation

Overall Structure

We decided to hold the superblock and inode/datablock bitmaps in-memory for convenience and less bio_reads.

Helper Functions

- **get_avail_ino()/get_avail_blkno()**
    - Since we have the bitmaps in memory, we do not need to read the bitmaps from disk. This allows us to simply traverse the bitmaps until we reach a valid bit represented by a ‘1’. Then, we mark this bit as used immediately and write the updated bitmap to disk. The number returned is relative to the starting point of each block using d_start_blk or i_start_blk, NOT relative to 0.
- **readi()**
    - We first find where the inode is in the disk by calculating the inode’s blocknum and offset within the block, then read blocknum from disk, and copy the inode into the passed inode argument.

- **writei()**
    - Similar to readi(), read the inode from the disk, but this time memcpy the passed inode argument contents into the disk inode.
- **dir_find()**
    - First, find the inode of the current directory using readi(). Then, iterate over every dirent inside the inode to see if a match is found to prevent duplicate inserts by reading the directptr[] array block by block using bio_read() and matching with strcmp(). We return 0 if found, -1 if not found.
- **dir_add()**
    - We first call dir_find() to check for a duplicate (which may not be needed- unsure about how FUSE handles duplicates). Then we get the current directory's inode and iterate over its directptr array to find an invalid datablock using bio_read(). Once an invalid block is found, we create a new dirent and fill it with the passed info and write the block to disk, returning 0. If there is no invalid dirent in our available datablocks, we find a new datablock with get_avail_blkno() and populate it with invalid dirents, then fill the first dirent with the info and write to disk.
- **dir_remove()**
    - We iterate over the dirents of the inode to find the target inode we wish to remove. If there is a match with strcmp(), we invalidate the dirent and the inode corresponding to the dirent, updating these changes on the inode bitmap, data region bitmap, and both the parent and removed inodes. Finally write the changes to disk and return 0. If a datablock is empty without any valid dirents, the bit is unset. If the target is not found in any dirents of the inode, we return -ENOENT.
- **get_node_by_path()**
    - We decided to implement get node by path recursively. One of our base cases is calling the function with path "/" for root, in which case we readi(0, inode). Otherwise, we truncate the path to remove the leading / and split the path by dir and basename. We first find the inode of the directory using dirname. Then, we iterate through the dir_inode to find a dirent with a matching basename. If a match is found, we have two scenarios. One is if there is no more "/"s left in the truncated path, meaning we are at the end of the path so we can return 0. If the inode type of the match is of type directory, we call get node by path recursively with the new truncated path without the directory.
    - If there is no match, we return -ENOENT.

TFS Functions

- tfs_init()
    - If the disk file has not been made yet, it simply calls tfs_mkfs. Otherwise, we read from the existing diskfile and initialize our in memory data structures (inode bitmap, data region bitmap, and superblock)
    - This function is also where we implement the logic to count the number of blocks used so far (for benchmark measuring in the first part of report)

- tfs_destroy()
    - We free all in memory data structures and call dev_close() to close the diskfile
- tfs_getattr()
    - We call get_node_by_path to get the inode corresponding to the file/directory passed in. If get_node_by_path returns a negative number, then we return ENOENT. Otherwise, we use the inode's information to initialize the stbuf passed in. Specifically, we initialize the mode (S_IFDIR or S_IFREG), number of links, uid, gid, ino, blksize, and st_size
- tfs_readdir()
    - We call get_node_by_path to get the inode corresponding to the directory passed in. After we have this inode, we go through all its dirents for each of its data blocks. For each dirent, if it is valid, we fill in the buffer with the name of this dirent with the given filler function.
- tfs_mkdir()
    - We separate the parent directory path and the base name path. We call get_node_by_path on the parent directory path to get the inode for the parent directory. We call get_avail_ino to get a new inode number for the new directory we are making. We initialize this new inode and call dir_add with the parent inode, new inode, and base name. We also have two more dir_add calls in order to add the . and .. dirent into our new directory. We also call writei to write our new inode into disk
- tfs_rmdir()
    - We call get_node_by_path with the entire path to get the inode of the target directory. We go through each data block of the target directory and unallocate it (unset it in the bitmap)
    - We also unallocate the target directory inode and make it invalid. We call bio_write() and writei to save these changes into disk.
    - We call dir_remove to remove the directory entry corresponding to the directory to be removed from the parent directory
- tfs_create()
    - We make a new inode for the new file. We then call dir_add with the parent directory inode, the new inode, and the name of the new file. We call writei and bio_write to save these changes on disk
- tfs_read()
    - We call get_node_by_path for the inode of the file we want to read. We use the offset and size to determine which indices to index into for the directptr array of our inode. We go through each data block - for each one, we memcpy its contents into the buffer passed in.
- tfs_write()
    - We call get_node_by_path to get the inode of the file we want to write to. Similar to read, we use offset and size to see which range of indices we have to index into in the directptr array. For each of the blocks referred in the directptr array, if they have not been allocated yet, we find a new block with get_avail_blkno. Otherwise, we use bio_read to read in the current data block and store it in a

    block buffer, and we modify this block buffer by memcpying the parameter buffer into this block buffer. After modifying, we bio_write this block back into the disk.
- tfs_unlink()
    - We call get_node_by_path to get the inode of the file we want to delete. We go through each data block of this file and invalide/unallocate it (unset it in the data region bitmap). We also unallocate the inode of the file. We use biowrite and writei to save these changes into disk.
    - We call get_node_by_path again to get the inode of the parent directory. We then call dir remove with the parent inode, and the name of the file we want to delete

Difficulties/Issues Faced (Resolved)

- Originally, we misinterpreted what we could store in-memory, so we started off by saving the entire disk to memory. Obviously this meant we were not using any helper functions we implemented since everything was in memory, so we reverted back to only holding the superblock and bitmaps in memory.
- When we initially ran the benchmark to see how many data blocks are used, we got an unexpectedly high amount (somewhere in the 200s) for simple_test. We realized that this was because we forgot to remove the “.” and “..” dirent entries for a directory when the directory is removed. **Furthermore, this was also due to the fact that we forgot to unallocate a data block (unset it in the data region bitmap) in the case that it no longer holds any valid dirents.** Once we implemented the solution for these two problems, our number of blocks decreased to 107.
- We often ran into errors due to inconsistent null terminating character placements to treat our char* ptrs as strings, but were rectified upon debugging.
- Adding support to multithreading led to deadlocks when trying to minimize the critical section to only locking specific helper functions. The tradeoff we decided was to implement coarse-grained locking on all tfs functions instead.