Paul Kotys, pjk151
Philip Okoh, pco23
Operating Systems CS416
Monday, December 13, 2021

FINAL REPORT FOR PROJECT 4
IMPLEMENTATION OF TYPE FILE SYSTEM (TFS)
TESTED ON cd.cs.rutgers.edu

All parts of the file system were implemented successfully.
The code is well documented, does not have any memory leaks, and perfectly sets the bitmaps.
The code passed all the benchmarks!

RUNNING THE BENCHMARKS

1) The source code of simple_test.c, test_case.c, bitmap_check.c was edited so that TESTDIR "/tmp/pjk151/mountdir" (pathname on my machine). In bitmap_check.c, line 57 had to be edited to have the correct path of the disk file: diskfile = open("/common/home/pjk151/code/DISKFILE", O_RDWR, S_IRUSR | S_IWUSR);

2) The benchmark are run and timed as follows:
    a) We use the linux time command.
    b) We start the file system using ./tfs -s /tmp/pjk151/mountdir. Note that we ARE NOT running with debugging because debugging slows down the system.
    c) We create a fresh file system to run the benchmarks.

3) simple_test.c (Uninitialized file system. Need to create DISKFILE)

```
pjk151@cd:~/code/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    0m0.498s
user    0m0.007s
sys     0m0.000s
pjk151@cd:~/code/benchmark$
```

4) simple_test.c (Under normal conditions, it is MUCH faster)

```
pjk151@cd:~/code/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    0m0.012s
user    0m0.000s
sys     0m0.004s
pjk151@cd:~/code/benchmark$
```

5) test_case.c

```
pjk151@cd:~/code/benchmark$ ./time test_case
-bash: ./time: No such file or directory
pjk151@cd:~/code/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    0m0.092s
user    0m0.000s
sys     0m0.019s
pjk151@cd:~/code/benchmark$
```

6) bitmap_test.c NOTICE abs(icount - icount_after) = 0 and abs(dcount - dcount_after) = 0. Our code does not mess up the blocks and inodes.

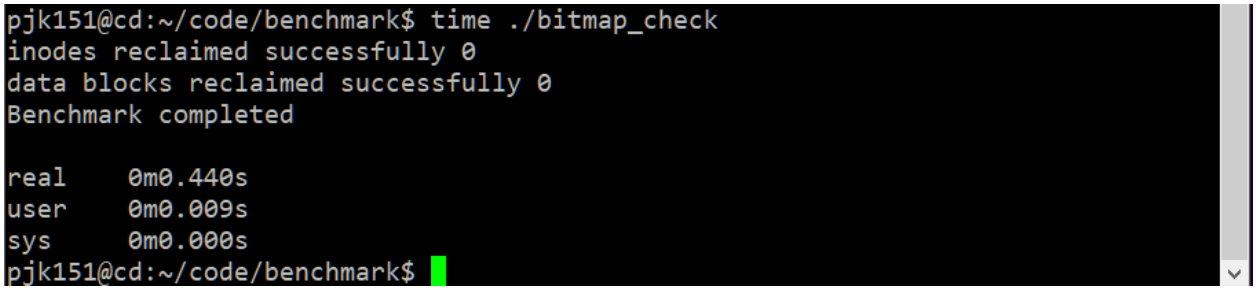

TABLE OF DATA.

TEST	REAL TIME	USER TIME	SYS TIME	PASSED	BLOCK / INODE LOST
simple_test	0.017s	0.008s	0.000s	YES	
simple_test	0.014s	0.005s	0.000s	YES	
simple_test	0.012s	0.000s	0.004s	YES	
test_case	0.092s	0.000s	0.019s	YES	
test_case	0.114s	0.008s	0.024s	YES	
test_case	5.686s	0.005s	0.075s	YES	
bitmap_check	0.484s	0.008s	0.000s	YES	0 / 0
bitmap_check	0.498s	0.003s	0.005s	YES	0 / 0
bitmap_check	0.162s	0.005s	0.015s	YES	0 / 0

For total blocks used, none of the benchmarks return the total blocks so I assumed you mean the number returned from bitmap_check.

COMPILE STEPS
The code should compile using the default makefile. We did not add any extra source file, everything is inside tfs.c.

CODE IMPLEMENTATION:

SuperBlockInit()

1. Initializes the value of the superblock into global space.

get_avail_ino()

1. Scans the inode bitmap until an available inode is found. Updated inode bitmap. Returns this ino.

get_avail_blkno()

1. Scan the block bitmap until an available block is found. Updated block bitmap. Returns the block number

readi()

1. Reads an inode from the disk into the passed in pointer.

write()

1. Reads the inode ino from disk and writes it into the passed in pointer.

dir_find()

1. Reads through the direct pointers of a directory, checking each directory entry to find the matching name. If found, it writes the directory entry to the passed in pointer.

dir_add()

1. Creates a directory entry and tries to add it to an existing allocated block pointed to by a direct pointer. If the block/s is full, it allocates a new block, updates the inode, and writes the directory entry to the block.

dir_remove()

1. Scans the direct pointers of a directory, checking each directory entry to find the matching name. If a matching name is found, we zero out the directory entry.

get_node_by_path()

1. Takes the passed in pathname and a path root, and begins traversing the directory structure. Basically, it splits the pathname by '/' and tries to find each directory/file in order. I implemented this iteratively and used dir_find() to check if an entry exists within a directory.
2. This function can handle cases like "/file/test", "//////file/test", "/file/test///", etc.

tfs_mkfs()

1. Here we initialized the structures such as the Superblock and the InodeTable and bitmaps. We also updated the root inode since this was necessary for mounting procedures in the future. This allowed us to easily extract the needed information to be mounted when calling tfs_getattr function. We also add the following directories to the root inode: "." and "..".

tfs_destroy()

1. I didn’t use any in-memory data structures except the global superblock, so all this function does is close the diskfile.

tfs_getattr()

1. First, we check if the request file/directory exists using get_node_by_path. If it does, we copy the struct vstat from the inode to the passed in pointer.
2. If the file/directory doesn’t exist, we return -ENOENT.
    a. WE NEED TO RETURN THIS ERROR CODE OTHERWISE THE FS WON’T WORK.

tfs_opendir()

1. We simply call get_node_by_path to check if the file exists and return the result.

tfs_readdir()

1. This function reads all the directory entries from a directory’s direct pointers. The only complicated thing here is the filler() function which is supplied by FUSE. Documentation is hard to find, but filler(buffer, directoryEntry->name, NULL, 0); seems to correctly accept the directory entries.
    a. Also, some documentation said to return 0 if filler doesn't return 0.

tfs_mkdir()

1. By finding the inode of the parent and using dir_find to make sure that the directory doesn’t already exist, we can then add the directory with the name using dir_add. Then we create a new inode which we must manually set. We also add “.” and “..” to the new directory. Finally, we write the inode to disk.

tfs_rmdir()

1. First we get the inode of the target directory. Then we clear the data block bitmap of the target directory (iterate through the direct pointers). Then we clear the inode bitmap. Then we get the inode of the parent directory, remove the directory entry, and commit the inode and block bitmaps to disk.

tfs_create()

1. Allocated a new inode for the new file, updates the parent inode by adding a directory entry. The new inode is manually initialized and then written to disk. Note we don’t allocate any block to the file until we actually write data to it.

tfs_open()

1. Basically, we just check if the path is valid or not.

tfs_read()

1. This is a complicated function. We iterate through both the direct pointers and the indirect pointers. Every block is either not read at all, read entirely, or partially read. We used the variables bufferIndex, size, offset, and fileSize to determine where we are in the file, how much data is still left to read, and if we should read the data at the place we are at.

tfs_write()

1. This is the most complicated function in the source code. We first calculate the size of the new file. ((fileSize > (offset + size)) ? fileSize : (offset + size)). Then we iterate through all the direct pointers and get an available block number. A block is either entirely skipped, entirely overwritten, or partially overwritten. We use the four variables bufferIndex, size, offset, and fileSize to determine when to write. Also, when a new block is needed (when we are writing passed the end of the file), we automatically allocate it and update the inode.

tfs_unlink()

1. Unlinking allows us to essentially lazily overwrite data as needed. Therefore this implementation focused heavily on just unsetting the bitmap data of both the inode table and the data block bitmap so when necessary these values can then be overwritten with the newer and necessary data.
2. We also update the inode of the parent directory and call dir_remove().