Christian Mitton

I worked by myself.

Summary

My code was built off of the foundation provided in tfs_mkfs(), since that is where the framework of my file system is created. So in order to illustrate my thought process I'll discuss tfs_mkfs() first and how it creates the file system, and then I'll explain how I envisioned the provided functions would interact with my file system.

tfs_mkfs():

Step 1.) Create super block (sb), then write it to the disk

- The super block is created and all it's fields are filled out. In the super block, the number that corresponds to the i_bitmap_blk, d_bitmap_blk, i_start_blk and d_start_blk refers to the index they will be located in on the disk. After the values are assigned, the superblock is written to the 0th index in the disk file using bio_write().

Step 2.) Create bitmaps, set 0th index in both bitmaps for root, write bitmaps to disk

- The inode and data bitmaps are created, and then their 0th index is set to 1 for the root. After this is done, they are both written to the disk.

Step 3.) Create Inode and dirent for root, write both to disk

- The inode and dirent for the root is created, the fields are filled out and it's ino is set to 0. The inode for the root is written to the disk at it's corresponding index (i_start_blk) and the dirent for root is written at it's corresponding index (d_start_blk)

Step 4.) Fill inode region with available inodes

- I considered the inode region to start at the i_start_blk index and end at the d_start_blk - 1 index. The data region is everyting after. This region of the disk is filled with free inodes. The number of inodes possible was caulated using this formula:

  spaceNeededForInodes = (sizeof(struct inode) * MAX_INUM)
  numOfInodes = spaceNeededForInodes / BLOCK_SIZE

When the above steps are completed and tfs_mkfs() finished, the file "disk" should have the following structure:

DISK						
			Inode Region		Data Region	
0	**1**	**2**	**3**	**...**	**69**	**...**
super block	**i_bitmap_blk**	**d_bitmap_blk**	**i_start_blk**	**...**	**d_start_blk**	**...**
Information about disk	100000000..	1000000000..	root inode	remaining possible inodes	root dirent	remaining possible datablocks

get_avail_ino():

This function traverses the i_bitmap_blk to find the next available index (indicated by a 0). Once it finds this index, it sets that index to 1 in the i_bitmap_blk, writes this updated bitmap to the disk, and then it returns the block number of the next available inode in the disk.

get_avail_blkno():

This function does the exact same as get_avail_ino(), only it traverses the d_bitmap_blk instead, and finds the next available data block on the disk.

readi():

Given an ino number and a pointer to an inode, this function will get the inode on the disk that corresponds to the ino, and then it'll load that inode into the specified pointer.

writei():

This function is similar to readi(), only instead of loading an inode from the disk into the specified pointer, it'll overrite the inode on the disk with the specified inode.

dir_find():

Given a name (f_name), this function will perform all the necessary steps to extract the dirent corresponding to the name on the disk, and load it to the provided dirent pointer.

int dir_add():

Given a name (f_name), inode number (ino) and a current directory(dir_inode) as arguments, this function will add a new dirent entry to the current directory's direct_ptr array. This entry will have the inode number that was passed into the function. It'll then write this new dirent to the data block region on the disk, and the name of this dirent will be assigned to f_name.

Int dir_remove():

This function will search the current directory (dir_inode), and if this directory contains a dirent with the name “fname” it will remove it by setting it’s value to 0 in the current directory’s direct_ptr array. It will also set the validity of the inode for the dirent to remove.

int get_node_by_path():

This is a recursive function that given a path, say: “./folder1/folder2/file.txt“, it will load the inode of file.txt into “inode”. This function assumes that the path will always start from root. There are many steps involved, but to summerize briefly:

1.) The path is split into tokens:

- “./folder1/folder2/file.txt” ===> “.” “folder1” “folder2” “file.txt”

2.) Each token is passed into the recursive function getPathRec() as “currentDirName.”

3.) With each recursive call, if currentDirName is a directory, it searches for the next token in the path

4.) If the last token is found in it’s parent directory it returns an inode to get_node_by_path(), and this inode is read into the original “inode” argument that was passed to get_node_by_path().

5.) If the last token is not found, the file doesn’t exsist and getPathRec() returns NULL. (The rest of the code assumes the path is always valid)