Itamar Levi, Macauley Pinto, Ryan Townsend
NetID's: il166, msp194, rrt51
CS 416: Operating Systems
6 May 2019

Project 4 Report

Get Avail Blkno & Get Avail Ino

These functions read from the disk at the respective blocks based off of the superblock initializations and search until there is an empty block and inode. The function then writes these back to the disk and returns the available inodes and blocks.

Readi & Writei

These functions both start by calculating the offset of the respective inode within the inode table region to get the block number and read that in from the disk. With that, the inode is read in and the parameters are written to it, including the size, validity, and direct/indirect pointers. This is then copied to the global "inodes" structure and is saved into that respective inode. *Readi* then is complete, but *Writei* is then written back to the disk, as the inode has to be written.

Helper Function: SearchiFile

The searchiFile function is the most integral aspect of the project as performs the directory traversals within the file. Initially, the project utilized a tree structure to reduce disk read calls, however, issues were encountered with the memory usage when using the tree structure in the memory space. Thus, the tree functions were subsequently removed except the search function which was instead manipulated to work as though it was performing a tree

traversal. Initially, the pathname passed is split by slashes to find the different levels in which the traversal must be performed. The outer for loop continues until the inner for loop exhausts all possible traversals in the file system hierarchy. Traversals begin from the root directory and file the dirent block of the root, comparing the subdirectories against the name of the split path names. If the mapping is found, the index associated with the split path names is incremented to confirm the subsequent path are within the trees hierarchy and a found flag is set to 0 to indicate a path has been found. The loop then performs a traversal of the dirent's in the directory that was found, attempting the locate the next path in the split path name. If the path is not found the, loop breaks and return 0, a number that no inode can be. Otherwise if the path is found and the index incremented in split path names is greater than length then the inode number is returned as all of the split path names have been traversed and located on the disk.

Helper Function: Split Path Names

The function Split Path Names takes in a path, and returns a two dimensional array containing all of the directory entries present within that path. The 0th index of the 2D array contains the number of directory entries present within the array, in a string form. The remaining strings are the directory entries.

Dir_Find

Dir_Find simply passes the path into our helper function, *SearchiFile*. This will return the inode number of the inode of the basename and checks to see if it exists within the diskfile. If it does, the *Readi* function is called to read this inode number and create an inode for it, which is then copied into the dirent structure.

Dir_Add

The dir add function is based on the count of links of a parent directory to add either a subdirectory or file. Thus, the parent inode is passed to the function and the link count is check against specific parameters related to directory creation. The tfs_mkdir() function will create 3 dirents associated with a directory, but the link count will be two for the current and parent directory. The third dirent is the name of the directory to help with traversal of the disk file. Thus, a link count less than 15 will need to traverse the first direct block at index starting at the third dirent entry. However, if the links count is greater than 15 it is simply divided by 16 to find the parent direct pointers index in which the entry should be added. If the direct pointer is not instantiated a data block will be assigned and instantiate the dirent's as invalid before writing the dirent to the disk. The dirent at the index is then read back in at the direct pointers data block to find the first invalid entry in the dirent block before subsequently marking the dirent as valid and copy over the parameters of the file or directory that is to be added as a dirent entry. Finally, the inode and dirent block is written back to disk before returning from the function.

Dir_Remove

The *DirRemove* function reads the data block of the directory inode, which is found with *SearchiFile*. Once this is found and read, the link is subtracted, the validity is set to invalid, the bitmap is unset, and the inode with the new removed information is updated back to the superblock (for the bitmap), inode, and the data block (directory).

Get_Node_By_Path

The purpose of this function is to traverse the path of the path provided as argument 1 of the function call, in order to find the inode of the terminal point, and load it into the inode*

passed as argument 3. We find this inode by calling the helper function searchiFile(). Once found, we update the inode's data within the disk.

TFS_MKFS & TFS_INIT

TFS_MKFS is called when the diskfile is first created, which is checked by the *TFS_INIT* function. If the diskfile has not been created, *TFS_MKFS* will create the diskfile using *dev_init*, initialize the bitmaps, the superblock, the inodes and their indirect/direct pointers, and the inodes paramers, which are also set through calling *get_avail_ino()*. The root direcotry is also created here, with the first 3 directory entries being '/', '.', and '..'. Thus, these are written to the disk, set to valid, and put into inodes. However, if the diskfile already exists (*TFS_MKFS* has already been called), *TFS_INIT* will simply reinitialize the bitmaps and the superblock, read these in from the disk and reinitialize the inodes and read them in if they already exists in the disk.

TFS_Destroy

This function simply writes the superblock and bitmaps to the disk and frees the inodes, as well as the superblock and bitmaps. The function then closes the diskfile.

TFS_Get-Attr

This function calls *get_node_by_path* to fill the inode of the st_buffer. If this node is found, the inode is updated with the stats, link count, size, and mode.

TFS_Opendir

This function calls *get_node_by_path* to get the inode from the path, which will then access its data blocks and make sure that the inode exists in that path.

TFS_Readdir

We first get the inode of the path of the directory passed into the function. We then malloc a BLOCK_SIZE'd chuck of data, cast as a (dirent *), to hold the directory entries of the directory. We read the directory's entries into that block, and traverse the memory chunk, printing the names of the entries within the parent subdirectory.

TFS_Mkdir

We first determine the inode of the parent directory, which will contain the the directory entry of the subdirectory we are making, using get_node_path. If that inode exists, we call dir_add to add the actual entry within the parent directories entries. Finally we initialize some metadata of the new directory entry, and then write the subdirectory and inode data into the disk.

TFS_Rmdir

We first determine the inode of the of the parent directory. We then call dir_remove on the inode we resolved, which handles the actual removal of the target directory entry.

TFS_Create

TFS_Create calls *get_node_by_path* on the parent directory, which gets the inode of this parent directory. With this, the new directory or file is added with *DirAdd*, and is put into the global inode data structure array. All of the parameters are set, and an inode is made for it with *get_avail_ino*. The direct pointers and indirect pointers are initialized to -1, indicating that they are empty. Later, however, the first direct pointer is set and found with *get_avail_blkno*, and thus the new file/directory is created and added onto the path, which is then written onto the disk as a dirent structure, malloced to hold BLOCK_SIZE amount.

TFS_Open

This function is very similar to *TFS_Opendir*, in that it simply calls *get_node_by_path* to get the inode of the path, and makes sure that it exists and is found.

TFS_Read

The purpose of TFS_Read is to read the data from a file into a buffer. First we locate the inode of the file we are reading from, using the absolute path provided as an argument to the function call. We then compute the location within the file's data blocks where we will begin reading from. We begin reading, tracking the amount of bytes read into the buffer. We continue reading in a loop, conditioned on whether the amount of bytes we have read has exceeded the size of the read request. On each iteration, we determine whether we are reading from a file's direct pointer or indirect pointer. If we are reading from a direct pointer, we load the BLOCK_SIZE chuck of memory pointed to by that direct pointer, and read the data from that block. If we are reading from an indirect pointer, we traverse the indirect block to locate the direct data blocks we will be reading from, load those blocks, and read from them normally. Between iterations of the loop, we update the number of bytes read. Once the number of bytes read has exceeded the size of the read request, we return the number of bytes read.

TFS_Write:

The purpose of TFS_Write is to write data from a buffer into a file. First we locate the inode of the file we are writing to, using the absolute path provided as an argument to the function call. We then compute the location within the file's data blocks where we will begin writing. We begin writing, tracking the amount of bytes written to the file. We continue writing

in a loop, conditioned on whether the amount of bytes we have written has exceeded the size of the write request. On each iteration, we determine whether we are writing to a file's direct pointer or indirect pointer. If we are writing to a direct pointer, we load the BLOCK_SIZE chuck of memory pointed to by that direct pointer, and write the data to that block. If we are writing to an indirect pointer, we first determine whether we need to allocate a new indirect block for that pointer, and subsequent direct data blocks. If not, we simply traverse the indirect block to locate the direct data blocks we will be writing to, we load those blocks, and write to them normally. Once we exceed the size of the write, we update the size of the file, write the inode back to the disk, and then return the number of bytes written.

TFS_Unlink

The *TFS_Unlink* function separates the parent directory using the *dirname* function and runs this as a parameter to *get_node_by_path*, which finds its node and calls *Dir_Remove* to "unlink" the parent directory and the child directory.

Blocks used for each test case?

This can be solved by creating a counter that increments every time the function *get_avail_blkno*, as this will be used whenever a new block needs to be allocated, indicating that there is a free block that can be written to for any operation.

Test_case: 2227 blocks used.

Simple_test: 193 blocks used.

Time to run benchmark?

This can be calculated by creating a start and end time using the *clock()* function, setting start to the beginning of the benchmark and end at the end of the benchmark completion. The calculated times for each test case is as follows:

Test_case: 0.277 seconds

Simple_test: .072 seconds