Connor Aleksandrowicz cja142
Ryan Berardi rtb100

TFS

Tfs_init:

Tfs_init begins by calling dev_open() on diskfile_path. If the return value is -1, we call tfs_mkfs. Otherwise we malloc space for the inode bitmap, data block bitmap, and the superblock, and then read the superblock from disk.

Tfs_destroy:

Tfs_destroy consists of four simple lines. We free the inode bitmap, data block bitmap, and superblock, and then call dev_close().

Tfs_getattr:

Tfs_getattr begins by locking our global mutex lock. It then attempts to get the target inode using the path provided and the function get_node_by_path(). If the return value of this function is -1, then we know that the target does not exist, so we unlock the mutex and return -ENOENT to signify that no target was found from the given path. Otherwise, we continue with the given target retrieved, and begin extracting data from the inode to fill into struct stat* stbuf. Once all required values are set/transferred we unlock the mutex and return 0 to signify completion.

Tfs_opendir:

Tfs_opendir begins by locking our global mutex lock. It then attempts to get the target inode using the path provided and the function get_node_by_path(). If the inode retrieved is not valid, we unlock the mutex and return -1, otherwise 0 is returned.

Tfs_readdir:

Tfs_readdir begins by locking our global mutex lock. It then attempts to get the target inode using the path provided and the function get_node_by_path(). If the inode retrieved is not valid, we unlock the mutex and exit. Otherwise, we browse the directory entries of this target, and add them to the buffer using the function filler(). Upon completion we unlock the mutex and return 0.

Tfs_mkdir:

Tfs_mkdir begins by locking our global mutex lock. It then creates two copies of path ***(This is a workaround to an issue that presented itself. When calling basename and dirname on the same string, the returned strings would be warbled)*** and uses them to get basename and dirname. Get_node_by_path() is then called to get the parent directory, if the returned value is -1 we unlock the mutex and exit. Otherwise, we get the next available inode, and call dir_add() to add a new directory into the parent directory. We then free malloc'd variables, unlock the mutex, and return 0.

Tfs_rmdir:

Tfs_rmdir begins by locking our global mutex lock. It then creates two copies of path ***(Same issue mentioned above)*** and gets basename and dirname from them. We then attempt to get the target directory, if it does not exist we free, unlock, and exit. If it does exist, we run through its direct_ptr's to find out if it is empty or not, if not we return with an error stating that you are attempting to remove a non-empty directory. If the directory is empty, we read the bitmaps from disk, make the necessary modifications, and write them back to disk. We then get the inode of the parent directory and call dir_remove(). Finally, we free, unlock, and return 0

Tfs_create:

Tfs_create begins by locking our global mutex lock. It then creates two copies of path ***(Same issue mentioned above)*** and gets basename and dirname from them. We then attempt to get the parent directory, if it does not exist we unlock and exit. Otherwise, we get the next available inode and call dir_add(). We then write the proper inode information to disk and set the direct_ptr array to 0. We then free, unlock, and return 0.

Tfs_open:

Tfs_open begins by locking our global mutex lock. It then attempts to get the inode via get_node_by_path(). If the inode exists, we unlock and return 0. Otherwise, we unlock and return -1.

Tfs_read:

Tfs_read begins by locking our global mutex lock. It then attempts to get the inode via get_node_by_path(). If the inode does not exist, we unlock and return -1. Otherwise, we create a temp block of memory large enough to hold all of the data in this file's direct_ptr's. We copy the data from disk to this temp variable and then call strncpy() to transfer size number of bytes from the offset within temp to the buffer variable. Once this is done we write the data back to disk, free temp, unlock the mutex, and return size.

Tfs_write:

Tfs_write begins by locking our global mutex lock. It then attempts to get the inode via get_node_by_path(). If the inode does not exist, we unlock and return -1. Otherwise, we create a temp block of memory large enough to hold all of the data in this file's direct_ptr's. We copy the data from disk to this temp variable and then call strncpy() to transfer size number of bytes from the buffer variable to the offset within temp. Once this is done we write the data back to disk, free temp, unlock the mutex, and return size.

Tfs_unlink:

Tfs_unlink begins by locking our global mutex lock. It then reads in both the data block bitmap and the inode bitmap from disk. We get the basename and the dirname from path and then attempt to get the inode of the target file, if it does not exist we unlock, free, and return ENOENT. If it is found, we clear the data block bitmap of the target file, and then clear the inode bitmap. We then get the node of the parent directory, if it does not exist we free, unlock, and return ENOENT. If it does exist, we call dir_remove() to remove the target from the parent, if this does not work we return an error, free, unlock, and exit. Otherwise, we free variables, write bitmaps to disk, unlock the mutex, and return 0.

Benchmark Results

Total Blocks Used:

When running the simple_test benchmark with a BLOCK_SIZE of 4,096 we make:

- 102 calls to get_avail_ino()
- 107 calls to get_avail_blkno()

When running the simple_test benchmark with a BLOCK_SIZE of 8,192 we make:

- 102 calls to get_avail_ino()
- 104 calls to get_avail_blkno()

When running the simple_test benchmark with a BLOCK_SIZE of 16,384 we make:

- 102 calls to get_avail_ino()
- 103 calls to get_avail_blkno()

Time to Run:

When running the simple_test benchmark with a BLOCK_SIZE of 4,096 the timer recorded **2,000 ms**.

When running the simple_test benchmark with a BLOCK_SIZE of 8,192 the timer recorded **4,000 ms**.

When running the simple_test benchmark with a BLOCK_SIZE of 16,384 the timer recorded **4,000 ms**.