Asst4 Documentation
Amali Delauney (adj298) Shyam Patel (spp12)
CS 416
Sudarsun Kannan

Execution

Run our Makefile as you would the benchmark. The only change we made was the addition of a -lm flag in the tfs portion of the Makefile.

Structures

We use all preprovided structures and no extra structures were implemented.

Globals

- **struct superblock* super_blokc**: It's our superblock
- **pde_t* page_dir**: It's our page directory
- **int dpb, ipd**: It is the dirents and inodes per block calculation
- **int num_iblk**: The number of inode blocks
- **int global_temp**: It's a global temp variable for when we need to pass an integer from one function to another without modifying the return value.
- **bitmap_t* inode_bitmap, block_bitmap**: Our bitmaps that are written back to disk once destroy is called and created when we make the file system

Functions

- **void get_avail_ino(), void get_avail_blkno()**: The function will iterate through the global inode_bitmap or block_bitmap until it finds an available slot. If found, it returns the index of the available slot; otherwise, it returns -1;
- **int readi(uint16_t ino, struct inode* inode)**: The function take an inode number, **ino**, can call bio_read() to get the block that the inode is contained within. They then use the calculated number in block to find the location of the inode in the block. In readi, it copies sizeof(struct inode) bytes at that start and into the parameter, **inode** and returns 0 on success or -1 on a failed bio_read
- **int writei(uint16_t ino, struct inode* inode)**: The function is the same as readi(), except, instead of copying the value into **inode**, it takes the provided inode struct and copies all of **inode**'s bytes into the block. The function finally issues a bio_write to write the changes into the "disk". The function returns 0 on success or -1 on a failed bio_read.
- **int dir_find(uint16_t ino, const char *fname, size_t name_len, struct dirent *dirent)**: We raed in the respective directory inode using **ino** and readi(). We then iterate through all of the dirents stored in the respective block numbers in the obtained inode's direct_ptr. This is done with a double for loop where the outer loop goes through

all of direct_ptr and the inner for loop will go through the respective data blocks. If we find a dirent with the same name as **fname**, we set **dirent** to be the curr_dirent at that time and we return 0 for success. On an error or failure to find the entry will return -1.

- **int get_avail_direct_ptr(struct inode* inode)**: Simply gets an unused split in an **inode**'s direct_ptr
- **int dir_add(struct inode dir_inode, uint16_t f_ino, const char *fname, size_t name_len)**: First, we check to see if an entry with the name **fname** exists. If it does, we return -1. Otherwise, we continue. We then obtain an available direct_ptr from dir_inode and data block entry.We obtain the respective block and execute the write to disk in the same manner. However, we update **dir_inode**'s size to represent what is being added. To write the new entry, we use the same logic writei(), except instead of writing to an inode block, we are writing to a data block.
- **int dir_remove(struct inode dir_inode, const char *fname, size_t name_len)**: We simply find the respective **fname**'s dirent's struct. We then take the structs ino value and set its respective position in the inode bitmap to 0 using unset_bitmap. Using global_temp, we don't need to re-find **fname**'s dirent's data block number. We simply set its respective position in the data block bitmap to 0.
- **int get_node_by_path(const char *path, uint16_t ino, struct inode *inode)**: We iterate through the path using the string.h function strtok(). With each token obtained from using **fname** we check to see if the current directory contains a dirent with the same name as that of the current token. This is done by using the function dir_find(). Should an entry be found, we check the valid bit and set the current inode number to be that dirent's inode number. We the repeat the previous steps until we reach the end of the path, hit a dirent with a valid bit of 0 or our placeholder dirent, curr_dirent, ends up being null. WE then obtain the respective inode using readi() and return 0 for success. On an error or invalid entry along the path, we return -1.
- **int tfs_mkfs()**: We invoke dev_init() and initialize our superblock, bitmaps, root directory inode and all necessary structs.
- **int tfs_init(struct fuse_conn_info* conn)**: All initialization steps: opening the disk file and setting up the superblock or creating the filesystem with a tfs_mkfs() all.
- **int tfs_destroy(void* userdata)**: We write our superblock to block 0, inode_bitmap to block 1, block_bitmap to block 2 and then free all necessary structures. We call dev_close() at the end.
- **int tfs_getattr(const char* path, struct stat* stbuf)**: We fill the attributes of the file into the **struct stat** of the inode
- **int tfs_opendir(const char* path, struct fuse_file_info* fi)**: Returns the inode number of the inode from the path. If an inode is not found the function returns -1
- **int tfs_readdir(const char* path, void* buffer, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info* fi)**:
- **int tfs_mkdir(const char* path, mode_t mode)**: We use **dirname()** and **basename()** to seperate the parent and target directory paths. We get the inode of the parent directory and add to it the inode returned by **get_avail_ino()**. Finally we update the inode as a directory entry and write it to the disk.

- **int tfs_rmdir(const char* path)**: **dirname()** and **basename()** are used to separate the path into parent directory path and target directory name. We then clear the bitmap entries of the target directory and finally remove the target directory from the parent directory.
- **int tfs_create(const char* path, mode_t mode, struct fuse_file_info* fi)**: We use **dirname()** and **basename()** to seperate the parent and target paths. We get the inode of the parent directory and add to it the inode returned by **get_avail_ino()**. Finally we update the inode and write it to the disk.
- **int tfs_open(const char* path, struct fuse_file_info* fi)**: Returns the inode number of the inode from the path. If an inode is not found the function returns -1.
- **int tfs_read(const char* path, char* buffer, size_t size, off_t offset, struct fuse_file_info* fi)**: First we retrieve the inode based on the path. Then using the offset and size we iterate through the appropriate number of data blocks and read them. Only the specific data specified is copied from the blocks to the buffer.
- **int tfs_write(const char* path, const char* buffer, size_t size, off_t offset, struct fuse_file_info* fi)**: First we retrieve the inode based on the path. Then using the offset and size we iterate through the appropriate number of data blocks and read them. Only the specific data specified is written from the buffer to the data block. The inode is updated and written back to disk.
- **int tfs_unlink(const char* path)**: **dirname()** and **basename()** are used to separate the path into directory path and file name. We then clear the bitmap entries of the file and finally remove the file from the directory.

Possible Issues with our Design

We are unable to handle very large files because we haven't implemented indirect pointers. There is fragmentation on the "disk" due to how we write and add entries into the data block. This is because when adding a new directory entry in dir_add() we look in the directory's inode's direct_ptr array to see if there are any available spots to place a dirent. As a result, a file that may only take up a few bytes may end up occupying a space that is many times larger than what is needed.

Benchmark

We passed all cases in the benchmarks (see picture below):

```
[spp128@kill benchmark]$ rm -rf /tmp/spp128/mountdir/files
[spp128@kill benchmark]$ ./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
```

```
[spp128@kill benchmark]$ ./test_cases
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
```

Extra Credit

We passed all cases, including tests 9 and 10, of test_cases when we set iters to 100.

```
[spp128@kill benchmark]$ grep ITERS test_cases.c
#define ITERS 100
#define ITERS_LARGE 2048
        for (i = 0; i < ITERS; i++) {
        if (st.st_size != ITERS*BLOCKSIZE) {
        for (i = 0; i < ITERS; i++) {
        for (i = 0; i < ITERS_LARGE; i++) {
        if (st.st_size != ITERS_LARGE*BLOCKSIZE) {
[spp128@kill benchmark]$ ./test_cases
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
```