Project3 report

Chenggang Li 521021910279

May 26, 2023

`disk.c` is relatively a trivial task. Here we'll talk about the implementation of `fs.c`.

I defined the super_block and inode as follows:

```
    struct super_block{
    uint32_t inode_count, data_count, free_inode_count, free_data_count;
    uint32_t root_inode, first_free_inode, first_free_data;
    uint8_t valid_bit;  //set as 'c' when formation is done
    uint8_t map[227];   //inode map and data map
};

struct inode{
    uint16_t type;          //directory or data
    uint16_t deleted_record;//number of removed record in the directory
    uint32_t creation_time;
    uint32_t file_size;
    uint32_t direct_block[4];
    uint32_t single_indirect;
};
```

Noted that we only have single indirect since the file in this system is too small for multiple indirects.

And I encapsulated a bunch of common-used functions:

```
    //find max of 2 unsigned integers
    uint32_t maxu(uint32_t a, uint32_t b){return a<b?b:a;}

    //send request of writing data to block ($block_num) to the disk simulator
    void send_write(uint32_t block_num, void* data);

    //send request of reading data of block ($block_num) to the disk simulator
    void send_read(uint32_t block_num, void* data);
```

```
9
10      //read the block where inode_id = ($inode) located and store it in void*
        buffer
11      void read_inode(uint32_t inode, void* buffer);
12
13      //write buffer to the block where inode_id = ($inode) located
14      void write_inode(uint32_t inode, void* buffer);
15
16      //read data from inode ($cur)
17      void read_file(struct inode* cur, void* buffer);
18
19      //write data to inode ($cur) and update the file size
20      void write_file(struct inode* cur, void* buffer, uint32_t cur_size);
21
22      //allocate new block and update the map. op can be directory or data
23      uint32_t find_new_block(int op);
24
25      //deallocate a block and update the map. op can be directory or data
26      void free_block(uint32_t num, int op);
27
28      //add record (id, type, name) to directory inode ($cur)
29      void add_to_directory(struct inode* cur, uint32_t id, uint8_t type, char*
        name);
30
31      //read . directory of inode ($cur) and stroe in ($name)
32      void read_cur_dir(struct inode* cur, char* name);
33
34      //find record with file_name = ($name) in directory inode ($cur)
35      uint32_t find_file(struct inode* cur, uint8_t type, char* name);
36
37      //delete record with file_name = ($name) in directory inode ($cur). Here
        we need to free the file inode and tag the record.
38      uint32_t delete_file(struct inode* cur, uint8_t type, char* name);
39
40      //write .. and . to directory file
41      void dir_init(void* cur, uint32_t father_id, char* father_name, uint32_t
        cur_id, char* cur_name);
42
43      //free the file inode
```

```
44    void free_inode(struct inode* tmp);
45
46    //null dir can't be removed. o.w. leak may occurs
47    uint32_t is_null_dir(struct inode* cur);
48
49    //clean fragment of the directory
50    uint32_t rearrange(uint32_t size);
```

record format of my directory is `inode_id_4, next_4, type_1, length_1, file_name_length`, which is `10 + length` byte long in total. first two record of one directory is always ..(parent directory) and .(current directory).

1 format

f command will format the disk. At first the super block will be initialized, and then the root directory will be initialized. We always choose block 0 to store the super block, and block 1 to store the root inode. Then root directory will be initialized by calling `dir_init()`. After formation, "Done!" will be sent to client.

2 mkdir and mk

first we'll call `find_new_block()` to allocate an inode. Then we'll add the inode to current directory by calling `add_to_directory()`. Then the inode is initialized and written to disk. If we are making a directory, we'll call `dir_init()` to write ..(current directory) and .(new directory) to the disk.

3 ls

we'll traversal current directory, read every record for the file name and type. Then we use insertion sort to guarantee the names are in lexicography order. Here I just used the same format of ls result in step2.

4 w, i, d, cat

for these 4 commands, we call `read_file()` to read the file data from inode, do the operations and (if command isn't cat)call `write_file()` to write back the file. `write_file()` is extremely complicated because I put allocate and deallocate block operations

in it. By comparing current file size and original file size, we can decide whether to allocate or deallocate. I don't want to dive into the details.

5 rm and rmdir

we call `delete_file()`. In the function we traversal current directory to search for the target file. If we are removing a directory, `is_null_dir()` will be called to check whether the subdirectory is null(if we have records in the subdirectory, block leak will occur). Then we free the inode and tag the delete file with inode id = 0xffffffff(max_uint). When there are too many deleted records, `rearrange()` will be called.

6 cd

Here we needs to read both absolute path and relative path. We make a copy of our current directory inode and do operations on it. If the first character is '/', we assert that it's absolute path and change the copy inode to root. Then we iteratively read a directory name between '/' and change copy inode to that directory. If we reach a valid directory, we'll write it back to our current directory.

7 summary

The real file system is far more complicated then I described above, and you can read my source code for more details(I have to admit that it's not that readable).