CS3104 File System Practical Report

170011474

November 2019

1 Overview

This practical requires students to implement a file system based on fuser operations in C. For the main practical, uses are able to execute some basic commands in terminal required by the Practical Specification Sheet. For the extension part, the file system adopts multi-level indexing scheme, renaming, hard link.

2 Design & Implementation (Including Extension)

2.1 Fundamental Data Structures

There are 4 fundamental data structures defined in my project.

1. `fcb` for File Control Block (FCB). It contains basic information, including permissions, ownership, size, timestamps, size, unique `uuid_t`s for its data blocks, *and one* ***`uuid_t`*** *for indirect block.*
2. `dir_entry` for Directory Entry. It contains a name of a link and its corresponding unique `uuid_t`-type `fcb` index.
3. `dir_data` to represent a data block of a directory. It contains `cur_len` (the current number of used entries) and an array of `dir_entry`s.
4. `file_data` to represent a data block of a file. It contains `size` (the current number of used bytes of this block) and `data`.

2.2 Database Functions

Database Functions include `read_from_db`, `write_to_db`, and `delete_from_db`. In this way, complicated read, write, and delete functions in fuser are hidden, and necessary status checking is provided. Therefore, code is more simpler and easy to follow.

2.3 Extension 1: Multi-level Indexing Scheme

2.3.1 Structure

Here my multi-level indexing scheme is the basis of following demonstration for other design. My file system adopts some direct blocks and one indirect block for each `fcb`. Further indirect blocks can be added, but here I made only 1 for simplicity. So in `fcb`, we have

```
......
mode_t mode; // file mode
off_t size; // file size
uuid_t data_blocks[NUMBER_DIRECT_BLOCKS]; // ids of direct blocks
uuid_t indirect_data_block;           // id of indirect data block
time_t atime; // time of last access
......
```

2.3.2 Iterating over Data Blocks in One FCB

To iterate over each data block in the current `fcb`, an `iterator` can be made with the information of the current `fcb`.

```
typedef struct
{
    fcb *fcb;
    int level;
    int index;
    int create; // for 1, it will create a new empty block
} fcb_iterator;
```

When to end iterating? So the iterator will iterate over all used data blocks. When it meets some data block is not used (i.e. the id of that data block is `zero_uuid`), if `create` is 1, then a new empty data block is created for this fcb. If `create` is 0, then the iterator should return NULL to indicate hitting the end. Besides, when it meets the last data block in the indirect data block. it also terminates.

When to jump to the indirect level? The index of the iterator will be checked. If the current index is out of valid range of length of direct blocks, then the `level` of the iterator will be incremented by 1 and the `index` is back to 0 again.

2.4 FCB-level and Directory-level Functions

There are some functions to create a new fcb (`make_new_fcb()`), get root fcb (`get_root_fcb()`), make an empty data block (`make_empty_data_block()`), etc. Here I want to point out some important and relatively complicated functions.

How to fetch a fcb given a path? `get_fcb_by_path()` This function uses a while loop to recursively retrieve the next-level `fcb` from the current-level `fcb`(directory). It starts from the root fcb, iterating over each directory entry in the current directory to find the matched link name. Then the current directory will be updated with the matched result, next iterating process will execute until the path name is all consumed.

```
while (cur_name != NULL)
{
    int rc = get_fcb_from_dir(&cur_dir, cur_name, cur_fcb_id, &cur_dir);
    if (rc != 0)
        return rc;

    cur_name = strtok_r(NULL, "/", &path_remaining); // keep extract the next name
}
```

How to add a fcb to a directory? `add_fcb_to_dir()` Given the fcb of the directory, we can iterate over the data blocks of the directory. **For each data block, we iterate over each directory entry until we find an empty directory entry.** Then we can fill it with fcb id and the corresponding link name. The updated directory should be write to database in the end. *Notice that each directory data block (`dir_data`) has an attribute `cur_len` to record the number of current used entries. If the `cur_len` is equal to `MAX_DIRECTORY_ENTRIES_PER_BLOCK`, it means the data block has been used up. So the iterator will continue to the next data block to find an empty entry.* In this way, search time will be significantly reduced.

How to delete a fcb from a directory? `delete_file_or_dir_from_dir()` Given the fcb of the directory, we can iterate over its data block to find the matched fcb id we want to delete. If the fcb is for a directory, then only empty directory can be deleted. If the fcb is for a file, then we iterate over the file fcb's data

blocks to delete them from database one by one. *Notice that if* ***delete_data*** *== 0, indicating currently we are trying to delete a* ***hard link****, so only the reference count will be decremented by 1.* When the reference count is 0, then its data blocks will be deleted from database. This hard link function is one of extensions.

```
else // delete a hard link
{
    fcb true_fcb;
    read_from_db(fcb_id, &true_fcb, sizeof(fcb));
    true_fcb.count -= 1;
    write_to_db(fcb_id, &true_fcb, sizeof(fcb));
}
```

2.5 Truncate Functions

To truncate a file, we firstly check if the new file will be larger or smaller. If the file becomes larger, then we only need to find an offset and then use `myfs_write()` to write some zeros to it. If the file becomes smaller, then the situation is more complicated.

1. Keep the common part and calculate a start block to edit. This can be done by:

```
 remaining_size = new_size % BLOCK_SIZE;
// from this block we are gonna edit, after this block everything will be deleted
int block_offset = (int)new_size / BLOCK_SIZE;
```

2. Iterate over each data block of the current file and record the index. If the index equals to the `block_offset`, then only first `remaining_size` bytes can be kept and following bytes will be replaced by 0 by a `memset()`.

3. Delete superfluous data blocks following `block_offset`. If the block `block_offset` is a direct block, then deletion will execute on subsequent direct blocks by `truncate_direct_level()` and on data blocks referred by indirect data block by `truncate_indirect_level()`.

4. If the block `block_offset` is a data block referred by indirect block, then only `truncate_indirect_level()` is needed since following blocks are all referred by the indirect block.

```
if (index == block_offset)
{
    memset(&cur_data_block.data[remaining_size], 0, BLOCK_SIZE - remaining_size);
    cur_data_block.size = remaining_size;
    write_to_db(cur_data_block_id, &cur_data_block, sizeof(file_data));
}
else if (index == block_offset + 1)
{
    if (iterator.level == 0)
        truncate_direct_level(&cur_fcb, index); // delete following direct blocks
    // deleting following data blocks referred by indirect block
    truncate_indirect_level(&cur_fcb, index - NUMBER_DIRECT_BLOCKS);
}
```

5. Finally, update the file fcb and its size in database.

2.6 Extension 2: Link Functions

The implementation of hard link has 3 steps:

1. given the `from` and `to` strings, check the destination file and the source file are not directories and the destination file's parent directory exists;

2. it does not create a new fcb, but it adds a directory entry to the parent directory of the destination file;

3. update the true fcb's reference count and update modified fcbs in databases.

To unlink a hard link, we only remove the directory entry of it and decrease the reference count. Only if when the reference count becomes 0, its data blocks can be deleted from database. The implementation of it has been illustrated above (see How to delete a fcb from a directory?)

3 Testing

3.1 Main Practical Testing

1. mkdir (create a directory)

```
(base) ThinkPad:~/hy30/mnt$ mkdir cs
(base) ThinkPad:~/hy30/mnt$ ls
cs
```

2. rmdir (remove a directory) and rm (romove a file)// This test will show what happens when removing a non-empty directory

```
(base) ThinkPad:~/hy30/mnt$ ls // original content
a.txt  cs
(base) ThinkPad:~/hy30/mnt$ cd cs
(base) ThinkPad:~/hy30/mnt/cs$ touch b.txt
(base) ThinkPad:~/hy30/mnt/cs$ cd ..
(base) ThinkPad:~/hy30/mnt$ rmdir cs // warning will be given
rmdir: failed to remove 'cs': Directory not empty
(base) ThinkPad:~/hy30/mnt$ cd cs
(base) ThinkPad:~/hy30/mnt/cs$ rm b.txt
(base) ThinkPad:~/hy30/mnt/cs$ cd ..
(base) ThinkPad:~/hy30/mnt$ rmdir cs/
(base) ThinkPad:~/hy30/mnt$ ls // current content
a.txt
```

3. touch (create a file)

```
touch a.txt
(base) ThinkPad:~/hy30/mnt$ ls
a.txt  cs
```

4. echo (write to a file)

```
(base) ThinkPad:~/hy30/mnt$ echo " Our group will operate on a sprint cycle of two
    weeks in ideally the following way: Every two weeks we will hold a Scrum
    meeting, during which we take one or more user stories from the product
    backlog and break it down into individual tasks. Each task is estimated to
    take roughly the same amount of time to complete" >> a.txt
(base) ThinkPad:~/hy30/mnt$ ll
total 4
drw-rw-r-- 1 hy30 hy30  0 November 14 14:45 ./
drwxr-xr-x 3 hy30 hy30 4096 November 5 09:47 ../
```

```
-rw-r--r-- 1 hy30 hy30 318 November 14 15:01 a.txt // see here
drwxr-xr-x 1 hy30 hy30  0 November 14 14:58 cs/
```

5. cat (read a file)

```
(base) ThinkPad:~/hy30/mnt$ cat a.txt
Our group will operate on a sprint cycle of two weeks in ideally the following
    way: Every two weeks we will hold a Scrum meeting, during which we take one or
    more user stories from the product backlog and break it down into individual
    tasks. Each task is estimated to take roughly the same amount of time to
    complete
```

6. stat

```
(base) ThinkPad:~/hy30/mnt$ stat a.txt
  File: a.txt
  Size: 318        Blocks: 0        IO Block: 4096 regular file
Device: 4ah/74d Inode: 5         Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1000/ hy30) Gid: ( 1000/ hy30)
Access: 2019-11-14 15:01:43.000000000 +0000
Modify: 2019-11-14 15:01:43.000000000 +0000
Change: 2019-11-14 15:01:43.000000000 +0000
 Birth: -
```

7. chmod

```
(base) ThinkPad:~/hy30/mnt$ ls -la a.txt
-rw-r--r-- 1 hy30 hy30 318 November 14 15:01 a.txt
(base) ThinkPad:~/hy30/mnt$ chmod 777 a.txt
(base) ThinkPad:~/hy30/mnt$ ls -la a.txt
-rwxrwxrwx 1 hy30 hy30 318 November 14 15:01 a.txt
```

8. utime (change modification time)

```
(base) ThinkPad:~/hy30/mnt$ ls -la a.txt
-rw-r--r-- 1 hy30 hy30 318 November 14 15:32 a.txt
(base) ThinkPad:~/hy30/mnt$ touch -d "2 hours ago" a.txt
(base) ThinkPad:~/hy30/mnt$ ls -la a.txt
-rw-r--r-- 1 hy30 hy30 318 November 14 13:32 a.txt
```

9. truncate

```
(base) ThinkPad:~/hy30/mnt$ cat a.txt
 Our group will operate on a sprint cycle of two weeks in ideally the following
     way: Every two weeks we will hold a Scrum meeting, during which we take one
     or more user stories from the product backlog and break it down into
     individual tasks. Each task is estimated to take roughly the same amount of
     time to complete
(base) ThinkPad:~/hy30/mnt$ truncate -s 50 a.txt
(base) ThinkPad:~/hy30/mnt$ cat a.txt
 Our group will operate on a sprint cycle of two w(base) ThinkPad:~/hy30/mnt$ ls
     -la a.txt
-rwxrwxrwx 1 hy30 hy30 50 November 14 15:01 a.txt
(base) ThinkPad:~/hy30/mnt$ truncate -s 500 a.txt
(base) ThinkPad:~/hy30/mnt$ ls -la a.txt
```

```
-rwxrwxrwx 1 hy30 hy30 500 November 14 15:01 a.txt
(base) ThinkPad:~/hy30/mnt$ cat a.txt
 Our group will operate on a sprint cycle of two w
```

When the new size is not valid: (to test, I set the `BLOCK_SIZE` as 300)

```
(base) ThinkPad:~/hy30/mnt$ truncate -s 50000 a.txt
truncate: failed to truncate 'a.txt' at 50000 bytes: File too large
```

3.2 Extension Testing: Hard Link

1. The linked file's parent does not exist

```
(base) ThinkPad:~/hy30/mnt$ ls
a.txt
(base) ThinkPad:~/hy30/mnt$ link a.txt ./cs/b.txt
link: cannot create link './cs/b.txt' to 'a.txt': No such file or directory
```

2. The linked file already exists

```
(base) ThinkPad:~/hy30/mnt$ mkdir cs
(base) ThinkPad:~/hy30/mnt$ cd cs
(base) ThinkPad:~/hy30/mnt/cs$ touch b.txt
(base) ThinkPad:~/hy30/mnt/cs$ cd ..
(base) ThinkPad:~/hy30/mnt$ link a.txt ./cs/b.txt
link: cannot create link './cs/b.txt' to 'a.txt': File exists
```

3. **After removing b.txt**

```
(base) ThinkPad:~/hy30/mnt$ ls
a.txt  cs
(base) ThinkPad:~/hy30/mnt$ link a.txt ./cs/b.txt
(base) ThinkPad:~/hy30/mnt$ cat a.txt
 Our group will operate on a sprint cycle of two weeks in ideally the following
     way: Every two weeks we will hold a Scrum meeting, during which we take one
     or more user stories from the product backlog and break it down into
     individual tasks. Each task is estimated to take roughly the same amount of
     time to complete
(base) ThinkPad:~/hy30/mnt$ cd cs
(base) ThinkPad:~/hy30/mnt/cs$ cat b.txt
 Our group will operate on a sprint cycle of two weeks in ideally the following
     way: Every two weeks we will hold a Scrum meeting, during which we take one
     or more user stories from the product backlog and break it down into
     individual tasks. Each task is estimated to take roughly the same amount of
     time to complete
(base) ThinkPad:~/hy30/mnt/cs$ stat b.txt
  File: b.txt
  Size: 318       Blocks: 0        IO Block: 4096 regular file
Device: 4ah/74d Inode: 11        Links: 2
Access: (0644/-rw-r--r--) Uid: ( 1000/ hy30) Gid: ( 1000/ hy30)
Access: 2019-11-14 15:32:14.000000000 +0000
Modify: 2019-11-14 13:32:36.000000000 +0000
Change: 2019-11-14 15:32:14.000000000 +0000
 Birth: -
(base) ThinkPad:~/hy30/mnt/cs$ truncate -s 30 b.txt
(base) ThinkPad:~/hy30/mnt/cs$ cd ..
```

```
(base) ThinkPad:~/hy30/mnt$ cat a.txt
 Our group will operate on a s
```

We can see that after linking b.txt to a.txt, they are connected. Truncating b.txt will affect a.txt successfully. And **the Links becomes 2** after using `stat b.txt`

4. rm (unlink a hard link)

```
ThinkPad:~/hy30/mnt$ ls
a.txt  cs
(base) ThinkPad:~/hy30/mnt$ rm a.txt
(base) ThinkPad:~/hy30/mnt$ cd cs
(base) ThinkPad:~/hy30/mnt/cs$ cat b.txt
 Our group will operate on a s(base) ThinkPad:~/hy30/mnt/cs$ stat b.txt
  File: b.txt
  Size: 30         Blocks: 0        IO Block: 4096 regular file
Device: 4ah/74d Inode: 11        Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1000/ hy30) Gid: ( 1000/ hy30)
Access: 2019-11-14 15:32:14.000000000 +0000
Modify: 2019-11-14 13:32:36.000000000 +0000
Change: 2019-11-14 15:32:14.000000000 +0000
 Birth: -
```

From above, we can see after removing a.txt, b.txt still retain the content. And the **Links becomes 1**.

4 Evaluation & Conclusion

1. The full functionality of the standard specification satisfies requirements.
   (a) Test cases are passed.
   (b) Code is clear and simple to follow, with sufficient comments and explanation.
   (c) There is no duplicate method.
2. Extension perform well according to requirements from the specification sheet.
   (a) Extensions have passed self-developed testings.

In conclusion, during these 4 weeks, I have strengthened C programming and acquired understandings about the file system inside operating system by building my own file system. I also have deeper insights into all errors and faults may occur.