CS202 Review Session 7 Based on previous notes of cs202 review session 7.

Yuxia Zhan - Fall 2025

Contents

Introduction

In this week and last week's classes, you learned a lot about file system. And in the lab5, you have the chance to dig into a real file system's code and complete some core functions inside this system. What's even better is, you can run your implementation of `ls` in lab2 in this file system.

Let's recall what happens when you run `ls` in lab2. What outputted are the files and subdirectories inside a directory. And if you recall more details, what's going on under the hood is that you use `opendir()`, `readdir()` syscalls to get the content inside a directory.

But according to what you learned about the disk, there's no such thing called "xxx directory" or "xxx file" inside a physical disk, there are only blocks of data. This "directory path and file name" thing is just some nice abstraction provided by the kernel's file system, such that user (you as a human) don't have to remember where data is physically located (e.g. on which platter or track of sector).

So, a file system is a way to provide an intuitive interface ("path and filename") to user to access data. The main job of a file system is to maintain the mapping between the "path and filename" and the underlying disk's blocks.

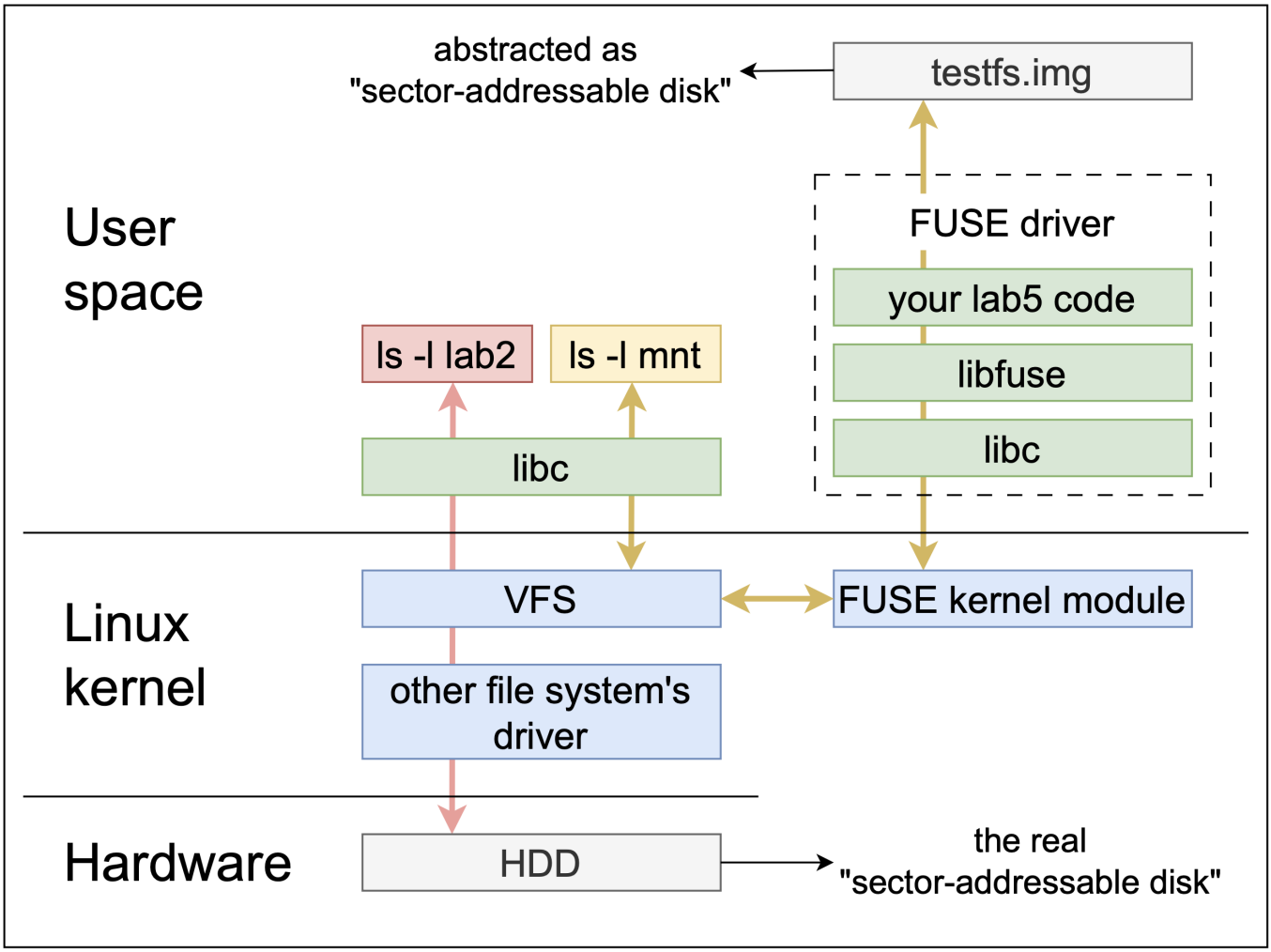
[Q&A] Recall the running example about disk in last Wed's class (https://cs.nyu.edu/~mwalfish/classes/25fa/lectures/scribble20.pdf), what's the most important abstraction (or say data structure) to maintain such mapping? Inode.

- [Q&A] When you run `ls lab2`, what happens inside the kernel?
    - handle the syscall `opendir(path)`
    - file system searches for this path, starting from root directory's inode (inode 2)
    - follow the inode, and goes to the corresponding subdirectory
    - repeat until it arrives at the lab2's inode
- [Q&A] When you run `cat lab2/main.c`, what happens inside the kernel?
    - same as above, additionally:
    - goes to main.c's inode, and follow the pointer to the corresponding data blocks

And because of the abstraction that file system makes, it doesn't matter whether the underlying storage is a real disk, or a file, or the data in memory; it's even possible that this storage is not in your local machine, as long as the "path and filename" abstraction remains unchanged, user can barely feel the difference.

In Linux, such a file system consists of two components, one is the VFS layer, the other is the different type of file system's driver. As you can see in the figure, user can use exactly the same code (like the `ls` you wrote in lab2) to read data from two storage of totally different file system types. One is the real disk, the other is just a normal file.

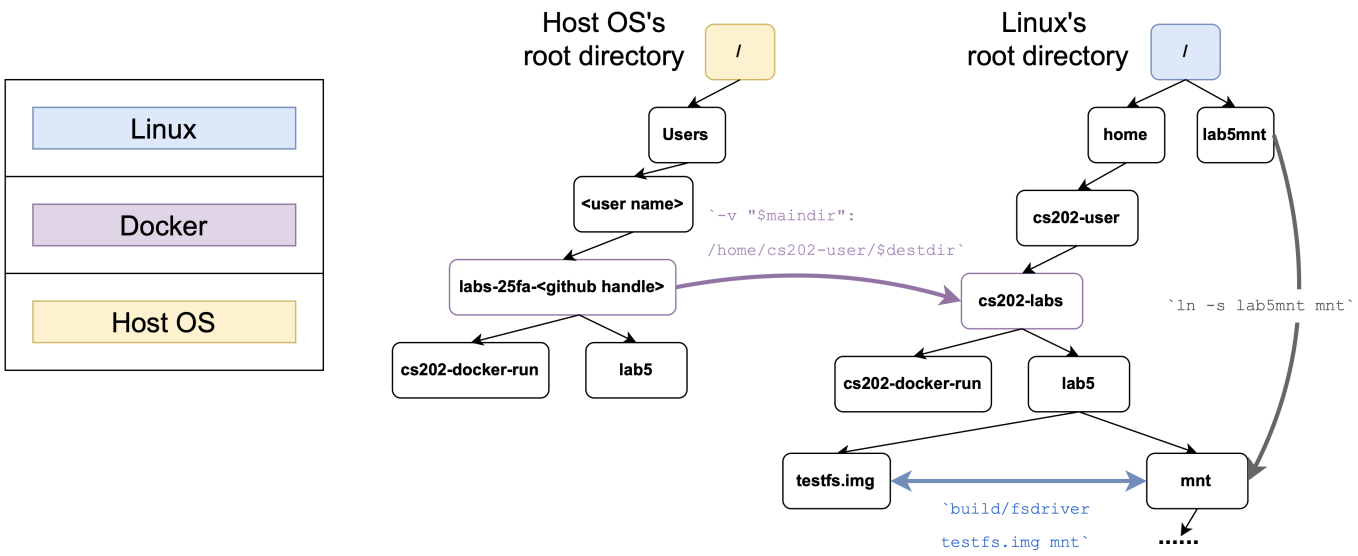
Figure 1 - s/w stack of file system.



In this lab, you will use VFS and the FUSE kernel module as is, and only have to modify the user-space fs driver part.

Before we go into more details about lab5, it's worth understanding how `mount` works in general, cause it will help you know what the test scripts are doing, and hence can better help you to debug.

Figure 2 - mount.



Actually, you've been using `mount` even if you might not realize it. Every time you run the container, a directory from your host OS is mounted to the container's Linux filesystem It's done by `-v "$maindir":/home/cs202-user/$destdir` (ref: https://docs.docker.com/reference/cli/docker/container/run/) inside `cs202-run-docker` script. This is why you can modify the same file both inside and outside the container (i.e., in your host OS).

In this lab, you will use mount more explicitly. For example, `build/fsdriver testfs.img mnt` mounts `testfs.img` at the `mnt` directory. You will complete fsdriver so that after testfs.img is mounted at mnt:

- when a user create a new file in mnt, your fsdriver can find some valid blocks for this file, and maintain the mapping accordingly
- when a user reads from mnt (by passing in the path), your fsdriver can locate the corresponding block (based on the mapping) in `testfs.img` and returns the data
- when a user delete a file in mnt, your fsdriver can do the clean up work and maintain the mapping accordingly

Lab5's file system design

[See https://cs.nyu.edu/~mwalfish/classes/25fa/labs/lab5.html] On-Disk File System Structure

For convenience, the file system design of this lab makes some simplifications compared to the file system you learned in class.

- There is only 1 region in which both inode and data block reside. Usually, they are divided into 2 separate regions: inode region and data block region. But we don't distinguish them in this lab.
- Each inode is allocated its own disk block. Usually, if the inode region and data block region are separated, indoes are packed in a single disk block.
    - [Q&A] What's the disadvantage of "allocating the whole disk block to an inode"?
    - Inefficient in space. Exercise 8.
- The file system read in chunk of 4KB block size.

```
// fs_types.h
// The size of a block in the file system.
#define BLKSIZE         4096

// How many bits are present in a block.
#define BLKBITSIZE      (BLKSIZE * 8)
```

- Superblock is block 0. It holds metadata about the FS and pointer to the root directory.

```
// disk_map.c
super = (struct superblock *)diskmap; // = diskmap(0)
bitmap = diskblock2memaddr(1);
```

- Bitmap is an array of bits. Each bit at index i indicates if block i is allocated or not. 1 indicates it's free and 0 indicates it's used.
    - How many blocks are there in total? `super->s_nblocks`.
    - fsformat will format the underlying image, such that superblock contains the correct metadata about this image.

```
// fs_types.h
struct superblock {
    uint32_t    s_magic; // Magic number: FS_MAGIC.
    uint32_t    s_nblocks; // Total number of blocks on disk.
    uint32_t    s_root; // Inum of the root directory inode.
} __attribute__((packed));
```

- Each inode contains:
    - pointers to 10 direct data block.
    - pointer to 1 indirect block.
        - How many direct blocks can be "addressed" by a indirect block? 4KB / 32bit = 4KB / 4B = 1K = 1,024.
        - 32bit is the size of "block number"
    - pointer to 1 double indirect block.
        - How many direct blocks can be "addressed" by a double indirect block? 1K * 1K = 1M = 1,048,576.

```
// fs_types.h
// The number of blocks which are addressable from the direct
// block pointers, the indirect block, and the double-indirect
// block.
#define N_DIRECT        10
#define N_INDIRECT      (BLKSIZE / 4)
#define N_DOUBLE        ((BLKSIZE / 4) * N_INDIRECT)
```

Lab5's file system implementation

As we mentioned earlier, VFS will dispatch the file operations to FUSE kernel module, and that will be handled accordingly by the fsdriver. You can find these file operations' handlers in `fsdriver.c`.

```
// fsdriver.c
struct fuse_operations fs_oper = {
    // ...
    .open       = fs_open,
    .read       = fs_read,
    .write      = fs_write,
    // ...
};
```

Roughly speaking, `fsdriver.c` implements all these handlers (`fs_*`), and it will will invoke the following helper functions to do the real job.

- `map_disk_image`, `diskblock2memaddr`, and `flush_block` in `disk_map.c`
- `alloc_diskblock` (Ex2), `diskblock_is_free`, `free_diskblock` in `bitmap.c`
- `inode_*` in `inode.c`
    - in which, `inode_block_walk` and `inode_get_block` (Ex3) are the real working horses.
    - `inode_truncate_blocks` (Ex4)
    - `inode_link` and `inode_unlink` (Ex5)
- `dir_*` and `walk_path` in in `dir.c`

Take `fs_open` as an example:

- When user call open(path), this syscall is dispatched to FUSE module, and later handled by `fs_open`, hence it has an additional input `struct fuse_file_info fi`.
- Inside `fs_open`, it calls `inode_open` with
    - `const char *path`: the pointer to the path string
    - `struct inode **pino`: the double-pointer to the inode structure. [Q&A] what does this double-pointer mean? how would you draw picture of the memory to indicate the pointer-to-pointer?

    ```
    alias | addr |      data
    ---------------------------
    pino  |  A   |        B
          |  B   |        C
          |  C   | struct inode
    ```

- Inside `inode_open`, it calls `walk_path`, with the `path` and `pino`.
    - `walk_path` can parse the path, and find the corresponding file's inode structure.
- Eventually, `fs_open` sets the corresponding inode filed in `fi`

There is a caveat though, how can fsdriver access the underlying testfs.img? Considering that in the "real disk" case, there will be a device driver, who is responsible for communicating with the disk, but apparently it's not the case with "testfs.img", which itself is nothing more than a file.

We can find answer in `map_disk_image.c`, it uses `mmap` to reserve a portion of the running process's virtual address space to provide read and write access to a file *as if that file were an array in memory*. This abstraction does (almost) the same thing as a device driver: abstract the underlying disk storage as an array of blocks.

To fill the gap between "array of blocks" (the real device driver) and "array of bytes" (mmap), fsdriver utilizes a rather straight forward way: use `diskblock2memaddr` to simulate it.

```
fd = open(imgname, O_RDWR)
diskmap = mmap(NULL, diskstat.st_size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, 0)
super = (struct superblock *)diskmap;
bitmap = diskblock2memaddr(1);
```

Lab5 excises details

- Ex2: `alloc_diskblock` in `bitmap.c`
    - `bitmap` is the bookkeeping for the blocks (just like the `pageinfo[]` is the bookkeeping for the physical pages)
    - you can learn how to interpret and manipulate bitmap's value in `free_diskblock`
- Ex3: `inode_block_walk` and `inode_get_block` in `inode.c`
    - Purpose of `inode_block_walk(struct inode *ino, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)`
        - Locate the global disk block number (`ppdiskbno`) corresponding to a local block number (`filebno`) of an inode (`ino`)
        - What would you do if `filebno` is within the range of direct blocks?
        - What if `filebno` exceeds range of direct blocks?
        - What if `filebno` exceeds range of indirect blocks?
        - It might help to visualize the direct, indirect, and double-indirect block structures in the memory, and how you would travel to a specific block using these data structures.
        - Allocate new block when needed (`bool alloc`), and remember to clear it before any use.
    - Purpose of `inode_get_block(struct inode *ino, uint32_t filebno, char **blk)`
        - Obtain the memory address of the `filebno`'th block for a given inode (`ino`).
        - Once you complete `inode_block_walk`, the implementation of this function will be very succinct.

- Ex4: `inode_truncate_blocks` in `inode.c`.
    - used to resize a file
- Ex5: `inode_link` and `inode_unlink` in `inode.c`
    - when user calls `ln target-file link-name`, the `inode_link` will be invoked.
    - Again, recall the running example about disk in last Wed's class (https://cs.nyu.edu/~mwalfish/classes/25fa/lectures/scribble20.pdf).
    - What does a hard link look like in disk? filename -> same inode.
    - What about a soft link? Allocate a new inode, which contains the target-file's path.

Debug for lab5

[Remember to run `./chmod-walk` and `sudo make grade`]

1. Setup and pre-test (before mount): run `test/testbasic.bash` to
    - prepare the `testfs.img`
        - `generate_test_msg`: write simple content into `build/msg`
        - `make_fsimg build/msg`: pack build/msg to a well-formed `testfs.img` [Q&A] what do you think need to be added to make `testfs.img`?

```
make_fsimg() {
    build/fsformat testfs.img 2048 $@ || fail "couldn't make test
image"
}
```

        - you can actually learn how is fs designed in fsformat.c (by initializing the disk); because what you will have to write in fsdriver.c is to maintain such a design when there are more modifications on the fs.
    - run `build/fsdriver testfs.img mnt --test-ops`
        - `fsdriver.c/fs_test` to test basic file operations, but don't mount
        - this can help you identify many bugs (if any) in you code before going to more complicated tests under `test` directory
2. Test (after mount)
    - since after mounting the `testfs.img` into a certain directory, user can directly read / write this directory, so to debug it, you will need two split panels in tmux, one for fsdriver, one for invoking the file operations on the directory.
    - (tmux panel 1)
        - `gdb build/fsdriver`
        - and then `run -d testfs.img mnt` (to mount testfs.img to mnt)
    - (tmux panel 2) run scripts or commands that try to access mnt directory to read / write the underlying `testfs.img`, e.g.,
        - your `ls` from lab2!
        - `test*.bash` under the `test` directory