Operating System

Assignment #3 Myfs

Software Dep.

201421016 Seung gon Jeon

1. Intro

In this assignment, I had understood about the implementation of the file system. The base of the file system is the “fuse” module, which is an interface to export a file system to the Kernel. There are lots of callback functions to know if the user want to make his/her own file system, and in this assignment most of the things to do was understanding about these functions.

2. FUSE and the libfuse functions

As I mentioned at the intro, FUSE is an interface for userspace programs to export a file system to the Linux Kernel[[1]](#footnote-2). The Fuse project works as two different components, one for the Kernel other for the user space library and the two communicate to each other talking which reference the kernel and the userspace need to see.

The first thing I was confused was this: I wondered what developing a file system means. In the first term I thought that developing a file system means that collecting the files together, linking the folder with the proper files, etc all by myself. But actually, it was not. Making a file system just meant that making a program as I done before. This program seems to be called as a standalone application that links with the “libfuse”, the library that saves functions that mounts/unmounts the file system, request/response with the kernel. These functions are the callback functions that the programmer needs to implement to make a new file system. From now on, I will talk about these callback functions and its roles that do in the system which is important during the implementation of the development.[[2]](#footnote-3)

-1. .access(const char\* path, int mask)

If the file system application starts executing, the first thing that had happened that I saw was this function. This is a callback function for calling the “access” function which checks the user’s permissions for a certain file(parameter “path”). It returns 0 for accessing success, if not -1. After access() returns, the callback function return 0 if access returned 0, if not, returns a errno

-2. .getattr(const char\* path, struct stat\* stbuf)

This is a callback function for getting the file’s status, which is saved as a structure named “stat”. The “stat” structure have information as number of hard links, owner’s id file system block size, latest time for doing something, etc. The stat saving task is done by the functions stat(), fstat() or lstat() which is called inside the .getattr callback function.

This function always happens when the user do something with the file system. For example, if the user does ‘ls’ command in the file system’s root directory, the first thing the file system does is this “.getattr” function and gets the status of the files which is mounted in the root. This makes the system to deceive that it seems to work in the mounted directory.

The usage is this. Gets a string that represents a certain file’s path, and save the file’s status in the buffer that stbuf points.

-3. .open(const char \*path, struct fuse\_file\_info \*fi)

This function opens a file that is represented as “path”. It uses fi->flags as the flags when opening the file. Because only the opening task must be done, the creation flags will be filtered by the kernel and then sended to this function, and access mode (like read only) will be used during the opening.

It seems to use fi→fh as a file handle, and use this for doing things with the opened file like read or write.

-4. .read(const char \*path, char \*buf, size\_t size, off\_t offset, struct fuse\_file\_info \*fi)

This is a function that reads a certain file. It happens when the user do any reading operations inside the mounted userspace. Of course, if the user wants to read a file the file must be opened, so the .open() function comes first when reading.

We can see that there is a string buffer where to save that the kernel passed and the size of this buffer. There is also a fuse\_file\_info struct pointer type parameter fi which is the same variable type that open used, also for the access mode. There is another parameter that we can see which is called offset. Offset is a parameter that notice the userspace where the reading must starts inside the file on the memory. In my test cases, this was passed as 0, the start of the file, and this means there is not a big reason that the kernel sends this information. But if there is a reason sending this information, I guess that it is used for the recovery of the improperly closed files, or there are some other reasons that I don’t know. But if the programmer wants to use this offset parameter inside read, he or she can use the pread() function which also has a parameter for offset, differently between the read() function. The read function always start at the file’s start point, but the pread function starts at the startpoint + offset so it will be more comfortable when using the offset parameter that is not 0.

After the file reading is done, the next thing that I saw was the .getattr function. It seems to get the status again of the read file to upload the stat structure’s time related fields such as st\_atim or st\_mtim. And after doing this, the .release callback function appeared.

-5. .write(const char \*path, const char \*buf, size\_t size, off\_t offset, struct fuse\_file\_info \*fi)

This is a function that writes in a certain file. Same as the read, it happens whenever the user do any writing operations inside the mounted userspace. Also the .open function must comes first.

The parameters are the same as .read function and everything is same with it except the operation is writing. As similar with .read(), there 2 functions for writing, write() and pwrite(). Pwrite has a parameter for offset as the same as pread(). But the usage with offset was slightly more practical with the offset parameter. If I write 4bytes in a new file, the first offset was 0, and the next time when I write again on the same file the offset was 4, and when I write a new 10 bytes, the next time I write the offset was 14, and so on. This means that when the user writes something in the system, offset points the start point where the writing will begin, not the start point of the file.

The difference with the read function I saw was the .getattr function. In read, after reading finished the getattr function appeared and uploaded the stat structure. In this time, before wirting starts the .getxattr function appeared and the .getattr did not appeared after the writing. Getxattr is a function that get an extended attribute like extended stat.

-6. .release(const char \*path, struct fuse\_file\_info \*fi)

This is a function that is called when there are no more references to an open file. All file descriptors are closed and all memory mappings are unmapped.

These are the most important operations that is used when only the issue is making or reading a file. If the programmer want to make things work with the folder too, he or she must implement the related callback functions like mkdir or readdir.

-7. .mkdir(const char \*path, mode\_t mode)

This function is used when the user makes a new directory inside the file system. When the user use linux command “mkdir”, it first checks the target directory if it is already existing. If it does not, it gets an error from the .getattr callback function and calls this .mkdir callback funtion to make it. If the target already exists, then the .getattr will have no error and the directory will not be build.

Inside this .mkdir callback function, the programmer can use mkdir() system call. The usage of this function is this:

int mkdir(const char \*pathname, mode\_t mode)

The mode parameter set from the kernel by the .mkdir callback function can be used directly for the new directory represented as the pathname.

-8. .readdir(const char \*path, void \*buf, fuse\_fill\_dir\_t filler,

off\_t offset, struct fuse\_file\_info \*fi, enum fuse\_readdir\_flags flags)

This is a function that reads the directory files and open it. There is 2 choices this callback function can do. It can ignore the offset parameter and passes 0 to the filler parameter which makes the the whole directory read in just one readdir() operation, or if it does not ignore the offset, the .readdir() callback operation will trak the offsets and read the directory until the buf is full or the directory ends.

In the implentation that was provided, was build as the latter method. First it opened a directory then read it using a loop until the directory end, and then inside the loop filled the filler parameter with the stat structure. When filling the filler with the buf cases an error(which means buf is full) then break out the loop. After the loop closing the directory folled and thats the end of this callback function.

After this .readdir is done, the files in the directory that the user has just read will be looked up, so the .getattr will happen for these files. After that, the .releasedir() callback happen and releases this directory just as .release() do.

There are also callback functions that create and destroy node or directories, changing modes of the files or directories, changing owners of the files or directories, etc. There are total 40 types of callback functions in the libfuse. Each of them has its specific purpose and needed to be implemented by the programmer if it is needed.

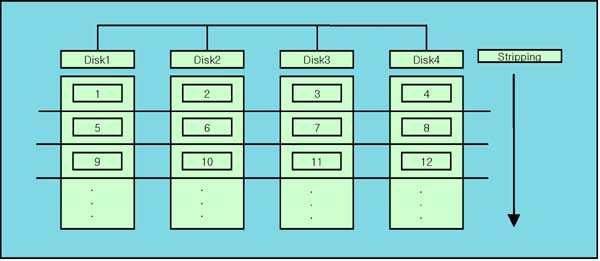
3. Raid[[3]](#footnote-4)

Raid stands for redundant array of independent/inexpensive disks, which means method of making data saved in multiple storages to make the system’s performance more superior. Because using multiple storages together, some people call Raid as a disk array. There are lots of ways to save the data in this disk array, each of them is called as level. In this assignment, the Raid 1 level had been provide and I must implement the Raid 0 using this Raid 1. Now I will talk about what kinds of levels that Raid has, and about those concepts.[[4]](#footnote-5)

-1. Raid 0

Raid 0 is a method that concentrates on the issue for enhancing the speed for read write operations. If our computer can do tasks in parallel, when the user write something in the file then the system can divide it into 2 parts when 1 block of the disk is filed and then save each parts in different storages at the same time. This approach will make the amount of spending time into half, so the speed will double. If it the system can work parallel 3 different task and has 3 different storage, the speed will triple, and so on. Figure 1 illustrates about this approach. People call this approach as ‘striping’.

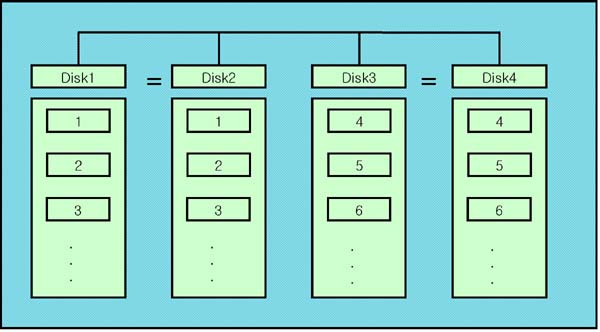
Because Raid 0 is using more than 1 different storage to save 1file in once, the speed suddenly soar. But because of the dividing part of this process, if only one of the disk fails for saving the parted data, the whole data will become unusable and all the other parts of that data stored in other disks becomes a garbage. This is a disadvantage of using Raid 0.

  
Figure 1: Raid 0

-2. Raid 1

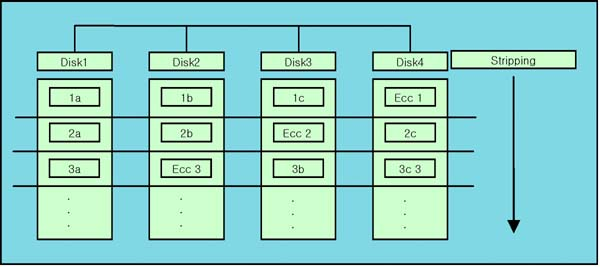
Raid 1 is a method that concentrates on the issue for safety. In Raid 0, because of the data splitting the whole data was in dangerous, but now in Raid 1 the data is not split and just save the same data in 2 or more different storages at the same time (if the computer supports mulitprocessing). This does not make the writing or reading speed higher but instead makes the user more relieved in saving the data. So the summary is this: even if one disk didn’t saved the file correctly there is another disk that know what that data was. Figure 2 shows about Raid 1. People call this approach ‘mirroring’.

But because using 2 different storage to save the same file, the system uses the storage twice as bigger than the original non-Raid using approach. So the space will be two times more expensive.

  
Figure 2: Raid 1

-3. Raid 5

Raid 5 is an advanced approach that consider all speed, safety and space. In this method, there is one disk for safety, and the other disks are for striping, just like in Raid 0. During the striping, the disk for safety saves some information that is calculates to check if the information in the striping disks are wrong or not. For example, lets suppose there is 2 disks A B for striping and disk C for safety. If the computer splits a file with 2bits, each disk will save 1bit. Then the system will do some operation for calculating a check bit like sum. For example if A needs to save 1 and B needs to save 0, the checkbit will be 1(1+0) and this will be saved in C. But when something happened gone wrong during the saving task and in result A saved 1 and B saved 1, then the system can no that something went wrong during the saving by checking the checkbit because 1+1 is 2 but it’s different with the checkbit, and can modify the wrong bit correctly. So the main concept of Raid 5 is saving some information that the system can know the information in the disk is wrong and correct it by itself. This approach is shown in Figure 3.

  
Figure 3: Raid 5

There are more levels in the Raid but I will stop here. It is quite interesting and seems to be a good research for many researchers.

4. Raid 0 implementation

I have written some code to change the Raid 1 code into Raid 0. The codes that I’ve changed are the callback functions that I explained on the second section of this report, “FUSE and the libfuse function”. In the Raid 1 codes, lots of these callback functions accesses to one of the storage that the user mounted in randomly, but for the Raid 0, the system must not only access one of the two, instead it needs to work on both disks. Do the first thing I had done is modifying the access and getattr function to check on both storages. It will still work when it is just left as the random way, but because I think this matches more with Raid 0 conceptly, I’d changed it.

The next thing that I made is the writing callback function. In the Raid 1 .writing code, the writing was done in both disks doing all the writings from the offset until the offset + buffer size. In Raid 0, the buffer needs to be written as the first block in the first disk, and the remained bytes must be written in the next disk. So I first checked the offset to find which disk the writing needs to be happened, and then calculated the writing start point and how much long the writings will be. After these is done, then the writing is happened.

The reading is similar to writing. In the Raid 1, the reading function just opened 1 of the file in any disk randomly, but now in Raid 0, the reading must be done in both disk if the reading overflows the block size in the first block(disk). So the first thing in reading is opening the first disk and read, checks if the reading is enough or not, if not opens the second disk and reads again inside the same buffer that the first read happened, but with different start point.

Some other functions are still in implementation. The system must do the same thing as writing when user makes a directory or reading a directory. It will be done until the deadline.

5. Feedbacks

This assignment was quite interesting. Until now, I did heard that we can make our own file system in Ubuntu, but I didn’t know how and finally my question was solved. But know I am curious how the kernel links the files with a folder so that represents the file is inside a certain folder. In first I was confused the meaning of making a file system was this. I think this will be similar with making the libfuse project. It will be hard, but challenging. If I have time, I want to compare the Linux kernel’s file system and the libfuse project, to know the real implementation of the cores.

The work amount was not that much big as I expected. Because the only thing that I needed to do was modifying some functions, making the codes was not a big problem. But instead, understanding how the system works was a problem. As I said before, I was confused about the assignment and took me some time to understand the fact that I was confused. The next time when I build a file system, I think I will not be confused again.

6. Reference

<https://github.com/libfuse/libfuse>

<https://libfuse.github.io/doxygen/structfuse__operations.html>

<https://ko.wikipedia.org/wiki/RAID>

<http://www.bodnara.co.kr/bbs/article.html?num=28067>

1. https://github.com/libfuse/libfuse [↑](#footnote-ref-2)
2. https://libfuse.github.io/doxygen/structfuse\_\_operations.html [↑](#footnote-ref-3)
3. https://ko.wikipedia.org/wiki/RAID [↑](#footnote-ref-4)
4. http://www.bodnara.co.kr/bbs/article.html?num=28067 [↑](#footnote-ref-5)