**Final project of operating system**

**Description**

FUSE (https://github.com/libfuse/libfuse) is a Linux kernel extension that allows for a user space program to provide the implementations for the various file-related syscalls. We will be using FUSE to create our own file system, managed a file that represent our disk device. Through FUSE and our implementation, it will be possible to interact with our newly created file system using standard UNIX/Linux programs in a transparent way.

From a user interface perspective, our file system will be a two level directory system, with the following restrictions/simplifications:

1. The root directory “\” will not only contain other subdirectories, but also regular files

2. The subdirectories will only contain regular files, and no subdirectories of their own

3. All files will be full access (i.e., chmod 0666), with permissions to be mainly ignored

4. Many file attributes such as creation and modification times will not be accurately

stored

5. Files cannot be truncated

6. Directories are treated as files

From an implementation perspective, the file system will keep data on “disk” via a linked allocation strategy, outlined below.

The operating system should be Ubuntu 18.04, FUSE version should be 3.3. You can download them at website.

**Installation of FUSE**

The first thing you should do is install ubuntu 18.04 in a vm, and then you should install following software packet:

Sudo apt-get install git gcc vim lrzsz openssh-server meson pkg-config make unity-tweak-tool libtool m4 autoconf pkg-config

Then download and install FUSE.

Tar -xvf libfuse-fuse-3.3.0.tar

mkdir build; cd build

meson ..

ninja

sudo ninja install

Edit file /etc/ld.so.conf, add a line at file tail :

include /usr/local/lib/x86\_64-linux-gnu

Execute:

ldconfig -v

**First FUSE Example**

Let us now walk through one of the examples. Enter the following:

cd libfuse-fuse-3.3.0 /examples

mkdir testmount

ls -al testmount

./hello testmount

ls -al testmount

You should see 3 entries: . , .. and hello. We just created this directory, and thus it was empty, so where did hello come from? Obviously the hello application we just ran could have created it, but what it actually did was lie to the operating system when the OS asked for the contents of that directory. So let’s see what happens when we try to display the contents of the file.

cat testmount/hello

You should get the familiar hello world quotation. If we cat a file that doesn’t really exist, how do we get meaningful output? The answer comes from the fact that the hello application also gets notified of the attempt to read and open the fictional file “hello” and thus can return the data as if it was really there.

Examine the contents of hello.c in your favorite text editor, and look at the implementations of readdir and read to see that it is just returning hard coded data back to the system.

The final thing we always need to do is to unmount the file system we just used when we are done or need to make changes to the program. Do so by:

fusermount -u testmount

**FUSE High-level Description**

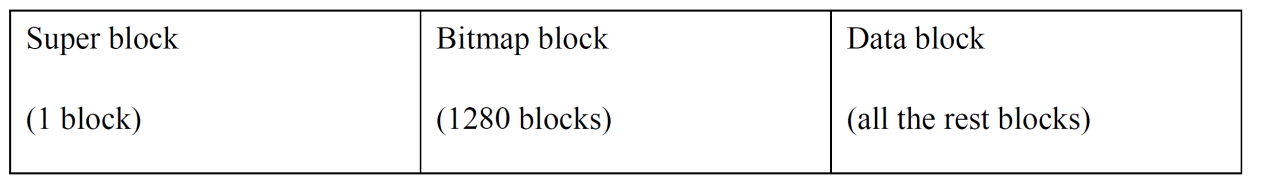
The hello application we ran in the above example is a particular FUSE file system provided as a sample to demonstrate a few of the main ideas behind FUSE. The first thing we did was to create an empty directory to serve as a mount point. A mount point is a location in the UNIX hierarchical file system where a new device or file system is located. As an analogy, in Windows, “My Computer” is the mount point for your hard disks and CD-ROMs, and if you insert a USB drive or MP3 player, it will show up there as well. In UNIX, we can have mount points at any location in the tree.

Running the hello application and passing it the location of where we want the new file system mounted initiates FUSE and tells the kernel that any file operations that occur under our now mounted directory will be handled via FUSE and the hello application. When we are done using this file system, we simply tell the OS that it no longer is mounted by issuing the above fusermount -u command. At that point the OS goes back to managing that directory by itself.

**What You Need to Do**

Your job is to create the u\_fs file system as a FUSE application that provides the interface described in the first section.

The u\_fs file system should be implemented using an image file, managed by the real file system in the directory that contains the u\_fs application. The layout of file system will be follow. We will consider the disk to have 512 bytes per blocks.所以平均每 block 大小为 512 bytes，然后整个5MB文件共 10240 块



**Super block**

Super block must be the first block of the file system. It descripts the whole file system. Info containing in super block should be:

struct sb {

long fs\_size; //size of file system, in blocks（以块为单位）

long first\_blk; //first block of root directory（根目录的起始块）

long bitmap; //size of bitmap, in blocks（以块为单位）

}

**Directories**

Directories should be also treated as a file. Each directory contains a list of u\_fs\_directory\_entry structures. There is no limit on how many directories we can have.

struct u\_fs\_file\_directory {

char fname[MAX\_FILENAME + 1]; //filename (plus space for nul)

char fext[MAX\_EXTENSION + 1]; //extension (plus space for nul)

size\_t fsize; //file size

long nStartBlock; //where the first block is on disk

int flag; //indicate type of file. 0:for unused; 1:for file; 2:for directory

}

Each directory entry will contain an 8-character maximum directory name, and then have a list of files that are in the directory.

Each file entry in the directory has a filename in 8.3 format. We also need to record the total size of the file, and the location of the file’s first block on disk.

所谓8.3格式短文件名规范，就是型如 PROGRA~1（目录）或者

啦啦啦~1.exe（文件）这样的名称——

“8”是指文件名或目录名的主体部分小于等于8个字节；

“3”是指文件名的扩展名部分小于等于3个字节。

另外还有一点，就是8.3文件名的有效字符不包括空格等特殊字符。

**Files**

Files will be stored in a virtual disk that is implemented as a single, pre-sized file called .disk with 512 byte blocks of the format:

struct u\_fs\_disk\_block {

size\_t size; // how many bytes are being used in this block（文件使用了这个块里面的多少Bytes）

long nNextBlock; //The next disk block, if needed. This is the next pointer in the linked allocation

list（该文件太大了，一块装不下，所以要有下一块的地址）long的大小为4Byte

char data[MAX\_DATA\_IN\_BLOCK];// And all the rest of the space in the block can be used for

actual data storage.

};

**Disk Management**

In order to manage the free or empty space, you will need to create some bitmap blocks on the disk that record whether a given block has been previously allocated （分配）or not. The total number of bitmap blocks should depends on the size of file system. You can do this however you like.

To create a 5MB disk image, execute the following:

dd bs=1K count=5K if=/dev/zero of=diskimg

This will create a file initialized to contain all zeros, named diskimg. You only need to do this once, or every time you want to completely destroy the disk.

You should also write a format program to init this file, i.e. write its super block and bitmap blocks data .

**Syscalls**

To be able to have a simple functioning file system, we need to handle a minimum set of operations on files and directories. The functions are listed here in the order that I suggest you implement them in.

The syscalls need to return success or failure. Success is indicated by 0 and appropriate errors by the negation of the error code, as listed on the corresponding function’s man page.

|  |  |  |
| --- | --- | --- |
| u\_fs\_getattr | **Description:** | This function should look up the input path to determine if it is a directory or a file. If it is a directory, return the appropriate permissions. If it is a file, return the appropriate permissions as well as the actual size. This size must be accurate since it is used to determine EOF and thus read may not be called. |
| **UNIX**  **Equivalent:** | man -s 2 stat |
| **Return**  **values:** | 0 on success, with a correctly set structure  -ENOENT if the file is not found |
| u\_fs\_readdir |  | This function should look up the input path, ensuring that it is a directory, and then list the contents.  To list the contents, you need to use the filler() function. For example: filler(buf, ".", NULL, 0); adds the current directory to the listing generated by ls -a In general, you will only need to change the second parameter to be the name of the file or directory you want to add to the listing. |
|  | man -s 2 readdir  However it’s not exactly equivalent |
|  | 0 on success  -ENOENT if the directory is not valid or found |
| u\_fs\_mkdir |  | This function should add the new directory to the root level, and should update the .directories file |
|  | man -s 2 mkdir |
|  | 0 on success  -ENAMETOOLONG if the name is beyond 8 chars -EPERM if the directory is not under the root dir only  -EEXIST if the directory already exists |
| u\_fs\_rmdir |  | Deletes an empty directory |
|  | man -s 2 rmdir |
|  | 0 read on success  -ENOTEMPTY if the directory is not empty  -ENOENT if the directory is not found  -ENOTDIR if the path is not a directory |
| u\_fs\_mknod |  | This function should add a new file to a subdirectory, and should update the .directories file appropriately with the modified directory entry structure. |
|  | man -s 2 mknod |
|  | 0 on success  -ENAMETOOLONG if the name is beyond 8.3 chars  -EPERM if the file is trying to be created in the root dir  -EEXIST if the file already exists |
| u\_fs\_write |  | This function should write the data in buf into the file denoted by path, starting at offset. |
|  | man -s 2 write |
|  | size on success  -EFBIG if the offset is beyond the file size (but handle appends) |
| u\_fs\_read |  | This function should read the data in the file denoted by path into buf, starting at offset. |
|  | man -s 2 read |
|  | size read on success  -EISDIR if the path is a directory |
| u\_fs\_unlink |  | Delete a file |
|  | man -s 2 unlink |
|  | 0 read on success  -EISDIR if the path is a directory  -ENOENT if the file is not found |

**Building and Testing**

Your source files should be included as part of the Makefile in your directory, so building your changes is as simple as typing make.

One suggestion for testing is to launch a FUSE application with the -d option (./u\_fs -d testmount). This will keep the program in the foreground, and it will print out every message that the application receives, and interpret the return values that you’re getting back. Just open a second terminal window and try your testing procedures. Note if you do a **CTRL+C** in this window, you may not need to unmount the file system, but on crashes (transport errors) you definitely need to.

Your first steps will involve simply testing with ls and mkdir. When that works, try using echo and redirection to write to a file. cat will read from a file, and you will eventually even be able to launch pico on a file.

Remember that you may want to delete your .directories or .disk files if they become corrupted. You can use the commands od -x to see the contents in hex of either file, or the command strings to grab human readable text out of a binary file.

**Notes and Hints**

• The root directory is equivalent to your mount point. The FUSE application does not see the directory tree outside of this position. All paths are translated automatically for you.

• sscanf(path, "/%[^/]/%[^.].%s", directory, filename, extension);

• Your application is part of userspace, and as such you are free to use whatever parts of C Standard Library you wish, including the file handling functions.

• Start early; this one will take a bit of time.

• Remember to always close your disk and directory files after you open them in a function. Since the program doesn’t terminate until you unmount the file system, if you’ve opened a file for writing and not closed it, no other function can open that file simultaneously.

• Remember to open your files for binary access.

**Requirements and Submission**

You need to submit:

• Your well-commented source program

• A technic report within 20 pages

You can write your technic report either by English or Chinese.