CSC 452 – Project 5: File Systems

Due: Wednesday, May 4, 2022, by 11:59pm

Description

FUSE (http://fuse.sourceforge.net/) 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 via a single file that represents 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 only contain other subdirectories, and no 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 (made shorter) but can be deleted and directories can be removed.

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

Installation of FUSE

FUSE consists of two major components: A kernel module that is part of Linux, and a set of libraries and example programs that you need to install. Once again, we will be using our VM as our development environment.

Start up your VM. Update the package list and install fuse and tmux:

```
pacman -Sy
pacman -S fuse tmux
```

Copy an example and our project skeleton to the current directory from lectura:

```
scp USERNAME@lectura.cs.arizona.edu:~jmisurda/original/csc452fuse.c .
scp USERNAME@lectura.cs.arizona.edu:~jmisurda/original/hello_fuse.c .
```

First FUSE Example

Let us now walk through the hello example. First, we need to compile it:

```
gcc -Wall `pkg-config fuse --cflags --libs` hello_fuse.c -o hello
```

There's a lot going on with this command. We're compiling with all warnings enabled (-Wall) and then we're using something special to help us build a complex application: pkg-config. This is a command that we can use to get the right compiler flags to build something that uses an installed package on our system. Here we're asking about the fuse package and requesting both the flags for the c compiler (--cflags) as wells as the linker flags (--libs).

This is enclosed in backticks (the character on the same key as the tilde on typical US keyboard layouts). The backticks say to run a program and capture its output as a string. Apostrophes won't work, so make sure you type it correctly.

After it builds and produces an output file executable named hello, enter the following:

```
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 file system 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 csc452 file system as a FUSE application that provides the interface described in the first section. A code skeleton has been provided as `csc452fuse.c`. It can be built as hello.c was before:

```
gcc -Wall `pkg-config fuse --cflags --libs` csc452fuse.c -o csc452
```

The csc452 file system should be implemented using a single file, managed by the real file system in the directory that contains the csc452 application. This file should keep track of the directories and the file data. We will consider the disk to have 512-byte blocks.

Disk Management

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

```
dd bs=1K count=5K if=/dev/zero of=.disk
```

This will create a file initialized to contain all zeros, named `.disk`. You only need to do this once, or every time you want to completely destroy the disk. (This is our "format" command.) As always, a file that begins with a dot is "hidden" in Unix and won't show up unless we do an `ls -a` to see all files. It's still there, even if we don't see it with a normal `ls`.

Root Directory

Since the disk contains blocks that are directories and blocks that are file data, we need to be able to find and identify what a particular block represents. In our file system, the root only contains other directories, so we will use block 0 of `.disk` to hold the directory entry of the root and, from there, find our subdirectories.

The root directory entry will be a struct defined as below (the actual one we provide in the code has additional attributes and padding to force the structure to be 512 bytes):

```
struct csc452_root_directory
{
	int nDirectories;   //How many subdirectories are in the root
	                    //Needs to be less than MAX_DIRS_IN_ROOT
	struct csc452_directory
	{
		char dname[MAX_FILENAME + 1];    //directory name (plus space for nul)
		long nStartBlock;            //where the directory block is on disk
	} directories[MAX_DIRS_IN_ROOT]; //There is an array of these
} ;
```

Since we are limiting our root to be one block in size, there is a limit of how many subdirectories we can create, MAX_DIRS_IN_ROOT.

Each subdirectory will have an entry in the directories array with its name and the block index of the subdirectory's directory entry.

Subdirectories

Directories will be stored in our .disk file as a single block-sized `csc452_directory_entry` structure per subdirectory.

The structure is defined below (again the actual one we provide in the code has additional attributes and padding to force the structure to be 512 bytes):

```
struct csc452_directory_entry
{
	int nFiles;                      //How many files are in this directory.
	                                 //Needs to be less than MAX_FILES_IN_DIR

	struct csc452_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
	} files[MAX_FILES_IN_DIR];           //There is an array of these
};
```

Since we require each directory entry to only take up a single disk block, we are limited to a fixed number of files per directory.

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

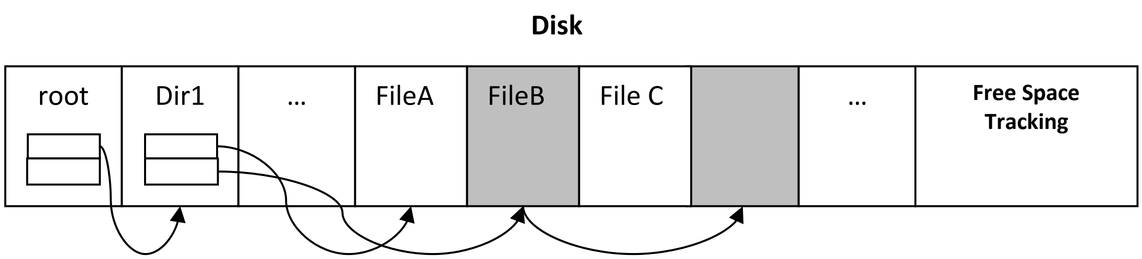
Files

Files will be stored alongside the directories in `.disk`. Data blocks are 512-byte structs of the format:

```
struct cs1550_disk_block
{
	//The next disk block, if needed. This is the next pointer in the linked
	//allocation list
	long nNextBlock;

	//All the space in the block can be used for actual data
	//storage.
	char data[MAX_DATA_IN_BLOCK];
};
```

This is how the resulting system is logically structured:



The root points to directory Dir1, which has two files, FileA and FileB. FileB spans two dis-contiguous blocks. FileC is referred to from some other directory, not shown.

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 last three do not need implemented beyond what the skeleton code has already.

The syscalls need to return success or failure. Success is generally (but not always) indicated by 0 and appropriate errors by the negation of the error code, as listed on the corresponding function's man page.

`csc452_mkdir`

Description:	This function should add the new directory to the root level directory entry.
UNIX Equivalent:	`man -s 2 mkdir`
Return values:	`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

`csc452_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

`csc452_readdir`

Description:	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.
UNIX Equivalent:	`man -s 2 readdir` However it's not exactly equivalent
Return values:	`0` on success `-ENOENT` if the directory is not valid or found

`csc452_rmdir`

Description:	Deletes an empty directory
UNIX Equivalent:	`man -s 2 rmdir`
Return values:	`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

`csc452_mknod`

Description:	This function should add a new file to a subdirectory, and should update the subdirectory entry appropriately with the modified information.
UNIX Equivalent:	`man -s 2 mknod`
Return values:	`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

`csc452_write`

Description:	This function should write the data in `buf` into the file denoted by `path`, starting at `offset`.
UNIX Equivalent:	`man -s 2 write`
Return values:	`size` on success `-EFBIG` if the offset is beyond the file size (but handle appends) `-ENOSPC` if the disk is full

`csc452_read`

Description:	This function should read the data in the file denoted by `path` into `buf`, starting at `offset`.
UNIX Equivalent:	`man -s 2 read`
Return values:	`size` read on success `-EISDIR` if the path is a directory

`csc452_unlink`

Description:	Delete a file
UNIX Equivalent:	`man -s 2 unlink`
Return values:	`0` read on success `-EISDIR` if the path is a directory `-ENOENT` if the file is not found

`csc452_open` This function should not be modified, as you get the full path every time any of the other functions are called.

`csc452_flush` This function should not be modified.

`csc452_truncate` This function should not be modified.

Building and Testing

For testing, we will want to launch a FUSE application with the `-d` option (`./csc452 -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.

We'd like to just open a second terminal window and try your testing procedures, but under our VM, we will have to use a terminal multiplexer to do this. A terminal multiplexer will allow us to see the output of our program in one "window" while having a shell in the other. You installed one called "tmux" before.

Do this:

1. Run `tmux`

2. Execute `./csc452 -d testmount`

3. Hit **CTRL+B**, release, and then type " (a quotation mark -- this requires holding shift down to type)
4. You should see a window with a shell appear at the bottom
5. When you're done testing, type exit to leave the shell and the bottom window will go away
6. Hit CTRL+C to end your csc452 program, and exit to leave that shell, which will exit tmux and drop you back to a normal command line

Doing a **CTRL+C** to csc452 will normally mean you do not need to unmount the file system, but on crashes (transport errors) you will.

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 nano on a file.

Remember that you may want to delete your `.disk` file if it becomes corrupted. You can use the commands `od -x` to see the contents in hex of a 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 and will all begin with a "/" representing the mount point directory (e.g., testmount) .

- You will need to co-implement `getattr()` and `mkdir()` at about the same time. The reason is that if you don't, you can get caught in an inconsistent state. Consider trying to make a directory named "foo":

  ```
  mkdir foo
  ```

  This `mkdir` shell command is a process that makes system calls to create the directory. It does three things:

  1. It asks if there is already something named "foo" by calling `getattr()` and checking that it returns `ENOENT` (not found). If not found, it's safe to make "foo" in step 2.
  2. It creates the directory via `mkdir()` which should return 0 (success)
  3. For some reason, it doesn't believe `mkdir()` actually worked so it calls `getattr()` again, which this time should return 0 (success) and fill in the struct with information that indicates that this is a directory (see the commented out code).

If you don't make `getattr()` work for returning existence/nonexistence for directories, step 3 will come back and say ENOENT (the current default return value for anything but the root), and the `mkdir` shell command will get confused and you'll get an error message.

- `sscanf(path, "/%[^/]/%[^.].%s", directory, filename, extension);` or you can use `strtok()`. Make sure you do not overrun any strings/buffers!
- Your application is part of userspace, and as such you are free to use whatever C Standard Libraries you wish, including the file handling ones.
- Remember to always close your disk file after you open it 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.
- Without the -d option, FUSE will be launched without knowledge of the directory you started it in, and thus won't be able to find your .disk file, if it is referenced via a relative path. This is okay, we will grade with the -d option enabled.

File Backups

I suggest making a directory on Lectura under your home directory that no one else can see.

If you have not done so for the other projects, on Lectura, do:

```
mkdir private
```

```
chmod 700 private
```

Backup all of your project files to your ~/`private/` directory frequently!

Loss of work not backed up is not grounds for an extension. YOU HAVE BEEN WARNED.

Requirements and Submission

You need to submit:

- Your well-commented program's source

```
turnin csc452-spring2022-p5 csc452fuse.c
```