Project 4: File System Implementation[1]

Submit a gzipped tarball of your code to CourseWeb.

Due: Saturday, August 1, 2020 @11:59pm
Late: Monday, August 3, 2020 @11:59pm with 10% reduction per late day

Table of Contents

[1] Based upon Project 4 of Dr. Misurda's CS 1550 course.

Project Overview

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 an 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.

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

Project Details

Installation of FUSE

FUSE consists of two major components: a kernel module that has already been installed, and a set of libraries and example programs that you need to install.

First, copy the source code to your `/u/OSLab/USERNAME` directory

```
cd /u/OSLab/USERNAME
cp /u/OSLab/original/fuse-2.7.0.tar.gz .
tar xvfz fuse-2.7.0.tar.gz
cd fuse-2.7.0
```

Now, we do the normal configure, compile, install procedure on UNIX, but omit the install step since that needs to be done as a superuser and has already been done.

```
./configure
make
```

(The third step would be `make install`, but if you try it, you will be met with many access denied errors.)

Setting up the Environment Variables

To be able to use fuse, you will need to do the following for it to work properly. Please enter the following commands after you first log in into thoth.cs.pitt.edu:

```
cd ~
chmod u+w .bash_profile
nano .bash_profile
```

This will give you write permissions to .bash_profile and open the nano editor. Now scroll down to the end of the file until you see the line:

```
# Define your own private shell functions and other commands here
```

Add the following lines (**spacing around the '[' and ']' characters need to be there!**):

```
if [ "$HOSTNAME" = "thoth.cs.pitt.edu" ]; then
source /opt/set_specific_profile.sh;
fi
```

Save the file and quit. Basically, you are directing the .bash_profile script to run another script located in /opt/set_specific_profile.sh, which if you look inside it, sets up the PATH environment variable.

Lastly, remove write permissions from .bash_profile for safety using the following command.

```
chmod u-w .bash_profile
```

Now, .bash_profile will not run until the next time you log in, so let's force run the script so that it is applied to this login session. Run the following command:

```
source /opt/set_specific_profile.sh
```

First FUSE Example

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

```
cd /u/OSLab/USERNAME/
cd fuse-2.7.0/example
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 cs1550 file system as a FUSE application that provides the interface described in the first section of this document. A code skeleton has been provided under the examples directory as `cs1550.c`. It is automatically built when you type `make` in the examples directory.

The cs1550 file system should be implemented using a single file, named `.disk` and managed by the real file system in the directory that contains the cs1550 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

In order to manage the free (or empty) space, you will need to create bookkeeping data in `.disk` that records the last block number that was allocated. (Please note that this is an overly simplistic way of keeping track of free blocks. In general, a scheme such as a block bitmap or a linked list of free blocks is needed.)

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.)

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 cs1550_root_directory
{
	long lastAllocatedBlock;	//The number of the last allocated block

	int nDirectories;	//How many subdirectories are in the root
				//Needs to be less than MAX_DIRS_IN_ROOT
	struct cs1550_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 on 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 `cs1550_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 cs1550_directory_entry
{
	int nFiles;			//How many files are in this directory.
					//Needs to be less than MAX_FILES_IN_DIR

	struct cs1550_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 nIndexBlock;		//where the index 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

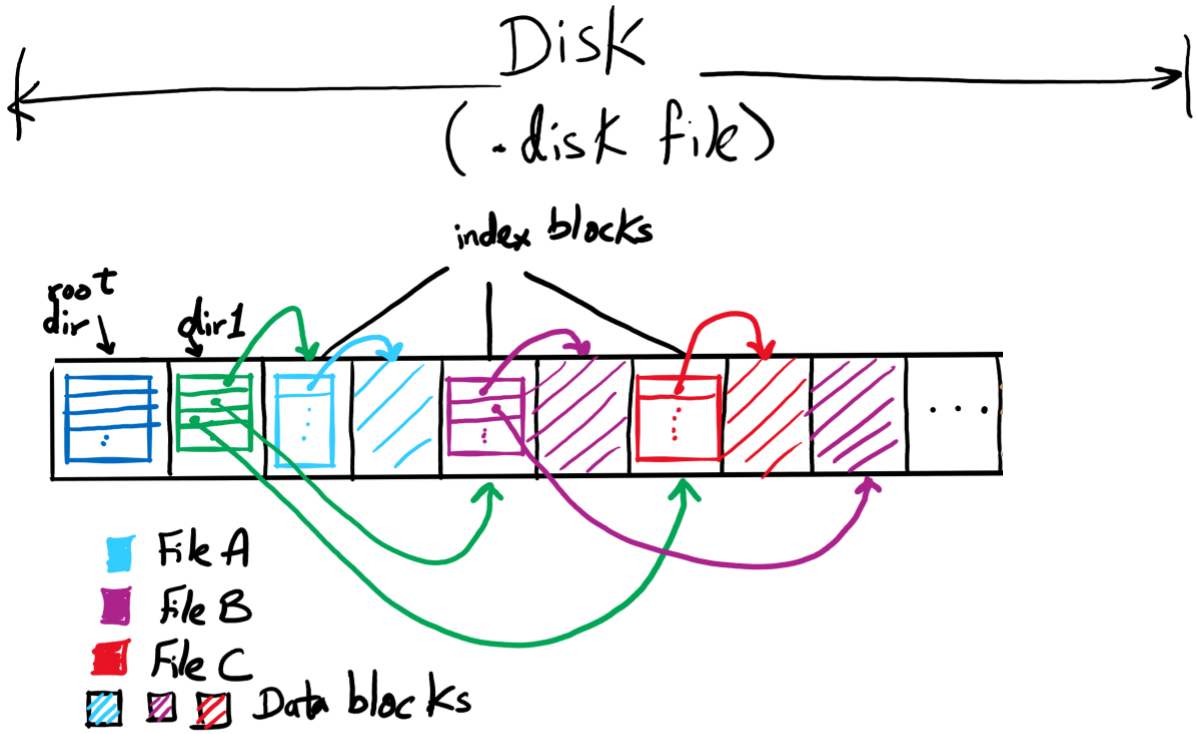
Files will be stored alongside the directories in the `.disk`. The size of the index and data blocks is 512 bytes. Each file has one index block and at least one data block. The index block is a struct of the format:

```
struct cs1550_index_block
{
	//All the space in the index block can be used for index entries. Each index
	//entry is a data block number.
	long entries[MAX_ENTRIES_IN_INDEX_BLOCK];
};
```

The data block is a struct of the format:

```
struct cs1550_disk_block
{
	//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 three files, File A, File B and File C, each having its own index block. File B has two data blocks. Both File A and File C have one data block each.

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 we 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 indicated by 0 and appropriate errors by the negation of the error code, as listed on the corresponding function's man page.

`cs1550_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

`cs1550_mkdir`

Description:	This function should add the new directory to the root level, and should update the `.disk` file appropriately.
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

`cs1550_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

`cs1550_rmdir` This function should not be modified.

`cs1550_mknod`

Description:	This function should add a new file to a subdirectory, and should update the `.disk` file appropriately with the modified directory entry structure.
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

`cs1550_unlink` This function should not be modified.

`cs1550_truncate` This function should not be modified.

`cs1550_open` This function should not be modified, as you get the full path every time any of the other functions are called.

`cs1550_flush` This function should not be modified.

`cs1550_init` This function includes code that is run when the file system loads (e.g., opening the .disk file, initializing a new .disk file)

`cs1550_destory` This function includes code that is run when the file system is stopped gracefully (e.g., closing the .disk file)

Building and Testing

The `cs1550.c` file is included as part of the `Makefile` in the examples 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 (`./cs1550 -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 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.
- `sscanf(path, "/%[^/]/%[^.].%s", directory, filename, extension);` or you can use `strtok()`
- 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.
- You can run gdb **gdb .libs/lt-cs1550** given that you inside the examples folder. You can then run the program (gdb) **r -d testmount**

File Backups

One of the major contributions the university provides for the AFS filesystem is nightly backups. However, the `/u/OSLab/` partition on thoth is not part of AFS space. Thus, any files you modify under your personal directory in `/u/OSLab/` are not backed up. If there is a catastrophic disk failure, all of your work will be irrecoverably lost. As such, it is my recommendation that you:

Backup all the files you change under `/u/OSLab` or QEMU to your `~/private/` directory frequently!

BE FOREWARNED: Loss of work not backed up is not grounds for an extension.

Requirements and Submission

You need to submit:

- Your well-commented `cs1550.c` program's source

Upload the file into Gradescope by the deadline.

Grading Sheet/Rubric

Item	Grade
cs1550_getattr	25%
cs1550_mkdir	25%
cs1550_readdir	20%
cs1550_mknod	20%
File System works correctly	10%

The test cases on Gradescope reflect the above rubrics. Please note that the score that you get from the autograder is not your final score. We still do manual grading. We may discover bugs and mistakes that were not caught by the test scripts and take penalty points off. Please use the autograder only as a tool to get immediate feedback and discover bugs in your program. Please note that certain bugs (e.g., deadlocks) in your program may or may not manifest when the test cases are run on your program. It may be the case that the same exact code fails in some tests then the same code passes the same tests when resubmitted. The fact that a test once fails should make you try to debug the issue not to resubmit and hope that the situation that caused the bug won't happen the next time around.