COSC 301 Project 5, Fall 2013: Amazon S3-backed File system

Due Thursday, 11 December 2013, 23:59:59

For the last project, you will build a fully-functional filesystem that stores file and directory content and metadata on the Amazon S3 storage system. Your filesystem will integrate with Linux using the FUSE library. Since you will use FUSE in this project (and because of other dependencies), you *must* use Linux for this project.

There are two stages of functionality with this project. Your quality of life will be improved if you follow the order of stages and fully test and debug the code in each stage before moving on to the next one. You are also very strongly advised to commit frequently using `git`, and to carefully manage your code so that you can always roll-back to a prior stage of development if something goes wrong. This project will be the most challenging one of the semester, so a little bit of planning and setup up- front will be worth it.

As with prior projects, you are welcome and encouraged to work with someone else. Please make a comment in your code to indicate who you worked with or who you got help from. Project submission is the same as with past projects: you'll need to keep your code in `git` and submit the repository name.

To get started, fork the git repo at <`https://github.com/jsommers/cosc301_proj05`>, and clone to your local workspace.

Overview of Amazon S3

Amazon's S3 service, or "Simple Storage Service", is a widely used cloud-based storage system. Users of S3 pay for transferring bytes in and out of S3, and for the storage of objects on S3.

The storage model is quite simple: all data are organized into **buckets**, and buckets contain **objects**. An object is the fundamental storage entity in S3. Each object must be uniquely identified within a bucket by a key (a name). So, the combination of the bucket name and the object key/name identifies a single object in S3. S3 is sometimes termed a "key-value" store, because each item in a bucket is composed of a key-value pair (where the value is an object). S3 objects also have version numbers, but we will not be concerned with those in this project.

For this project, each of you will be assigned your own S3 bucket. Within this single bucket you will store directory data, file data, and all metadata. Since you will be working within a single bucket, each item in the bucket must have a unique key/name. Conveniently, the name `"/a/path/to/a/file.txt"`, which represents a file path name, is also a valid key. This observation is key (yes, pun intended) for successfully completing this project.

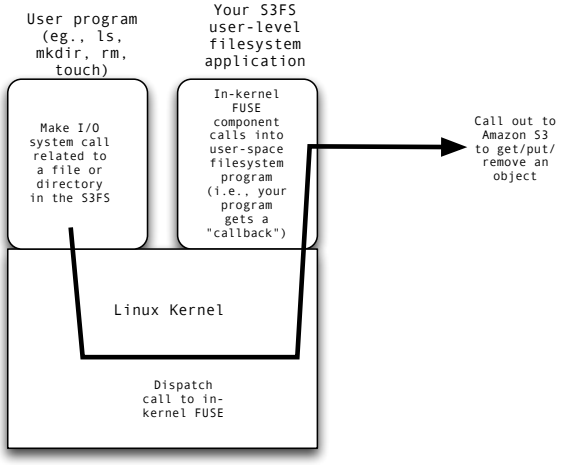
You will be provided with a set of functions (a library) that abstracts many of the details of using S3. You will simply need to call functions in this library to retrieve (`get`), store (`put`), and (`remove`) objects from a bucket (along with a few other calls).

Overview of FUSE

FUSE, or "Filesystems in User SpacE", is a Linux-based system for implementing a complete file system as a user-space program. We'll use FUSE to tie together the Linux file system and Amazon S3. We'll call the filesystem you create "`s3fs`".

The way FUSE works is as follows: your user-space program that implements s3fs must register itself with an in-kernel FUSE component. Upon registration, a new subtree of the file system is created and "mounted". For example, you might create a subdirectory called `mnt` inside your project directory (e.g., `s3fs-project`). If you start up your user-space s3fs program to use the `./mnt` subdirectory as a "mount point", then any I/O actions to files or directories (read, write, remove) within that subdirectory will first be directed into the kernel. The in-kernel file system (and

kernel FUSE component) will figure out that the action needs to be routed to your program. Your program will then receive a **callback** (i.e., a function you register in your program will get invoked by FUSE). Your s3fs code will then translate those actions into Amazon S3 actions, and interpret and pass back any results. The figure below gives an overview of this process.



Your main tasks for this project are to:

1. Design how you want to store file data, directory data, and metadata using the Amazon S3 key-value store (in your bucket). (One suggested design is described below, though you can implement a different design if you want.)

2. Implement and register FUSE callbacks to handle actions like `mkdir`, `rmdir`, `getattr` (like `stat`), and other calls, translating those calls into appropriate Amazon S3-related actions.

Project Setup in Linux

Before describing a simple design for storing filesystem data on S3, and implement FUSE callbacks, here are details on getting an environment set up in a Linux virtual machine. As noted above, use of Linux for this project is non-negotiable. I'd also strongly suggest that you use a virtual machine environment for development rather than the lab Linux machines, since you will be able to do anything as `root` if necessary. (You shouldn't need to, but just in case...)

You should have received an email with your Amazon S3 credentials in the form of a file named `env.sh`. (I have emailed or will email each of you a separate `env.sh` containing your bucket name and credentials.) Save this file in your project folder. **Please do not add this file to your git repo**: it contains sensitive/secret information about your S3 access. If you want to be sure that git ignores the file, you can create a text file named `.gitignore` in your repo and add the line `env.sh`.

Next, install a couple additional Ubuntu packages required for the Amazon S3 library that your code will indirectly use:

```
$ sudo apt-get install libcurl3-gnutls-dev libxml2-dev libfuse-dev
```

(`libfuse-dev` should already be installed. The above line will ensure that that's the case.)

Next, there should be a subfolder named `libs3-2.0` in your repo. `cd` into that directory and type `make`. Once compilation is done, type `sudo make install` then `cd ..` to get back to the main directory of your repo.

Now, type `. ./env.sh` to have your shell "source" (read) the contents of the file `env.sh`. Type `env | grep S3`. You should see something like:

```
S3_ACCESS_KEY_ID=AKkasdf323kasdf
S3_SECRET_ACCESS_KEY=234jalskdjflajsjfiwhefiahsdf
S3_BUCKET=edu.colgate.cosc301.username
```

(Note that your secret s3 information - key ID and secret access key - will be different (the ones above are garbage). Note also that the last line should list your username as the last component in the `S3_BUCKET` name.)

If all that worked, try one more test to verify you're up and ready to go:

```
$ s3 list $S3_BUCKET
```

This command should use the `s3` program (installed as part of compiling and installing `libs3-2.0`) and list the contents of your bucket on S3. It should be initially empty, but the command should still succeed.

If you type `make` in your project folder, the skeleton code for your file system, as well as a test program named `libs3_wrapper_test` will be compiled. Running the latter program (`./libs3_wrapper_test`) will cause some actions to take place with your bucket (a little object is written, then read back, then removed).

Please note: before you use Amazon S3 in testing, you will *always* need to type `. ./env.sh` to set up your shell environment. You'll need to do this for every shell you're testing in.

A Suggested Filesystem Structure

Here is one suggestion for how to organize directory data, file data, and metadata on S3. You do not need to follow this structure; it is merely a suggestion.

One thing to note before the design is described is that when you mount your s3fs filesystem, most callbacks you get from FUSE include the path name of the file or directory involved in the I/O operation. The path given in the callback is relative to the mount point that you use when initializing your s3fs program. So, for example, if you mount your s3fs filesystem on `/home/jsommers/proj4-s3fs/mnt`, and your working directory is the project folder `proj4-s3fs`, when you type `ls -l`, your program will get a callback for the path `"/"`, since the "root" of your filesystem is referenced from the containing folder. All paths that you receive as parameters to callbacks will start with `"/"`, and will be relative to the mountpoint.

In the example design described here, every name in the file system, e.g., `"/x"` is represented by some object in your S3 bucket. If the name refers to a directory, the data object stored on S3 will contain information about 1 or more directory entries (the current directory, as well as any other items contained within that directory). If the name refers to a file, the S3 object will contain the raw file data.

Each directory entry contains a name, a type (whether the entry refers to a file, a directory, or if the entry is unused), and metadata. The metadata include items such as the file size, access and modification times, owner, permissions, etc. You will need to decide exactly what metadata you want to store, but I suggest you look at the man page for `stat` (`man 2 stat`) for the metadata available from that system call (in the `struct stat` structure) as a guide for what you should/could include. A convenient way to implement this design is to treat directory data as an array of C `struct`. For example:

```
typedef struct {
    char type;          // file, directory, or unused
    char name[256];     // reasonable upper-bound on a name
    // metadata items would go here, too
} s3dirent_t;
```

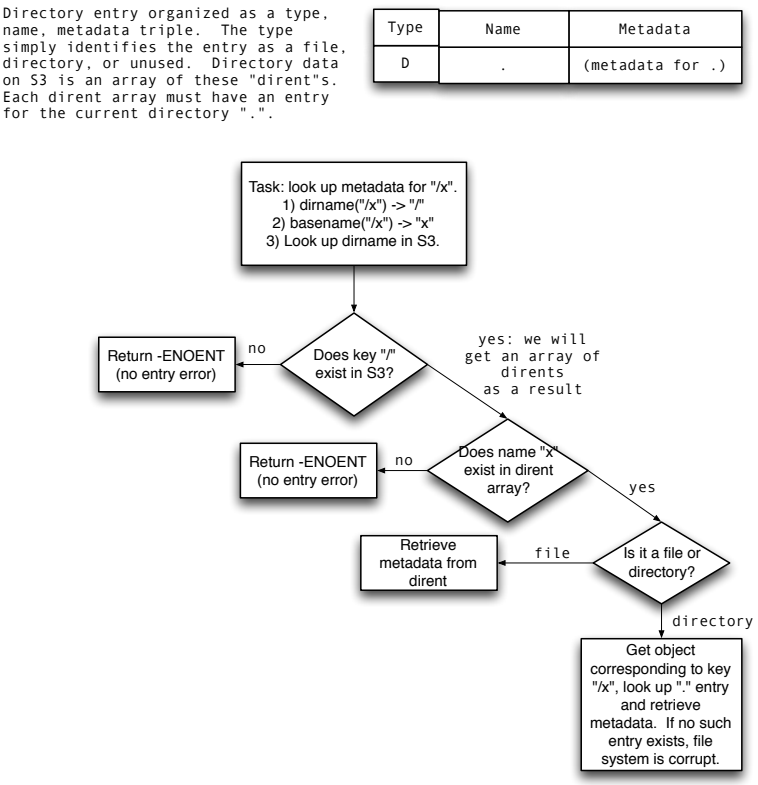
A few key points about this design:

- For every directory object you store on S3 (i.e., as an array of `s3dirent_t` objects), the first entry should always be for the current directory, and have the name `"."`.
- You should *not* have a fixed upper limit on the number of entries in a directory. When you retrieve the data for a directory, you can easily find out how many directory entries are present by dividing the retrieved object size by the size of a `s3dirent_t`.
- Each directory (except for the root directory `/`) will actually be referred to by two `s3dirent_t` structs: the parent directory will have an entry for it (with the directory name, and type directory), and the directory object itself will have a `"."` entry for itself. You should only store the metadata for the directory in *one* place. I would strongly suggest storing these metadata in the `"."` entry.

With a simplistic design like this, determining whether a given name refers to a file or a directory requires possibly two lookups (object retrievals) from S3. First, we split the name into two components: everything up to the last slash (`/`) is the first component, and everything after the last slash is the second component. Linux provides two built-in functions for obtaining these two items: the `dirname()` function can return the first item, and the `basename()` function returns the second item. (Read the `man` pages for these functions - they have good and useful examples in them.)

For example, say you got a callback to get the file attributes for `"/x"`. Up front, you don't know whether this name refers to a file or to a directory. What you can do is first split the name into a directory component and a basename component (`/` and `x`, respectively). Next, look up the directory object for the directory name (`/`). Assuming this object exists (and it's the root directory, so it had better exist!), search the list of directory entries for the name `x`. If the entry doesn't exist, you should return an error. If it does exist, you can find out from the directory entry whether the name `/x` refers to a file or a directory. If it is a file, you can retrieve the file attributes directly from the directory entry. If `x` refers to a directory, you can retrieve the object named `/x` on S3, which should contain directory entry data. The entry for `.` should have the metadata you are looking for.

The diagram below depicts this process:



Existing source code, running, and debugging

A list of the (important) source files in the project directory and what's in them is described below:

`s3fs.h` The main header file for your code. You should define your `s3dirent_t` structure here, as well as any other data type definitions or constants.

`s3fs.c` All the callbacks for your filesystem can be implemented here. There are skeleton functions already coded for maximum set of callbacks that you need to handle. (There are different stages of implementing these callbacks that you should follow. See below for more discussion on that.) A `main` skeleton function is already coded.

`libs3_wrapper.h` This header file contains the function prototypes for interacting with Amazon S3. The header file contains explanations for each function, and there is also a short sample program (`libs3_wrapper_test.c`)

that shows examples of using these functions. **Reading the code in this sample program will help for understanding how to use s3 from your code.**

There is also a `Makefile` provided that can build your filesystem program. Just typing `make` should do it. In fact, you should be able to compile and start up your filesystem even without adding any new code. (The `libs3_wrapper_test` program is also compiled for you to test with.) Once you do that, you can start things up by typing:

```
$ ./s3fs -d ./mnt
```

The `-d` option puts your filesystem in *debugging* mode, which basically means that you will see quite a bit of output from FUSE, and you can additionally write `printf` statements to show debugging output. If you do not supply the `-d` option, your program will start as a "background" job, and you will not see any `printf` output. For that reason, you should always test using the `-d` option.

You can type `Ctrl+C` to stop your `s3fs` program. This should unregister your filesystem from FUSE as a byproduct. If your program unexpectedly crashes, you will need to "unmount" your filesystem from the OS. Just type `fusermount -u ./mnt` to unmount it. (Otherwise, you won't be able to start your program again for testing.)

To test different callbacks in your filesystem, just use built-in Linux programs to provoke different callbacks. For example, `ls` will cause a callback for `opendir`, `readdir`, and `releasedir`, as well as to `getattr` (get file or directory attributes). Indeed, just getting a simple `ls -l` to work correctly should be your first goal. After that, you can try to create a new directory (and do another `ls -l`). You should also follow the project stages described below, which are aligned with the process described here (i.e., getting `ls` working, then `mkdir`, etc.)

Development Stages

To help manage the complexity of this project, you should proceed in stages. Each of the 3 stages below includes the portion of credit you can earn (at maximum) for reaching that stage. The skeleton file `s3fs` has some more information about each callback function to implement.

Stage 1 (worth 80 out of 100 points)

For the initial stage of this project, you will need to handle file system initialization, destroying the file system, getting attributes (metadata) for a directory or file, reading directory data, and creating/removing new directories. No files are involved in the project for this first stage. Each of the callbacks you will need to implement are described below. For now, you should implement the functions in the most naive way possible. (Strange advice, I know.) Don't try to be efficient. Just go for correctness.

`fs_init` Initialize the file system. At initialization you should **completely destroy everything in the bucket**. There is a `s3fs_clear_bucket` function in `libs3wrapper.h` to do this. You should then create a directory object to represent your root directory, and store that on S3.

`fs_destroy` This function is called when the filesystem is shut down, so you should clean up anything you need to and go away. You probably will not need to modify this function.

`fs_getattr` Given a path name, return metadata information about the named directory or file.

`fs_opendir` Open a directory. Hint: don't do anything in this function except check whether the named directory exists.

`fs_readdir` Read the entries of a directory object. This function is a little weird in that one of the callback function arguments is a pointer to a function named `filler` (of type `fuse_fill_dir_t`) and a buffer (`buf`) for filling in directory items. Inside `fs_readdir`, you should retrieve the directory object, then set up a `for` loop in which you call the `fill` function for every item in the directory. For example:

```
// objsize is the size of the retrieved directory object
int numdirent = objsize / sizeof(s3dirent_t);
int i = 0;
```

```
for (; i < numdirent; i++) {
    // call filler function to fill in directory name
    // to the supplied buffer
    if (filler(buf, dir[i].name, NULL, 0) != 0) {
        return -ENOMEM;
    }
}
```

fs_releasedir Close (release) an open directory. Hint: don't do anything except return success.

fs_mkdir Create a new directory. You should check whether the directory already exists (and return `-EEXIST` if that's the case). Otherwise, create a new directory object, and a new directory entry in the parent directory.

Note that the permissions (mode) for any directory *must* include `S_IFDIR` (defined in `sys/stat.h` --- see man 2 stat). You'll notice that at the top of the mkdir callback, there's already a line to explicitly do a bit-wise OR operation to ensure that that bit is part of the mode.

In your initialization of mode (i.e. part of your metadata) for your root directory, a reasonable setting is:

```
mode_t mode = (S_IFDIR | S_IRUSR | S_IWUSR | S_IXUSR);
```

(The macros are defined in `<sys/stat.h>`, again see `man 2 stat`.)

fs_rmdir Remove an existing directory. Only remove the directory if there are no entries other than `"."` (i.e., the directory is empty). Remember to remove the directory entry in the parent directory object.

Stage 2 (worth 20 out of 100 points)

For this stage, similar advice as in the first stage. Go for correctness and ignore efficiency.

fs_mknod Create a new file. This callback should only create an empty file. That's it.

fs_open Open an existing file. Hint: similar to `opendir`, just check whether the named file exists. Don't do anything else.

fs_read Read file data. You'll be passed a buffer to fill data in to, the amount of data to read, and the "offset" (number of bytes from the beginning of the file) to start reading from.

fs_write Write data to a file. Similar to `fs_read`, you'll be passed a buffer with data to write to a file, the number of bytes to write, and the offset from the beginning of the file for writing.

fs_release Equivalent to closing the file. If you take the path of least resistance, you shouldn't do anything in this file except return success.

fs_rename Rename a file.

fs_unlink Remove (unlink) a file.

fs_truncate Truncate a file (remove its contents, turning it into a zero-length file).

fs_ftruncate Truncate a file (remove its contents, turning it into a zero-length file). (The difference with `ftruncate` and `truncate` is that `ftruncate` is for truncating an open file. You can treat these calls as identical.)

fs_access Check access permissions for a file. In the template code, this function is set up initially to return success (0), even though that's not necessarily true.

For 5 points bonus, you can fix this callback so that it properly checks permissions. This function is pretty hard to write correctly. See `man 2 access` for the meaning of this system call and how it should behave.

Turning in the project

Just post the name of your github repo to Moodle.