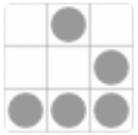
FUSE Linux Filesystem C

Writing a Simple Filesystem Using FUSE in C

Posted by Mohammed Q. Hussain on May 21, 2016.



Mohammed Q. Hussain

A Programmer.

FUSE (Filesystem in Userspace) is an interface that let you write your own filesystem for Linux in the user space. Of course being in the user space is a big advantage, you could use any of available libraries when you build your own filesystem in contrast of kernel space which needs a deep understand of the kernel which you are working with. Actually you can't build a native filesystem which can store data to disk directly [1] instead you need to use an already exist filesystem to do so [2]. However; you still could build interesting stuff, for example GDFS is a filesystem which let you mount your Google Drive in your system and access the files there as normal files. You can find a list of filesystems that implemented using FUSE here. For me I found ext4fuse really interesting.

Personal Experience

In one of my projects I needed a way to track any modifications on some specific files and after a simple search I found the system call **inotify**. For the requirements of the project inotify was not enough, yes it is going to catch any modifications on the files but I also needed to know in which offset of the file the changes have been written and I found no way to do that with inotify, thus I decided to use FUSE where I could implement my own "write" function therefore I could track any

changes on the files in question and easily get the offset of changes which was the critical information in the main part of the project.

Let's Start!

First you need to make sure that FUSE is installed in your Linux box. Note that I'm using version 2.9.4 here, so it might be some simple differences if you're using a newer version but the main concepts will be probably the same. FUSE has a structure called "fuse_operations" with the following definition [3]:

```
struct fuse_operations {
        int (*getattr) (const char *, struct stat *);
        int (*readlink) (const char *, char *, size_t);
        int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
        int (*mknod) (const char *, mode_t, dev_t);
        int (*mkdir) (const char *, mode_t);
        int (*unlink) (const char *);
        int (*rmdir) (const char *);
        int (*symlink) (const char *, const char *);
        int (*rename) (const char *, const char *);
        int (*link) (const char *, const char *);
        int (*chmod) (const char *, mode_t);
        int (*chown) (const char *, uid_t, gid_t);
```

```
int (*truncate) (const char *, off_t);
int (*utime) (const char *, struct utimbuf *);
int (*open) (const char *, struct fuse_file_info *);
int (*read) (const char *, char *, size_t, off_t,
             struct fuse_file_info *);
int (*write) (const char *, const char *, size_t, off_t,
              struct fuse_file_info *);
int (*statfs) (const char *, struct statvfs *);
int (*flush) (const char *, struct fuse_file_info *);
int (*release) (const char *, struct fuse_file_info *);
int (*fsync) (const char *, int, struct fuse_file_info *);
int (*setxattr) (const char *, const char *, const char *, size_t, int);
int (*getxattr) (const char *, const char *, char *, size_t);
int (*listxattr) (const char *, char *, size_t);
int (*removexattr) (const char *, const char *);
int (*opendir) (const char *, struct fuse_file_info *);
int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t,
                struct fuse_file_info *);
int (*releasedir) (const char *, struct fuse_file_info *);
int (*fsyncdir) (const char *, int, struct fuse_file_info *);
void *(*init) (struct fuse_conn_info *conn);
void (*destroy) (void *);
int (*access) (const char *, int);
int (*create) (const char *, mode_t, struct fuse_file_info *);
int (*ftruncate) (const char *, off_t, struct fuse_file_info *);
int (*fgetattr) (const char *, struct stat *, struct fuse_file_info *);
```

```
        int (*lock) (const char *, struct fuse_file_info *, int cmd,
                     struct flock *);
        int (*utimens) (const char *, const struct timespec tv[2]);
        int (*bmap) (const char *, size_t blocksize, uint64_t *idx);
        int (*ioctl) (const char *, int cmd, void *arg,
                      struct fuse_file_info *, unsigned int flags, void *data);
        int (*poll) (const char *, struct fuse_file_info *,
                     struct fuse_pollhandle *ph, unsigned *reventsp);
        int (*write_buf) (const char *, struct fuse_bufvec *buf, off_t off,
                          struct fuse_file_info *);
        int (*read_buf) (const char *, struct fuse_bufvec **bufp,
                         size_t size, off_t off, struct fuse_file_info *);
        int (*flock) (const char *, struct fuse_file_info *, int op);
        int (*fallocate) (const char *, int, off_t, off_t,
                          struct fuse_file_info *);
};
```

You can see that all the fields of this structure are pointers to functions. Each one of them will be called by FUSE when a specific event happens on the filesystem; for instance when the user writes on a file the function which is pointed by the field “write” in the structure will be called. If you take a quick look on the fields of the structure and try to figure out what event each field represents you’re going to find yourself familiar with many of those events. For example, obviously the function of “mkdir”

entry will be called when the user tries to create a new directory, “unlink” will be called when the user tries to delete a file and so on.

To implement your filesystem you need to use this structure and you need to define the functions of this structure then to fill the structure with the pointers of your implemented functions. Most of the functions here are optional; you don’t need to implement them all; even though some of them are essential for a functional filesystem (e.g. getattr). We’re going here to examine the most important functions that must be implemented to write our example simple filesystem and these functions are getattr, readdir and read.

The function of **getattr** event will be called when the system tries to get the attributes of the file. The comment on FUSE library source code says [3]: “Similar to **stat()**. The ‘st_dev’ and ‘st_blksize’ fields are ignored. The ‘st_ino’ field is ignored except if the ‘use_ino’ mount option is given.”

If we read the **manual page** of stat() we can see that it receives two parameters, the first one is the path of the file which its attributes must be returned, the second one is a pointer to the structure which will contain the attributes after the call of the function finishes. In case of success this function must return 0.

If you need more information about the attributes themselves I recommend the following links **Unix Stat Command: How To Identify File Attributes**, **Stat (system call)** and **The meaning of the File Attributes**

The function of **readdir** event will be called when the user tries to show the files and directories that reside in a specific directory. And as we can see from the names, the function of **read** event will be called when the system tries to read a chunk of data from a file.

Implementing Our Functions

What's Our Example Filesystem Going To Do?

Nothing special nor useful! it's a read-only filesystem, there are two files on it; namely "file54" and "file349". The user can open these files and read their contents. I'm going to call it **S**imple **S**tupid **F**ile**s**ystem (SSFS).

Implementing "getattr"

Well, let's start. Now we are going to implement the function of the event getattr, it is essential to write a functional filesystem. Once again the function of this event is going to return important information about each file that resides in our filesystem by filling a structure of type **stat**. In sake of

simplicity we're going to cover the essential fields of this structure just to make our filesystem works. maybe there are some uncovered fields in this tutorial that you are interested in, so I recommend you to read the documentation of the structure "stat".

Before we start the real job let's define FUSE version macro and include the required files to write SSFS:

```
#define FUSE_USE_VERSION 30

#include <fuse.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
```

The function "do_getattr" is the one which will be called when the system asks SSFS for the attributes of a specific file. We, of course, start with the header of the function:

```
static int do_getattr( const char *path, struct stat *st )
{
```

We can see that it takes two parameters and returns an integer. The first parameter is the path of the file which the system asked SSFS for its attributes. The other parameter is the “stat” structure which must be filled with the attributes of that file. On success the return value must be 0, otherwise it is -1 and “errno” must be filled with the right error code.

Well, let’s fill the first bunch of stat’s fields.

```
st->st_uid = getuid();
st->st_gid = getgid();
st->st_atime = time( NULL );
st->st_mtime = time( NULL );
```

The field “st_uid” represents the owner of the file in question, In SSFS the owner of all files/directories is the same user who mounted the filesystem. Also for “st_gid” which represents the owner group of the files/directories, it will be the same group of the user who mounted the filesystem. We use the functions **getuid** and **getgid** to get the user id and group id of the current user (the user who mounted the filesystem).

The field “st_atime” represents the last access time for the file in question and “st_mtime” represents the last modification time of the file. We fill both of them with the time which the function has been called in (the current time) by using the function **time**. Please note that the value of these two fields must be a **Unix time**.

Now, let’s fill the rest of the fields:

```
if ( strcmp( path, "/" ) == 0 )
{
        st->st_mode = S_IFDIR | 0755;
        st->st_nlink = 2;
}
else
{
        st->st_mode = S_IFREG | 0644;
        st->st_nlink = 1;
        st->st_size = 1024;
}

return 0;
```

As you can see from the last piece of code that we fill both “st_mode” and “st_nlink” fields with different information according to the path of the file in question. The field “st_mode” specifies if the file is a regular file, directory or **other**. In addition, “st_mode” specifies the permission bits of that file. The field “st_nlink” specifies the number of **hardlinks**. Finally, the field “st_size” specifies the size of that file in bytes.

Back to the previous code we can see that the first block of the if-statement is going to execute when the file in question is the root directory of **SSFS**. In this case we fill “st_mode” with the macro **S_IFDIR** which indicates that the file in question is a **directory**, then we set the **permission bits** as the following: only the owner of the file could read, write and execute the directory, the group’s users and other users could only read and execute the directory, If you don’t know what I’m talking about you could read “**Understanding Linux File Permissions**”. After filling “st_mode” for the root directory we specify the number of hardlinks, you might wonder why 2 hardlinks not just 1, you can find the answer **here**.

The block of else will be executed for all files other than the root directory, in **SSFS** namely: “file54” and “file349”. In “st_mode” for those files we use the macro **S_IFREG** to indicate that they are just regular files, then we specify the permission bits as the following: the owner could read and write the file, the group’s users and other users could only read the file. After that we set the number of hardlinks as 1. And finally we specify the size of the files as 1024 bytes.

Now, we got a simple & stupid implementation of getattr, the whole code with comments and debug messages:

```
static int do_getattr( const char *path, struct stat *st )
{
        printf( "[getattr] Called\n" );
        printf( "\tAttributes of %s requested\n", path );

        // GNU's definitions of the attributes
(http://www.gnu.org/software/libc/manual/html_node/Attribute-Meanings.html):
        //              st_uid:         The user ID of the file's owner.
        //              st_gid:         The group ID of the file.
        //              st_atime:       This is the last access time for the
file.
        //              st_mtime:       This is the time of the last modification
to the contents of the file.
        //              st_mode:        Specifies the mode of the file. This
includes file type information (see Testing File Type) and the file permission
bits (see Permission Bits).
        //              st_nlink:       The number of hard links to the file.
This count keeps track of how many directories have entries for this file. If the
count is ever decremented to zero, then the file itself is discarded as soon
        //                                                      as no process still holds
it open. Symbolic links are not counted in the total.
        //              st_size:        This specifies the size of a regular file
```

```
in bytes. For files that are really devices this field isn't usually meaningful.
For symbolic links this specifies the length of the file name the link refers to.

        st->st_uid = getuid(); // The owner of the file/directory is the user who
mounted the filesystem
        st->st_gid = getgid(); // The group of the file/directory is the same as
the group of the user who mounted the filesystem
        st->st_atime = time( NULL ); // The last "a"ccess of the file/directory
is right now
        st->st_mtime = time( NULL ); // The last "m"odification of the
file/directory is right now

        if ( strcmp( path, "/" ) == 0 )
        {
                st->st_mode = S_IFDIR | 0755;
                st->st_nlink = 2; // Why "two" hardlinks instead of "one"? The
answer is here: http://unix.stackexchange.com/a/101536
        }
        else
        {
                st->st_mode = S_IFREG | 0644;
                st->st_nlink = 1;
                st->st_size = 1024;
        }
```

```
        return 0;
}
```

Implementing “readdir”

In readdir we could list the files/directories which are available inside a specific directory. In SSFS there is only one directory (the root directory). Let’s start with the header of do_readdir function:

```
static int do_readdir( const char *path, void *buffer, fuse_fill_dir_t filler,
off_t offset, struct fuse_file_info *fi )
{
```

As you can see, do_readdir has five parameters, here; we are interested in the first three of them. The first parameter “path” is the path of the directory in question, that is, the directory which the system requested the list of files that reside under it. In the second parameter “buffer” we are going to fill the names of the files/directories which are available inside the directory in question. The third parameter “filler” is a function sent by FUSE and we could use it to fill the “buffer” with available files in “path”. The function returns 0 on success.

The declaration of “filler” is the following [3]:

```
typedef int (*fuse_fill_dir_t) (void *buf, const char *name,
                                const struct stat *stbuf, off_t off);
```

The first parameter is a pointer to the buffer which we want to write the entry (filename or directory name) on. The second parameter is the name of the current entry. The third and the fourth parameters will not be covered here.

```
filler( buffer, ".", NULL, 0 );
filler( buffer, "..", NULL, 0 );
```

We filled the list of available entries in “path” with two entries: “.” which represents the current directory, while “..” represents the parent directory. It’s a known convention in Unix world. Let’s continue:

```
if ( strcmp( path, "/" ) == 0 )
{
        filler( buffer, "file54", NULL, 0 );
        filler( buffer, "file349", NULL, 0 );
}
```

```
        return 0;
```

In case the directory in question is the root directory, we are going to add entries for the files: “file54” and “file349”. And that’s it!

The code of “do_readdir”:

```
static int do_readdir( const char *path, void *buffer, fuse_fill_dir_t filler,
off_t offset, struct fuse_file_info *fi )
{
        printf( "--> Getting The List of Files of %s\n", path );

        filler( buffer, ".", NULL, 0 ); // Current Directory
        filler( buffer, "..", NULL, 0 ); // Parent Directory

        if ( strcmp( path, "/" ) == 0 ) // If the user is trying to show the
files/directories of the root directory show the following
        {
                filler( buffer, "file54", NULL, 0 );
                filler( buffer, "file349", NULL, 0 );
        }
```

```
        return 0;
}
```

Implementing “read”

The last function we are going to implement is “read”. As we know, through this function the system could read the content of a specific file. Let’s start with the header:

```
static int do_read( const char *path, char *buffer, size_t size, off_t offset,
struct fuse_file_info *fi )
{
```

The first parameter “path” is the path of the file which the system wants to read. In the second parameter “buffer” we are going to store the ***chunk*** which the system interested in, the third parameter “size” represents the size of this chunk and the fourth parameter “offset” is the place in the file’s content where we are going to start reading from. “do_read” must return the number of the bytes that have been read successfully.

For the clarity I’m going to give a more detailed example about the parameters “size” and “offset”. Let’s assume that we have a file with the following content: “Filesystem in Userspace (FUSE) is a

software interface for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a “bridge” to the actual kernel interfaces.”.

If size = 35 and offset = 0, we are going to read the content from the first character, because offset = 0, and we are going to read the first 35 characters, so the content of “buffer” will be the following: “Filesystem in Userspace (FUSE) is a”.

Another example, let’s assume that size = 35 but the offset = 40, so we are going to skip the first 41 characters, because offset = 40, and start reading from the character 42 to 77, because the size = 35, the value of the “buffer” will be “ware interface for Unix-like comput”. Well, let’s start by defining some local variables:

```
char file54Text[] = "Hello World From File54!";
char file349Text[] = "Hello World From File349!";
char *selectedText = NULL;
```

The first string is the content of “file54” and the second is the content of “file349”. The job of “selectedText” will be shown in a moment:

```
        if ( strcmp( path, "/file54" ) == 0 )
                selectedText = file54Text;
        else if ( strcmp( path, "/file349" ) == 0 )
                selectedText = file349Text;
        else
                return -1;
```

If the file in question is “file54” then we are going to assign the pointer of “file54” content to the “selectedText” which is obviously represents the content which will be returned to the system. The same is going to happen with “file349”. In case that the system requested to read another file an error will be sent because SSFS only has those to files.

```
        memcpy( buffer, selectedText + offset, size );

        return strlen( selectedText ) - offset;
}
```

In the last part of “do_read”, we are going to copy - into “buffer” - the content of the file in question using **memcpy** starting from the “offset” until reaching “size”. Then we return the number of bytes that have been read.

Filling “fuse_operations” & Telling FUSE About It

Now simply we fill “fuse_operations” structure and call the main function of FUSE which is going to run our filesystem:

```
static struct fuse_operations operations = {
    .getattr    = do_getattr,
    .readdir    = do_readdir,
    .read       = do_read,
};

int main( int argc, char *argv[] )
{
        return fuse_main( argc, argv, &operations, NULL );
}
```

And we got our first filesystem! :-)

Compiling & Mounting The Filesystem

You can use GCC to compile SSFS as the following:

```
gcc ssfs.c -o ssfs `pkg-config fuse --cflags --libs`
```

The “pkg-config” part is going to provide the compiler the proper arguments to include “fuse” library.

To mount the filesystem after compilation:

```
./ssfs -f [mount point]
```

Using the option “-f” will let you see the debug messages which are printed using “printf”.

Source Code

You can find the source code of **SSFS** in my GitHub account here:

https://github.com/MaaSTaaR/SSFS.

References

[1] **FUSE in Wikipedia**.

[2] **The Source Code of “ext4fs”**.

[3] **The Source Code of “libfuse 2.9.4”**.