CS135 FUSE Documentation

There is very little FUSE documentation on the FUSE Web site. A bit more, which is unfortunately very outdated, is available from an IBM article from 2006. If you come across anything more complete or more current, I'd appreciate hearing about it so I can add a link to it from this site.

Note: Be sure to read the Gotchas list before starting your code, and refer back to it frequently when you run into troubles.

Writing a FUSE Client

The best way to write a fuse client is to start with an example or an existing client; I recommend `fusexmp.c` or `fusexmp_fh.c` (the latter implements file handles, so it's a better choice if you're developing a complex filesystem). The existing clients provide a scaffolding for you to work from, but you'll still need to understand what all the functions are supposed to do, and also how to compile and run your client. That's what this Web page is for.

Unix Manual Pages

May of the FUSE functions are closely related to Unix system calls. Rather than repeating the full specification (especially error conditions) here, it's better for you to refer to the Unix man page. You can do this on any Unix machine with the "man" command. By convention, if I refer you to the "stat(2) system call", that means you should type "man 2 stat" to get the necessary information.

FUSE File Handles

Many FUSE functions offer two ways to identify the file being operated upon. The first, which is always available, is the "`path`" argument, which is the full pathname (relative to the filesystem root) of the file in question. If you choose to do so, all your functions can use that argument to locate a file.

However, pathname lookup is often a very expensive operation, so FUSE sometimes provides you another option: a "file handle" in the "`fuse_file_info`" structure. The file handle is stored in the "`fh`" element of that structure, which is an unsigned 64-bit integer (`uint64_t`) uninterpreted by FUSE. If you choose to use it, you should set that field in your `open`, `create`, and `opendir` functions; other functions can then use it. In many FUSE implementations, the file handle is actually a pointer to a useful data structure, which is typecast to an integer to keep the compiler happy. But you can make it an index into an array, a hash key, or pretty much anything else you choose.

Getting FUSE Context

For many operations, it is useful to have a relevant "context" in which to execute them. For historical reasons, the context isn't passed as an argument; instead you must call `fuse_get_context` with no argument, which returns a pointer to a `struct fuse_context` with the following usable elements:

`uid`
: The (numeric) user ID of the process invoking the operation.

`gid`
: The (numeric) group ID of the process invoking the operation.

`pid`
: The thread (process) ID of the process invoking the operation.

`private_data`
: A pointer (`void*`) to the private data returned by the `init` function.

`umask`
: The `umask` of the process invoking the operation.

FUSE Functions

The following is a brief description of all the API functions you can create in a FUSE filesystem. Note that many are unnecessary, especially if you are implementing a partial filesystem like the one in this assignment. However, I have tried to provide full documentation here. Unless otherwise specified, all functions return an integer 0 or a positive number for success, or a negative value selected from `errno.h` if there is an error.

All of your functions should be named with a prefix that is closely related to your filesystem name. For example, in an SSH filesystem you should use `ssh_getattr`, `ssh_read`, etc.

`void* init(struct fuse_conn_info *conn)`
: Initialize the filesystem. This function can often be left unimplemented, but it can be a handy way to perform one-time setup such as allocating variable-sized data structures or initializing a new filesystem. The `fuse_conn_info` structure gives information about what features are supported by FUSE, and can be used to request certain capabilities (see below for more information). The return value of this function is available to all file operations in the `private_data` field of `fuse_context`. It is also passed as a parameter to the destroy() method. (Note: see the warning under Other Options below, regarding relative pathnames.)

`void destroy(void* private_data)`
: Called when the filesystem exits. The `private_data` comes from the return value of `init`.

`getattr(const char* path, struct stat* stbuf)`
: Return file attributes. The "stat" structure is described in detail in the stat(2) manual page. For the given pathname, this should fill in the elements of the "stat" structure. If a field is meaningless or semi-meaningless (e.g., `st_ino`) then it should be set to 0 or given a "reasonable" value. This call is pretty much required for a usable filesystem.

`fgetattr(const char* path, struct stat* stbuf)`
: As `getattr`, but called when fgetattr(2) is invoked by the user program.

`access(const char* path, mask)`
: This is the same as the access(2) system call. It returns -ENOENT if the `path` doesn't exist, -EACCESS if the requested permission isn't available, or 0 for success. Note that it can be called on files, directories, or any other object that appears in the filesystem. This call is not required but is highly recommended.

`readlink(const char* path, char* buf, size_t size)`
: If `path` is a symbolic link, fill `buf` with its target, up to `size`. See readlink(2) for how to handle a too-small buffer and for error codes. Not required if you don't support symbolic links. **NOTE:** Symbolic-link support requires only `readlink` and `symlink`. FUSE itself will take care of tracking symbolic links in paths, so your path-evaluation code doesn't need to worry about it.

`opendir(const char* path, struct fuse_file_info* fi)`
: Open a directory for reading.

`readdir(const char* path, void* buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info* fi)`
: Return one or more directory entries (`struct dirent`) to the caller. This is one of the most complex FUSE functions. It is related to, but not identical to, the readdir(2) and getdents(2) system calls, and the readdir(3) library function. Because of its complexity, it is described separately below. Required for essentially any filesystem, since it's what makes `ls` and a whole bunch of other things work.

`mknod(const char* path, mode_t mode, dev_t rdev)`
: Make a special (device) file, FIFO, or socket. See mknod(2) for details. This function is rarely needed, since it's uncommon to make these objects inside special-purpose filesystems.

`mkdir(const char* path, mode_t mode)`
: Create a directory with the given name. The directory permissions are encoded in `mode`. See mkdir(2) for details. This function is needed for any reasonable read/write filesystem.

`unlink(const char* path)`
: Remove (delete) the given file, symbolic link, hard link, or special node. Note that if you support hard links, `unlink` only deletes the data when the *last* hard link is removed. See unlink(2) for details.

`rmdir(const char* path)`
: Remove the given directory. This should succeed only if the directory is empty (except for "." and ".."). See rmdir(2) for details.

`symlink(const char* to, const char* from)`
: Create a symbolic link named "`from`" which, when evaluated, will lead to "`to`". Not required if you don't support symbolic links. **NOTE:** Symbolic-link support requires only `readlink` and `symlink`. FUSE itself will take care of tracking symbolic links in paths, so your path-evaluation code doesn't need to worry about it.

`rename(const char* from, const char* to)`
: Rename the file, directory, or other object "`from`" to the target "`to`". Note that the source and target don't have to be in the same directory, so it may be necessary to move the source to an entirely new directory. See rename(2) for full details.

`link(const char* from, const char* to)`

Create a hard link between "`from`" and "`to`". Hard links aren't required for a working filesystem, and many successful filesystems don't support them. If you *do* implement hard links, be aware that they have an effect on how [`unlink`] works. See `link(2)` for details.

`chmod(const char* path, mode_t mode)`
: Change the mode (permissions) of the given object to the given new permissions. Only the permissions bits of `mode` should be examined. See chmod(2) for details.

`chown(const char* path, uid_t uid, gid_t gid`
: Change the given object's owner and group to the provided values. See chown(2) for details. **NOTE:** FUSE doesn't deal particularly well with file ownership, since it usually runs as an unprivileged user and this call is restricted to the superuser. It's often easier to pretend that all files are owned by the user who mounted the filesystem, and to skip implementing this function.

`truncate(const char* path, off_t size)`
: Truncate or extend the given file so that it is precisely `size` bytes long. See truncate(2) for details. This call is required for read/write filesystems, because recreating a file will first truncate it.

`ftruncate(const char* path, off_t size)`
: As `truncate`, but called when ftruncate(2) is called by the user program.

`utimens(const char* path, const struct timespec ts[2]`
: Update the last access time of the given object from ts[0] and the last modification time from ts[1]. Both time specifications are given to nanosecond resolution, but your filesystem doesn't have to be that precise; see utimensat(2) for full details. Note that the time specifications are allowed to have certain special values; however, I don't know if FUSE functions have to support them. This function isn't necessary but is nice to have in a fully functional filesystem.

`open(const char* path, struct fuse_file_info* fi)`
: Open a file. If you aren't using file handles, this function should just check for existence and permissions and return either success or an error code. If you use file handles, you should also allocate any necessary structures and set `fi->fh`. In addition, `fi` has some other fields that an advanced filesystem might find useful; see the structure definition in `fuse_common.h` for very brief commentary.

`read(const char* path, char *buf, size_t size, off_t offset, struct fuse_file_info* fi)`
: Read `size` bytes from the given file into the buffer `buf`, beginning `offset` bytes into the file. See read(2) for full details. Returns the number of bytes transferred, or 0 if `offset` was at or beyond the end of the file. Required for any sensible filesystem.

`write(const char* path, char *buf, size_t size, off_t offset, struct fuse_file_info* fi)`
: As for [`read`] above, except that it can't return 0.

`statfs(const char* path, struct statvfs* stbuf`
: Return statistics about the filesystem. See statvfs(2) for a description of the structure contents. Usually, you can ignore the `path`. Not required, but handy for read/write filesystems since this is how programs like `df` determine the free space.

`release(const char* path, struct fuse_file_info *fi)`
: This is the only FUSE function that doesn't have a directly corresponding system call, although close(2) is related. `Release` is called when FUSE is completely done with a file; at that point, you can free up any temporarily allocated data structures. The IBM document claims that there is exactly one `release` per `open`, but I don't know if that is true.

`releasedir(const char* path, struct fuse_file_info *fi)`
: This is like `release`, except for directories.

`fsync(const char* path, int isdatasync, struct fuse_file_info* fi)`
: Flush any dirty information about the file to disk. If `isdatasync` is nonzero, only data, not metadata, needs to be flushed. When this call returns, all file data should be on stable storage. Many filesystems leave this call unimplemented, although technically that's a Bad Thing since it risks losing data. If you store your filesystem inside a plain file on another filesystem, you can implement this by calling fsync(2) on that file, which will flush too much data (slowing performance) but achieve the desired guarantee.

`fsyncdir(const char* path, int isdatasync, struct fuse_file_info* fi)`
: Like `fsync`, but for directories.

`flush(const char* path, struct fuse_file_info* fi)`
: Called on each close so that the filesystem has a chance to report delayed errors. **Important:** there may be more than one `flush` call for each `open`. **Note:** There is no guarantee that `flush` will ever be called at all!

`lock(const char* path, struct fuse_file_info* fi, int cmd, struct flock* locks)`
: Perform a POSIX file-locking operation. See details below.

`bmap(const char* path, size_t blocksize, uint64_t* blockno)`
: This function is similar to bmap(9). If the filesystem is backed by a block device, it converts `blockno` from a file-relative block number to a device-relative block. It isn't entirely clear how the `blocksize` parameter is intended to

be used.

`setxattr(const char* path, const char* name, const char* value, size_t size, int flags)`
: Set an extended attribute. See setxattr(2). This should be implemented only if `HAVE_SETXATTR` is true.

`getxattr(const char* path, const char* name, char* value, size_t size)`
: Read an extended attribute. See getxattr(2). This should be implemented only if `HAVE_SETXATTR` is true.

`listxattr(const char* path, const char* list, size_t size)`
: List the names of all extended attributes. See listxattr(2). This should be implemented only if `HAVE_SETXATTR` is true.

`ioctl(const char* path, int cmd, void* arg, struct fuse_file_info* fi, unsigned int flags, void* data`
: Support the ioctl(2) system call. As such, almost everything is up to the filesystem. On a 64-bit machine, `FUSE_IOCTL_COMPAT` will be set for 32-bit ioctls. The size and direction of data is determined by `_IOC_*()` decoding of `cmd`. For `_IOC_NONE`, `data` will be `NULL`; for `_IOC_WRITE` `data` is being written by the user; for `_IOC_READ` it is being read, and if both are set the data is bidirectional. In all non-NULL cases, the area is `_IOC_SIZE(cmd)` bytes in size.

`poll(const char* path, struct fuse_file_info* fi, struct fuse_pollhandle* ph, unsigned* reventsp);`
: Poll for I/O readiness. If `ph` is non-NULL, when the filesystem is ready for I/O it should call `fuse_notify_poll` (possibly asynchronously) with the specified `ph`; this will clear all pending `poll`s. The callee is responsible for destroying `ph` with `fuse_pollhandle_destroy()` when `ph` is no longer needed.

Init Function

The initialization function accepts a `fuse_conn_info` structure, which can be used to investigate and control the system's capabilities. The components of this structure are:

`proto_major` and `proto_minor`
: Major and minor versions of the FUSE protocol (read-only).

`async_read`
: On entry, this is nonzero if asynchronous reads are supported. The initialization function can modify this as desired. Note that this field is duplicated by the `FUSE_CAP_ASYNC_READ` flag; asynchronous reads are controlled by the logical OR of the field and the flag. (Yes, this is a silly hangover from the past.)

`max_write`
: The maximum size of a write buffer. This can be modified by the init function. If it is set to less than 4096, it is increased to that value.

`max_readahead`
: The maximum readahead size. This can be modified by the init function.

`capable`
: The capabilities supported by the FUSE kernel module, encoded as bit flags (read-only).

`want`
: The capabilities desired by the FUSE client, encoded as bit flags.

The capabilities that can be requested are:

`FUSE_CAP_ASYNC_READ`
: Use asynchronous reads (default). To disable this option, the client must clear both this capability (in the `want` flags) and the `async_read` field. If synchronous reads are chosen, Fuse will wait for reads to complete before issuing any other requests.

`FUSE_CAP_POSIX_LOCKS`
: Set if the client supports "remote" locking via the [`lock`] call.

`FUSE_CAP_ATOMIC_O_TRUNC`
: Set if the filesystem supports the `O_TRUNC` open flag.

`FUSE_CAP_EXPORT_SUPPORT`
: Set if the client handles lookups of "." and ".." itself. Otherwise, FUSE traps these and handles them.

`FUSE_CAP_BIG_WRITES`
: Set if the filesystem can handle writes larger than 4 KB.

`FUSE_CAP_DONT_MASK`
: Set to prevent the `umask` from being applied to files on create operations. (Note: as far as I can tell from examining the code, this flag isn't actually implemented.)

Readdir Function

The `readdir` function is somewhat like `read`, in that it starts at a given offset and returns results in a caller-supplied buffer. However, the offset not a byte offset, and the results are a series of `struct dirents` rather than being uninterpreted bytes. To make life easier, FUSE provides a "filler" function that will help you put things into the buffer.

The general plan for a complete and correct `readdir` is:

1. Find the first directory entry following the given `offset` (see below).
2. Optionally, create a `struct stat` that describes the file as for `getattr` (but FUSE only looks at `st_ino` and the file-type bits of `st_mode`).
3. Call the `filler` function with arguments of `buf`, the null-terminated filename, the address of your `struct stat` (or NULL if you have none), and the offset of the *next* directory entry.
4. If `filler` returns nonzero, or if there are no more files, return 0.
5. Find the next file in the directory.
6. Go back to step 2.

From FUSE's point of view, the `offset` is an uninterpreted `off_t` (i.e., an unsigned integer). You provide an offset when you call `filler`, and it's possible that such an offset might come back to you as an argument later. Typically, it's simply the byte offset (within your directory layout) of the directory entry, but it's really up to you.

It's also important to note that `readdir` can return errors in a number of instances; in particular it can return -EBADF if the file handle is invalid, or -ENOENT if you use the `path` argument and the path doesn't exist.

Lock function

The [`lock`] function is somewhat complex. The `cmd` will be one of `F_GETLK`, `F_SETLK`, or `F_SETLKW`. The fields in `locks` are defined in the fcntl(2) manual page; the `l_whence` field in that structure will always be `SEEK_SET`.

For checking lock ownership, the `fi->owner` argument must be used.

Contrary to what some other documentation states, the FUSE library does not appear to do anything special to help you out with locking. If you want locking to work, you will need to implement the lock function. (Persons who have more knowledge of how FUSE locking works are encouraged to contact me on this topic, since the existing documentation appears to be inaccurate.)

The Rest of a FUSE Client

Once you've written your operations, you need some boilerplate. As mentioned above, all of your functions should be named with a sensible prefix; here I use "`prefix`" to represent that. Create a `fuse_operations` struct that lists the functions you implemented (for any unimplemented ones, simply delete the relevant lines):

```
static struct fuse_operations prefix_oper = {
    .init        = prefix_init,
    .destroy     = prefix_destroy,
    .getattr     = prefix_getattr,
    .fgetattr    = prefix_fgetattr,
    .access      = prefix_access,
    .readlink    = prefix_readlink,
    .readdir     = prefix_readdir,
    .mknod       = prefix_mknod,
    .mkdir       = prefix_mkdir,
    .symlink     = prefix_symlink,
    .unlink      = prefix_unlink,
    .rmdir       = prefix_rmdir,
    .rename      = prefix_rename,
    .link        = prefix_link,
    .chmod       = prefix_chmod,
    .chown       = prefix_chown,
    .truncate    = prefix_truncate,
    .ftruncate   = prefix_ftruncate,
    .utimens     = prefix_utimens,
    .create      = prefix_create,
    .open        = prefix_open,
    .read        = prefix_read,
    .write       = prefix_write,
```

```
    .statfs      = prefix_statfs,
    .release     = prefix_release,
    .opendir     = prefix_opendir,
    .releasedir  = prefix_releasedir,
    .fsync       = prefix_fsync,
    .flush       = prefix_flush,
    .fsyncdir    = prefix_fsyncdir,
    .lock        = prefix_lock,
    .bmap        = prefix_bmap,
    .ioctl       = prefix_ioctl,
    .poll        = prefix_poll,
#ifdef HAVE_SETXATTR
    .setxattr    = prefix_setxattr,
    .getxattr    = prefix_getxattr,
    .listxattr   = prefix_listxattr,
    .removexattr = prefix_removexattr,
#endif
    .flag_nullpath_ok = 0,                /* See below */
};
```

Set `flag_nullpath_ok` nonzero if your code can accept a NULL path argument (because it gets file information from `fi->fh`) for the following operations: `fgetattr`, `flush`, `fsync`, `fsyncdir`, `ftruncate`, `lock`, `read`, `readdir`, `release`, `releasedir`, and `write`. This will allow FUSE to run more efficiently.

Finally, since your client is actually an executable program, you need a `main`:

```
int main(int argc, char *argv[])
{
    umask(0);
    return fuse_main(argc, argv, &prefix_oper, NULL);
}
```

Compiling Your Program

You can do your development on any machine you choose that supports FUSE. Mac users can try macfuse; Linux users should be able to find FUSE as part of their distribution.

Compiling a FUSE program requires a slightly complicated command:

```
/usr/bin/gcc -g `pkg-config fuse --cflags --libs` my_hello.c -o my_hello
```

A better approach, of course, is to use `make`. This truly minimal `Makefile` will let you type "`make foo`" for any `foo.c`. You are encouraged to use it and extend it to be more sensible. **NOTE:** On Wilkes, be sure to use "/usr/bin/gcc" rather than just "gcc". Wilkes is specially configured so that plain gcc produces 32-bit code, but for this assignment you need 64-bit code.

Running & Testing

To run a FUSE program, you'll need two windows and a scratch directory. You'll run your filesystem under a debugger in window #1; window #2 will be used for testing. The scratch directory is needed because you must have an empty directory on which to mount your shiny new filesystem.

The simplest (and incorrect, for our purposes) way to run a FUSE program is to make a scratch directory and then pass that as an argument to the program. For example, if you're running the "hello, world" filesystem (`hello.c`):

```
$ mkdir testdir
$ ./hello testdir
$ ls testdir
hello
$ cat testdir/hello
hello, world
$ fusermount -u testdir
$ rmdir testdir
```

When you run your program this way, it automatically goes into the background and starts serving up your filesystem. After you finish testing, the `fusermount` command unmounts your filesystem and kills the background program.

As a practical matter, it's easier to leave `testdir` hanging around rather than making it and removing it every time. Most systems have a number of empty directories hanging around just in case you want to mount on top of them (often, either `/mnt` or inside `/mnt`).

Of course, it's unlikely that your program will work perfectly the first time, so it's better to run it under the debugger. To do that, you'll need two windows. In window #1, do:

```
$ mkdir testdir         # if necessary
$ gdb hello
[gdb noise]
(gdb) [set breakpoints, etc.]
(gdb) run -s -d testdir
```

The `-s` switch means "single-threaded", which makes gdb behave in a much friendlier fashion. The `-d` switch means "debug"; in addition to printing helpful debugging output, it keeps the program in the foreground so gdb won't lose track of it.

Now, in window #2 you can do:

```
$ ls testdir
...                    # Other trial commands
$ fusermount -u testdir
```

IMPORTANT: You need to do the `fusermount` even if your program crashes or you abort it. Otherwise you'll get a confusing "Transport endpoint not connected" message the next time you try to mount the test system.

If you have set breakpoints, when you do "ls testdir", your window may seem to hang. That's OK; just go over to the gdb window and step through your code. When it returns a result, your test window will come alive again.

Other Options

Your new FUSE client has a lot of options. The simplest invocation just specifies a mount point. For example, if your client is named `fuse_client` and you're mounting on "~/foo", use:

```
./fuse_client ~/foo
```

There are tons of switches available; use `./fuse_client -h` to see them all. The important ones are:

-d
: Enable debugging output (implies -f).

-f
: Run in foreground; this is useful if you're running under a debugger. **WARNING:** When -f is given, Fuse's working directory is the directory you were in when you started it. Without -f, Fuse changes directories to "/". This will screw you up if you use relative pathnames.

-s
: Run single-threaded instead of multi-threaded. This makes debugging vastly easier, since gdb doesn't handle multiple threads all that well. It also protects you from all sorts of race conditions. Unless you're trying to write a production filesystem *and* you're a parallelism expert, I recommend that you *always* use this switch.

-o [no]rellinks
: Transform absolute symlinks to relative (or don't, if `norellinks` is given).

Contrary to what the help implies, switches can be specified before the mount point, in standard Unix fashion.

Gotchas

There are several common problems that plague programmers new to Fuse. This is a partial list:

Multithreading

By default, Fuse is multithreaded. That's handy for production filesystems, because it lets client (or file access) A proceed even if client B is hung up. But multithreading introduces the possibility of race conditions, and makes debugging harder. Always run with the `-s` switch to avoid this problem.

`getattr`
: Fuse calls `getattr` like crazy. Implement it first, or nothing will work.

Truncate
: Unless you're writing a read-only filesystem, you need to implement the `truncate` system call to make writes work correctly.

Working directory
: When it starts, Fuse changes its working directory to "/". That will probably break any code that uses relative pathnames. To make matters worse, the `chdir` is suppressed when you run with the -f switch, so your code might appear to work fine under the debugger. To avoid the problem, either (a) use absolute pathnames, or (b) record your current working directory by calling `get_current_dir_name` before you invoke `fuse_main`, and then convert relative pathnames into corresponding absolute ones. Obviously, (b) is the preferred approach.

Printf
: Your `printf`/`fprintf` debugging code will only work if you run with the `-f` switch. Otherwise, Fuse disconnects `stdout` and `stderr`.

Unimplemented functions
: It is very tempting to just leave functions undefined if your filesystem doesn't need them, or if you just haven't gotten around to writing them yet. Don't. If a function isn't listed in your `fuse_operations` struct, Fuse will silently generate a failure when it is called, and you'll never find out that you need to write it. Instead, write every unimplemented function as a stub that prints a message to `stderr` and returns an error code. When you see the message, you'll know what extra functions you need to write.

This page is maintained by Geoff Kuenning.