Distributed Systems Group
Department Software Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

Operating Systems (TI2726-C)
Lab Session 6: File systems

M. van de Ruit, O. W. Visser & J.S. Rellermeyer

March 29, 2018

Deadline: April 5 2018

Introduction

Session 6 of this lab will focus on the file system, through the introduction of *File Systems in User Space* (FUSE). You will have access to a (somewhat) functional data structure called the *Dumb File System* (DFS), and will have to make it accessible through the file system. In order, the purposes of this lab session are for you to:

- Become familiar with FUSE.
- Become familiar with basic file and directory operations.
- Make an underlying data structure accessible through FUSE.

Before you start working on the assignments, please **very carefully** read the provided explanations, **and familiarize yourself with the provided code**, as this will save you immeasurable time.

File Systems in User Space (FUSE)

FUSE, short for File Systems in User Space [1], is an interface for programs in user space to create and present virtual file systems. Traditionally, file systems are built within kernel space, which (as you may remember from a previous assignment) brings with it several limitations. As such, FUSE comes with a number of advantages and disadvantages:

- A FUSE file system can use userspace code and libraries.
- A FUSE file system can be loaded, mounted and unmounted by ordinary users.
- If a FUSE file system crashes, it will not crash the kernel.
- A FUSE file system is likely far slower than a kernel-based file system.

FUSE file systems are often used for writing virtual file systems, which can act as a specific view or translation of an existing underlying file system or other such component. For example, two file systems built with FUSE are *GmailFS* [2] (which stores data in e-mail format on Gmail servers) and *SSHFS* [3] (which provides access to data on a remote server through *SSH*).

A file system built with FUSE can be mounted to a directory by executing it's built program. For example:

```
pi/ $ ls
      dir fuse_program.o fuse_program.c
pi/ $ ./fuse_program dir
```

Afterwards, the file system is directly accessible in the folder into which it is mounted:

```
pi/ $ cd dir/
pi/dir/ $ ls
          subdir1 subdir2 Hello-there.txt
```

To unmount the file system again, `fusermount` or `umount` can be used:

```
pi/dir/ $ cd ..
pi/ $ sudo umount dir/
```

[1] FUSE: `https://github.com/libfuse/libfuse`
[2] GmailFS: `http://sr71.net/projects/gmailfs/`
[3] SSHFS: `https://github.com/libfuse/sshfs`

FUSE File operations

To function as a FUSE file system, a program must implement a number of *operations* that mirror common file system operations. FUSE has both a high-level and a low-level API for this. In this lab-session, we will use the high-level API. There is documentation [4] available online for specifics. Some of the high-level operations are shown below:

```
struct fuse_operations {
    int (*getattr) (const char *, struct stat *);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*create) (const char *, mode_t,
        struct fuse_file_info *);
    int (*rename) (const char *, const char *);
    int (*truncate) (const char *, off_t);
    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 *);
};
```

For a specification of all the operations FUSE supports, please refer to the specific documentation [5]. As stated previously, almost all of these operations map directly to existing operations in the `C` library. For example: `mkdir()` creates a directory, while `rename()` renames a file. Some operations are different, however, such as `getattr()`, which is similar to the existing `stat()` in `C`.
For the coming assignments, it is important that you understand the basics behind the following FUSE operations, as you will have to implement parts of them. Any function arguments marked with `...` are not relevant for this lab session.

- `int mkdir(const char * path, ...)`
  This function shall create a new directory with the name `path`.
- `int rmdir(const char * path)`
  This function shall remove a directory whose name is given by `path`.
- `int create(const char * path, ...)`
  This function shall create a file with the name `path`.
- `int unlink(const char * path)`
  This function shall remove a file at `path`.
- `int read(const char * path, char * buff, size_t size, off_t off, ...)`
  This function shall read data from a file at `path`. Specifically, it reads a chunk of bytes of size `size` from the file into `buff`, starting from offset `off`. It returns the exact number of bytes that was read (this is not always the same as `size`, as the function can encounter an end of file).

[4] FUSE Documentation: `http://libfuse.github.io/doxygen/`
[5] fuse_operations: `http://libfuse.github.io/doxygen/structfuse__operations.html`

- `int write(const char * path, char * buff, size_t size, off_t off, ...)`
  This function shall write data to a file at `path`. Specifically, it writes a chunk of bytes of size `size` from `buff` to the file, starting from offset `off`. It returns the exact number of bytes that was written.

- `int readdir(const char * path, void * buff, fuse_fill_dir_t fill, ...)`
  This function provides directory data for a directory at `path`. Specifically, it fills `buff` with the correct number of items in the directory (subdirectories, files) through use of `fill`, which is a function.
  For example, to add a subdirectory *documents* to `buff`, use `fill(buff, "Documents", NULL, 0);`.

The Dumb File System (DFS)

For these assignments, you will be implementing the mentioned FUSE operations to work on the DFS. The DFS is a tiny and completely in-memory data structure that stores directories and files in a tree-like structure. You can read the code, which is located in `os_dfs.c`, but you do not have to ever modify it for the assignment. The basic structures of the DFS are:

```
dir_t {
    char * name;
    list_t * files;
    list_t * dirs;
    dir_t * parent;
};

file_t {
    char * name;
    char * contents;
    dir_t * parent;
};
```

As can be seen, directories can store subdirectories and files. All items can have a pointer towards their containing parent. Files store their contents in a character array.
Furthermore, the DFS has a number of functions that can be used to, among others, add or remove files and directories, or search through the structure. These are briefly listed below, but the actual code has more expansive comments.

```
// Creates a file with name and content. Returns pointer to file.
file_t * create_file(const char * name, const char * contents);

// Destroys a file, freeing memory.
void destroy_file(file_t * file);

// Creates a directory with name. Returns pointer to directory.
dir_t * create_dir(const char * name);
```

```
// Destroys a directory, all subdirectories and subfiles, freeing memory.
void destroy_dir(dir_t * dir);

// Adds a file to a given directory.
void add_file_to(dir_t * parent, file_t * file);

// Adds a directory to another given directory.
void add_dir_to(dir_t * parent, dir_t * dir);

// Takes a path string, and returns a list of strings. Returns pointer.
list_t * parse_path(const char * path_s);

// Takes a list of strings, and returns a path string. Returns pointer.
char * unparse_path(list_t * path_l);

// Takes a path string, and returns the very last part, aka the name.
char * get_name_from_path(char * path_s);

// Takes a path string, and returns all but the last part, aka a path
// towards what is on the path.
char * get_dirs_from_path(char * path_s);

// Find a directory (by name), located directly inside another directory.
// Returns NULL if not found.
dir_t * find_dir_direct(dir_t * dir, const char * name);

// Find a file (by name), located directly inside another directory.
// Returns NULL if not found.
file_t * find_file_direct(dir_t * dir, const char * name);

// Find a directory (by path), located somewhere (deep) inside a directory.
// Returns NULL if not found.
dir_t * find_dir(dir_t * dir, const char * path_s);

// Find a file (by path), located somewhere (deep) inside a directory.
// Returns NULL if not found.
file_t * find_file(dir_t * dir, const char * path_s);
```

The DFS relies heavily on a very simple linked-list structure called `list_t`, which stores pointers to data. The list is displayed below, together with useful functions to modify or search it. Once again, the code `os_list.c` has more expansive comments.

```
// Nodes are the links inside the list.
node_t {
    void * data;
    node_t * next;
};
```

```
// The list itself stores a head, tail and length.
list_t {
    unsigned int length;
    node_t * head;
    node_t * tail;
};

// Create and allocate a list, returns pointer.
list_t * create_list();

// Destroy a list, deallocating all items inside.
// Also deallocates the pointers these items store!
void destroy_list(list_t * list);

// Push a pointer on the front of the list.
void push_front_p(list_t * list, void * data);

// Push a pointer on the back of the list.
void push_back_p(list_t * list, void * data);

// Pop a pointer from the front of the list
// Returns NULL if empty.
void * pop_front_p(list_t * list);

// Pop a pointer from the back of the list.
// Returns NULL if empty.
void * pop_back_p(list_t * list);

// Get a pointer stored at position i in the list.
// Returns NULL if empty or out of bounds.
void * get_p(list_t * list, unsigned int i);

// Remove and return a pointer stored at position i in the list.
// Returns NULL if empty or out of bounds.
void * remove_p(list_t * list, unsigned int i);
```

It should be clear to you that the DFS is extremely limited. It is not a file system, as it has no systems in place for symbolic links, file attributes, permissions (users, groups) etc. Furthermore, it doesn't store data, as it is completely in-memory. Finally, it is not even accessible to us from the shell!

This is where FUSE comes in, as we can expose the DFS to a user as if it were a file system.

Debugging your FUSE program

While working on your assignments, you are bound to break something, or encounter a crash. You may wish to add debug statements to see what's happening. However, FUSE has the nasty tendency to not output `printf()` statements while it is used normally. Thankfully, there are workarounds available:

- FUSE programs can be run in the foreground by passing the `-f` or `-d` flags, although you will have to find another way to use your file system while it is doing so (if you have multiple shells available through `SSH`, this is easy).
- `syslog()` can be used to, well, log. For more information on system logging, please see this link [6].
- FUSE programs can be run in `gdb`, which is extremely helpful when debugging crashes. For example, to get started, you can use:

```
pi/ $ gdb os_assignment
...
(gdb) r -d dir/
...
```

[6] Linux Log File Locations: `https://www.cyberciti.biz/faq/linux-log-files-location-and-how-do-i-view-log`

Assignments

For the assignments, all code has to be written in `C`, preferrably in the file `os_assignment.c`, and has to be able to run on the Raspberry Pi. Only the code has to be submitted to CPM. **A new makefile** is provided to you this week. You will also require the provided files `os_list.c` and `os_dfs.c` to build the code. Please rename `os_assignment.c` to `os_assignment_6_1.c` or similar before uploading your code.

As you can see in `os_assignment.c`, a part of the FUSE code (specifically, `truncate()` and `getattr()`) is already in place. During these assignments, you will implement some specific functions we require to interact with DFS through the normal file system. Until you do so, large parts of your file system will not work, or will be invisible to you!

Please read the assignments carefully and make sure your code exactly implements the given descriptions. We will assess the functionality and clarity of your code and comments. Answers to questions are allowed in both Dutch and English.

Please read **FUSE File operations** and **The Dumb File System** (and probably the attached code) before starting on the assignments. You can try to complete the assignments very quickly and with very short code (less than 70 lines of code), without ever modifying strings, traversing any data structure manually, or allocating memory, by using the functions from the DFS. Doing this can save you immeasurable time.

6.1: Reading directories

The first function you will have to complete is the function `os_readdir()`. As explained previously, you can use `fill()`, which is provided by FUSE, to add the correct items to `buff`. Some (incorrect) code is already inside this function, to show you some simple usage of DFS.
Your implementation should:

1. Fill the buffer with ".", which is a hard link from a folder to itself, and "..", which is a hard link from a folder to its parent.
2. Fill the buffer with all subdirectories and files in the directory specified by `path`.

If your implementation is correct, you can now see all the files and folders defined in `create_dfs()`, through `ls` in the shell. Beforehand, you could see a fake folder called "Documents".

6.2: Adding and removing directories

Now, you will have to implement `os_mkdir()` and `os_rmdir()`.

Your implementation should:

1. In `os_mkdir()`, find the parent directory to add your new directory to, create the new directory, and add it to the parent directory.
2. In `os_rmdir()`, find a directory and remove it.

If your implementation is correct, you can now add and remove directories to and from your file system, through the shell.

6.3: Reading and writing files

Now, you will have to implement `os_read()` and `os_write()`. Usage of `memcpy()`, `strlen()` and `realloc()` is recommended!

Your implementation should:

1. In `os_read()`, find the file to read from, copy the correct chunk of bits to the buffer, and return the correct number of read bits. If the file does not exist, do not forget to return an error.
2. IN `os_write()`, find the file to write to, copy the correct chunk of bits from the buffer into the file, and return the correct number of written bits. If the file does not exist, do not forget to return an error.

If your implementation is correct, you can now open up one of the arbitrary files in your system (say, in `nano`), add some text to it and then save it.

6.4: Adding and removing files

Now, you will have to implement `os_create()` and `os_unlink()`.
Your assignment is to:

1. In `os_create()`, find the parent directory where you create a file, create the new file, and add it to the directory.
2. In `os_unlink()`, find the file and remove it.

If your implementation is correct, you can now add and remove simple files to your system. For example:

```
/pi/dir $ echo "Hello!" > hi.txt
/pi/dir $ ls
          subdir readme.txt hi.txt
/pi/dir $ rm hi.txt
/pi/dir $ ls
          subdir readme.txt
```

Bonus: Dumber Dumb File System (DDFS)

One of the major advantages of FUSE is that it runs in user space. Hence, you can reuse libraries and code from previous assignments without any problem. For example, you can use *wiringPi* to control a Pi's LEDS through a file system.

The (sadly completely anonymous) developer of DFS is asking you to dumb down their system a bit more. They want to keep track of the size of DFS, and hence ask you to use the Pi's LEDs to display how many files there are inside DFS. Unfortunately, there are only 4 LEDs on the Pi, so you will simply have to limit the number of files in the system!

Your objective is to:

1. Either modify/expand DFS or your FUSE implementation such that you know exactly how many files there are in the system.
2. Display and update the number of files in the system on the Pi's LEDs in binary.
3. Limit the number of files in the system to a maximum of 16.