HOMEWORK 5

A read-only File System Driver

Issued	3/20/14
Due Date	~~4/8/14~~ 4/10/14 **before class**

NOTE: As a reminder, homework is considered handed in when it is committed to your personal subversion repository before the deadline. Changes made after the deadline will be ignored.

Goal

For this homework, we will be designing our own file system and develop (part of) a driver to access the file stored on our file system.

Evaluation

The homework will be evaluated using the following factors:

- Code Quality: Your code should compile without warnings or errors
- Documentation of the code -- The code is clearly documented, explaining what the intent is and how it is accomplished.
- Correctness of execution -- The code performs the functions requested.
- Creativity -- Your solution is different. Make sure to **document** your code or we might miss your special trick!
- Optional tasks -- You implemented some or all of the optional requests, or went beyond what was requested in some other way.

What to hand in

As for all the other homework assignments, you will need to commit your homework to your assigned subversion repository, under the correct directory (`homework/homework5`).

Use EXACTLY the requested file and directory names, including case!

Please commit the following files:

- `mkfs.c`, `yourfs.c`, `yourfs.h` and any other files required to build the test program and the mkfs program. FILES SHOULD COMPILE WITHOUT WARNINGS!
- The `Makefile`
- Makefile used to build your floppy disk image
- A text file `DESCRIPTION` containing a description of what your code does, how you achieved it, clearly listing which tasks (see above) you completed.

Is is OK if you commit more files than requested, but make sure the files requested above are present and exactly named as specified.

Details

Environment

Your code needs to compile under your virtual machine (ubuntu). While a part of your code will eventually be used in our own kernel, for the purpose of this homework, all development happens under linux.

A simple `Makefile` is provided. Feel free to enhance this makefile (for example to automate invoking your `mkfs` program or to automate testing.

Storage Abstraction

As discussed during class, the file system is typically implemented on top of a storage abstraction. For the homework, the way to access the underlying storage device is through the `device.h` header. This header contains functions to obtain the blocksize of the device, the number of blocks and to read or write a block on the device.

Please see `device.h` for a detailed description of the functions and their parameters.

You should not modify `device.h` or `device.c`.

Task 1: mkfs

The file system developed for this homework is a **read-only** file system. This means that it is intended for devices which do not support modifying device contents, such as a CDROM.

However, we still need to be able to create the file system. Since we can't create it the normal way (i.e. format the device with the file system and then copy the files onto the file system or similar), we will use a separate program which, given a set of files and directories, creates the final file system.

Part of this program is already provided; see the `mkfs.c` file in the shared repository.

This program takes a number of options:

- `-v` and `-d`: these options set the `opt_verbose` and `opt_debug` variables. You can use these in your code to enable extra debug output if you want.
- `-i` filename: this option specifies **an existing file** in which the tool will write the file system image. You can use `make image1.img` and `make image2.img` to create empty image files which you can use for this.
- `-c` number: This sets the `opt_clear` variable; If this variable is not equal to 0, your `mkfs` programs needs to fill every unused disk block (i.e. every block not storing file data or data structures for you file system) with this byte.

So, for example, given a directory `test` containing the following files:

```
a
b
c
```

and an image file `image1.img`, calling your `mkfs` program as:

```
./mkfs -i image1.img test
```

should create a file system containing 3 files, `a`, `b` and `c`, where each file contains the contents of the equally named file in the `test` directory.

You can make the following assumptions:

- File and directory names can only contain letters and numbers (a-z,A-Z,0-9) and each directory or file name is at most 254 characters.
- The file system does not need to support modification, once created.
- For meta data / attributes, only the file size needs to be stored. So there is no need to store permission, owner or other information.
- Each file will be smaller than 2 GB in length.
- Each directory will contain at most 255 files.

Task 2: yourfs.h and yourfs.c

The `yourfs.h` and `yourfs.c` files provide the read access to the file system that was created by the `mkfs` tool.

The test program uses these two files to access the file system created by `mkfs`, and to validate that each file was correctly stored on your file system.

See the `yourfs.h` file for a list of functions to implement and how these functions should behave.

There is no need to modify =yourfs.h=

An example implementation of `yourfs.c`, which calls the existing POSIX I/O functions is provided. This is just **an example**. You should replace `yourfs.c` with your own version which provides the same functions and functionality, but uses `device.h` to obtain the file data and directory listings instead.

Tasks

Everybody needs to implement the 'minimal' functionality. For extra credit, one or more of the optional tasks can be completed.

Approximately 75% **of the points for the code** are allocated for to the minimal task, and each optional task counts for an additional 25%. If you successfully complete the two additional tasks, extra credit will go towards other homeworks.

The following tasks need to be completed:

Minimal

The following tasks need to be completed:

- Create a working `mkfs` program
- Create a working `yourfs.h` and `yourfs.c`
- Running the test program finds no errors.
- All code compiles without warnings.
- **No subdirectories need to be supported**; However, **if subdirectories are present in the directory given to the mkfs tool, they need to be ignored by the tool**, and your `mkfs` program should still function correctly.
- The `readdir` functionality does not need to be completed.

Optional 1: Readdir

For extra credit, implement the directory listing functionality in `yourfs.h` and `yourfs.c`.

```
yourfs_dir_open
yourfs_dir_close
yourfs_dir_getentry
```

See `yourfs.h` for more information. The special files `.` and=..= should not be returned by your `yourfs_dir_getentry` function.

Optional 2: Subdirectories

For extra credit, implement subdirectories. Subdirectories should be able to nest to an arbitrary depth.

Self-Evaluation

To help you to develop and test your code, a program `testfs` is provided. The idea is that given a directory containing files (and possibly other subdirectories), and an image file (created by `mkfs`) which contains the same files and directories, `testfs` will use your `yourfs.c` file to verify that all the files are present in the image and that the contents of the files etc. is correct (when accessed through your `yourfs.c` functions).

By default, the program only tests the minimal functionality (no subdirectories, no readdir). If you also want to test the optional functionality, pass `-1` or `-2` to the test program.

To grade the homework, the same `testfs` program will be used.

Help and hints

- A program to test your `mkfs` tool and `yourfs.c` and `yourfs.h` files is provided.
- The `valgrind` tool can help you find bugs in your program. http://valgrind.org/

It can be installed using `apt-get` under ubuntu.

- In the `yourfs_xxx_open` functions, you can `malloc` a `yourfs_xxx_handle_t` structure, and return it. Store enough information to identify the file or directory, so when later your `yourfs_xxx_read` function is called, you can retrieve the information from the structure.

  In the `yourfs_xxx_close` function, you can `free()` the structure again.
- If you do not implement subdirectories, you can ignore the `yourfs_dir_handle_t` parameter as it will always be equal to `YOURFS_ROOT_DIRECTORY`.
- In case you do implement subdirectories: take some time to think about how to best implement this. Hint: even if you don't allow subdirectories, there will at least be one directory in your file system
  the root directory. So, maybe, adding support for subdirectories should not be that much work since you will already have code to handle the root directory.