For details on the fsx600 file system format and the FUSE library please see the accompanying document.

Materials

You will be provided with the following files in your team repository:

- Makefile
- fsx600.h – structure definitions
- homework.c – skeleton code
- misc.c – additional support code
- libhw3.c, hw3test.py – python-based unit test framework
- fsck.hw3 – file system checker
- mktest.c – creates a small test file system image (with files)
- mkfs-hw3.c – creates an empty file system
- read-img.c – utility for reading and checking disk image files

Additional information beyond the accompanying documents may be found in source file comments.

Usage and other hints:

- compiling – type ‘make’ to compile everything, ‘make clean’ to delete all the output files.
- the ‘mktest’ program is used to create a 1MB disk image containing several files and directories:
  `./mktest foo.img`
  creates an image file named ‘foo.img’
- to create an empty image of arbitrary size use the ‘mkfs-hw3’ program:
  `./mkfs-hw3 -size 50m big.img`
- To mount the file system in 'foo.img' on the directory 'testdir':
  `./homework -image foo.img testdir`
- When testing write functionality, create a clean disk image each time you re-run the program.

Question 1 – Read-only access

Implement code for read-only command-line access to fsx600 disk images. Test it using (a) the unit test framework, creating a script test-q1.py, and (b) via shell commands, creating the script test-q1.sh

You will need to implement the following methods:

- getattr
- read
- readdir
- statfs

Deliverable: After finishing this part, tag it in Git using the command 'git tag PART1'. (don't forget to push after doing this) If you absolutely need to make a change after doing that, you can create new tags PART1a, PART1b…

Question 2 – Write access

For this question you will need to implement full read/write access and test it.

Suggestions:

- Use fsck.hw3 or read-img to check whether you are writing to the disk image correctly.
- Beware of corrupted disk images. You may want to regenerate the disk image with mktest or mkfs-x6 each time you recompile your code.

Deliverable: The completed homework.c, plus unit tests in test-q2.py and full tests in test-q2.sh

Implementation hints

Initialization – in the fs_init function you should read in the superblock to get the file system parameters. It's also a good idea to allocate memory and read in the inode and block bitmaps, as well as the inode table itself. (note that you'll have to re-write the appropriate blocks to disk whenever you modify these structures, but it's still easier to manipulate them in memory)

Path argument – the 'path' argument to every method is read-only ('const'), so you can't use the normal C library functions (strtok, strsep) to split it – you'll have to make a copy first. The easiest way to make a copy is with the 'strdupa' function, which uses a local variable (instead of calling malloc) so you don't have to remember to free the memory::

```
getattr(const char *path, …) {
    char *_path = strdupa(path);
    ...call function that splits '_path'...
```

Path translation – **please** factor out your path translation code, so you don't have 10 different versions in 10 different places. (and factor path splitting into its own function, too) Note that there are really two different things that you want to do with a path – for most (e.g. getattr, read) you want to translate "/a/b/c" to the inode for "c", while for most of the rest (e.g. mkdir, unlink) you want to translate "/a/b/c" to the inode for "/a/b", plus the string "c".

Additional information may be found in the skeleton file comments, in additional documentation provided, and on Piazza.