For details on the fs5600 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
- homework.c – skeleton code
- misc.c – additional support code
- image.c, blkdev.h – the disk image device and blkdev header file (see docs)
- mktest.c – creates a simple file system image
- test.sh – file system testing framework
- 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.
- To create a sample image file (containing 1 file) named "foo.img", or re-write an existing damaged one:
  ```
  ./mktest foo.img
  ```
  (or you can create an empty image of arbitrary size with the 'mkfs-x6' program)
- To run the file system in a simple command-line mode:
  ```
  ./homework -cmdline foo.img
  ```
- When testing write functionality, create a clean disk image each time you re-run the program.

Question 1 – command line read-only access

Implement code for read-only command-line access to fs5600 disk images. (note that this is the same as the code for read-only FUSE access; however you will debug and test with the command-line interface, which is simpler and easier to debug)

See the end of this document for implementation hints.

Deliverable: the question 1 functionality in your code should work. You will need to implement a test script, test-1.sh, which tests this functionality. (see homework-3-test.pdf)

Question 2 – command line read/write access

For this question you will need to implement read/write access through the command line interface.

Suggestions:

- Use the read-img utility 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 read/write functionality should work. You will need to implement a test script, test-2.sh, to test this. (again, see homework-3-test.pdf)

Question 3 – FUSE access

Test and debug FUSE access.

Suggestions:

- The first thing that FUSE does after mounting a file system is to call getattr("/"), so you need to handle 'getattr' properly for the root directory.
- To run the file system:
  - create a directory: `mkdir dir`
  - run the program: `./homework -d -s -image foo.img dir`

    the '-d' (debug) option causes it to run in the foreground (you'll need another window to access the directory), and the '-s' specifies single-threaded. These are also the options you would use running under the debugger.
- After running the file system, you have to umount it with the command 'fusermount -u dir' (e.g. from another window). This is the case even if your program crashed. If you don't do that, attempts to access the mount directory or to re-run your program will fail with "transport endpoint is not connected".

Deliverable: fully functional FUSE functionality (which is pretty much the same functionality as for question 2, but with FUSE instead), and script test-3.sh to test it. In addition please generate the code coverage report homework.c.gcov, as described in the testing document, and add it to your repository.

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. The easiest way to make a copy of it 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. 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".

Read and write – these will be easier to implement if you factor out a function that translates from block number within the file (i.e. 0, 1, 2 for bytes 0..1023, 1024..2047, 2048..3071) to disk block numbers. Or maybe two functions, because for 'write' you want to allocate the block (and indirect block if necessary) if it doesn't exist.