LOUISIANA STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CSC 4103: Operating Systems
Prof. Golden G. Richard III

Programming Assignment # 3: A Simple Filesystem
Due Date: LAST DAY OF LECTURE @ Class Time
NO LATE SUBMISSIONS

**** TEAMS OF 1 or 2 STUDENTS ARE ALLOWED ****
BE SURE THE NAMES OF ALL TEAM MEMBERS ARE IN THE SOURCE FILES.
SUBMIT ONLY *ONE* SOLUTION PER TEAM.

START RIGHT AWAY.

The Mission

In this assignment you will implement a simple filesystem from scratch. I will provide the implementation of a persistent "raw" software disk (available in `softwaredisk.h` and `softwaredisk.c`), which supports reads and writes of fixed-sized blocks (numbered `0..software_disk_size() - 1`). This component simulates a hard drive.

Your goal is to wrap a higher-level filesystem interface around my software disk implementation—that API is in the file `filesystem.h`. You must not change this file. Your implementation of the API will go in a new file, `filesystem.c`.

Requirements:

- Obviously you must store all data persistently on the software disk and use only the software disk API. You **may not** use standard filesystem operations like `fopen()`, etc. at all—you are implementing the filesystem from scratch, above a block-oriented storage device.
- You must provide a file system initialization program called `formatfs.c` which initializes your file system. Part of this initialization will include initializing the software disk via `init_software_disk()`, which destroys all existing data.
- You **must** use a bitmap for tracking free disk blocks and inodes.
- Your filesystem will provide a flat namespace. There is only a single root directory and no subdirectories.
- You **must** use an inode-based block allocation strategy for files. Use **13** 16-bit direct block numbers and a single indirect block number in the inode. You do not have to implement double and triple indirect blocks in the inode.
- Your filesystem must gracefully handle out-of-space errors (e.g., operations that overflow the capacity of the software disk) and set appropriate error conditions.

- Your filesystem must support filenames of at least 256 characters in length. These are NULL-terminated strings composed of printable ASCII characters.
- Your implementation **does not** have to be thread-safe for full credit, but it's not a bad idea to make it thread-safe if you have time—it's good experience. If you want to do this, consider using pthreads read/write locks.
- Pay very special attention to the error conditions outlined in the `FSError` definition below! For example, you must catch attempts to open a file that is already open, to delete a file that is currently open, etc.
- Again, your filesystem MUST use the interface in `filesystem.h`. You may NOT change `filesystem.h`. You are implementing the `filesystem.c`.

What Do I Get?

An implementation of the software disk is provided. Do not implement this yourself and do not make modifications. You can obtain `softwaredisk.h`, `softwaredisk.c`, and `filesystem.h` from Moodle. The program `exercisesoftwaredisk.c` (available in the same place) tests the functionality of the software disk and illiustrates how to use the software disk API.

Using the Filesystem API

When you create test cases, compile like this:

```
$ gcc -g -o testcase testcase.c filesystem.c softwaredisk.c
```

Submission/Grading

In addition to your implementation in (`filesystem.c`, `formatfs.c`), you must submit a concise, typed design document describing your filesystem's physical layout on the software disk. This should include a description of which blocks are used to track metadata such as the bitmaps, inodes, etc., how the directory structure is maintained, etc. You should also document any implementation-specific limits (such as limits on the size of filenames, maximum number of files, etc.). Please name your design document `filesystem_design.pdf` and include it with your submission to `classes.csc.lsu.edu`.

A good grade on this assignment depends on a proper design for your filesystem, your filesystem working flawlessly, and on high-quality, well-designed code.