LOUISIANA STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CSC 4103: Operating Systems
Fall 2018
Programming Assignment # 3: A Simple Filesystem
Prof. Golden G. Richard III
Due Date: 11/26/2017 @ class time
** TEAMS OF 1 or 2 STUDENTS ARE ALLOWED **
NO LATE SUBMISSIONS
(AND START RIGHT AWAY)

The Mission

In this assignment you will implement a simple filesystem from scratch. Your filesystem will provide a flat namespace (e.g., there is only a single root directory and no subdirectories) and a set of file operations. You may implement either a FAT or *inode*-based block allocation strategy for files, but you must document which strategy is used. I will provide the implementation of a persistent "raw" software disk (available via `softwaredisk.h / softwaredisk.c`), which supports reads/writes of fixed-sized blocks (numbered `0..software_disk_size() - 1`). Your goal is to wrap a higher-level filesystem interface around my software disk. It is your responsibility to allocate blocks for file allocation, the directory structure, and file data. You may use any sensible technique for tracking free disk blocks.

Further 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.
- 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.
- Your filesystem **must** handle out of space errors (e.g., operations which overflow the capacity of the software disk) and set appropriate error conditions.
- Your filesystem **must** allow filenames of at least 255 characters in length, composed of printable ASCII characters.
- Your filesystem **must** provide the following filesystem interface. This is the interface expected by the sample applications—if your code deviates from this interface, it it is incorrect and applications will break:

filesystem.h:

```
// main private file type: you implement this in filesystem.c
struct FileInternals;

// file type used by user code
typedef struct FileInternals* File;

// access mode for open_file() and create_file()
typedef enum {
      READ_ONLY, READ_WRITE
} FileMode;

// error codes set in global 'fserror' by filesystem functions
typedef enum  {
      FS_NONE,
      FS_FILE_OUT_OF_SPACE,    // the operation caused the software disk to fill up
      FS_FILE_NOT_OPEN,        // attempted read/write/close/etc. on file that isn't open
      FS_FILE_OPEN,            // file is already open. Concurrent opens are not
                               // supported and neither is deleting a file that is open.
      FS_FILE_NOT_FOUND,       // attempted open or delete of file that doesn't exist
      FS_FILE_READ_ONLY,       // attempted write to file opened for READ_ONLY
      FS_FILE_ALREADY_EXISTS,  // attempted creation of file with existing name
      FS_EXCEEDS_MAX_FILE_SIZE,// seek or write would exceed max file size
      FS_ILLEGAL_FILENAME,     // filename begins with a null character
      FS_IO_ERROR              // something really bad happened
} FSError;

// function prototypes for filesystem API

// open existing file with pathname 'name' and access mode 'mode'.  Current file
// position is set at byte 0.  Returns NULL on error. Always sets 'fserror' global.
File open_file(char *name, FileMode mode);

// create and open new file with pathname 'name' and access mode 'mode'.  Current file
// position is set at byte 0.  Returns NULL on error. Always sets 'fserror' global.
File create_file(char *name, FileMode mode);

// close 'file'.  Always sets 'fserror' global.
void close_file(File file);

// read at most 'numbytes' of data from 'file' into 'buf', starting at the
// current file position.  Returns the number of bytes read. If end of file is reached,
// then a return value less than 'numbytes' signals this condition. Always sets
// 'fserror' global.
unsigned long read_file(File file, void *buf, unsigned long numbytes);

// write 'numbytes' of data from 'buf' into 'file' at the current file position.
// Returns the number of bytes written. On an out of space error, the return value may be
// less than 'numbytes'.  Always sets 'fserror' global.
unsigned long write_file(File file, void *buf, unsigned long numbytes);

// sets current position in file to 'bytepos', always relative to the beginning of file.
// Seeks past the current end of file should extend the file. Returns 1 on success and 0
// on failure.  Always sets 'fserror' global.
int seek_file(File file, unsigned long bytepos);

// returns the current length of the file in bytes. Always sets 'fserror' global.
unsigned long file_length(File file);

// deletes the file named 'name', if it exists. Returns 1 on success, 0 on failure.
// Always sets 'fserror' global.
int delete_file(char *name);
```

```
// determines if a file with 'name' exists and returns 1 if it exists, otherwise 0.
// Always sets 'fserror' global.
int file_exists(char *name);

// describe current filesystem error code by printing a descriptive message to standard
// error.
void fs_print_error(void);


// filesystem error code set (set by each filesystem function)
extern FSError fserror;
```

The `softwaredisk` upon which you will build your simple filesystem obeys the following interface. You are not allowed to modify the software disk interface or implementation:

softwaredisk.h:

```
#define SOFTWARE_DISK_BLOCK_SIZE 512

// software disk error codes
typedef enum  {
      SD_NONE,
      SD_NOT_INIT,              // software disk not initialized
      SD_ILLEGAL_BLOCK_NUMBER,  // specified block number exceeds size of software disk
      SD_INTERNAL_ERROR         // the software disk has failed
} SDError;

// function prototypes for software disk API

// initializes the software disk to all zeros, destroying any existing
// data.  Returns 1 on success, otherwise 0. Always sets global 'sderror'.
int init_software_disk();

// returns the size of the SoftwareDisk in multiples of SOFTWARE_DISK_BLOCK_SIZE
unsigned long software_disk_size();

// writes a block of data from 'buf' at location 'blocknum'.  Blocks are numbered
// from 0.  The buffer 'buf' must be of size SOFTWARE_DISK_BLOCK_SIZE.  Returns 1
// on success or 0 on failure.  Always sets global 'sderror'.
int write_sd_block(void *buf, unsigned long blocknum);

// reads a block of data into 'buf' from location 'blocknum'.  Blocks are numbered
// from 0.  The buffer 'buf' must be of size SOFTWARE_DISK_BLOCK_SIZE.  Returns 1
// on success or 0 on failure.  Always sets global 'sderror'.
int read_sd_block(void *buf, unsigned long blocknum);

// describe current software disk error code by printing a descriptive message to
// standard error.
void sd_print_error(void);

// software disk  error code set (set by each software disk function).
extern SDError sderror;
```

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.

Submission/Grading

In addition to your implementation in (`filesystem.h`, `filesystem.c`, `formatfs.c`), you must submit a concise, typed design document that describes the physical layout of your filesystem on the software disk. This should include a description of which blocks are used for file allocation, how free blocks are tracked, 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.