LOUISIANA STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

# CSC 4103: Operating Systems
## Prof. Aisha Ali-Gombe

**Programming Assignment # 4: A Simple Filesystem**
**Assigned – April 8th, 2025 - Due Date: on or before May 6th @ noon**
**Instruction – Students are strongly encouraged to work with a PA buddy but can also to work on their own. Only one student can submit on behalf of their team**
**NO LATE SUBMISSIONS will be accepted**

## The Mission

In this assignment you will implement a simple filesystem from scratch in a file name `filesystem.c`. 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 the software disk implementation. It is your responsibility to implement –

(a) an API for reading, writing…. and all other file operations (see below in `filesystem.h`)
(b) track files that are created
(c) allocate blocks for file allocation
(d) the directory structure
(e) file data

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 which 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 nul-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. To see how to use the pthreads library to do this, examine the code for `prioque.c`, which you used in the last assignment.
- 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.
- 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:

```
#if ! defined(__FILESYSTEM_4103_H__)
#define __FILESYSTEM_4103_H__

// private
struct FileInternals;

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

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

// error codes set in global 'fserror' by filesystem functions
typedef enum  {
  FS_NONE,
  FS_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 to 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 (implied) access
// mode READ_WRITE.  Current file position is set to byte 0.  Returns
// NULL on error. Always sets 'fserror' global.
File create_file(char *name);

// 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);

// extra function to make sure structure alignment, data structure
// sizes, etc. on target platform are correct.  Should return 1 on
// success, 0 on failure.  This should be used in disk initialization
// to ensure that everything will work correctly.
int check_structure_alignment(void);

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

#endif
```

## What Do I Get?

An implementation of the software disk is provided. Do not implement this yourself and do not make modifications. The code for `softwaredisk.h`, `softwaredisk.c`, `filesystem.h` and `testfs0.c` and `testfs4a.c` (as test cases) are available 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.

The software disk uses the following API, defined in:

softwaredisk.h:

```
#if ! defined(SOFTWARE_DISK_BLOCK_SIZE)
#define SOFTWARE_DISK_BLOCK_SIZE 4096

// 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;
#endif
```

## Using the API

Compile the test cases (to perform file operations like reading, writing, deleting etc) like this:

```
$ gcc -g -o testfs0 testfs0.c filesystem.c softwaredisk.c
```

## Filesystem Implementation Cheatsheet

Your software disk has 4096 block, each block has 4096 bytes. See screenshot below for block apportionment in the `filesystem.c` implementation.

```
#define MAX_FILES                   512
#define DATA_BITMAP_BLOCK             0  // max SOFTWARE_DISK_BLOCK_SIZE*8 bits
#define INODE_BITMAP_BLOCK            1  // max SOFTWARE_DISK_BLOCK_SIZE*8 bits
#define FIRST_INODE_BLOCK             2
#define LAST_INODE_BLOCK              5  // 128 inodes per block,
                                         // max of 4*128 = 512 inodes, thus 512 files
#define INODES_PER_BLOCK            128
#define FIRST_DIR_ENTRY_BLOCK         6
#define LAST_DIR_ENTRY_BLOCK         69
#define DIR_ENTRIES_PER_BLOCK         8 //8 DE per block
                                       //max 0f 64*8 = 512 Directory entries corresponding to max file for single-level directory structure
#define FIRST_DATA_BLOCK             70
#define LAST_DATA_BLOCK            4095
#define MAX_FILENAME_SIZE           507
#define NUM_DIRECT_INODE_BLOCKS      13
#define NUM_SINGLE_INDIRECT_BLOCKS   (SOFTWARE_DISK_BLOCK_SIZE / sizeof(uint16_t))

#define MAX_FILE_SIZE  (NUM_DIRECT_INODE_BLOCKS + NUM_SINGLE_INDIRECT_BLOCKS) * SOFTWARE_DISK_BLOCK_SIZE
```

## 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 to track metadata such as the bitmaps, inodes, et al, 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.