

## 1 Instructions

- All programs should be written in C (C++ is acceptable, but you may not use any object oriented feature, i.e. classes, inheritance, etc.). It is your responsibility to make sure your code compiles and runs correctly on a **linux** system.
- You may assume the input is valid, and no input checks are needed.
- Submit your source code file(s) through blackboard by the due date at 11:59PM. If more than one submission, only the last one will be graded.
- Only one member of each team needs to submit.

## 2 The Assignment

In this project, you will build a user-level library, libFS, **that implements a simple file system.** Your file system will be built inside of a library that applications can link with to access files and directories. Your library will in turn link with a layer that implements a "disk"; we provide this library, LibDisk, which you **must** use.

### 2.1 LibFS Specification

Here is a description of the LibFS API to the file system. There are three parts to the API: two generic file system calls, a set of calls that deal with file access, and a set of calls that deal with directories.

Applications (e.g., your own test applications, and certainly our test applications) will link with LibFS in order to test out your file system. Your library will be tested on its functionality and also particularly on how it handles errors. When an error occurs (possible errors are specified below in the API definition), your library should set the global variable `osErrno` to the error described in the API definition below and return the proper error code. This way, applications that link with your library have a way to see what caused the error.

#### 2.1.1 Generic File System API

- `int FS_Boot(char *path)` should be called exactly once before any other LibFS functions are called. It takes a single argument, the path, which either points to a real file where your "disk image" is stored or to a file that does not yet exist and which must be created to hold a new disk image. Upon success, return 0. Upon failure, return -1 and set `osErrno` to `E_GENERAL`.
- `int FS_Sync()` makes sure the contents of the file system are stored persistently on disk. More details on how this is accomplished using libDisk are available below. Upon success, return 0. Upon failure, return -1 and set `osErrno` to `E_GENERAL`.

#### 2.1.2 File Access API

Note that a number of these operations deal with pathnames. To simplify, we will make a few assumptions relating to pathnames. All pathnames are absolute. That is, anytime a file or directory is specified, the full path starting at the root is expected. Also, assume that the maximum length

of a file name is 16 bytes (15 characters plus one for an end-of-string delimiter). Finally, assume that the maximum length of a path is 256 total characters.

- `int File_Create(char *file)` creates a new file of the name pointed to by *file*. If the file already exists, you should return -1 and set `osErrno` to `E_CREATE`. Note: the file should not be "open" after the create call. Rather, `File_Create()` should simply create a new file on disk of size 0. Upon success, return 0. Upon a failure, return -1 and set `osErrno` to `E_CREATE`.
- `int File_Open(char *file)` opens up a file (whose name is pointed to by *file*) and returns a non-negative integer file descriptor, which can be used to read or write data to that file. If the file doesn't exist, return -1 and set `osErrno` should be set to `E_NO_SUCH_FILE`. If there are already a maximum number of files open, return -1 and set `osErrno` to `E_TOO_MANY_OPEN_FILES`.
- `int File_Read(int fd, void *buffer, int size)` should read *size* bytes from the file referenced by the file descriptor *fd*. The data should be read into the buffer pointed to by *buffer*. All reads should begin at the current location of the file pointer, and file pointer should be updated after the read to the new location. If the file is not open, return -1, and set `osErrno` to `E_BAD_FD`. If the file is open, the number of bytes actually read should be returned, which can be less than or equal to *size*. (The number could be less than the requested bytes because the end of the file could be reached) If the file pointer is already at the end of the file, zero should be returned, even under repeated calls to `File_Read()`.
- `int File_Write(int fd, void *buffer, int size)` should write *size* bytes from *buffer* into the file referenced by *fd*. All writes should begin at the current location of the file pointer and the file pointer should be updated after the write to its current location plus *size*. Note that writes are the only way to extend the size of a file. If the file is not open, return -1 and set `osErrno` to `E_BAD_FD`. Upon success of the write, all of the data should be written out to disk and the value of *size* should be returned. If the write cannot complete (due to a lack of space), return -1 and set `osErrno` to `E_NO_SPACE`. Finally, if the file exceeds the maximum file size, you should return -1 and set `osErrno` to `E_FILE_TOO_BIG`.
- `int File_Seek(int fd, int offset)` should update the current location of the file pointer. The new location is given as an *offset* from the beginning of the file. If *offset* is larger than the size of the file or negative, return -1 and set `osErrno` to `E_SEEK_OUT_OF_BOUNDS`. If the file is not currently open, return -1 and set `osErrno` to `E_BAD_FD`. Upon success, return the new location of the file pointer.
- `int File_Close(int fd)` closes the file referred to by file descriptor *fd*. If the file is not currently open, return -1 and set `osErrno` to `E_BAD_FD`. Upon success, return 0.
- `int File_Unlink(char *file)` this should delete the file referenced by *file*, including removing its name from the directory it is in, and freeing up any data blocks and inodes that the file was using. If the file does not currently exist, return -1 and set `osErrno` to `E_NO_SUCH_FILE`. If the file is currently open, return -1 and set `osErrno` to `E_FILE_IN_USE` (and do NOT delete the file). Upon success, return 0.

### 2.1.3 Directory API

- `int Dir_Create(char *path)` creates a new directory as named by *path*. Again, in this assignment, all paths are absolute paths, i.e., you can assume that you don't have to track the

current working directory or anything similar. Creating a new directory takes a number of steps: first, you have to allocate a new file (of type directory), and then you have to add a new directory entry in the current directory's parent. Upon failure of any sort, return -1 and set `osErrno` to `E_CREATE`. Upon success, return 0. Note that for simplicity `Dir_Create()` is not recursive - that is, if only `"/"` exists, and you want to create a directory `"/a/b/"`, you must first create `"/a"`, and then create `"/a/b"`.

- `int Dir_Size(char *path)` returns the number of bytes in the directory referred to by *path*. This routine should be used to find the size of the directory before calling `Dir_Read()` (described below) to find the contents of the directory.
- `int Dir_Read(char *path, void *buffer, int size)` can be used to read the contents of a directory. It should return in the *buffer* a set of directory entries. Each entry is of size 20 bytes, and contains 16-byte names of the directories and files within the directory named by *path*, followed by the 4-byte integer inode number. If *size* is not big enough to contain all of the entries, return -1 and set `osErrno` to `E_BUFFER_TOO_SMALL`. Otherwise, read the data into the buffer, and return the number of directory entries written.
- `int Dir_Unlink(char *path)` removes a directory referred to by *path*, freeing up its inode and data blocks, and removing its entry from the parent directory. Upon success, return 0. Note: `Dir_Unlink()` should only be successful if there are no files or directories within the directory. If there are still files within the directory, return -1 and set `osErrno` to `E_DIR_NOT_EMPTY`. If someone tries to remove the root directory (`"/"`), don't allow it!!! Return -1 and set `osErrno` to `E_ROOT_DIR`.

## 2.2 Notes


When reading or writing a file, you will have to implement a notion of a current file pointer. The idea here is simple: after opening a file, the current file pointer is set to the beginning of the file (byte 0). If the user then reads *N* bytes from the file, the current file pointer should be updated to *N*. Another read of *M* bytes will return the bytes starting at offset *N* in the file, and up to bytes *N+M*. Thus, by repeatedly calling `read` (or `write`), a program can read (or write) the entire file. Of course, `File_Seek()` explicitly changes the location of the file pointer to given location.

Note that you do not need to worry about implementing any functionality that has to do with relative pathnames. In other words, all pathnames will be absolute paths. Thus, all pathnames given to any of your file and directory APIs will be full ones starting at the root of the file system, i.e., `/a/b/c/foo.c`. Thus, your file system does not need to track any notion of a "current working directory".

## 3 Implementation

### 3.1 The Disk Abstraction

One of the first questions you might ask is "Where am I going to store all of the file system data?" A real file system would store it all on disk, but since we are writing this all at user-level, we will store it all in a "fake" disk, provided to you. In `LibDisk.c` and `LibDisk.h` you will find the implementation of a "disk" that you need to interact with for this assignment.



The "disk" that we provide presents you with `NUM_SECTORS` sectors, each of size `SECTOR_SIZE`. (These are defined as constants in `LibDisk.h`) Thus, you will need to use these values in your file system structures. The model of the disk is quite simple: in general, the file system will perform disk reads and disk writes to read or write a sector of the disk. In actuality, the disk reads and writes access an in-memory array for the data; other aspects of the disk API allow you to save the contents of your file system to a regular Linux file, and later, restore the file system from that file.



## 3.2 Disk API

- `int Disk_Init()` should be called exactly once by your OS before any other disk operations take place.
- `int Disk_Load(char* file)` is called to load the contents of a file system in *file* into memory. This routine (and `Disk_Init()` before it) will probably be executed once by your library when it is "booting", i.e., during `FS_Boot()`.
- `int Disk_Save(char* file)` saves the current in-memory view of the disk to a file named *file*. This routine will be used to save the contents of your "disk" to a real file, so that you can later "boot" off of it again. This routine will likely be invoked by `FS_Sync()`.
- `int Disk_Write(int sector, char* buffer)` writes the data in *buffer* to the sector specified by *sector*. The buffer is assumed to be of size `SECTOR_SIZE` exactly.
- `int Disk_Read(int sector, char* buffer)` reads a sector from *sector* into the buffer specified by *buffer*. As with `Disk_Write()`, the buffer is assumed to be of correct size.

For all of the disk API: All of these operations return 0 upon success, and -1 upon failure. If there is a failure, `diskErrno` is set to some appropriate value - check the code in `LibDisk.c` for details.


## 3.3 On-Disk Data Structures

A big part of understanding a file system is understanding its data structures. Of course, there are many possibilities. Below is a simple approach, which may be a good starting point.


### 3.3.1 superblock

First, somewhere on disk you need to record some generic information about the file system, in a block called the superblock. This should be in a well-known position on disk – in this case, make it the very first block. For this assignment, you don't need to record much there. In fact, you should record exactly one thing in the superblock – a magic number. Pick any number you like, and when you initialize a new file system (as described in the booting up section below), write the magic number into the super block. Then, when you boot up with this same file system again, make sure that when you read that superblock, the magic number is there. If it's not there, assume this is a corrupted file system (and that you can't use it).

### 3.3.2 inode blocks and data blocks



To track directories and files, you might need two types of blocks: inode blocks and data blocks. First, let's examine inodes. In each inode, you need to track several things about each file. First, you should track the size of the file. Second, you need to track the type of the file (normal or directory). Third, you should track which file blocks are allocated to the file. For this assignment, you can assume that the maximum file size is 30 blocks. Thus, each inode should contain 1 integer (size), 1 integer (type), and 30 integers (pointing to data blocks locations). You might also notice that each inode is likely to be smaller than the size of a disk sector – thus, you should put multiple inodes within each disk sector to save space.



Second, there are data blocks. Assume that each data block is the exact same size as a disk sector. Thus, part of disk must be dedicated to these blocks. Of course, you also have to track which inodes have been allocated, and which data blocks have been allocated. To do this, you should probably use a bit map for each, i.e., the first block after the superblock should be the inode bitmap, and the second block after the superblock should be the data block bitmap.

One painful part about any file system is pathname lookup. Specifically, when you wish to open a file named /foo/bar/file.c, first you have to look in the root directory ("/"), and see if there is a directory in there called "foo". To do this, you start with the root inode number (which should be a well-known number, like 0), and read the root inode in. This will tell you how to find the data for the root directory, which you should then read in, and look for foo in. If foo is in the root directory, you need to find its inode number (which should also be recorded in the directory entry). From the inode number, you should be able to figure out exactly which block to read from the inode portion of the disk to read foo's inode. Once you have read the data within foo, you will have to check to see if a directory "bar" is in there, and repeat the process. Finally, you will get to "file.c", whose inode you can read in, and from there you will get ready to do reads and writes.

### 3.3.3 Open File Table

When a process opens a file, first you will perform a path lookup. At the end of the lookup, though, you will need to keep some information around in order to be able to read and write the file efficiently (without repeatedly doing path lookups). This information should be kept in an open file table. When a process opens a file, you should allocate it the first open entry in this table – thus, the first open file should get the first open slot, and return a file descriptor of 0. The second opened file (if the first is still open) should return a descriptor of 1, and so forth. Each entry of the table should track what you need to know about the file to efficiently read or write to it – think about what this means and design your table accordingly. It is OK to limit the size of your table to a fixed size.

## 3.4 Disk Persistence

The disk abstraction provided to you above keeps data in memory until `Disk.Save()` is called. Thus, you will need to call `Disk.Save()` to make the file system image persistent. A real OS commits data to disk quite frequently, in order to guarantee that data is not lost. However, in this assignment, you only need to do this when `FS.Sync()` is called by the application which links with your `LibFS`. Still, you do need to call `Disk.Read` and `Disk.Write` for every `File.Read`, `File.Write`, and other file system operations that interact with the disk.

### 3.5 Booting Up

When "booting" your OS (i.e., starting it up), you will pass a file name that is the name of your "disk". That is, it is the name of the linux file that holds the contents of your simulated disk. If the file exists, you will want to load it (via `Disk_Load()`), and then check and make sure that it is a valid disk. For example, the file size should be equal to `NUM_SECTORS` multiplied by `SECTOR_SIZE`, and the superblock should have the information you expect to be in there (as described above). If any of those pieces of information are incorrect, you should report an error and exit.

However, there is one other situation: if the disk file does not exist, this means you should create a new disk, initialize its superblock and other structures, and create an empty root directory in the file system. Thus, in this case, you should use `Disk_Init()` followed by a few `Disk_Write()` operations to initialize the disk, and then a `Disk_Save()` to commit those changes to disk.

### 3.6 Other Notes

- Caching: Your file system should not perform any caching. That is, all operations should read and write the Disk API.
- Directories: Treat a directory as a "special" type of file that happens to contain directory information. Thus, you will have to have a bit in your inode that tells you whether the file is a normal file or a directory. Keep your directory format simple: a fixed 16-byte field for the name, and a 4-byte entry as the inode number.
- Maximum file size: 30 sectors. If a program tries to grow a file (or a directory) beyond this size, it should fail. This can be used to keep your inode quite simple: keep 30 disk pointers in each inode. You don't have to worry about indirect pointers or anything like that (that a real file system would have to deal with).
- Maximum element length in pathname: 16 characters. This means 15 characters plus one for an end-of-string delimiter. You don't have to worry about supporting long file names or anything fancy like that. Thus, keep it simple and reserve 16 bytes for each name entry in a directory.
- The maximum length of a path: 256 characters. Including the file's name
- If `File_Write()` only partially succeeds (i.e. some of the file got written out, but then the disk ran out of space), it is OK to return -1 and set `osErrno` appropriately.
- You should not allow a directory and file name conflict in the same directory (i.e. a file and a directory of the same name in the same directory).
- The maximum number of open files is 256.
- Filename: legal characters for a file name include letters (case sensitive), numbers, dots ("."), dashes ("-"), and underscores ("\_").
- You should enforce a maximum limit of 1000 files/directories. Define a constant internal to your OS called `MAX_FILES` and make sure you observe this limit when allocating new files or directories (1000 total files and directories).

## 4 Provided Materials (Summary)

The following files are provided here for you.

- LibDisk.c The source file for the disk abstraction
- LibDisk.h The header file for the disk abstraction
- Make.LibDisk An example Makefile for LibDisk
- LibFS.h The LibFS header
- LibFS.c The LibFS source file (skeleton, that you need to edit)
- Make.LibFS An example Makefile for LibFS
- main.c An example main.c
- Make.main An example Makefile

Note: You MUST NOT change a single line of code in LibDisk. You MUST NOT change a thing about the interface of LibFS (as defined in LibFS.h).

Note: To use a makefile not named "makefile" or "Makefile", just type "make -f Make.main" (for example).

## 5 Submission and grading

Your implementation will be graded along three main axes:

Functionality: Approximately 70% of your project grade will be devoted to how well your implementation matches the specification above; that is, this part of your grade depends upon correctly implementing the specified functions.

Documentation and Style: Approximately 20% of your project grade will be for the documentation in your README file and the style and comments of your code.

Application: Approximately 10% of the grade will be for two interesting applications you will write for your library. This means you need to supply two c files with a main function. Each links to your library, and use it to perform certain file system operation. Examples could be: linux commands ls, cp, mv (with or without copying), or anything else you may think of. Grade will be given for creativity and quality of implementation.

You need to submit only one copy of the project (your source code, a README file, and a Makefile) through blackboard. Do not submit any .o files.

In your README file you should have the following five sections:

The names and id for all team members.

A brief description of how you divided the work between you.

Design overview: A few simple paragraphs describing the overall structure of your code and any important structures and algorithms. For example, include a brief description of the data structures you use to map between addresses and memory objects, a brief description of the policy that use to perform allocations (e.g., first-fit, best-fit, rotating first-fit, buddy, etc).

Known bugs or problems: A list of any features that you did not implement or that you know are not working correctly

**Remember: No late projects will be accepted!**