Project 2: File System Management

COP 5614 – Operating Systems Principles

The original design of this assignment can be accredited to Prof. Andrea C. Arpaci-Dusseau at University of Wisconsin-Madison. Modifications were made to fit with the current course whenever appropriate.

You should submit the required deliverable materials on Canvas by **11:55pm, November 29, 2019.**

1. Objectives:

1. To understand how does file system work, specifically the directory hierarchy and storage management.
2. To understand some of the performance issues file systems must deal with.

2. Overview:

In this project, we will build a user-level library, `libFS`, that implements a 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. The original system has been in the **provided material folder**.

The Disk Abstraction

One of the first questions you might ask is "Where am I going to store the file system?" 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. In `LibDisk.c` and `LibDisk.h` you will find the "disk" that you need to interact with for this assignment.

Note that you must NOT change anything in `LibDisk`. You should simply use it.

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.

Actually, 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.

Here is the basic disk API:

- **`int Disk_Init()`**
  Disk_Init() should be called exactly once by your OS before any other disk operations takeplace.
- **`int Disk_Load(char* file)`**
  Disk_Load() 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(), which we see later.
- **`int Disk_Save(char* file)`**

Disk_Save() 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 it again. This routine is expected be invoked by FS_Sync(), which we see later.

- **`int Disk_Write(int sector, char* buffer)`**
  Disk_Write() writes the data in *buffer* to the sector specified by *sector*. The buffer is assumed to be of size sector exactly.

- **`int Disk_Read(int sector, char* buffer)`**
  Disk_Read() reads a sector from *sector* into the buffer specified by buffer. As withDisk_Write(), the buffer is assumed to be of size sector exactly.

For all disk operations listed above, all functions return 0 upon success, and -1 upon failure. If there is a failure, **`diskErrno`** (a global variable) is set to a specific value to indicate the error.

You should read the source code, `LibDisk.c` and `LibDisk.h`, to better understand the implementation of the disk abstraction (note again that you are not allowed to make any changes to them).

You can run "`make -f Makefile.LibDisk`" to generate the shared library for disk.

LibFS Specification

We now describe the `LibFS` API to your file system. Applications (e.g., your own test applications, and certainly our test applications) will link with `LibFS` to test out your file system. Your library will be tested on the functions and on how it handles errors. When an error occurs (each possible error is 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 happened when noticing the error code returned from the functions.

The `LibFS` library are three parts to the API:

1) two generic file system functions

2) a set of functions that deal with file access

3) a set of functions that deal with directories

The prototype of these functions is defined in `LibFS.h`. You should not change anything inside this header file; it defines the user interface. Your code and our test code will be linked with your library and this header file defines the prototype of the API functions.

Generic File System API

- **`int FS_Boot(char *path)`**

  FS_Boot() 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 **osErrno** is set to E_GENERAL.

- **`int FS_Sync()`**
  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 **osErrno** is set to E_GENERAL.

File Access API

Note that several following operations deal with *pathnames***.** Here we make a few assumptions about them. First, all pathnames are ***absolute*****.** That is, anytime a file is specified, the full path starting at the root (/) is

expected. Second, the maximum name length of a single file name is 16 bytes (15 characters plus one for an end-of-string delimiter). Finally, the maximum length of a pathname is 256 total characters (including the end-of-string delimiter).

- **`int File_Create(char *file)`**

  File_Create() 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 -1and set **osErrno** to E_CREATE.

- **`int File_Open(char *file)`**
  File_Open() opens up a file (whose name is pointed to by *file*) and returns an integer file descriptor (a number greater than or equal to 0), which can be used to read or write data to that file. If the file doesn't exist, return -1 and set **osErrno** to E_NO_SUCH_FILE.

- **`int File_Read(int fd, void *buffer, int size)`**
  File_Read() 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)`**
  File_Write() should write *size* bytes from *buffer* and write them 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 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 on disk), return -1 and set **osErrno** to E_NO_SPACE. Finally, if the file exceeds the maximum file size, you should return -1and set **osErrno** to E_FILE_TOO_BIG.

- **`int File_Seek(int fd, int offset)`**
  File_Seek() should update the current location of the file pointer. The 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)`**
  File_Close() 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 function is the opposite of File_Create(). This function 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 has been 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.

Directory API

- **`int Dir_Create(char *path)`**

Dir_Create() creates a new directory as named by *path.* As we mentioned before, all paths must be absolute paths. Creating a new directory takes two steps: first, you must allocate a new file (of type directory), and then you have to add a new entry in the parent directory. Upon failure of any sort, return -1 and set **osErrno** to E_CREATE. Upon success, return 0.

Note that Dir_Create() is *not* recursive. For example, if only "/" exists, and you want to create a directory "/a/b/", one must first create "/a", and then create "/a/b", in two separate steps.

- `int Dir_Unlink(char *path)`
  Dir_Unlink() 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. If the directory does not currently exist, return -1 and set **osErrno** to E_NO_SUCH_DIR. Dir_Unlink() should only be successful if there are no files within the directory. If there are still files within the directory, return -1 and set **osErrno** to E_DIR_NOT_EMPTY. It's not allowed to remove the root directory ("/"), in which case the function should return -1 and set **osErrno** to E_ROOT_DIR.

- `int Dir_Size(char *path)`
  Dir_Size() returns the number of bytes in the directory referred to by *path.* This function 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)`
  Dir_Read() 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 files (or directories) within the directory named by *path,* followed by the 4-byte integer inode number.

  If *size* is not big enough to contain all 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 that are in the directory (e.g., 2 if there are two entries in the directory).

3. Implementation Hints

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. The disk can be divided into five parts.

First, we need to record some generic information about the file system installed on the disk. The information should be stored in a well-known position on disk -- in this case, make it the very first block (sector). We call it the *superblock***.** 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).

Second, we need to keep track of all the files and directories on the disk. Each file or directory corresponds to an **inode**. Each inode is a data structure that contains the necessary information about the file/directory (such as file size, type, etc.) On disk, these inodes are stored consecutively; therefore, we can basically refer to an inode by its index. We must track which inodes have been allocated. To do this, we can use a bitmap. For this assignment, since the file system has a maximum limit of 1000 files and directories, we only need

to use 1000 bits for the bitmap (that's 125 bytes). Depending on the sector size, the inode bit map may occupy one or more sectors.

Third, the content of the file or directory is stored in a data block. We assume that each data block is the exact same size as a disk sector. Like the above inode bitmap, we also need to a bitmap to track which sectors of the disk have been allocated. Depending on the disk size, the data block bitmap (or sector bitmap) may occupy one or more sectors. For example, if the disk has 10,000 sectors, that'll require 10,000 bits (or 1,250 bytes). If each sector is 512 bytes, it'd need 3 sectors to store the bit map.

Fourth, improvement of inode: we use bitmap to track which sectors of the disk have been allocated. There are different ways to complete the bitmap. You need implement a **static map (lookup table)** for **fast lookup**, which means the bitmap contains types with first i bits set. If you cannot implement the static map, you can use a simpler bitmap to complete this function. But it will have a 10% deduction.

Fifth, we need to store the inodes. In each inode, we need to track at least three things about each file. First, we should track the size of the file. Second, we need to track the type of the file (normal or directory). Third, we should track which data blocks got allocated to the file. For this assignment, you can assume that the maximum file size is 30 data blocks. Thus, each inode data structure should contain 1 integer (size), 1 integer (type), and 30 pointers to data blocks (the pointers here are integer sector numbers). You might also notice that each inode is likely to be smaller than the size of a disk sector -- thus, we should put multiple inodes in each disk sector to save space (there may be a small fragmentation with in a sector).

Last, the remaining disk sectors should be data blocks. There can be simply indexed by their sector numbers.

IMPORTANT: For this assignment, your file system should not perform any caching. That is, all operations should read/write from the disk using the LibDisk API (one disk sector at a time).

Booting Up

When "booting" your OS (i.e., starting it up), you will pass a filename that is the name of your "disk". That is, it is the name of the 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 equivalent to NUM_SECTORS times SECTOR_SIZE, and the superblock should have the information you expect to be in there (in our case, the correct magic number). If any of these 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 and initialize its superblock 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.

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, 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. You do need to call Disk_Read and Disk_Write for every File_Read, File_Write, and other FS operations that interact with the disk.

Directories and Files

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.

Maximum element length in path name: 16 characters. 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.

Pathname lookup. For instance, 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. 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.

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".

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 to be able to read and write the file efficiently (without repeatedly doing path lookups). This information should be kept in a **per-file 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. That is, the file description, which you use for File_Read, File_write, File_Seek, and File_Close, are actually the index to the open file table. 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. (For this assignment, we limit open files to 256.)

Current Read/Write Location of an Open File

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() exists to explicitly change the location of the file pointer.

Miscellaneous Notes

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).

If the maximum name length of a file is 16 bytes means 15 characters plus one for an end-of-string delimiter.

The maximum length of a path is 256 characters.

The maximum number of open files is 256.

Legal characters for a file name include letters (case sensitive), numbers, dots ("."), dashes ("-"), and underscores ("_").

You should enforce a maximum limit of 1000 files/directory. 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

The following files have been provided here for you (on Canvas):

- The source file for the disk abstraction: `LibDisk.c` (not allowed to change)
- The header file for the disk abstraction: `LibDisk.h` (not allowed to change)
- An example Makefile for LibDisk: `Makefile.LibDisk`
- The LibFS header: `LibFS.h` (not allowed to change)
- The LibFS source file: `LibFS.c` (this is the main file for you to change)
- An example Makefile for LibFS: `Makefile.LibFS`
- An example: `main.c`
- An example Makefile: `Makefile`
- Some simple programs for testing purpose

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

You should create your test program to make sure your LibFS can be as thoroughly tested as possible.

5. Deliverables

- A README file, which describes how we can compile and run your code;
- **A PDF report**, which identifies the group members and provides detailed description on the design and implementation of your file management system. Discussion of the output of running the test program with screenshots (screenshots should contain your personal identification of computer, system, or ID). **Individual contributions should be clearly identified in the report**, otherwise your group will lose 10 point at the final. Report should contain your name and pantherID.

- Your source code, including a Makefile.
    - Do not submit your binary code --- your grader will compile your code.
    - No grade will be given if your code fails to compile.
    - Provide sufficient comments in your code to help the TA understand your code; this is important in case you want to get partial credit when your submitted code does not work properly.
- Compress all the above into a single file and upload it into Canvas. Each team will submit one and only one such file.
- **Donot plagiarism!!!** We will do the code checking. Do not copy any contents or codes from other materials.

6. Grading Criteria

Your implementation will begraded along two main axes:

- Functionality: 75% of your project grade will be devoted to how well your implementation matches the specification above; this part of your grade depends upon correctly implementing the specified functions.
- Documentation and Style: Approximately 25% of your project grade will be for the documentation files and the style and comments of your code.