ARIZONA STATE UNIVERSITY

CSE 430 SLN 70967— **Operating Systems** — Fall 2014

Instructor: Dr. Violet R. Syrotiuk

Project #3

Available Tuesday 11/04/2014; due Tuesday 12/02/2014

This project has two objectives: (1) To understand how file systems work, specifically the directory hierarchy and storage management; and (2) To understand some of the performance issues related to file systems.

1 Overview

In this project, you will build a user-level library, `LibFS`, that implements a portion of a file system. Your file system is built inside a library that applications can link with to access files and directories. Your library in turn links with an implementation of a "disk;" this library, `LibDisk`, is provided to you.

2 LibFS Specification

There are three parts to the `LibFS` API to the file system: 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., test applications) link with `LibFS` in order to test out your file system.

Your library will be tested on how it functions and also on how it handles errors. When an error occurs, your library must set the global variable `osErrno` to the error described in the API definition and return the proper error code. This way, applications that link with your library have a way to see what happened when the error occurred.

2.1 Generic File System API

- `int FS_Boot(char *path)`
  `FS_Boot()` must 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()`
  `FS_Sync()` ensures the contents of the file system are stored persistently on disk. More details on how this is accomplished using `LibDisk` are described in §3.1. Upon success, return 0. Upon failure, return -1 and set `osErrno` to `E_GENERAL`.

2.2 File Access API

A number of the file access operations deal with pathnames. Therefore, we must make a few assumptions about pathnames. *All pathnames are absolute.* That is, any time a file is specified, the full path starting from the root is expected. Also, assume that the maximum name length of a single file name is 16 bytes (15 characters plus one for an end-of-string delimiter), and the maximum length of a path is 256 characters.

- `int File_Create(char *file)`
  `File_Create()` creates a new file of the name pointed to by `file`. Upon success, return 0. 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()` simply creates a new file on disk of size 0.

- `int File_Open(char *file)`
  `File_Open()` opens 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`. If there are already a maximum number of files open, return -1 and set `osErrno` to `E_TOO_MANY_OPEN_FILES`.

- `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`. 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_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.

2.3 Directory API

- `int Dir_Create(char *path)`
  `Dir_Create` creates a new directory as named by path. In this project, all paths are absolute paths. 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 `_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".

2.4 Notes

When reading or writing a file, you must 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.

3 Implementation Hints

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 this library is at user-level, we store it in a "fake" disk. The files `LibDisk.c` and `LibDisk.h` provide the "disk" that you need to interact with for this project.

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 must use these values in your file system structures. The model of the disk is simple: in general, the file system performs 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.

Here is the disk API:

- `int Disk_Init()`
  `Disk_Init()` must be called exactly once by your OS before any disk operations take place.

- `int Disk_Load(char* file)`
  `Disk_Load()` is called to load the contents of a file system in file into memory. This function (and `Disk_Init()` before it) must be executed once by your library when it is "booting", i.e., during `FS_Boot()`.

- `int Disk_Save(char* file)`
  `Disk_Save()` saves the current in-memory view of the disk to a file named `file`. This function is used to save the contents of your "disk" to a real file, so that you can later "boot" from it. This function is likely invoked by `FS_Sync()`.

- `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 exactly the size of a sector.

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

All of the disk APIs 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.

3.2 On-Disk Data Structures

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

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 project, 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).

To track directories and files, you might need two types of blocks: *inode* blocks and `data` blocks. In each inode, you need to track at least two 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 project, you can assume that the *maximum file size is 30 blocks*. Thus, each inode should contain 1 integer (size), 1 integer (type), and 30 pointers (to data blocks). You might also notice that each inode is likely to be smaller than the size of a disk sector — thus, you must put multiple inodes within each disk sector to save space.

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 and data blocks have been allocated. To do this, you must 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.*

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 is given 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 it's 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 Open File Table

When a process opens a file, first you will perform a path lookup. At the end of the lookup, though, you need to keep some information in order 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. 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. You may 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. That is, you 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 project, you only need to do this when `FS_Sync()` is called by the application which links with your LibFS. You need to call `Disk_Read` and `Disk_Write` for every `File_Read`, `File_Write`, and other file systems operations that interact with the disk.

3.5 Booting Up

When "booting" your OS, you 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 (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 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 `Disk_Write()` operations to initialize the disk, and then a `Disk_Save()` to commit those changes to disk.

3.6 Other Notes

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.

If `File_Write()` only partially succeeds (i.e. some of the file got written out, but then the disk ran out of space), 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.

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

The following files are provided for you: (1) The header and source files for the disk abstraction: `LibDisk.h` and `LibDisk.c`. (2) The header and source files for `LibFS`: `LibFS.h` and `LibFS.c` (3) An example `main.c`.

You must not change a single line of code in `LibDisk`. In addition, you must not change anything about the interface of `LibFS` (as defined in `LibFS.h`).

5 For Honours Credit or Bonus Credit

For honours or bonus credit, implement following file and directory API, respectively.

- `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 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 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 that are in the directory (e.g., 2 if there are two entries in the directory).

6 Hand-in instructions

Submit electronically, before 9:00am on Tuesday, 12/02/2014 using the submission link on Blackboard for Project #3, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following items:

Design and Analysis (30%): Provide a description of the methodology you followed to implement your `LibFS`. Specifically:

1. Provide a few paragraphs describing the overall structure of your code and any important data structures. For example, include brief description of the data structures you use to map between addresses and memory objects and a brief description of the policy that use to perform allocations (e.g., first-fit, best-fit, etc.).
2. Describe how you handled ambiguities, if any, in the specification.
3. A list of any features that you did not implement or that you know are not working correctly

Implementation (50%): Provide a makefile to compile your program, and your *documented* C/C++ source code implementing `LibFS`. Your grade will be related to how well your implementation matches the specification.

You may write your code only in C or C++. You must not alter the requirements of this project in any way.

Correctness (20%) Your code **must** run on `general.asu.edu` using a makefile you provide. Sample test applications will be provided.

[1] **Do not** use any other archiving program except `zip`.