This is a **C/C++ Language, UNIX**-based project to learn about file systems, and their structure.

Submission

Due Date: Sunday, 07/01/11 (11:59pm ET)

This submission should be done through the MyCourses dropbox. Note, this will not build your files upon submission; make sure it works, if I cannot compile it, you will not get credit. I will simply extract your project to its own folder and run *make.* Please zip up your submission (in .zip format before submitting it).

Please follow the file and program name requirements in this document and include all the files necessary to compile your program.

Attribution and Documentation

This is an individual effort project. You must do all the work yourself without assistance from other students.

You may find things online or in books that help you in certain areas; you need to document your sources. **This means you must acknowledge contributions from these other sources**. I expect that you will provide comments in each file containing information about the program, including all contributing sources.

Style, Documentation and Formatting

All functions, classes, and files must have comments; all parameters must be defined in the comments section. I am not terribly concerned with the format of your code, though it should be readable, variable names should make sense, and the code should be documented. If you still aren't sure if your code is properly formatted, ask me. Be sure to include your name in ALL source files.

Grading

This project is worth 100 points.

It is possible to earn up to 10 points extra credit on this assignment, details are below.

Project Resource Materials

There are several resources on the internet you can use for this project. I suggest using gmakemake to create your makefile. This should be self-explanatory, and will greatly assist your efforts.

Project Description

Overview

Will be the creation of a shell and a file system. Your file system must be able to create, delete and edit files. Files will be able to be copied within your file system, and to/from the native file system. File system management and performance of the system is of utmost importance.

Program I/O

A user starts the program by running 'os1shell' from the command line. The program will take up to one command line parameter, the name of a file system to open. If the file system does not exist, the user should be prompted to create a new file system. The file system structure details will be given below, but users should be able to manage files on this system in much the same way that they manage files on the native file system.

Requirements for the program are:

- Users can create a file system by giving a file system name on the command line when starting the shell
  - If the file system does not exist, users will be prompted to create it
  - If the file system already exists, it will be opened for the user, and the user will be placed in the root of that file system
- The maximum size for the file system is 50MB
- The minimum size for the file system is 5MB
- The minimum size for a cluster is 8KB
- The maximum size for a cluster is 16KB
- Cluster sizes should be even multiples of powers of 2 in KB (8K, 9K, 10K, etc), such that they are divisible by 128
- Users can execute the following commands on the file system, and all functions should work between the native and user file system, and any mixture of the two
  - ls – display a list of files in the current directory
  - touch – create a new 0 byte file
  - cp – copy a file from <source> to <destination>
  - mv – move a file from <source> to <destination>
  - rm – remove a file
  - df – show the structure of the file system
  - cat – display the contents of a file

Details

File System

A filesystem in a unix system can be implemented in many different ways. Linux, for example, has the ability to read and write nearly any file system on the market, and is capable of reading every commonly used file system. Each of these file system implementations is usually implemented in the kernel level code (often through a module), and the file I/O is then dealt with through an API call. In order to increase your understanding of how file systems are implemented, you will be creating a file system that is handled entirely within the shell, since you will not have access to implement a kernel level file system.

There are many types of file systems supported by systems today, though we will be focusing on a FAT like file system for this project. A FAT file system is composed of a File Allocation Table. The FAT32 file system is a common system used on many devices these days, including digital cameras. While the standard FAT16 and FAT32 are much more complicated than what you will need to implement, the basic structure will be used in this particular project.

In a FAT file system, there is a structure that maps file to the system's hard disk, indicates when clusters on the hard drive are bad, manages the files copying, deleting, moving and the access of applications to the system.

The original FAT File system was FAT12, and was used on floppy disks and hard drives up to 15MB in size. The FAT16 file system allowed for access to 2GB and FAT32 allows up to 2TB drives. In any of these systems, the FAT table is a fixed size, and the cluster size determines that maximum size of a disk. For example if we have 2000 records for files, and each cluster is 1k, we can have a 2MB drive. If each cluster is 8kb, we could have a 16MB drive.

When creating a file system, there is a compromise between the cluster size and performance. The large cluster size means than the performance will be faster as fewer lookups will need to occur (just because FAT16 supported 2GB did not mean that every drive was 2GB, a 500MB drive would have ¼ the number of entries of a 2GB drive with the same cluster size). A larger cluster size is not always ideal though, as only one file can occupy a cluster at a time. If you have a 8kb clusters, and are writing 1kb files, there will be 7kb of space that isn't used. This is a tradeoff that must be made when selecting cluster sizes, and is something you will need to consider when you create your file system. If a file is larger than a cluster, which is recorded in the fat table, there is a link to the next cluster used by the file, which may or not be the next cluster on the system.

A final point to consider when creating file systems is how to handle directories or folders. Directories are typically a hierarchical structure of virtual files that house files and other directories. In the FAT file systems, directories are special entries within the FAT table that give information about directories, including permissions, access rights, and last modified and access times. In other words, directories are just special cases of files, but remember, every file lives in a directory. The FAT file system has changed how they handle root file systems throughout the years, but it basic concept has remained the same.

For more detailed information about the FAT system, and for ideas about how to implement your project, the wikipeidia page on FAT file systems is quite comprehensive: http://en.wikipedia.org/wiki/File_Allocation_Table

The Program

You will be required to implement a file system for the shell you created in Project 1. Since the file system will reside entirely inside the shell, it will be a simplified version of a FAT system. You will need to support files and directories and management of those files and directories.

The os1shell create in project 1 should now take a command line parameter, which will be the name of the file system to open. If the file system does not exist, the user should be prompted for several parameters before the shell is opened, in the following manner:

Are you sure you want to create a new file system [Y]?

Enter the maximum size for this file system in MB [10]:

Enter the cluster size for this file system in KB [8]:

os1shell -> _

If the user specifies all the parameters for create a file system, it will be created on parent file system where it can be accessed from your shell. Be sure to take note of the note on page 5.

Once the File System has been created, it should consist of several pieces in order to be able to access the system.

The first thing you need is a Boot Record. This should reside at address 0 in your file system, and contain information such as the cluster size, the disc size and the location of the root directory table entry. Based on this structure, the first cluster will always be allocated to the structure of the file system.

Type	Name	Description
Unsigned int	Cluster Size	The size of the cluster you specified when creating the file system in bytes
Unsigned int	Size	The size of the disc in bytes
Unsigned int	Root Directory	Index to the cluster that stores the root directory
Unsigned int	FAT	Index to the cluster containing the FAT Table

The second thing you will need is a directory table. The directory table stores the information about files and directories in your file system. This should be stored in the first available cluster, and as long as there is free space available in the file system, it can grow to expand in the same way as a file. You should note that the directory table is 128 bytes large, and that a cluster is also a power of 2. **For performance reasons, it makes sense to always fill a cluster with directory entries and mark them as available for use, if they aren't used. This will make reading and writing of the directory tables much easier, and is a requirement for the project,**

The **struct**ure would then consist of multiple entries that look like the following

Type	Name	Length	Description
char	Name	112	Filename The first byte can have a special value Value: 0x00 — This entry is available for use

<table>
<tr><td></td><td></td><td></td><td><table><tr><td>0xFF</td><td>This entry is deleted</td></tr></table></td></tr>
<tr><td>Unsigned int</td><td>Index</td><td></td><td>The index of first cluster for this file or the location of a directory structure</td></tr>
<tr><td>Unsigned int</td><td>Size</td><td></td><td>The size of the file in bytes (0 for directories)</td></tr>
<tr><td>Unsigned int</td><td>Type</td><td></td><td>File attributes<table><tr><th>Value</th><th>Description</th></tr><tr><td>0x00</td><td>File</td></tr><tr><td>0xFF</td><td>Directory</td></tr></table></td></tr>
<tr><td>Unsigned int</td><td>Creation</td><td></td><td>The creation date of the file (unix epoch format)</td></tr>
</table>

A properly defined directory table will be defined as:

```
struct directory_table[MAX_FILES];
```

Aside from the directory table, you will need a File Allocation Table, which consists of an array – the index in the array corresponds to the cluster for that data, the 2nd element is a pointer to the next cluster containing data.

```
int FileAllocationTable[ClusterCount];
```

FileAllocationTable[i] is the reference to the next cluster in the series, and i is the index of cluster on the disk we are looking at

Some common value for FileAllocationTable[i] are:
0x0000 – a free cluster
0x0001 – 0xFFFD – the index of the next cluster in the chain
0xFFFE– a reserved cluster, do not use
0xFFFF – The last cluster in the chain

Note: When creating the file system, keep in mind that the entire FAT table must fit inside a single cluster. Since we have 1 record in our FAT for each cluster, and it is an array of integers, it should be trivial to find out if the FAT table will fit; if it won't, you should not allow the user to create the file system.

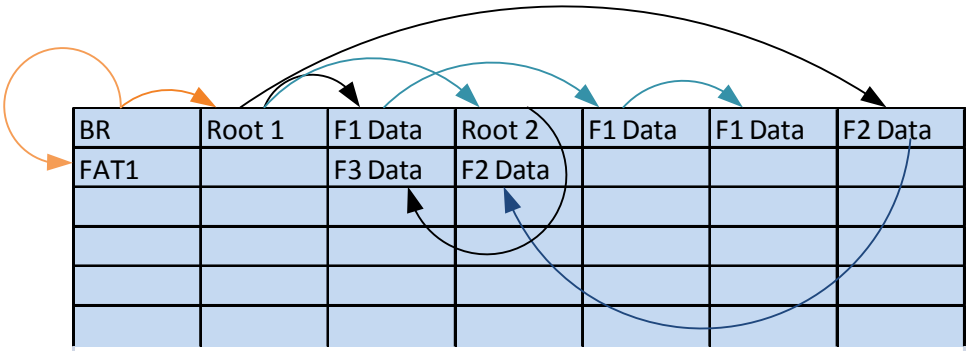
Given the above two items, we can read or write files with the simple code:

```
currentCluster = directory_table[i].index;

while (FileAllocationTable[currentCluster] != 0xFFFF)
{
      DataLocation = currentCluster;
      currectCluster = FileAllocationTable[currentCluster];
      // handle reading and writing here
}
```

You can keep track of free clusters by setting the FileAllocationTable[currentCluser] = 0x0000; Doing so indicates there is no data and that it can be used for data storage. It is critical that you keep track of where the directory table and FAT are stored (it should be stored in your file system) and ensure that you don't overwrite them. As shown here, the cluster count should be 0 indexed. The location of the boot record should always be stored in cluster 0.

Both the FAT and the Directory_Table must be read and written to disk as files. They should occupy clusters in the same manner as regular files, which means your file I/O routines will have to handle periodic reading and writing of these entries.



BR = Boot Record
Root - The Root File System, this can exist anywhere on the disk
F(x) Data - indicates data for a file on the system

You will be required to print the information in the FAT table, and the Directory table in a user readable format. It should be clear how the clusters are linked, and what information is in the directory table. The format of this output is up to you.

Commands

When you first start your file system, with the command: *os1shell myfs* you will have created a file system call myfs. Your mountpoint for this file system is /myfs. When you first start your shell, you should be in the file system, so an *ls* command will give a directory listing from the file system.

The cd functionality should work, and a valid command would be *cd /home/fac/jsb*. This is usually implemented by storing the current path in a local variable and passing that to functions such as ls, cat, touch and rm when calling execvp.

If I have executed a cd to change to a path out of my file system, I should be able to return to the file system by executing *cd /myfs* where myfs is the name of the file system specified on the command line. Commands should then be executed in the local file system.

Below is an example of a run of the program

-bash-3.2$ os1shell myfs

```
os1shell> cp /home/fac/jsb/a.txt /myfs/a.txt
os1shell> touch b.txt
os1shell> ls
            a.txt
```

```
b.txt
```

Other sample commands:

```
cp /home/fac/jsb/a.txt /myfs/a.txt // copy from parent fs to local fs
cp /myfs/a.txt /home/fac/jsb/a.txt // copy local fs to parent fs
cp /myfs/a.txt /myfs/b.txt // copy a file on the local file system
```

Sample Code

There is a starter filesystem class available to you.
http://cs.rit.edu/~jsb/fsystem.h
http://cs.rit.edu/~jsb/fsystem.cpp

YOU WILL BE REQUIRED TO BE ABLE TO READ AND WRITE ANOTHER STUDENT'S FILE SYSTEM. FAILURE TO DO SO WILL RESULT IN A LOSS OF POINTS!

Extra Credit

Most file systems allow users to create directories to house and organize files. In the above scenario, we only require one directory, and every file gets added to the same directory table. While this is a solution for this project, it is not an ideal solution for a real file system. Implementing directories is not terribly difficult and is simply done by creating a directory table that has a link in it to another directory table (instead of a file). This new entry will have a link back to the parent directory as well as any files that reside in the new directory. Moving files between directories means that an entry will need to be removed from one directory table and added to another. When removing an item from a directory entry, it is a matter of marking the first byte of the entry to 0x00 to indicate it is free to use. New entries should checking for entries with a 0x00 or 0xFF entry before creating a new one.

Directories, if implemented, must be done entirely within the confines of the project. No new tables may be created to store information, and user should be able to move, copy, cat, etc, any file within a directory. rm must also function and provide the ability to remove a directory, or rmdir must be implemented. If directories are to be implemented, wildcard support should also be implemented for file management (cp /directory/* /home/mydir/code/). This will be worth up to 10 points.

____ : (12 points) Creating a file system that is divided into clusters, with a size specified on the command line. You are able to open an existing file system by specifying its name on the command line

____ : (4 points) creation of a 0 byte file using 'touch'

____ : (8 points) copying/moving a file to and from the parent file system. Copying within the current file system.

____ : (8 points) correcting setting the deleted flag in the FAT table and not showing those entries, using rm

____ : (12 points) Correctly reading the FAT table and allowing files to span multiple, non-adjacent clusters

____ : (4 points) df will show the parameters of the current disk, including the cluster size and file system size

____ : (6 points) printing of the FAT table, and Directory Table in a user readable format

____ : (4 points) using correct functions to read/write the file on the disk

____ : (8 points) proper implementing of 'cat' to output a given file to standard out.

____ : (8 points) overall functionality. Does the application handle strange conditions and work as expected

____ : (6 points) Ability to read and write a standard file system format

____ : (12 points) design: modular architecture, clearly documented

____ : (8 points) style: consistency and coding standards usage

total is 100 points (before extra credit).

____ : (10 extra points) allow users to create, remove, rename and put files in sub directories.

____ : (10 extra points) submission of project before 02/13/11 at 11:59pm