CSI 385
Fall 2016

Semester Project

Assigned Date: Tuesday, Sept 6, 2016
Due Date: 23:59 EST, Tuesday, Nov 22, 2016

The coded solution to the following project is to be done by your team and only your team. You may ask help only from your teammate and your professor, but no one else. You may use your notes, texts, online tutorials, etc., but the code must be your own. Bear in mind that your professor is there to clarify information for you not to write your program!

Learning Objectives

In the process of completing this project, students will develop their abilities to:

- Specify, design, test and document the design of a file system, a key component of an operating system.
- Use shared memory as a means of inter-process communication.

Problem Statement

You will be working in a group of **two** members. As a team, you will design and implement the FAT12 file system. Also, your team must design a test plan to evaluate your system as carefully as possible. You will test and debug the system according to your test plan. Finally, you will demonstrate your system using an image of a floppy disk. Throughout the project, you will maintain up-to-date documentation of your design and test plan. At the end of the project, you must submit a copy of your design and test plan along with your complete project.

An overview of FAT12[†]

The File Allocation Table (FAT) is a table stored on a hard disk or floppy disk that indicates the status and location of all data clusters that are on the disk. The FAT can be considered to be the table of contents of a disk, and it is used by the operating system to find a file, store the number of segments that each file has, and the location of all the pieces. It also marks the sectors as used, full, or bad so that the system can determine where to place a file. If the FAT is damaged or lost, then a disk is unreadable.

For this project, the FAT12 file system is to be implemented. The number 12 is derived from the fact that the FAT consists of 12-bit entries. The storage space on a floppy disk is divided into units called sectors. In larger storage devices, a bunch of sectors form a cluster. However, for the floppy disk, the number of sectors in a cluster is one. Also, the size of a sector (and hence a cluster) is 512 bytes for a floppy disk.

1. Disk Organization

The floppy for FAT12 consists of four major sections: the boot sector, FAT tables, root directory, and data area as shown in Figure 1.

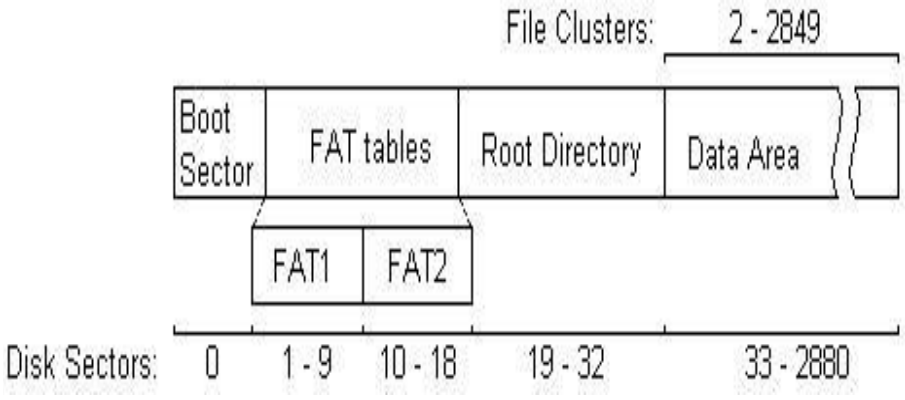

Figure 1 Disk organization of FAT12 file system[‡]

[†] This description was originally provided by Dr. Chidanandan of the Rose-Hulman Institute of Technology

[‡] This diagram is obtain from the course web site for CS 324 from Brigham Young University

- **Boot Sector:** The boot sector consists of the first sector (sector 0) on the volume or disk. The boot sector contains specific information on the organization of the file system, including how many copies of the FAT tables are present, how big a sector is, how many sectors are in a cluster, etc.

- **FAT:** FAT tables contain pointers to every cluster on the disk, and indicate the number of the next cluster in the current cluster chain, the end of the cluster chain, whether a cluster is empty, or has errors. The FAT tables are the only method of finding the location of files and directories on the disk. There are typically two redundant copies of the FAT table on disk for data security and recovery purposes. On a floppy, since a cluster consists of just one sector, there is a FAT entry pointer to every sector on the disk.

- **Root Directory:** The root directory is the primary directory of the disk. Unlike other directories located in the data area, the root directory has a finite size to restrict the total number of files or directories that can be created therein. The FAT12 system has 14 sectors, and each sector can have 16 entries. Thus, a FAT12 floppy can have 224 possible entries.

- **Data Area:** The first sector or cluster of the data area corresponds to cluster 2 of the file system (the first cluster is always cluster 2). The data area contains file and directory data and spans the remaining sectors on the disk.

A summary of the disk organization is given below:

Logical Sector	Content
0	Boot Sector
1	First sector in the (first) FAT
10	First sector in the second FAT
19	First sector in the floppy disk's root directory
XX	Last sector in the root directory (see bytes 17 and 18 in the boot sector)
XX + 1	Beginning of data area for the floppy disk

For FAT12, XX = 32 as 14 sectors are reserved for the root directory.

2. The Boot Sector

The boot sector exists at sector 0 of the disk and contains the basic disk geometry, which is the set of information needed by the operating system to use the disk correctly. Whenever the disk is used, the information from the boot sector is read and any needed information is extracted from it. The boot sector on a DOS formatted floppy is a sequence of bytes that looks as follows:

Starting Byte	Length (in bytes)	Stored Data
0	11	Ignore
11	2	Bytes per sector
13	1	Sectors per cluster
14	2	Number of reserved sectors
16	1	Number of FATs
17	2	Maximum number of root directory entries
19	2	Total sector count[1]
21	1	Ignore
22	2	Sectors per FAT
24	2	Sectors per track
26	2	Number of heads
28	4	Ignore
32	4	Total sector count for FAT32 (0 for FAT12 and FAT16)
36	2	Ignore
38	1	Boot signature[2]
39	4	Volume ID[3]
43	11	Volume label[4]
54	8	File system type[5] (e.g. FAT12, FAT16)
62	-	Rest of boot sector (ignore)

1. **Total Sector Count:** This field is the 16-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. For FAT12 and FAT16 volumes, this field contains the sector count. For FAT32, see bytes 32-35.

2. **Boot Signature:** Extended boot signature. This is a signature byte that indicates that the following three fields in the boot sector are present. The value should be 0x29 to indicate that.

3. **Volume ID:** Also the Volume serial number. This field, together with *volume label*, supports volume tracking on removable media. These values allow FAT file system drivers to detect that the wrong disk is inserted in a removable drive. This ID is usually generated by simply combining the current date and time into a 32-bit value.

4. **Volume Label:** This field matches the 11-byte volume label recorded in the root directory. NOTE: FAT file system drivers should make sure that they update this field when the volume label file in the root directory has its name changed or created. The setting for this field when there is no volume label is the string “NO NAME”.

5. **File System Type:** One of the strings “FAT12”, “FAT16”, or “FAT”. Many people think that the string in this field has something to do with the determination of what type of FAT, FAT12, FAT16, or FAT32, that the volume has. This is not true. This string is informational only and is not used by Microsoft file system drivers to determine FAT type because it is frequently not set correctly or is not present. This string should be set based on the FAT type though, because some non-Microsoft FAT file system drivers do look at it.

3. FAT (File Allocation Table)

The FAT, as stated earlier, is a data structure that maps the data sectors of the storage device. It is similar to an array and each entry in the FAT corresponds to a cluster of data on the disk. The values in each entry of the FAT that are of interest are:

- A value signifying that this data cluster is the last cluster of a file
- A value signifying that this data cluster is currently unused
- A value signifying where the NEXT data cluster of the current file is located. Specifically, the FAT entry values signify the following:

Value	Meaning
0x00	Unused
0xFF0-0xFF6	Reserved cluster
0xFF7	Bad cluster
0xFF8-0xFFF	Last cluster in a file
(anything else)	Number of the next cluster in the file

Translation from physical to logical data sector numbers:

- The FAT works off logical data sector values. For the FAT12 system, while determining the logical sector number from the physical sector number, the following two factors need to be taken into account:
  1. From the organization of the disk described earlier, the first 33 sectors are predefined. The actual data sector that holds user data does not exist in these first 33 sectors and starts at sector number 33 (remember we start with 0).

  2. The entries in positions 0 and 1 of the FAT are reserved. Therefore, it is the second entry of the FAT that actually contains the description for physical sector number 33.

- Therefore, physical sector number = 33 + FAT entry number – 2. For example, the fifth entry of the FAT would actually refer to physical data sector number 36.

4. Directories

Directories (such as the root directory) exist like files on the disk, in that they occupy one or more sectors. Each sector (512 bytes) of a directory contains 16 directory entries (each of which is 32 bytes long). Each directory entry describes and points to some file or subdirectory on the disk. Thus, the collection of directory entries for a directory specifies the files and subdirectories of that directory.

Each directory entry contains the following information about the file or subdirectory to which it points.

Offset (in bytes)	Length (in bytes)	Description
0	8	Filename
8	3	Extension
11	1	Attributes
12	2	Reserved
14	2	Creation Time
16	2	Creation Date
18	2	Last Access Date
20	2	Ignore in FAT12
22	2	Last Write Time
24	2	Last Write Date
26	2	First Logical Cluster
28	4	File Size (in bytes)

Note: We have already established that in the FAT12 system a cluster holds just one sector. Therefore, the two words are used interchangeably in the rest of this document.

Notes on directory entries:

1. The First Logical Cluster (FLC) field specifies where the file or subdirectory begins. Thus the directory entry points to a file or subdirectory. Note that it gives the value of the FAT index. For example, if the FLC value is 2, then it implies that the index to the FAT array should be 2, which is physical cluster 33 in the FAT12 system. If the value of the FLC is 0, then it refers to the first cluster of the root directory and that directory entry is therefore describing the root directory. Keep in mind that the root directory is listed as the ".." entry i.e. the parent directory in all its sub-directories.

2. If the first byte of the *Filename* field is 0xE5, then the directory entry is free (i.e., currently unused), and hence there is no file or subdirectory associated with the directory entry.

3. If the first byte of the *Filename* field is 0x00, then this directory entry is free and all the remaining directory entries in this directory are also free.

4. The *Attributes* field of a directory entry is an 8-bit quantity where each bit refers to an attribute (property) of the file or subdirectory pointed to by this directory entry, as follows:

Bit	Mask	Attribute
0	0x01	Read-only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Unused
7	0x80	Unused

   i. If a bit in the *Attributes* field is set (i.e., is 1), that means that the file or subdirectory to which this directory entry points has the attribute associated with that bit. For example, if the *Attributes* field is 0001 0010, then the file/subdirectory pointed to by this directory entry is a hidden subdirectory.

(Bit 1 is on, indicating that it is hidden. Bit 4 is also on, indicating that it is a subdirectory and not a file. Remember, bits are numbered right-to-left.)

ii. If the *Attributes* byte is 0x0F, then this directory entry is part of a long file name and can be ignored for purposes of this assignment. (The updated version of the Microsoft white paper on FAT systems includes details about long file names in FAT12, if you want to deal with them.)

5. The formats for the time and data fields are specified in a Microsoft white paper on FAT systems. (But you do not need to know those formats for this assignment.)

6. The directory entry specifies where the file or subdirectory starts (FLC field) and the length of the file or subdirectory (File Size field). However, the file or subdirectory is not stored contiguously, in general. For a file that is more than 1 cluster long, you need to use the FAT to find the remaining clusters, per the next section of this document.

5. FAT12 Filename and Extension Representation

Filenames in DOS traditionally have a limit of 8 characters for the name, and 3 characters for the extension. There are a few things to be aware of:

- File/directory names and extensions are not null-terminated within the directory entry
- File/directory names always occupy 8 bytes – if the file/directory name is shorter than 8 bytes (characters) pad the remaining bytes with spaces (ASCII 32, or Hex 0x20). This also applies to 3-character extensions.
- File/directory names and extensions are always in uppercase. Always convert given file/directory names to uppercase.
- Directory names can have extensions too.
- "FILE1" and "FILE1.TXT" are unique (the extension does matter).
- Files and directories cannot have the same name (even though the attributes are different). Here are examples of how some file names would translate into the 11 bytes allocated for the file/directory name and extension in the directory entry (white space between quotes should be considered as spaces).
  - "foo.bar" → "FOO BAR"
  - "FOO.BAR" → "FOO BAR"
  - "Foo.bar" → "FOO BAR"
  - "foo" → "FOO"
  - "foo." → "FOO"
  - "PICKLE.A" → "PICKLE A"
  - "prettybg.big" → "PRETTYBGBIG"
  - ".big" → illegal! file/directory names cannot begin with a "."

6. Why Do We Need a FAT?

The directory entry has a field called the First Logical Cluster (FLC) field which specifies where the file or subdirectory begins. Since files and directories can be larger than a sector, a directory or file may have to be stored across more than one sector. The data sectors belonging to a file or a directory are not always stored in contiguous locations in memory. A FAT, therefore, is used to keep track of which sectors are allocated to which file.

To retrieve the entire contents of a file, for example, the FLC field would point to the sector number that holds the first 512 bytes of data. The data from this sector needs to be read in. To determine if there is more data, one must examine the FAT entry that corresponds to the FLC. By examining the FAT entry value, it can be determined if there is another sector allocated to this file. If there is, then the logical sector value is translated to physical sector value and the data from that sector is read in. Next, the FAT entry for the second data sector is examined to see if it is the end of the file. If not, the process is continued. Therefore, the FAT allows accessing data stored in non-contiguous sectors of the storage device.

In Figure 2, File1.txt is stored in logical sectors 2, 4, 6 and 7. The directory entry field — Start Cluster (which is our FLC) field points to sector number 2 which is the first data sector. In the FAT, the value at the second FAT entry is 4, indicating that the next data sector of File1.txt is stored in logical sector 4. The last sector is sector 7, which is evident as the seventh FAT entry holds the EOC (End of Cluster) value.

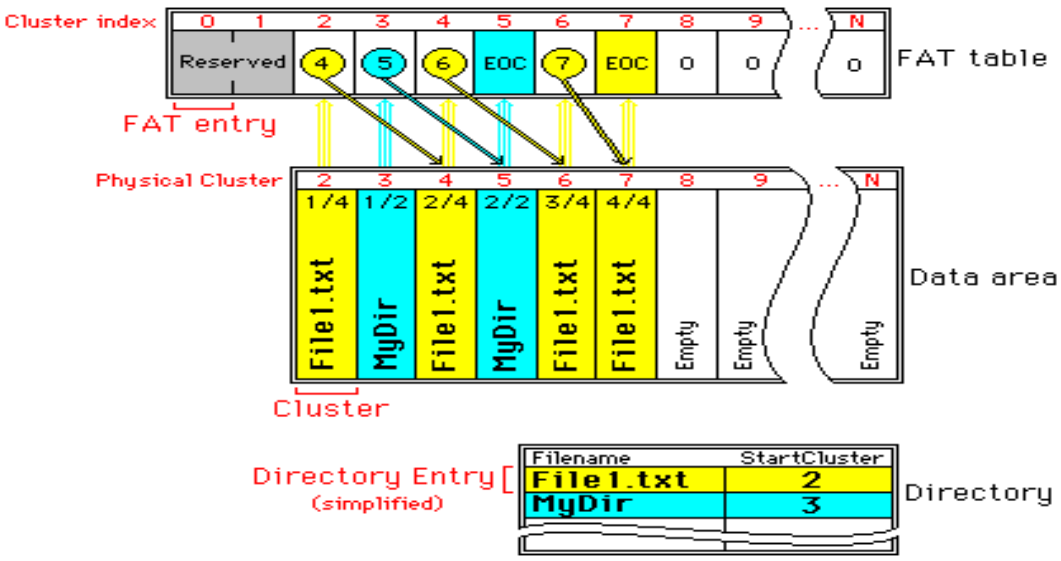

Figure 2 Illustration on the usage of a FAT[§]

7. FAT Packing

In this section, the choice of the value 12 is explained followed by a description on how a 12-bit value is stored in the FAT.

- The space on a floppy disk = 1.44 Mbytes
- The number of bytes in a sector = 512
- The number of sectors in 1.44 Mbytes = $x \approx 2812$
- Therefore, the minimum number of bits required to address x sectors = 12 bits ($2^{11} < 2812 < 2^{12}$)

It can be seen from the above computations that 12 bits is the minimum number of bits needed to access the entire 1.44M space of a floppy disk.
The challenge of 12 bits is that computers store everything in multiples of 8 bits (1 byte). So when storing

[§] This diagram is obtained from the course web site for CS 324 from Brigham Young University

the 12 bit quantity, the option of using 16 bits to store the 12 bits is unsatisfactory as it would leave 4 bits unused for every FAT entry. Since disk space is already at a premium on floppies another solution was designed. This solution involves packing 2 FAT entries (a total of 24 bits) into three 8 bit locations. This is great from an efficiency point of view but it means you have to do a little bit of work to extract a single entry. To further clarify examine a snapshot of the FAT. 8-bit entries are examined:

Position	Byte
0	76543210
1	54321098
2	32109876

This space holds 2 FAT entries. The first entry would be 109876543210 where the first 4 bits come from position 1. The second entry is 321098765432 where the last 4 bits come from position 1. Since the FAT was developed for IBM PC machines, the data storage is in little-endian format i.e. the least significant byte is placed in the lowest address.

So how do we work with the FAT? First, we think of the FAT as an array of bytes (8 bit quantities) since that is the only way we will be able to represent it in C. Now, if we want to access the n^{th} FAT entry then we need to convert between 12 bit and 8 bit values.

- If n is even, then the physical location of the entry is the low four bits in location $1 + (3\times n)/2$ and the 8 bits in location $(3\times n)/2$
- If n is odd, then the physical location of the entry is the high four bits in location $(3\times n)/2$ and the 8 bits in location $1 + (3\times n)/2$

You are provided with the functions to read and write values to the FAT.

Requirements

Your team must design and implement the software for the FAT12 file system. The following is the list of operations that your file system must perform:

1. Execute a simple shell that will display a prompt and wait for user input. When input is received, the shell must ensure that the input is a valid input, **fork off a child process** to perform the operation required, and then display the prompt again. If the input is invalid, then an appropriate error message must be displayed and the prompt displayed again.

Following is the list of commands that your file system must be able to execute. Each of these commands is a valid input to your shell. You may also create a command to quit the shell, i.e. “exit” and/or “logout”. All other inputs should be considered invalid. Note: Each of the following command must be a program of their own.

1. cat *x*
   - Abbreviation for con**cat**enate.
   - If *x* is a file, print the contents of *x* to the screen.
   - If *x* is a directory, print an error message.
   - If number of arguments is not equal to one, print an error message.
   - *x* can be an absolute path or a relative path.
   - If *x* does not exist, print an appropriate error message.

- Example:

```
command> cat course/eccs/eccs164/assign/assign1/solution/pa1.cpp
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
   double principle, interestRate, monthlyPayment;
   int duration;

   cout.setf(ios::showpoint | ios::fixed);
   cout.precision(2);

   cout << "Enter the principle amount:  ";
   cin >> principle;

   cout << "   Enter the interest rate:  ";
   cin >> interestRate;

   cout << "   Enter the loan duration:  ";
   cin >> duration;

   interestRate /= 100;
   interestRate /= 12;
   duration *= 12;

   monthlyPayment = interestRate * pow(interestRate + 1, duration);
   monthlyPayment /= pow(interestRate + 1, duration) - 1;
   monthlyPayment *= principle;

   cout << "Monthly payment:  $" << monthlyPayment << endl;
   return 0;
}
```

2. cd *x*
   - Acronym for change directory.
   - Change to specified directory.
   - If there is no argument available, change to the home directory.
   - Example:

```
command> cd subdir
/SUBDIR
```

3. df
   - Acronym for disk free.
   - Print the number of free logical blocks.
   - If there are any arguments, ignore them.
   - Possibilities:
     - Sum up the number of free clusters on the disk when it is first mounted, place that value in some variable, and then just keep track of every new FAT table entry allocated and deleted throughout the life of your program (assuming that you are the only one that is accessing the drive, you'll be fine).
     - Create a free-space bitmap, similar to the one explained in your text on page 303 that associates a bit to each cluster in the FAT table.

- Implement your own creative method!
- Example:

```
command> df
512K-blocks          Used          Available     Use %
       2847          7                2840      0.25
```

Note: Keep in mind that the boot sector and the sectors allocated to the root directory are not counted as logical sectors. In the MS-DOS FAT file system, there is no structure or system that keeps track of how much free space is available on the disk (FAT32 volumes attempt to help you with this by providing a last-known free space variable). Thus, in order to manage this information yourself, you have several choices.

4. ls *x*
   - Acronym for list.
   - When *x* is the name of a file, list the name of the file, with its extension, the file type, FLC and file size.
   - When *x* is the name of a directory, then list the names of the entries (with extensions) within the directory along with FLC, file size and file type (file or directory). Also, list the "." (current directory) and ".." (parent directory) entries.
   - If *x* is a blank space, for each entity in the current directory, lists its name with extension, FLC, file size in bytes and file type. Also, list the "." and ".." entries.
   - If there are two or more arguments, return an error message.
   - Keep in mind that "/", "." and ".." are valid values for *x*.
   - *x* can be a relative or absolute file name.
   - Example:

```
command> ls
Name           Type   File Size     FLC
SUBDIR          Dir           0       7
EXAMPLE.C      File        1929       2
```

Note: ls has a number of flags that list different types of information. You are required to implement a modified version of ls, as described above. This version will allow you to use ls to test other commands.

5. mkdir *x*
   - Acronym for make directory.
   - If *x* exists, then print a message indicating that it already exists.
   - *x* may be a relative or absolute pathname.
   - If the number of arguments is not equal to one, then quit the operation with an appropriate error message displayed.
   - If *x* does not exist, then create a directory *x* in the target directory.
   - This involves the following steps:
     i. Finding a free entry in the target directory. If the target directory is full, then if space exists on the disk, allocate another sector to the target directory. If there is insufficient space on disk, print an appropriate message and quit the operation.
     ii. Look for space on the disk for the file directory. If there is insufficient space, quit the operation with an appropriate error message.
     iii. If the right number of free sectors is found, allocate them to the file directory and update the required data structures.

iv. In the target directory, add the entry for *x* along with its file name and extension, type, size and the FLC. You may fill the other fields with zeros or ignore them.
v. In the first data sector of the new directory, add entries for "." and "..".

- Example:

```
command> ls
Name         Type   File Size     FLC
EXAMPLE.C    File        1929       2

command> mkdir subdir
```

```
command> ls
Name         Type   File Size     FLC
SUBDIR        Dir           0       7
EXAMPLE.C    File        1929       2
```

6. pbs
   - Acronym for print boot sector.
   - Print the contents of the boot sector (at a minimum the parts shown in the example below)
   - Any arguments specified must be ignored.
   - Example:

```
command> pbs
Bytes per sector            = 512
Sectors per cluster         = 1
Number of FATs              = 2
Number of reserved sectors  = 1
Number of root entries      = 224
Total sector count          = 2880
Sectors per FAT             = 9
Sectors per track           = 18
Number of heads             = 2
Boot signature (in hex)     = 0x29
Volume ID (in hex)          = 0x244f429d
Volume label                = NO NAME
File system type            = FAT12
```

Note: pbs is not a real command. However, you will implement it as an exercise to familiarize yourself with the boot sector and the FAT12 system.

Hint:

1. For this command, create a structure that has all the attributes listed as data stored in the boot sector. You may ignore the attributes marked as ignore in the FAT12 document, provided to you. You must define this structure in an appropriately named header file.
2. Declare a global variable of this type.
3. Write a function called readBootSector. The function will read the data in the boot sector, assign values to the global variable defined above. Note that the number of bytes to be read (sector size) can be made available through the global variable BYTES_PER_SECTOR and hence need not be passed as a parameter to the function. It will use the read_sector() function to read from the boot sector.
4. Write a function printBootSector. The function will print the values read from the boot sector in the format as shown in the example.
5. Write a program pbs.c that will use the above functions to read and print the contents of the boot sector. The program must ignore any command line arguments.

7. pfe *x y*
   - Acronym for print fat entries.
   - Print the 12-bit FAT entry values representing logical sectors *x* to *y*.
   - If *x* > *y*, then an error message must be displayed.
   - If *x* < 2, an error message must be displayed.
   - Example:

```
command> pfe 2 8
Entry 3: 4
Entry 4: 5
Entry 5: 6
Entry 6: FFF
Entry 7: FFF
Entry 8: 0
```

Note: pfe is not a real command. However, you will implement it as an exercise to familiarize yourself with the boot sector and the FAT12 system. Moreover, this command can be used by other commands.

Hint:

1. For this command, write a function checkRange, which accepts the values of *x* and *y* as parameters. It must validate the values of *x* and *y* as stated earlier in the specifications. The function must return a value to the calling function, indicating if *x* and *y* are valid inputs.

2. Write a function readFAT12Table that will read the entire FAT table into a buffer and return the buffer. The input to the function will be a value indicating which FAT table should be read. It will use the read_sector() function to read the data in the sectors that hold the FAT table.

   Note: The FAT should not be stored as a global value. When a function has to access the FAT table, it must call the above function.

   Hint: To read data from multiple sectors, do not read a sector at a time into a buffer and then concatenate buffers together with strcat. Instead use read_sector in the following manner:

   fat – sufficiently large character array
   i – must vary from 0 to the number of FAT sectors.
   read_sector(i+1, &fat[i * BYTES_PER_SECTOR]);

3. Write a program pfe.c that will accept two integers *x* and *y* as command line arguments, use the above functions to validate *x* and *y* as per the specifications, and then print the FAT entry values for FAT entries *x* to *y*. If *x* and *y* are invalid inputs, an appropriate error message must be displayed.

8. pwd
   - Acronym for print working directory.
   - Print the absolute path to the current working directory.
   - If there are any arguments, ignore them.
   - Example:

```
command> pwd
/
```

   Note: When you start up the system, the current working directory must be the root directory.

9. rm *x*
   - Acronym for remove.
   - If *x* does not exist, print an appropriate message and quit the operation.
   - *x* can be an absolute or relative pathname.
   - If the number of arguments is more than one, then quit the operation with an appropriate error message.
   - If *x* is a directory, print an appropriate message and quit the operation.
   - If *x* is a file, then remove the file from the target directory.
   - This involves the following operations:
     i. Remove the entry from the parent directory.
     ii. Optimize the space usage of the parent directory.
     iii. Free the data sectors of the target file.
     iv. Update appropriate data structures.
   - Example:

```
command>ls
Name           Type   File Size     FLC
SUBDIR          Dir           0       7
EXAMPLE.C      File        1929       2

command> df
512K-blocks           Used          Available     Use %
       2847           7                2840      0.25

command> rm example.c
command> ls
Name           Type   File Size     FLC
SUBDIR          Dir           0       7

command> df
512K-blocks           Used          Available     Use %
       2847           6                2841      0.21
```

10. rmdir *x*
    - Acronym for remove directory.
    - If *x* does not exist, print an appropriate message and quit the operation.
    - *x* can be an absolute or relative pathname.
    - If the number of arguments is more than one, then quit the operation with an appropriate error message.

- If *x* is a file, print an appropriate message and quit the operation.
- If *x* is a directory, but has entries other than "." and ".." i.e. it is not empty, then print an appropriate message and quit the operation.
- If *x* is an empty directory, then remove the directory from the parent directory. This involves the following operations:
  i. Remove the entry from the parent directory.
  ii. Optimize the space usage of the parent directory.
  iii. Free the data sectors of the target directory.
  iv. Update appropriate data structures.
- Example:

```
command> ls
Name           Type   File Size    FLC
SUBDIR          Dir           0      7
EXAMPLE.C      File        1929      2

command> df
512K-blocks           Used          Available     Use %
       2847           7                2840       0.25

command> rmdir subdir
command> ls
Name           Type   File Size    FLC
EXAMPLE.C      File        1929      2

command> df
512K-blocks           Used          Available     Use %
       2847           1                2846       0.04
```

11. touch *x*
    - If *x* exists, then print a message indicating that it already exists.
    - *x* may be a relative or absolute pathname.
    - If *x* does not exist, then create a file *x* in the target directory.
    - This involves the following steps:
      i. Find a free entry in the target directory. If the target directory is full, and if space exists on the disk, allocate another sector to the target directory. If there is insufficient space on disk, print an appropriate message and quit the operation.
      ii. Look for space on the disk for the file. If there is insufficient space, quit the operation with an appropriate error message.
      iii. If the right number of free sectors is found, allocate them to the file and update the required data structures.
      iv. In the target directory, add the entry for *x* along with its file name and extension, type, size and the FLC. You may fill the other fields with zeros or ignore them.
    - Example (For brevity, the "." and ".." entries are not listed):

```
command> ls
Name           Type   File Size    FLC
SUBDIR          Dir           0      7
EXAMPLE.C      File        1929      2

command> touch test

command> ls
Name           Type   File Size    FLC
SUBDIR          Dir           0      7
TEST           File          0      8
```

```
EXAMPLE.C      File        1929      2
```

Documentation

Documentation is an important part of your project. Through your documents, you will communicate your design to the rest of the world. Here are the documents that you must maintain throughout the project. These documents must be completed in *LaTex* with the exception of the implementation log. However, it is a good idea to put the implementation log in *LaTex* as well.

1. Design Phase

In your design document, you must list all the new functions you will be using, with details on their behavior. You must also list, for every command, the sequence of operations to be performed and which function will perform the operations. A well thought out and detailed design will save you a lot of work in the implementation phase of your project. You will realize that a number of operations to be performed are common to a number of the commands. A good design will have functions that can be re-used by a number of the commands.

For this part, the design document will include the following:

- A description of how the shell works.
- A list of all new functions. State the function prototype, what each input parameter is and what the return value is. Describe the operations to be performed by the function.
- For each command other than pbs and pfe, describe (in steps) how the command execution will be performed using the functions you described above.

Note: You are required to keep updating your design if you change it. You will have to turn in a copy of your final design document by the end of the project.

2. Test Plans

In order to make sure that your system is working properly, you must implement a test plan. Your team will develop a systematic procedure to test your system. A test plan should have detail information on what is the expected output for a given input for all commands.

As you are testing your system, you know where a problem may occur and you will have to fix it. An exhaustive test plan will catch any features that you might have missed in your design phase and save you many hours of re-writing code, during the implementation phase.

At the end, your program will have to pass all your test cases to be able to claim that it is well tested. These test cases are important, thus your team must create as many critical test cases as possible. Then, it will help you to make sure your system is robust. If you change your test plan, make sure you have your plan updated because you must turn in your final test plan documentation.

3. Implementation Log

In order to document the work that you have done, use a log file. The entry of the log should contain the following information:

1. Date
2. Starting and ending time for the log entry
3. Brief description of the work completed during this period of time (this can be done in point

form)

Since you will be working in a group, each member of the group must have his/her own log file. You will need to attach this log file as an appendix to your final report.

Hint: It is a good idea to keep a daily log, otherwise you may not be able remember what you have completed a week ago.

4. Commit History

You are required to maintain all your code and document under version control (git is recommended). You should commit early and often and provide meaningful commit messages. You will then be required to submit your commit history along with the rest of the project documentation.

Additional Requirements

1. Your code must comply with the published C Programming Standard.
2. Your code must have header information in all your files.
3. Your code must be in **C** and it must be compliable under **Hawk**.
4. You can include a **readme** file describing how to use your program.
5. You must include the **makefile** used to compile your program.
6. You submit all materials on **and** put a copy of the solution in the **submit directory on hawk**.

Supplement

You are provided with the following files in a file named ***package.tar.gz*** on Angle:

1. fat.c
   - The program fat.c reads the first 13 bytes of the boot sector and determines the number of bytes per sector. This value can be used to determine if the file system is a FAT12, FAT16 or FAT32 system. If the number is 512, then we are dealing with the FAT12 system. Use the program to understand how to read data from the disk and how to manipulate the data read in from the disk.
   - This will serve as an excellent example to help you to kick start your pbs and pfe commands.
2. fatSupport.h and fatSupport.c
3. dda.h
4. floppy1, floppy2 and floppy3

Description of the Floppy Images

Three floppy images are provided to you to test your system. These images are:

floppy1

- root
  - subdir
    - example.c
  - example.c

floppy2

- root
  - subdir1
    - subsub
      - sssub1
        - ssssub1
          - test1
          - test2
        - ssssub2
        - test1.txt
        - test2.txt
      - sssub2
        - test1
      - sssub3
      - sssub4
        - test1
        - test2
      - test1.txt
      - test2.txt
    - example.c
  - subdir2
  - subdir3
  - subdir4
  - example.c

floppy3

- root
  - class
    - class14
      - j-doe
        - private

- public
  - html
- w-chen
  - private
    - large.txt
    - small.txt
  - public
    - html
      - index.html
- class15

- course
  - acc
  - blw
  - cit
  - cor
  - csi
    - csi140
      - assign
        - assign1
          - solution
            - pa1.cpp
          - spec.txt
        - assign2
        - assign3
        - lab1
          - spec.txt
        - lab2
        - lab3
        - lab4
      - readme.txt
    - csi385
      - osproj.txt
  - egp
    - egp200
  - for
  - gdd
  - his
    - his110
  - ixd
  - lan
  - mth
    - mth230
    - mth240
  - psy
  - rad
  - sci
  - course.txt
- large.txt
- small.txt

Project Timeline**

Date	Goal
9/09/16	Identify your partner and decide on a name for your team**. Send an email to the professor with the name of your teammate and the name of your team.** One email per team is sufficient.
9/19/16	**Submit a paper (2 pages max) describing how the FAT table works**, how to use the FAT table to locate a file and what the differences are between the root directory and the FAT table.
9/26/16	Complete the pbs and pfe commands.
10/03/16	**Hand-in a copy of your initial design and test plans** (One set of documents per team is sufficient). Also, you are required to present your design to your peers. Feedback will be given to you so that you can incorporate them into your project.
10/12/16	Complete the initial simulation of a shell. For this part, you do not have to use the FAT12. However, at this point there are only two commands available under your shell, which are pbs and pfe.
10/19/16	Complete the cd and pwd commands. Update shell for these commands. Update design and test plans as needed.
10/26/16	Complete the ls command. Updated shell for this command. Updated design and test plans as needed. **Mid-project evaluation for the shell, cd, pwd, ls, pbs, and pfe commands.**
11/02/16	Complete the rm and rmdir command. Update shell for these commands. Update design and test plans as needed.
11/09/16	Complete the mkdir, touch, cat, and df commands. Update shell for these commands. Update design and test plans as needed. **At this point all functionality is implemented and your project will undergo peer evaluation.**
11/16/16	Complete peer evaluation for specified groups. **Hand in all evaluation reports.**
11/22/16	Hand-in a final version of your design and testing plans, and also a copy of your project with any bugs identified by your peers fixed. Complete a teammate evaluation. Hand in all documents (one set per group). However, the teammate evaluations must be completed and handed in independently.
11/29/16	Presentations start

** The timeline is very tight so make sure you plan accordingly.
Also note that whenever something is to be submitted it is due by 23:59 on the day specified.