



Operating Systems Development Series

# Operating Systems Development - FileSystems and the VFS

by Mike, 2010

This series is intended to demonstrate and teach operating system development from the ground up.

## Introduction

Welcome to the 22'th chapter in a never-ending series for operating system development! This is more than chapter 22 but also year 2 for the OS Development Series.

This is yet another filesystem related tutorial (Dont worry, its the last one ;) ). The first one was needed so we can load our main bootloader program from the bootcode, The second one was for our main boot program so it can load our kernel. Now we need one more for our kernel so our kernel can load programs and execute them. There is a difference between this chapter and the other two, however - this one will be in C instead of assembly language. :)

To spice things up, however, and introduce something new, we will also be looking into **Virtual FileSystems (VFS)**. This will allow us to interface with any filesystem driver and different disk devices in the same way. It can be used for both local disk drives, but can also be used to interface with any network filesystem.

*Ready?*

## File Systems

### Abstract

#### File System

A **File System** defines a logical way to read and write information. In this way, it can be considered a **specification**. Most PC file systems are based off of the desktop concept of files and folders.

There are a lot of different kinds of file systems. Some are widely used (Like FAT12, FAT16, FAT32, NTFS, ext (Linux), HFS (Used in older MACs); other filesystems are only used by specific companies for in house use (Like the GFS - Google File System). Some filesystems are used in networking only (NFS). You can also develop and design your own file system implementation.

File Systems are used for data storage and organizing data. They help provide a straightforward way to access files and directories on removal media (floppies, flash drives, CDs, DVDs), local drives (hard disk drives), and network clients. File Systems can also exist as an in-memory image. For example, you can load a file that contains a "foot print" of a special type of file system in it.

## Files and Folders

A **file** is a group of data that represents something to a program or to the user. This data can be anything we want it to be. It all depends on how we interpret the data. For example, a **text file** contains text information. A file can also be an image of something. A **folder** is a logical group of files. It is also known as a **directory**.

Directories provide us a way to manage a large amount of files. Directories typically form a **tree** structure. This is known as a **directory tree**. There is only one directory that is the parent of all directories and files: the **Root Directory**. A **File Path** is the location of a file in the directory tree. For example, the file **a:\myfile.txt**, myfile.txt is the filename. It is in the root directory at the device known as "a:". **a:\mydir\myfile.txt** is a file, myfile.txt, located in the subdirectory, mydir, that is, in turn, located in the root directory on device "a:".

## File and Folder Naming

The name of a folder or file is a string representing that file or folder, usually by its contents. File Systems implement file naming and folder naming differently, and each has their own constraints. For example, FAT12 stores filenames and folder names in a directory entry as an array of 11 bytes (8 for filename, 3 for extension. This is also known as the **8.3 naming convention**) This limits file names and folder names to 11 characters. On the other hand, NTFS is limited to 255 characters with **Long File Name (LFN)** support. NTFS, for another example, stores file names along with file attributes in a **Master File Table**.

Most filesystems file names are not case sensitive. However, some filesystems may store filenames differently internally. For example, you may have found out that you can have an 8.3 lowercase file name for a file on the floppy disk, but be able to load the file from your OS by using an all uppercase file name. Windows displays the LFN of the file name, while FAT12's 8.3 file entry only contains its 8.3 all-uppercase file name. This is what makes it possible.

## File Types

### Symbolic link's

Symbolic links are a way to provide shorten paths. For example: a:/folder/link.lnk points to a:/otherfolder/subfolder/subsubfolder/yet another folder/link.txt. Now you can access the text file easily. Symbolic links are also very often used to make folder organized. Like the Windows Start Menu. Contains symbolic links to your programs. A symbolic link is not very hard to implement. You find the node given (which is the link). It seems to be a link, so you get the real path and read that file instead.

Windows Shortcuts are a type of symbolic link.

### Pipes

A type of **InterProcess Communication (IPC)** is called a pipe. A pipe is a **virtual file**, usually between two or more processes. The best example may be stdout, stdin and stderr on Unix. They are handled as normal files, but the data written to stdout show up onto the screen (or in stdout.txt).

## Special File Types

### Metafiles

Some filesystems also impliment special files and folders specifically for filesystem use. Typically you cannot have two files or folders with the same name (nor a filename sharing the same name as a folder) in the same directory. Because of this, naming a file or folder with one of these hidden files may also not be possible depending on implimentation.

For example, NTFS provides several metafiles for filesystem use. These files are located on the root directory of the system drive (typically C:). \$MFT, \$MFTMirr, and \$LogFile are a few of these files. While they do not ever show up even when view hidden and system files are checked, watch what happens when you create a file with one of the above names there. You can create those files anywhere else, but you will get a "file already exists" error when creating one on the root directory do to the metafiles.

### Device Files

Unix-like systems, DOS (and, in turn, Windows) has **Device Files** which are special "files" that represent a device. For example, NUL (null device), CLOCK\$, PRN (printer), etc. Here is the list of device files:

- CON
- PRN
- AUX
- CLOCK\$
- NUL
- COM0, COM1, ... COM9
- LPT0, LPT1, ... LPT9

Because these names have special meaning in DOS and Windows, you cannot name a file or folder any of the above names.

### . and ..

. and .. are special files some file systems impliment. '.' is the file name of a file that contains file information that refers to the current directory. '..' is the file name of a file that contains information that refers to the parent directory of that file. For example, if there is a file located at **c:\mydir\file.txt**, and **c:\mydir** was the current directory, the pathname .. will refer to C: while the pathname . will refer to c:\mydir.

## File System Types

### Flat File Systems

A **Flat File System** is a filesystem that does not support subdirectories. Instead, all of the files are in the same (root) directory. Many early computer systems used flat file systems. Modern operating systems typically impliment more advanced hierarchical file systems. While small and easy to impliment, flat file systems are hard to orginize.

### Hierarchical File Systems

This type of file systems supports subdirectories. Most modern file systems (including FAT12, FAT16, FAT32, etc., NTFS) fit into this category. (The first version of FAT12 was a flat file system. Later versions support subdirectories however.)

## Journaling File Systems

This type of file system uses a "journal" of file system changes. This is a log of changes the system intends to make to files or directories prior to completing the steps. This insures that, if a crash occurs during a filesystem operation (like writing a file), the journal can be read to undo the changes made to repair the filesystem.

## File System Drivers

While a **file system** defines a specification for reading and writing "files" and "directories", a **file system driver** contains the implementation of a specific type of file system. A good example of a file system driver is **ntfs.sys** which contains Microsoft's implementation of the NTFS File System. File system drivers are also sometimes implemented as minidrivers inside of larger software. Bootloaders are a good example. Because boot loaders have to be able to load files from disk without a separate driver program, they contain several filesystem minidrivers for different types of filesystems inside of the bootloader itself. If you developed the bootloader in the series, you have already experienced the FAT12 file system and developed a FAT12 minidriver for our bootloader.

## Virtual FileSystem (VFS)

### Abstract

A **Virtual File System (VFS)** is an abstraction layer on top of specific filesystem implementations. The software accesses storage devices through a VFS. This allows the software to read or write to different storage devices without any knowledge of the device or filesystem that is being used. It also allows the same code to work with any number of installed filesystems or devices.

The basic idea is to allow a single system interface to work with any filesystem in a uniform way. Windows, Linux, and Mac OS all support VFS in different ways.

### Implementation

There are different ways to implement a VFS.

### Mount Point List

A **mount point list** is a list of mounted file systems and where they are mounted. For example, if a file needs to be read from, the OS typically calls the VFS ReadFile() function which searches through the list of mounted file systems to locate the device and file system the file is in. It then passes the read request to that file system's ReadFile() function.

### Node Graph

A **Node Graph** contains a graph of nodes that represent files of different types: files, folders, mount points, etc. Each file node structure typically

contains function pointers to file system-specific routines for reading and writing files.

For example, we can create a FILE structure like this:

```
typedef struct _FILE {  
    char        name[32];    //filename  
    uint32_t     flags;       //flags  
    uint32_t     fileLength;  //length of file  
    read_func_t  read;        //function pointers to read,write,open,close file  
    write_func_t write;  
    open_func_t  open;  
    close_func_t close;  
}FILE, *PFILE;
```

Notice the function pointers are stored in this FILE structure. Lets say we want to read a file, so we call fopen(), which, eventually, calls our VFS OpenFile() function. All the VFS file operation routines ever need to do is pass control to that specific FILE's function pointers:

```
void VfsOpenFile (PFILE file, const char* filename) {  
    if (file)  
        file->open (filename);  
}
```

This allows the filesystem-defined routine to be called.

## DOS and Windows

DOS and Windows assigns a letter from 'a' through 'z' to represent a mounted file system. Windows keeps a symbolic link between a drive letter and its Object Manager name. For example, the drive letter c: (symbolic link name \\GLOBAL??\C:) may be mapped to the Object name \Device\HardDiskVolume1 device object. A File System can register itself to own a device object. If a file system is found to own the object, the rest of the file path name ("myfile.txt" in this example) is passed to that filesystem's FileOpen() function.

### Drive letter assignment

Windows supports assigning drive letters to devices and partitions representing mounted file systems. (During boot, if no filesystem driver registers to own a device object, Windows uses its RAW minidriver for the devices.) Drive letters can also refer to network shared drives, virtual disk images, or a symbolic link to another location in the local or a network client. However, they are limited to 26 devices do to only 26 letters that can be used from 'a' to 'z'.

## Interface

For simplicity, we will be using drive letter assignment along with a mount point list in our VFS implementation. Our implementation needs to be simple because we do not have device management nor I/O management in the OS presented in the series.

I personally recommend developing the VFS first prior to the filesystem driver. This way the interface and framework of the VFS will have already been completed.

## FILE

Anyone that has used C is already familiar with the infamous FILE\* data type. FILE\* is an **Abstract Data Type (ADT)** that represents a pointer to a file object. ISO C defines that C implementations must define a FILE type, however does not define what is inside of the structure. That is, while FILE\* is ISO C, the structure contents is implementation-defined.

We can define a file structure that will represent the current state of a file any way we want. So lessee... a file has a name and a size, so that's two members already. We need a way to flag if it's the **End of File (EOF)**, and file-specific flags, so that's two more members. We also need a way to keep track of a file's current position (its cluster and the cluster's offset), and now we have something like this:

```
typedef struct _FILE {  
    char        name[32];  
    uint32_t    flags;  
    uint32_t    fileLength;  
    uint32_t    id;  
    uint32_t    eof;  
    uint32_t    position;  
    uint32_t    currentCluster;  
    uint32_t    device;  
}FILE, *PFILE;
```

That was easy, huh? **id** can be used for identification purposes if you like. **device** represents the device the file resides on.

## Types of files

There are a lot of different types of files that we have talked about: files, directories, symbolic links, etc. For simplicity, we will only focus on files and directories. These will be used in the **flags** member of our FILE structure above to represent the type of file.

```
#define FS_FILE        0  
#define FS_DIRECTORY  1  
#define FS_INVALID    2
```

## Operations

There are some typical operations we can perform on a file:

- Open
- Close
- Read
- Write
- Mount
- Unmount

Open and Close operations perform opening and closing a file object (file or directory, whatever the file type is), while reading and writing operations perform reading and writing the file type. All of these are exposed to the programmer through the standard C file I/O functions.

For our VFS, they are exposed through a **Volume Manager** located in fsys.h:

```
extern FILE volOpenFile (const char* fname);
extern void volReadFile (PFILE file, unsigned char* Buffer, unsigned int Length);
extern void volCloseFile (PFILE file);
extern void volRegisterFileSystem (PFILESYSTEM, unsigned int deviceID);
extern void volUnregisterFileSystem (PFILESYSTEM);
extern void volUnregisterFileSystemByID (unsigned int deviceID);
```

For example, let's say we call the C fopen() routine. That will call our volOpenFile() routine which returns a FILE object. We passed a path to the file, like "a:\myfile.txt". The Volume Manager indexes into the mount point list and verifies that a file system has registered for the device ID that represents 'a'. If it has, it calls that filesystem drivers FileOpen() method passing "myfile.txt". Don't worry if it sounds complicated. It can be; but the design of how it's implemented in the demo is very easy.

## Volume Manager Implementation

### File System Abstraction

The first thing we need is a way to abstract filesystem-specific information. This includes the name of the filesystem and the operations that can be performed on files. This is done using function pointers.

```
typedef struct _FILE_SYSTEM {
    char Name [8];
    FILE      (*Directory) (const char* DirectoryName);
    void      (*Mount)     ();
    void      (*Read)      (PFILE file, unsigned char* Buffer, unsigned int Length);
}
```

```

        void          (*Close)      (PFILE);
        FILE          (*Open)       (const char* FileName);
    }FILESYSTEM, *PFILESYSTEM;

```

## Implimentation

The Volume Manager impliments our VFS in the demo. Its in the files fsys.h and fsys.cpp. Remember that we will be using drive letter assignment to represent devices? Because there are 26 possible devices, it is helpful to make a constant, **DEVICE\_MAX**. Because each device can only have one mountable file system, we store them in a list (like a mount point list).

```

#define DEVICE_MAX 26

//! File system list
PFILESYSTEM _FileSystems[DEVICE_MAX];

```

Here is how it works. Because we are storing the filesystems as a list of pointers, if a pointer is valid, the filesystem has been registered there. Each element in the array represent the drive letter that it refers to. So 'a' is at \_FileSystems[0], 'b' is at \_FileSystems[1], etc. It is the filesystems responsibility to manage the disk that they are writing on.

Using this method provides a very basic but easy way of accessing devices. For example volOpenFile() only needs to check the first character of the path (the drive letter) and do a lookup into the list to see if a filesystem is registered for that device. If it is, it can call that filesystem's open() method and pass the filename to the driver. We default to using 'a', however if the input path contains an ':' then we use the first character for the device instead. This allows us to call volOpenFile in two ways: passing a string like "**myfile.txt**" and "**a:myfile.txt**", where "a" is the device the file is in. Cool, huh?

```

FILE volOpenFile (const char* fname) {
    if (fname) {
        //! default to device 'a'
        unsigned char device = 'a';

        //! filename
        char* filename = (char*) fname;

        //! in all cases, if fname[1]==':' then the first character must be device letter
        if (fname[1]==':') {
            device = fname[0];
            filename += 2; //strip it from pathname
        }
    }
}

```



```

    }

    //! call filesystem
    if (_FileSystems [device - 'a']) {

        //! set volume specific information and return file
        FILE file = _FileSystems[device - 'a']->Open (filename);
        file.deviceID = device;
        return file;
    }

    FILE file;
    file.flags = FS_INVALID;
    return file;
}

```

All of the other file operation routines are basically the same. Knowing how our VFS is storing filesystems, you can probably guess how `volRegisterFileSystem()` family of routines work. All they basically do is store a pointer to the filesystem in the list or clear it.

```

void volRegisterFileSystem (PFILESYSTEM fsys, unsigned int deviceID) {

    if (deviceID < DEVICE_MAX)
        if (fsys)
            _FileSystems[ deviceID ] = fsys;
}

```

Alright then! So we initialize the filesystem driver, which calls `VolRegisterFileSystem()` to register itself. We call `fopen()`, which calls `VolOpenFile()`, which in turn calls our filesystem's `open()` method. Everything is now in place but we are missing something... something very important... the filesystem driver itself!

Right, I suppose we should go into it .. again...

## FAT12 - Take Three

### Introduction

We have looked at and implemented FAT12 two times in the past throughout the series. Because of this, I do not plan on covering FAT12 in great detail again. However, this will be a review of FAT12 along with the C driver code and how it works.

If needed, please reference [Chapter 11](#) while reading.

## Boot Sector

Remember that a lot of important filesystem information is stored in the boot sector along with our boot strap program? More specifically, it is located in the **Bios Parameter Block (BPB)** located in the boot sector.

When we mount our filesystem, we will need to read from the BPB and store this information for later use. To do this, we can create a structure that matches the boot sector:

```
typedef struct _BOOT_SECTOR {
    uint8_t          Ignore[3];           //first 3 bytes are ignored (our jmp instruction)
    BIOSPARAMATERBLOCK Bpb;              //BPB structure
    BIOSPARAMATERBLOCKEXT BpbExt;        //extended BPB info
    uint8_t          Filler[448];        //needed to make struct 512 bytes
}BOOTSECTOR, *PBOOTSECTOR;
```

A good example of what the boot sector looks like is to think about what our Stage 1 Bootloader program looks like in memory. The very first instruction in Stage1 (Please see [Chapter 4's demo](#), Stage1.asm) was **jmp loader**. This is a three byte instruction, so the first 3 bytes in the above structure is the **Operation Code (OPCode)** of our jmp instruction.

Also remember from [Chapter 4](#) that we have covered the OEM Parameter Block (aka, Bios Parameter Block (BPB)). The BPB is located right after our 3 byte jump instruction. Because of this, the BIOSPARAMATERBLOCK is next in this structure. I also provide the BIOSPARAMATERBLOCKEXT structure which is an extension to the BPB for some other file systems, such as FAT32.

The last 448 bytes of the bootsector contain the rest of our boot sectors program code. Because it's not important to us right now, we just treat it as padding in the **Filler** member. This insures the BOOTSECTOR structure is exactly the same size as our on-disk boot sector (512 bytes).

BIOSPARAMATERBLOCK is a structure that defines the format for a BPB. It is the same structure that is in the boot sector and has been covered in more depth in [Chapter 5](#).

```
typedef struct _BIOS_PARAMETER_BLOCK {
    uint8_t          OEMName[8];
    uint16_t         BytesPerSector;
    uint8_t          SectorsPerCluster;
    uint16_t         ReservedSectors;
    uint8_t          NumberOfFats;
    uint16_t         NumDirEntries;
    uint16_t         NumSectors;
    uint8_t          Media;
    uint16_t         SectorsPerFat;
```

```

        uint16_t      SectorsPerTrack;
        uint16_t      HeadsPerCyl;
        uint32_t      HiddenSectors;
        uint32_t      LongSectors;
    }BIOSPARAMATERBLOCK, *PBIOSPARAMATERBLOCK;

```

The above structure should look familiar :) If not, please read its description in [Chapter 5](#)

BIOSPARAMATERBLOCKEXT, however, may be new. While we have already covered the BPB in depth and used it in the past for FAT12 parsing, FAT12 bootsectors do not rely on the BPB extended members. FAT32, however, does.

```

typedef struct _BIOS_PARAMATER_BLOCK_EXT {

    uint32_t      SectorsPerFat32;    //sectors per FAT
    uint16_t      Flags;              //flags
    uint16_t      Version;            //version
    uint32_t      RootCluster;        //starting root directory
    uint16_t      InfoCluster;
    uint16_t      BackupBoot;         //location of bootsector copy
    uint16_t      Reserved[6];

}BIOSPARAMATERBLOCKEXT, *PBIOSPARAMATERBLOCKEXT;

```

Thats everything :) There is nothing special here-everything has already been covered in detail in previous chapters. These structures provide the filesystem driver an easy way of referencing data in the BPB for later filesystem use. All we need to do is read in the bootsector, and accessing the data through a PBOOTSECTOR. :)

We read the sector using our floppy disk driver that we developed in the previous chapter.

```

    //! Boot sector info
    PBOOTSECTOR bootsector;

    //! read boot sector
    bootsector = (PBOOTSECTOR) flpydisk_read_sector (0);

```

That is all that is needed :) All of our important information is now in **bootsector.bpb**. All thats needed is mounting the filesystem...

## Mounting the filesystem

Now that we have our BPB information in memory, we need to prepare the filesystem for use. We start this by first deciding what information we need.

Okay.. let see, we will need to total number of sectors on disk. We will also need to know the total number of directory entries. Other helpful information can be for use with the **File Allocation Table (FAT)** and the Root Directory:

```
typedef struct _MOUNT_INFO {
    uint32_t numSectors;
    uint32_t fatOffset;
    uint32_t numRootEntries;
    uint32_t rootOffset;
    uint32_t rootSize;
    uint32_t fatSize;
    uint32_t fatEntrySize;
}MOUNT_INFO, *PMOUNT_INFO;
```

Okay... Remember that we already have the bootsector stored in our BOOTSECTOR structure? Knowing this, we can simply copy over some of the information from the BPB to our MOUNT\_INFO structure.

Alright.. Lets locate the location of the first FAT and root directory in a FAT12 formatted disk:

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.
-------------	------------------------	-------------------------	-------------------------	-----------------------------------	--

Notice that there are two FATs. The first FAT is right after the boot sector on disk. Because of this, we set **fatOffset** in MOUNT\_INFO to 1. Also note that the Root Directory is right after both FATs. Knowing this, we can come up with a simple calculation to find the starting sector of the root directory. **(NumberOfFATS \* sectorsPerFAT) + 1**. We need to add 1 for the bootsector.

We now have the location of the first FAT and root directory. To find the size of the root directory, all we need is the number of entries in the root directory and the size of each entry. Each directory entry in FAT12 is a specific structure format that is 32 bytes in size. So all we have to do is **bootsector->Bpb.NumDirEntries \* 32**. This is the number of bytes the directory takes up. We divide it by the bytes per sector to convert it to a sector count.

```
//! store mount info
_MountInfo.numSectors    = bootsector->Bpb.NumSectors;
_MountInfo.fatOffset     = 1;
_MountInfo.fatSize       = bootsector->Bpb.SectorsPerFat;
_MountInfo.fatEntrySize  = 8;
_MountInfo.numRootEntries = bootsector->Bpb.NumDirEntries;
_MountInfo.rootOffset    = (bootsector->Bpb.NumberOfFats * bootsector->Bpb.SectorsPerFat) + 1;
```

```
_MountInfo.rootSize = ( bootsector->Bpb.NumDirEntries * 32 ) / bootsector->Bpb.BytesPerSector;
```

That is all that there is to it. We have our FAT12 driver initialized. Easy, huh? We have the important filesystem information in MOUNT\_INFO so all that's needed is to parse the directories and load a file. :)

## Directory parsing

### Format

A directory in FAT12 is composed of 32 byte structures that provide information about a file or subdirectory. Each directory entry has the following format:

```
typedef struct _DIRECTORY {
    uint8_t  Filename[8];           //filename
    uint8_t  Ext[3];                //extension (8.3 filename format)
    uint8_t  Attrib;                //file attributes
    uint8_t  Reserved;
    uint8_t  TimeCreatedMs;         //creation time
    uint16_t TimeCreated;
    uint16_t DateCreated;          //creation date
    uint16_t DateLastAccessed;
    uint16_t FirstClusterHiBytes;
    uint16_t LastModTime;           //last modification date/time
    uint16_t LastModDate;
    uint16_t FirstCluster;         //first cluster of file data
    uint32_t FileSize;              //size in bytes
} DIRECTORY, *PDIRECTORY;
```

That is all that there is to it :) This is a directory entry - the information stored in our DIRECTORY structure can be a subdirectory or a file.

**Filename** and **Ext** contains the file or directories 8.3 format name.

**Attrib** contains the attributes of a file or directory. It has the following values for reference:

- Read only: 1
- Hidden: 2
- System: 4
- Volume Label: 8
- Subdirectory: 0x10
- Archive: 0x20
- Device: 0x60

Please note that we will not be using this in the series as it is not needed. However, you can provide support for working and setting file attributes in your own system if you like.

All **date** members in this structure follow a specific bit format:

- **Bits 0-4**: Day (0-31)
- **Bits 5-8**: Month (0-12)
- **Bits 9-15**: Year

All **time** members in this structure follow a specific bit format:

- **Bits 0-4**: Second
- **Bits 5-10**: Minute
- **Bits 11-15**: Hour

Because we have no need in modifying or retrieving file or directory date or time information we are not using them in the series. However, I encourage our readers to add the functionality themselves later on if they like.

Remember that, for a FAT12 formatted floppy disk, a cluster is the same size as a sector (512 bytes). Because of this, the **FirstCluster** field in **DIRECTORY** also points to the first sector of a file. Thus, by reading this sector, you effectively read the first 512 bytes of the file.

Now let's parse our directory and find our file...

## Parsing

Remember that a directory contains a list of directory entry structures. Knowing this, parsing the directory to find a file or directory becomes very easy.

We begin with loading the root directory. Remember that we retrieved the root directory sector from the BPB when we mounted the filesystem and stored it into **\_MountInfo.rootOffset**. Thus, all we need to do is to load the sector, and use a **DIRECTORY\*** to access the directory entries.

Then we loop and compare filenames to find a match. We convert the input filename to its DOS 8.3 filename format using **ToDosFileName()**. For example, turning the input filename of "Myfile.txt" to the FAT12 internal format "MYFILE TXT".

We read in a sector and compare each entry in the sector. You will also notice that we turn the filenames into C strings so we can use a simple **strcmp()** call to test if filenames match. When we found a match, we fill out our **FILE** structure and return it.

Let's take a look:

```
FILE fsysFatDirectory (const char* DirectoryName) {  
    FILE file;  
    unsigned char* buf;  
    PDIRECTORY directory;
```

```

    //! get 8.3 directory name
    char DosFileName[11];
    ToDosFileName (DirectoryName, DosFileName, 11);
    DosFileName[11]=0;

```

**DirectoryName** contains the directory or file name we are wanting to find. We convert the input filename, like "myfile.txt" into its DOS 8.3 filesystem format "MYFILE TXT" and store it in DosFileName.

```

    for (int sector=0; sector<14; sector++) {

        //! read in sector
        buf = (unsigned char*) flpydisk_read_sector ( _MountInfo.rootOffset + sector );

        //! get directory info
        directory = (PDIRECTORY) buf;

```

We are reading from the root directory. The root cluster is stored in `_MountInfo`, which contains information obtained from the **Bios Parameter Block (BPB)** when the file system was mounted. **\_MountInfo.rootOffset** contains the first cluster of the root directory. The root directory contains, at most, 224 DIRECTORY entries. A DIRECTORY entry is 32 bytes,  $224 \times 32 = 7168$  bytes,  $7168 \text{ bytes} / 512 \text{ bytes (512 bytes in a cluster)} = 14$ . This means the root directory consists of 14 clusters.

Knowing this, rather than loading the entire directory at once, we can load it sector by sector and parse each part of the directory.

```

    //! 16 entries per sector
    for (int i=0; i<16; i++) {

        //! get current filename
        char name[11];
        memcpy (name, directory->Filename, 11);
        name[11]=0;

        //! find a match?
        if (strcmp (DosFileName, name) == 0) {

```

Knowing that a DIRECTORY entry is 32 bytes,  $512 \text{ bytes per cluster} / 32 \text{ bytes} = 16$ . This means there are 16 DIRECTORY entries in one sector. So, we loop through each entry and compare filenames to locate the file or directory that we are looking for. If they match, we can create a new **FILE** object and return it. **file.currentCluster** will contain the first cluster of the file for reading later, **file.fileLength** contains the size of the file, in bytes. **directory->Attrib** contains the files attributes. We set it based on its DIRECTORY entry attribute.

```

        //! found it, set up file info

```

```
strcpy (file.name, DirectoryName);
file.id      = 0;
file.currentCluster = directory->FirstCluster;
file.eof     = 0;
file.fileLength = directory->FileSize;

//! set file type
if (directory->Attrib == 0x10)
    file.flags = FS_DIRECTORY;
else
    file.flags = FS_FILE;

//! return file
return file;
}
```

Almost there... If we have not found the file or directory yet, we just move onto the next DIRECTORY entry. If we never find the file, we set FS\_INVALID and return.

```
        //! go to next directory
        directory++;
    }
}

//! unable to find file
file.flags = FS_INVALID;
return file;
}
```

Thats it! The above routine works for directories and files in FAT12. By calling it, it will search the root directory for any folder or file name and return its information.

## SubDirectories

While the old version of FAT12 was flat, new versions of this file system supports subdirectories. This allows us to be able to use directories and manage a lot of files more easily. For example, it would be a good idea in a large OS to separate OS-specific files in a **system** directory, or a **user** directory containing user profiles.

A subdirectory is just an ordinary file with the DIRECTORY flag set. Because of this, we first need to know how to read files so lets look at that now.

## File Reading



## Format

Okay, so we can now parse directories and locate files. We now need a way to read the file's contents. Remember that, technically, we can already read the first 512 bytes of any file by just the **FirstCluster** field in the directory entry structure for that file. To read more than one cluster, we have to parse the **File Allocation Table (FAT)**.

Recall that FAT consists of a number of entries containing cluster numbers. The size of these entries depends on the filesystem. FAT12 has 12 bits per entry, FAT16 has 16 bits, FAT32 has 32 bits per entry.

Think of the FAT as - not as a linked list, but rather a table of entries that represent the whole physical disk. The first cluster of the disk is represented by the first entry of the FAT. The second cluster is represented by the second entry, and so on. This means there is a one to one relationship between a cluster and a FAT entry. This makes reading and writing files in FAT12 easy.

## Reading a file

To read a file, we just read the current cluster of the file. We try to locate its next cluster on disk by parsing the FAT table. After we find the next cluster, update the "current cluster" for the next file read.

The cluster to read was set when the file is opened. On the first call to this routine, **file->currentCluster** is the same as **DIRECTORY->FirstCluster**.

This cluster is an offset into the data area on disk. Lets recall the format of a FAT12 formatted disk and locate our FAT and data area:

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.
-------------	------------------------	-------------------------	-------------------------	-----------------------------------	--

Remember that each FAT takes 9 sectors. Because there are two FATs,  $9+9=18$ . We have also concluded that our root directory is 14 sectors in the previous section.  $18+14=32$ . This is the amount of sectors both FATs and the root directory take up. So far our equation is  **$32 + \text{file->currentCluster}$** . We need to subtract 1 and we have  **$32 + (\text{file->currentCluster} - 1)$** . This is the sector to read in and contains the file data.

```
void fsysFatRead(PFILE file, unsigned char* Buffer, unsigned int Length) {
    if (file) {
        //! starting physical sector
        unsigned int physSector = 32 + (file->currentCluster - 1);

        //! read in sector
        unsigned char* sector = (unsigned char*) flpydisk_read_sector ( physSector );

        //! copy block of memory
        memcpy (Buffer, sector, 512);
    }
}
```

To read in the next cluster we have to parse the FAT tables. Because a FAT table is 9 sectors, rather than reading all 9 sectors we determine what sector we need to read.

We first get a byte offset into where the next cluster will be in. To do this, we multiply the cluster value by the size of a cluster. This gets stored in **FAT\_Offset**. The size of a FAT32 cluster is 4 bytes, so we would multiply by 4 if we are using FAT32. We would multiply by 2 if we were using FAT16 as that uses 2 bytes per cluster entry. That's all fine of course, but what about FAT12? FAT12 uses 12 bits per cluster entry. That's 8 bits (for the 1st byte) and 4 bits (for the 2nd byte. 4 bits is half of 8 bits, so it's 0.5) so it's 1.5 bits per cluster entry.

After this, just divide this byte offset by the size of a sector to obtain the sector of the FAT to read in. The remainder is the offset in this sector, which is the cluster to read from the FAT. This is in **entryOffset**.

**FAT** is defined as **uint8\_t FAT [SECTOR\_SIZE\*2]**. Notice that we read 2 sectors of our FAT into memory instead of one. Why do this? Knowing a sector size is 512 bytes, 512 bytes \* 8 = 4096 bits per sector. 4096 bits / 12 bits (for a FAT entry) and we have 341.3333...etc. This means that an entry will sit between the 1st and 2nd sector. This will cause problems when loading files. Because of this, we have to load an additional sector so the last cluster value of the 1st sector will not be corrupt.

```

unsigned int FAT_Offset = file->currentCluster + (file->currentCluster / 2); //multiply by 1.5
unsigned int FAT_Sector = 1 + (FAT_Offset / SECTOR_SIZE);
unsigned int entryOffset = FAT_Offset % SECTOR_SIZE;

//! read 1st FAT sector
sector = (unsigned char*) flpydisk_read_sector ( FAT_Sector );
memcpy (FAT, sector, 512);

//! read 2nd FAT sector
sector = (unsigned char*) flpydisk_read_sector ( FAT_Sector + 1 );
memcpy (FAT + SECTOR_SIZE, sector, 512);

```

After the FAT sector has been read, we read in the cluster number.

Now we run into a problem. If we read an 8 bit value, we will not be able to read the whole 12 bits of a cluster value. So, we read 16 bits instead using an **uint16\_t**. Of course, now we have the problem of having too much bits of our 12 bit value.

Let's take a look closer. Let's say this is our FAT. We will separate our FAT into bytes but mark out the 12 bit entries. (This is taken from [Chapter 6](#))

Note: Binary numbers separated in bytes.  
Each 12 bit FAT cluster entry is displayed.

01011101	0111010	01110101	00111101	00111101	0111010	00111110	00111110
1st cluster		2nd cluster---		3rd cluster-		4th cluster----	
-0 cluster ----							

**Notice all even clusters accopy all of the first byte, but part of the second. Also notice that all odd clusters occopy a part of their first byte, but all of the second!**

With this in mind, this means if the cluster is even, **Mask out the top 4 bits, as it belongs to the next cluster.** If the cluster is odd, **shift it down 4 bits (to discard the bits used by the first cluster.)**

Now that we have all of that out of the way, lets finish off this function:

```

    //! read entry for next cluster
    uint16_t nextCluster = *( uint16_t*) &FAT [entryOffset];

    //! test if entry is odd or even
    if( file->currentCluster & 0x0001 )
        nextCluster >>= 4;    //grab high 12 bits
    else
        nextCluster &= 0x0FFF; //grab low 12 bits

    //! test for end of file
    if ( nextCluster >= 0xff8) {
        file->eof = 1;
        return;
    }

    //! test for file corruption
    if ( nextCluster == 0 ) {
        file->eof = 1;
        return;
    }

    //! set next cluster
    file->currentCluster = nextCluster;

```

## Writing a file

[To be completed in the chapter update!]

## SubDirectories

A **SubDirectory** is a file with the DIRECTORY attribute set. To read from a subdirectory, all we need to do is locate the FAT12 file on disk with that directory name and read it in the same way as with other files using the FAT.

After the file is loaded, from the first byte to the last is just an array of DIRECTORY entries. Parse the DIRECTORY entries the same way that we did with the root directory to read this directory :-). These will be the files and folders inside of the directory.

Lets take a look:

```
FILE fsysFatOpenSubDir (FILE kFile,
                      const char* filename) {

    FILE file;

    //! get 8.3 directory name
    char DosFileName[11];
    ToDosFileName (filename, DosFileName, 11);
    DosFileName[11]=0;
```

**filename** contains the file or directory that we want to find. **kFile** is the subdirectory that we want to parse. We convert the input filename, like "myfile.txt" into its DOS 8.3 filesystem format "MYFILE TXT" and store it in **DosFileName**.

```
    //! read directory
    while (! kFile.eof ) {

        //! read directory
        unsigned char buf[512];
        fsysFatRead (&file, buf, 512);

        //! set directort
        PDIRECTORY pkDir = (PDIRECTORY) buf;
```

**file** is our subdirectory that we want to parse. Remember that it is just an ordinary file in FAT12, so we read in a sector of the file. The file consists of an array of DIRECTORY entries. To make the DIRECTORY members easy to access, we use **pkDir** to point to the sector contents. Now, we search through the directory...

```
        //! 16 entries in buffer
        for (unsigned int i = 0; i < 16; i++) {

            //! get current filename
            char name[11];
            memcpy (name, pkDir->Filename, 11);
```

```

name[11]=0;

//! match?
if (strcmp (name, DosFileName) == 0) {

```

Each DIRECTORY entry is 32 bytes. A sector (also cluster in FAT12) is 512 bytes. 512 bytes / 32 bytes = 16 DIRECTORY entries per sector. So, we loop through all 16 entries to compare names. Once we find a filename matching the one we are searching, we have found the file.

```

        //! found it, set up file info
        strcpy (file.name, filename);
        file.id = 0;
        file.currentCluster = pkDir->FirstCluster;
        file.fileLength = pkDir->FileSize;
        file.eof = 0;

        //! set file type
        if (pkDir->Attrib == 0x10)
            file.flags = FS_DIRECTORY;
        else
            file.flags = FS_FILE;

        //! return file
        return file;
    }

```

When we have found the file, we fill in our FILE structure - first file cluster (so we can read it later on), file size (so we know when EOF is) and its attribute (file or directory).

If the file has not been found, we just move onto the next entry. This loop will continue until the end of the file. If no file is found, we set FS\_INVALID and return.

```

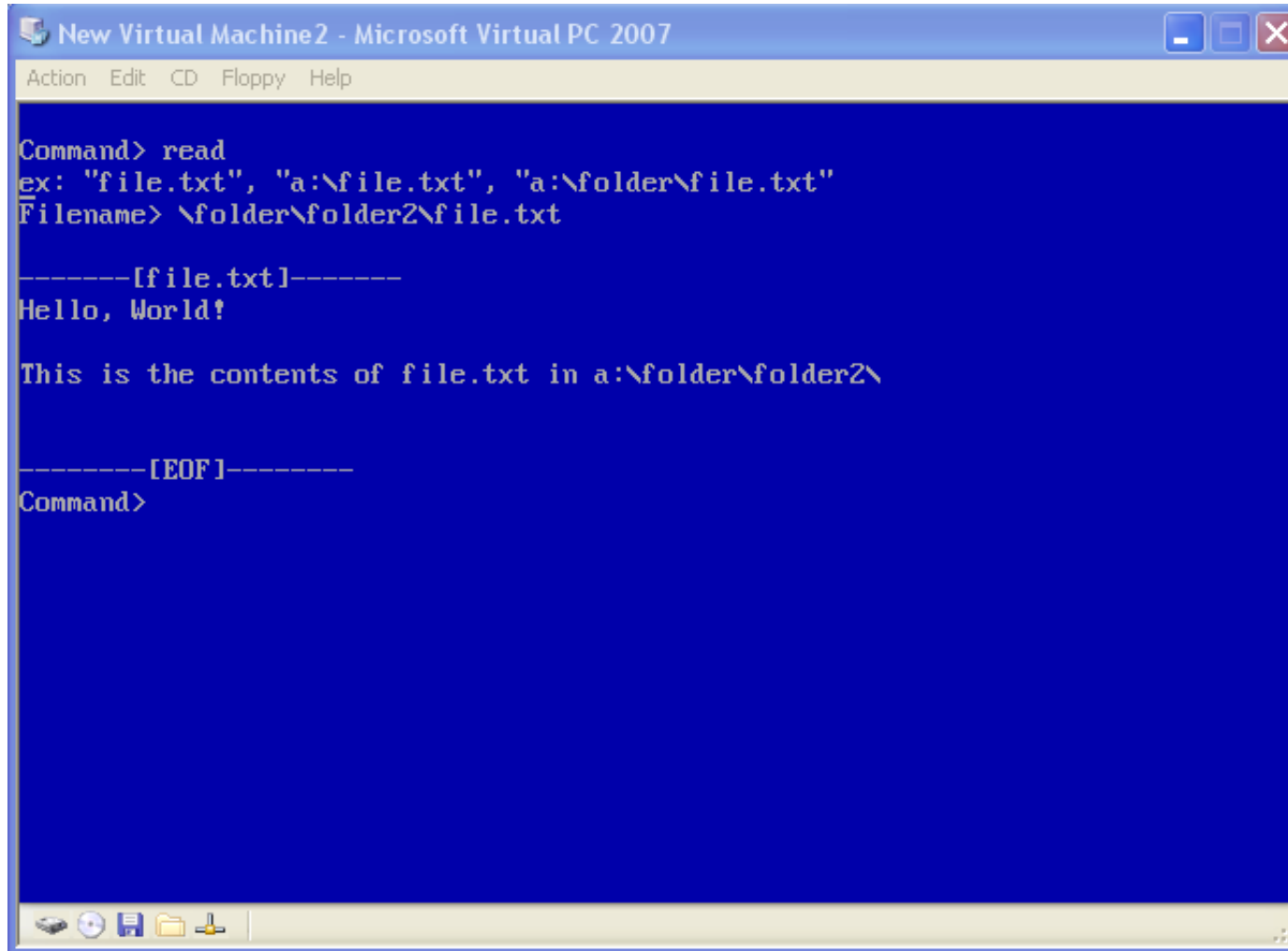
        //! go to next entry
        pkDir++;
    }

    //! unable to find file
    file.flags = FS_INVALID;
    return file;
}

```

Notice the similarities between this routine and our **FsysFatDirectory** routine.

## Demo



```
Command> read
ex: "file.txt", "a:\file.txt", "a:\folder\file.txt"
Filename> \folder\folder2\file.txt

-----[file.txt]-----
Hello, World!

This is the contents of file.txt in a:\folder\folder2\

-----[EOF]-----
Command>
```

*Viewing a file in our OS*  
[DEMO DOWNLOAD](#)

This chapter's demo puts everything we covered and implements a VFS and FAT12 minidriver. It is capable of supporting multiple filesystems, disk devices, subdirectory support, and loading and displaying files.

The demo is also capable of displaying large files and implements a "press a key to continue" feature for multi-cluster files.

This demo implements the **strchr()** ISO C routine in our CRT **string.c** to help with text parsing. It also upgrades our **read** command so it is capable of locating and displaying files instead of raw sectors.

The Volume Manager is very simplistic in this demo, implemented in **fsys.cpp**. It manages the registering and unregistering of file systems, and file system abstraction. You can call **volOpenFile()** to open a file. It defaults to opening **a:file.txt** but it will also work if you call it to open any file on any directory.

Not all file systems support subdirectories. Because of this, we leave subdirectory support to the file system drivers. Instead, the volume manager only handles the drive letter part of a path name. For example, if you call **volOpenFile ("a:\folder\file.txt")**, volOpenFile will pass **"\folder\file.txt"** to the file system registered on device 'a'. The file system driver is responsible for parsing the directory path name and opening subdirectories and files.

In the case of our FAT12 minidriver, this special routine is **fsysFatOpen()** which is responsible for parsing the directory path (like **"\folder\folder\file.txt"**) and calls its other file system routines for parsing and reading files and directories.

Thats it :-) This is possibly our last chapter covering FAT12. Because of this I do plan for an update covering writing files and directories on disk a little later.

## Conclusion

This was a fun chapter, huh? We are now able to load files from disk. I know, I know, "About time!" :) We are almost now ready to make the big leap into multitasking and executing programs. Before going multitasking, however, we should cover Loaders. A Loader is responsible for loading and executing a program, and mapping it into an address space. We also need to cover heap management and stack management in address spaces.

Because I plan to update the memory management chapter heavily, I might move heap and stack management in a chapter following the memory management chapter. In any case, I will be sure to keep you updated on changes.

This does, however, mean that it is almost time for us to dive into multitasking. Afterwords? User mode!

Until next time,

~Mike

*BrokenThorn Entertainment. Currently developing DoE and the [Neptune Operating System](#)*

*Questions or comments? Feel free to [Contact me](#).*

Would you like to contribute and help improve the articles? If so, please [let me know!](#)



