CS444 3/04/05

Linux Virtual File System

VFS Intro

The Linux Virtual File System (VFS) is not a file system at all. It is a generic interface that can be used to create file systems for the Linux kernel. All file systems that are supported within the Linux kernel must comply with the VFS interface. This allows any number of completely different file systems to coexist and interoperate freely.

Interoperatility

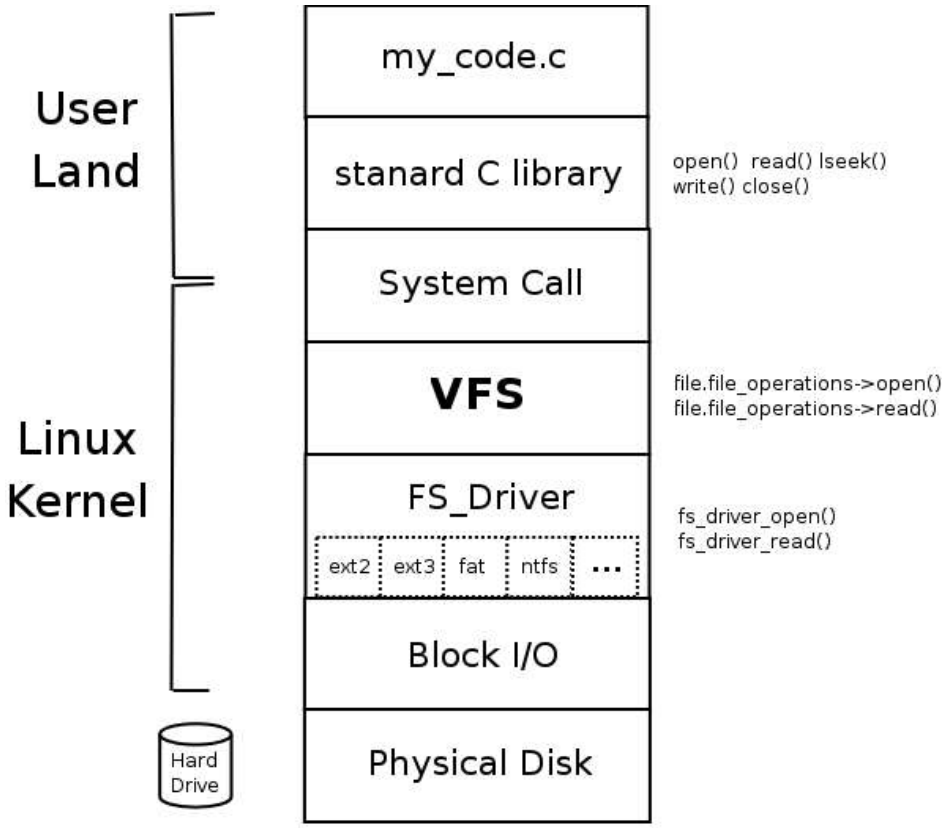
Consider this. You write a very simple C program to open a file and read the first 512 bytes.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>

int main()
{
        char buffer[512];
        int infile = open("hello.txt", O_RDONLY);

        read(infile, buffer, 512);
        printf("%s\n",buffer);
        close(infile);
}
```

Take a look at what is going on here. We open the file, read from the file and then close the file. Three basic file operations. Now, consider the following. What file system does “hello.txt” live on (ReiserFS)? Do the open(), read(), and close() functions know exactly how to perform those operations within the ReiserFS file system? What if “hello.txt” were to have been on EXT2 or EXT3, standard Linux file systems? What if it had been on a FAT32 or an NTFS file system? Obviously the task of opening reading and writing files will be different across all of these file systems.

Also consider this. The open() read() write() and close() functions are part of the standard C library. The standard C library works under Windows too, as well as many many other operating systems! Does the standard C library need to know how to read and write to every single type of file system ever created. No!!! Of course not :) That would be crazy. This is where VFS comes in.

When you make a call to the read function (for example), the standard C library actually calls the read system call that exists within the Linux kernel. That system call is responsible for figuring out which file system we are reading from and passing the read request onto the appropriate file system driver. All file systems have drivers that must know how to read files. This is the purpose of the VFS.



VFS Interface

The VFS is made up of exactly four different types of objects. Each file system in the Linux kernel must provide these data structures to the kernel when it requests them. All of these VFS objects are completely separate from whatever happens to live on the actual disk. For example, if you were writing your UserFS assignment to use VFS, your “inodes”

that live on disk would be separate from the “inode” in the VFS layer. Now, your on-disk inode would probably contain almost the exact same data that is needed for the VFS inode, but they are not the same. It is necessary got the VFS to operate this way in order to support file systems (like FAT32) that do not have inodes. FAT32 must construct an inode (by reading the relevant information from its own file system structures) on the fly when they are needed.

1. Superblock
2. inode
3. dentry
4. file

VFS Superblock

The VFS superblock contains information about the file system as a whole. It contains information like the file system's block size, the maximum file size, the file systems mount point, and most importantly the superblock_operations struct (we'll discuss later). A superblock must be created whenever a file system is mounted, One superblock exists for each mounted file system. So if you were writing UserFS using the VFS, you would need a function to read your superblock from the disk and convert it to a VFS superblock structure for the kernel. Pretend you wrote this function and called it “get_sb()”.

Inode

The VFS inode structure represents a single file. It holds information like the size of the file, access permissions, who owns the file, and most importantly the inode_operations struct (later). All of this is information about the file that is not really part of the file. It is just metadata that helps to describe the file. The purpose of the inode is to hold all of this metadata. One inode needs to be created for every file that needs to be accessed. So every time a user opens a file, an inode needs to be created. If you were writing UserFS using the VFS you would need a function to read your own inode off of the disk and use its data to create a VFS inode struct. Pretend you wrote this function and called it “read_inode()”.

Dentry

The VFS dentry object represents a single file or folder. it represents a single component of a path name. Look at this example:

/usr/bin/nano

If you were to attempt to access this file in Linux, four different dentry objects would have to be constructed. One for each of the following path conponenets:

- / (root of file system, mount point)
- usr (directory)
- bin (directory)
- nano (file)

Dentrys can be used to verify that a file or directory does actually exist, get the exact name of a file or directory, and traverse a file systems directory tree by looking at a dentry's parents and children. A dentry object contains information like the name of the file or folder it represents, the inode associated with the file/folder it represents, list of child dentries (like files in a directory), and parent dentry. Unlike the superblock and inode VFS objects, the dentry does not represent a unique on-disk structure, Instead, the file system must create dentries from a string representation of a path name, like "/usr/bin/nano". The file system will probably have to refer to some of its on-disk data in order to create dentries, even though the dentry does not represent an on-disk structure. Oh, and a dentry object also contains a dentry_operations struct that (you know what I'm going to say next, right??) we'll discuss later.

File

The VFS file object is an in-memory representation of an open file. A file object needs to be created for every file that is opened by some process. It contains data like the dentry for the file, file access mode (read, write, read/write) offset position (where will we read from next), and finally a file_operations struct.... You know... later :-D.

What does it all mean??

So you are writing a file system for the Linux kernel (UserFS). At some point in time someone is going to try to mount your file system. At that time, the kernel will see that UserFS is the file system type and it

will go to your UserFS driver and say “hey! I need your superblock! Run your get_sb() function and give me a VFS superblock object!” You're get_sb() function will then have to create a superblock struct, read in the needed data from the disk, and fill in that superblock struct, You will then need to return in to the kernel.

Next, some user tries to access a file “/usr/bin/nano” on your file system. The kernel will say “Hey! You need to see if this file exists.” You'll have to call your “read_inode()” function to read the inode / (the root directory of your FS). The location of this inode should be contained in your superblock! Your read_inode() function should be able to find the inode you are looking for on the disk, and then create and fill in a VFS inode struct with the needed data. You will need to do this for /, usr, bin, and nano.

Also, dentry objects will need to be made while you are creating and looking up the inodes.

Now that user is ready to open that file. The kernel will say to your file system driver “Hey! I need to open this file!! Give me a file object that represents the file /usr/bin/nano!” Your file system driver will have to create and initialize the VFS file struct and return it to the kernel so the kernel can then tell the user “Hey, this is your file! Its ready!”.

This seemingly loud “Hey! I'm the kernel!” voice is actually the VFS layer. It connects the kernel to a file system in a very specific way with a very specific interface. As long as a file system conforms to this VFS interface, Linux can access and use the file system seamlessly and it can interoperate with any other Linux file system.

Operations

Remember all those *_operations structs we kept putting off until later? Well, later has come. The VFS objects that we just discussed are a generic model of a file system. They allow us to view a particular file system in a generic and non-file system specific way. But what happened when we want to actually read data from a file, or write data to a file, or perform some other operations that might rely on the actual on-disk structure of the underlying file system? Lets pretend that the superblock has been updated and needs to be written to the disk. We will need to call some function that knows how to write our superblock to the disk. This is where the superblock_operations structure comes into play.

The superblock_operations structure containd a list of function pointers. Each one has a unique name and they each point to a function (exactly which function is determined by the actual file system driver)

that executes a specific task. The super_block operations struct has a function pointer called write_super() that will write an updated superblock to the disk.

Do you see the importance of these structures? Each file system that works with Linux provided its own superblock_operations structure and when Linux needs to execute a certain file system specific operation (like writing the superblock to the disk) it can just call the function pointed to by the appropriate function pointer in the superblock_operations struct!!

There are four operations structs that each define functions that can be used to carry out file system specific tasks. Here is a list of them and a few of their function pointers.

superblock_operations
- alloc_inode()
- destroy_inode()
- read_inode()
- write_inode()
- statfs()
- put_super()
- write_super()

inode_operations
- lookup()
- mkdir()
- rmdir()
- rename()

dentry_operations
- d_revalidate()
- d_compare()
- d_delete()

file_operations (this one will look most familiar to you)
- read()
- llseek()
- write()
- open()

Looking at VFS code.

Now that we have some idea how VFS works. Lets take a look at some of the source code that makes up VFS.

Take a look at the following files in your Linux kernel source code tree (you can browse this online at http://lxr.linux.no).

include/linux/fs.h

Line 429: The VFS inode struct definition lives here.
Line 577: The VFS file struct definition lives here.
Line 754: The VFS superblock struct definition lives here.

Line 908: file_operations struct definition lives here.
Line 936: inode_operations struct definition lives here.
Line 973: super_operations struct definition lives here.

include/linux/dcache.h

Line 83: The VFS dentry struct denifintion
Line 111: dentry_operations struct definition lives here.

fs/filesystems.c

Line 66: register_filesystem() function.

This function is what a file system driver calls in order to tell the kernel "Hey, I am the driver for the X file system". This means that the kernel will now recognize and allow the use of file system X.

Looking at implemented File systems

EXT2 is the most widely used Linux file system to date. Actually, EXT3 is the most widely used Linux file system, but EXT2 is forwards compatible and slightly simpler. Lets take a look at some of the EXT2 code and see if we can see how it implements VFS.

fs/ext2/super.c

Line 1015:

This is the code that actually loads the file system driver. As you can see, a call to register_filesystem() is made to tell the kernel to recognize the EXT2 file system and to use this driver when it needs to work with an EXT2 file system. Notice that a pointer to something called 'ext2_fs_type' is passed as an argument.

Line 1007:

Here is where this argument is defined. It is a struct of type file_system_type. It is what is used to tell the kernel about the existence of a new file system driver. It lists two very important things. 1) The name of the file system, “ext2” in this case, and 2) the function that will handle reading the superblock, “ext2_get_sb()” in this case.

Line 206:

Here you'll find the super_operations structure for EXT2. Remember, the purpose of this struct is to define which functions will handle certain file system specific events related to the superblock. Lets look at ext2_write_super().

Line 881:

This function is supposed to write a modified superblock back to the disk. Notice the nature of the data it is working with here. It is storing information like free block count and free inode count. Some of this info is kept in the ext2 superblock (the on-disk superblock), but NOT in the VFS superblock structure. Why is this? it is because VFS cannot contain fields for file system specific information. If it did it would not be generic. And yes blocks and inodes are file system specific because it is possible to write a file system without inodes! The FAT file system is an inode-less file system.

Now lets go back to the super_operations structure and take a look at the function that handles reading an inode. ext2_read_inode.

fs/ext2/inode.c

Line 1025:

This function has two main variables that are used. 'inode' which is the VFS inode passed as argument. The caller must set the inode number (inode->i_ino) and the superblock field (inode->i_sb) before calling this function.

The second variable it uses is raw_inode which represents the actual on disk inode structure. Remember, these two things are different!!! The goal of this function is to obtain the data needed from the on-disk inode and report it using the VFS inode data structure.

The call to the ext2_get_inode() function is what actually reads the inode off of the disk and stores it in the raw_inode variable.

Next we'll look at the FAT file system. Before Windows 2000 and Windows XP, Windows used the FAT32 file system. Before FAT32 was FAT16. FAT16 (or just FAT) is the file system that is used on floppy disks and very old Windows (Windows 95 and earlier) versions.

fs/fat/inode.c

Line 773:

This is where the super_operations structure is declared for the fat file system. Lets look at the fat_put_super() function.

Line 171:

Since the fat fs does not have an on disk super block, this should be a relatively short function. All it really does (aside from a few initialization tasks) is call the fat_clusters_flush() function. Look at this function in fs/fat/misc.c on line 51. All it does is, according to the comment, is write the number of free clusters (places to store data within the fat fs) to the physic disk.

Line 533:

The fat_read_root() function is supposed to read the root directory entry from the disk and fill in the argument inode appropriately. If it is a fat32 file system, then the location of the root directory is stored in the msdos_sb_info structure. Otherwise, the root directory is always located at cluster 0.