

CS169 Programming Assignment 2: Virtual File System

<i>Assignment Out:</i>	Monday, Oct. 6, 2008
<i>Helpsession:</i>	Tuesday, Oct. 7, 2008 (CIT 165, 8pm)
<i>Assignment Due:</i>	Monday, Oct. 27, 2008 (11:59 pm)

1 Introduction

The Virtual File System (VFS) is an interface providing a clearly defined link between the operating system kernel and the various file systems. The VFS makes it simple to add many different filesystems to your kernel and give them the same basic UNIX-style interface: with a VFS, you can play music by writing to `/dev/audio`, you can list your processes by reading `/proc/`, and redirect flames to `/dev/null`. One of the reasons Linux has taken over the world¹ is that it supports lots and lots of different filesystems — e.g. users who don't want to give up all their Windows files can keep them safely on a `fat32` partition.

In this assignment, as before, we will be giving you a bunch of header files and you will supply much of the code. You'll be working in the `vfs/` directory of your source tree. You will be creating the internal data structures (*files* and their associated *vnodes*), and writing the operations on the *vnodes* (*vnnode_ops*). We will be providing you with a simple in-memory filesystem for testing, (`testfs`), and you will need to write the special files to interact with devices. Finally, you will implement support for the system calls that operate on files: `open`, `close`, `read`, `write`, `lseek`, `mknod`, `mkdir`, `rmdir`, `dup`, `dup2`, `link`, `unlink`, `rename`, `chdir`, and `getdents`.

2 VFS Notes

In general, we have tried to place all information about the structure of VFS in the Kernel Hacker's Guide. However, there are a few notes specific to this assignment that just don't belong in that oh-so-holy tome. So here you go.

¹This is admittedly debatable.

2.1 Mounting

Before a file can be accessed, the file system containing the file must be *mounted* (a scary-sounding term for setting a couple of pointers). In standard UNIX, a superuser can use the system call `mount(2)`; in Weenix, mounting will be performed internally, at bootstrap time.

In `idleproc` a call is made to `vfs_init()`. This in turn calls `mountproc()` to mount the filesystem of type `VFS_ROOTFS_TYPE`. In the TA Weenix, the root filesystem type is `s5fs`. Since your implementation of `s5fs` will occur in a later assignment, for now your root filesystem is `testfs`, which provides all the operations of a filesystem except `link` and `getpage`.

The mounted file system is represented by an `fs_t` structure that is dynamically allocated at mount time.

Note that you do not have to handle mounting a file system on top of an existing file system, or deal with the mount point issues. There will be extra credit involved for those who implement this (it is a significant amount of work).

3 The Assignment

The following is a brief check-list of the features which you will be adding to your implementation of Weenix in this assignment.

- Setting Up the Filesystem: `vfs.c vnode.c file.c`
- The `testfs` Filesystem: `testfs.c` (mostly provided)
- Names to Vnodes: `namev.c`
- Opening Files: `open.c`
- VFS Syscall Implementation: `vfs_syscall.c`

4 Code Overview

We now go into a function-by-function breakdown of the assignment.

4.1 Setting Up The Filesystem

Functions you should take note of in `vfs/vfs.c`:

- `int vfs_init(void)`

Functions you should take note of in `vfs/vnode.c`:

- `void vnode_init()`
- `vnode_t *vget(struct fs *fs, int vnum)`
- `void vput(struct vnode *vn)`
- `void vref(vnode_t *vn)`

Functions you should write in `vfs/vnode.c`:

- `int special_file_read(vnode_t *file, int offset, void *buf, int count)`
- `int special_file_write(vnode_t *file, int offset, void *buf, int count)`

Functions you should take note of in `vfs/file.c`:

- `void file_init()`
- `file_t *fget(int fd)`
- `void fput(file_t *f)`

4.2 The testfs Filesystem

The `testfs` filesystem is an extremely simple filesystem that provides basic implementation of all the filesystems operations except `link` and `getpage`. Of course, there is no need for `getpage` until VM and `link` isn't implemented because it would require reference counting. We decided not to implement reference counting because it reduces the complexity of the code. This makes the code easier to understand and makes the one function you have to implement easier.

Of course this simplicity comes at a cost. For instance, `testfs` does not support multiple processes very well. Since there are no reference counts, one process can delete a file that another process has open without any complaint from the kernel. Keep this in mind when testing.

Functions you need to write in `vfs/testfs.c`:

- `static int testfs_readdir(vnode_t *dir, int offset, struct dirent *d)`

4.3 Names to Vnodes

At this point, you will have a filesystem mounted, but still have no way to convert a pathname into the vnode associated with a file at that path. This is where the namev functions come into play, as you probably guessed.

There is a series of functions which implement all your name-to-vnode converting needs. We'll give you an executive summary (see comments in the source code), and you'll have to implement them. Keeping track of vnode reference counts can get hairy here. Copious commenting and `dbg()`s will serve you well. Finally, don't forget that `lookup()` takes care of the special case when name is `".."` for a root dir.

Functions you need to write in `vfs/namev.c`:

- `int lookup(vnode_t *dir, const char *name, int len, vnode_t **result)`
- `int dir_namev(const char *pathname, int *namelen, const char ** name, vnode_t *base, vnode_t **res_vnode)`
- `int open_namev(const char *pathname, int flag, int mode, vnode_t **res_vnode, vnode_t *base)`

4.4 Opening Files

The functions herein will allow you to actually open files in a process. When you have open files you should have some sort of protection mechanism so that one process cannot delete a file that another process is using. You can do that either here or in the S5 layer. Although the choice is up to you, it's somewhat easier to do it in the S5 layer and not worry about it here.

Functions you will need to write in `vfs/open.c`:

- `int do_open(const char *filename, int oflags)`

4.5 VFS Syscall Implementation

At this point, you've mounted your `testfs`; now you want to write the code that will allow you to test your code vigorously. The syscall functions act as the link between user and kernel mode. When a user program makes a call to `read`, `do_read` will eventually be called. Thus you must be vigilant in checking for any and all types of errors that might occur along the way and return appropriate error codes.

Note: for each of these functions, it is understood that you will handle error conditions by returning `-errno` if there is an error. A return value less than zero is assumed to be an error. You should read corresponding syscall man pages for hints on both `errno` return values and the implementation of these functions. Read the manpage for `errno` for a specific breakdown of what all the error values mean. This may look like a lot, but once you get one of the syscalls, it's easy to get all of them. Pay special attention to the comments and use lots of `dbg()`.

Functions you will need to write in `vfs/vfs_syscall.c`:

- `int do_read(int fd, void *buf, size_t nbytes)`
- `int do_write(int fd, const void *buf, size_t nbytes)`
- `int do_close(int fd)`
- `int do_dup(int fd)`
- `int do_dup2(int ofd, int nfd)`
- `int do_mknod(const char *path, int mode, unsigned devid)`
- `int do_mkdir(const char *path)`
- `int do_rmdir(const char *path)`
- `int do_unlink(const char *path)`
- `int do_link(const char *from, const char *to)`
- `int do_rename(const char *oldname, const char *newname)`
- `int do_chdir(const char *path)`
- `int do_getdent(int fd, struct dirent *dirp)`
- `int do_lseek(int fd, int offset, int whence)`
- `int do_stat(const char *path, struct stat *buf)`

5 Test Code

You should have lots of good test code. What does this mean? It means that you should be able to demonstrate fairly clearly — via tty i/o and `printfs` — that you have successfully implemented as much as possible without having a `s5fs` yet. Moreover, you want to be sure that your refcounts are correct — that is, when your simulator shuts down, `vfs_shutdown()` will check refcounts. Basically, convince *yourself* that this mostly works; and get ready for the next assignment — implementing a real file system.

To further help you test your code, we have included a kernel mode version of the `vfstester` you'll see again in fi. You can find it at `vfs/vfs_privtest.c`. `Privtest` is not a complete test suite. If

you are able to run the `privtest` and have your filesystem unmount and shut down properly, you're off to a good start, but `privtest` alone is not sufficient.

6 Notes on Compilation

Open up `Makefile.defines` and uncomment the `VFS = true` line and you should be good to go.

7 Handing In

As with `kern`, you are required to hand in your kernel when you are done with this assignment. This is because we like to keep a close eye on your progress as you go through the semester. Remember, your grade is based on your demo and not the code you handin, so don't bomb your demo. Hand in using the following command:

```
make clean;/course/cs169/bin/cs169_handin vfs
```