VE482, Lab 8 Report, Group 9

1. What is a filesystem?

   A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the filesystem[1].

2. How is the Linux VFS working?

   In Linux, all files are accessed through VFS. This is a layer of code which implements generic filesystem actions and vectors requests to the correct specific code to handle the request[2]. Two main types of code modules take advantage of the VFS services, which are device drivers and filesystems.

   The role of the VFS is:

   - Keep track of available filesystem types
   - Associate (and dissociate) devices with instances of the appropriate filesystem.
   - Do any reasonable generic processing for operations involving files.
   - When filesystem-specific operations become necessary, vector them to the filesystem in charge of the file, directory, or inode in question.

   The interaction between the VFS and specific filesystem types occurs through two main data structures, the super_block structure and the inode structure, and their associated data structures, including super_operations , inode_operations, file_operations , and others.

   The source code for the VFS is in the fs/ subdirectory of the Linux kernel source, along with a few other related pieces, such as the buffer cache and code to deal with each executable file format. Each specific filesystem is kept in a lower subdirectory.

   It uses init_name_fs() to register filesystem with the registry kept in fs/super.c. It's also possible for a filesystem to support more than one type of filesystem. For instance, in fs/sysv/inode.c, three possible filesystem types are supported by one filesystem with the following codes:

```
static struct file_system_type sysv_fs_type[3] = {
        {sysv_read_super, "xenix", 1, NULL},
        {sysv_read_super, "sysv", 1, NULL},
        {sysv_read_super, "coherent", 1, NULL}
};

int init_sysv_fs(void)
{
        int i;
        int ouch;

        for (i = 0; i < 3; i++) {
                if ((ouch = register_filesystem(&sysv_fs_type[i])) != 0)
                        return ouch;
        }
        return ouch;
}
```

   To connect the filesystem to a disk, it will call a function(for example, ext2_read_super() for ext2 filesystem) to see if a device can be mounted to the system, then if it succeeds in reading the superblock and is able to mount the filesystem, it fills in the super_block structure with information that includes a pointer to a structure called super_operations, which contains pointers to functions which do common operations related to superblocks. When a filesystem is mounted, do_umount() calls read_super, which

[1] https://www.tldp.org/LDP/sag/html/filesystems.html
[2] https://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html

ends up calling the reading function, which returns the superblock, which points to that structure of pointers to functions in _sops.

The inode_operations structure is:

```
struct inode_operations {
        struct file_operations * default_file_ops;
        int (*create) (struct inode *,const char *,int,int,struct inode **);
        int (*lookup) (struct inode *,const char *,int,struct inode **);
        int (*link) (struct inode *,struct inode *,const char *,int);
        int (*unlink) (struct inode *,const char *,int);
        int (*symlink) (struct inode *,const char *,int,const char *);
        int (*mkdir) (struct inode *,const char *,int,int);
        int (*rmdir) (struct inode *,const char *,int);
        int (*mknod) (struct inode *,const char *,int,int,int);
        int (*rename) (struct inode *,const char *,int,struct inode *,const char *,int);
        int (*readlink) (struct inode *,char *,int);
        int (*follow_link) (struct inode *,struct inode *,int,int,struct inode **);
        int (*readpage) (struct inode *, struct page *);
        int (*writepage) (struct inode *, struct page *);
        int (*bmap) (struct inode *,int);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*smap) (struct inode *,int);
};
```

Most of these functions map directly to system calls. There are also many system calls related to files and directories which are in the file_operations structure. This is the same one used when writing device drivers and contains operations that work specifically on files, as the following codes show:

```
struct file_operations {
        int (*lseek) (struct inode *, struct file *, off_t, int);
        int (*read) (struct inode *, struct file *, char *, int);
        int (*write) (struct inode *, struct file *, const char *, int);
        int (*readdir) (struct inode *, struct file *, void *, filldir_t);
        int (*select) (struct inode *, struct file *, int, select_table *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
        int (*fasync) (struct inode *, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
};
```

3. What is FUSE, and how does it interact with the VFS? Can you sketch it quickly to make it clearer?

   Fuse is the abbreviation for Filesystem in Userspace, which is a software interface for Unix-like computer operating systems that lets non-privileged users create their own files systems without editing kernel code[3]. It provides a "bridge" between the file system running in user space with the kernel interfaces.

   The FUSE project consists of two components: the fuse kernel module(maintained in the regular kernel repositories) and the libfuse userspace library. libfuse provides the reference implementation for communicating with the FUSE kernel module[4].

   A FUSE file system is typically implemented as a standalone application that links with libfuse. libfuse provides functions to mount the file system, unmount it, read requests from the kernel, and send responses back. libfuse offers two APIs: a "high-level", synchronous API, and a "low-level" asynchronous API. In both cases, incoming requests from the kernel are passed to the main program using callbacks. When using the high-level API, the callbacks may work with file names and paths instead of inodes, and processing of a request finishes when the callback function returns. When using

[3] https://en.wikipedia.org/wiki/Filesystem_in_Userspace
[4] https://github.com/libfuse/libfuse

the low-level API, the callbacks must work with inodes and responses must be sent explicitly using a separate set of API functions. Figure 1 shows how it works[5].

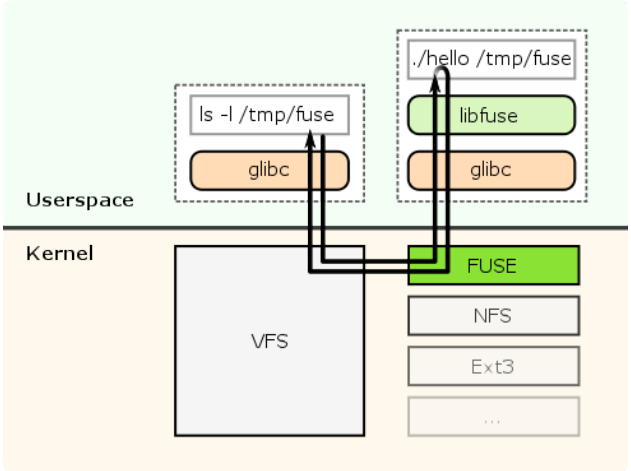

Figure 1: A flow-chart diagram illustrating how FUSE works

This is the illustration corresponds to the "hello world" file system on the FUSE website. At a high level the "hello world" file system is compiled to create a binary called "hello". This binary is executed in the upper right hand corner of the illustration with a file system mount point of /tmp/fuse. Then the user executes an ls -l command against the mount point (ls -l /tmp/fuse). This commands goes through glibc to the VFS. The VFS then goes to the FUSE module since the mount point corresponds to a FUSE based file system. The FUSE kernel module then goes through glibc and libfuse (libfuse is the FUSE library in user space) and contacts the actual file system binary ("hello"). The file system binary returns the results back down the stack to the FUSE kernel model, back through the VFS, and finally back to the ls -l command. There are some details I have glossed over of course, but the figure illustrates the general flow of operations and data.

[5] http://www.linux-mag.com/id/7814/