LINUX INTERNALS

VFS : OPEN FILE KERNEL DATA STRUCTURES

Important Notice : Courseware - Legal

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

The Virtual Filesystem Switch (VFS) and some of it's important Data Structures

The virtual file system (VFS) is an interesting aspect of the Linux kernel because it provides a common interface abstraction for file systems. The VFS provides a switching layer between the SCI (system call interface) and the file systems supported by the kernel.

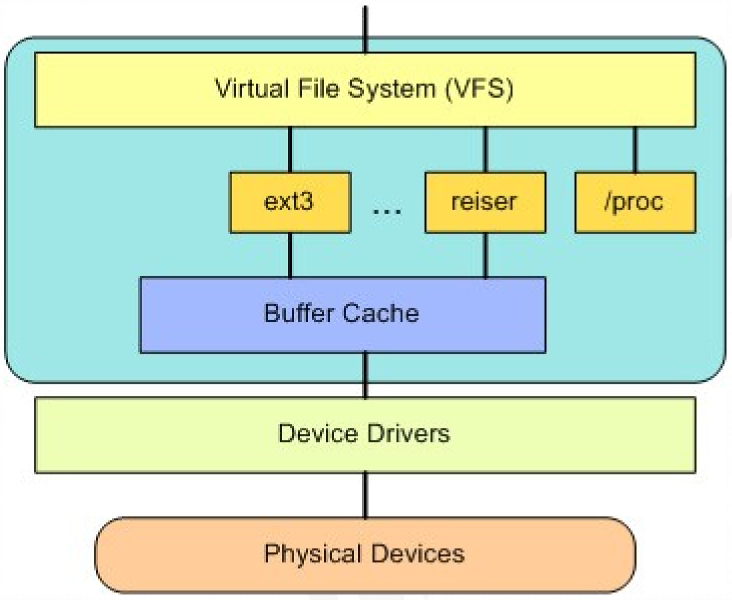

Figure : The VFS provides a switching fabric between users and file systems [this para sourced from: http://www.ibm.com/developerworks/linux/library/l-linux-kernel/]

<< 'Buffer Cache' is the old name for 'Page Cache' >>

At the top of the VFS is a common API abstraction of functions such as open, close, read, and write. At the bottom of the VFS are the file system abstractions that define how the upper-layer functions are implemented. These are plug-ins for the given file system (of which over 50 exist). You can find the file system source in the fs/ branch of the kernel source tree.

Below the file system layer is the buffer cache, which provides a common set of functions to the file system layer (independent of any particular file system). This caching layer optimizes access to the physical devices by keeping data around for a short time (or speculatively read ahead so that the data is available when needed). Below the buffer cache are the device drivers, which implement the interface for the particular physical device.

Each process on the system has its own list of open files, root filesystem, current working directory, mount points, and so on. Three data structures tie together the VFS layer and the processes on the system: the files_struct, fs_struct, and namespace structure.

<< We also mention the 'file' structure- the object that represents an open file (by a process or thread) on the OS; it is pointed to by the first structure above, namely, the 'files_struct'. >>

The Open Files Table - the files_struct structure

The files_struct is defined in *<linux/fdtable.h>*. This table's address is pointed to by the *files* entry in the process descriptor. All per-process information about open files and file descriptors is contained therein.

*<< The **`files_struct`** structure implements the Unix **OFDT (open file descriptor table)** of a Linux process. >>*

```
...
/*
 * The default fd array needs to be at least BITS_PER_LONG,
 * as this is the granularity returned by copy_fdset().
 */
#define NR_OPEN_DEFAULT BITS_PER_LONG

/*
 * The embedded_fd_set is a small fd_set,
 * suitable for most tasks (which open <= BITS_PER_LONG files)
 */
struct embedded_fd_set {
        unsigned long fds_bits[1];
};

struct fdtable {
        unsigned int max_fds;
        struct file ** fd;      /* current fd array */
        fd_set *close_on_exec;
        fd_set *open_fds;
        struct rcu_head rcu;
        struct fdtable *next;
};
```

```
/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
        atomic_t count;
        struct fdtable *fdt;
        struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
        spinlock_t file_lock ____cacheline_aligned_in_smp;
        int next_fd;
        struct embedded_fd_set close_on_exec_init;
        struct embedded_fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];
};
```

The fd array points to the list of open file objects. By default, this is the *fd_array* array. Because NR_OPEN_DEFAULT is equal to 32, this includes room for 32 file objects. If a process opens more than 32 file objects, the kernel allocates a new array and points the fd pointer at it. In this fashion, access to a reasonable number of file objects is quick, taking place in a static array. In the case that a process opens an abnormal number of files, the kernel can create a new array. If the majority of processes on a system open more than 32 files, for optimum performance the administrator can increase the NR_OPEN_DEFAULT preprocessor macro to match.

<<

Using **procfs** to see the open files held by a process (the “evince” PDF viewer, in the example below):

```
$ ls -l /proc/$(pgrep soffic)/fd
total 0
lr-x------. 1 kai kai 64 May 14 07:55 0 -> /dev/null
lrwx------. 1 kai kai 64 May 14 07:55 1 -> socket:[42094]
lrwx------. 1 kai kai 64 May 14 07:55 10 -> anon_inode:[eventfd]
lr-x------. 1 kai kai 64 May 14 07:55 11 -> /usr/lib64/libreoffice/program/types/oovbaapi.rdb
lr-x------. 1 kai kai 64 May 14 07:55 12 -> /usr/lib64/libreoffice/program/types/offapi.rdb
lrwx------. 1 kai kai 64 May 14 07:55 13 -> socket:[115243]
lrwx------. 1 kai kai 64 May 14 07:55 15 -> socket:[116116]
lrwx------. 1 kai kai 64 May 14 07:55 16 -> anon_inode:[eventfd]
lrwx------. 1 kai kai 64 May 14 07:55 17 -> anon_inode:[eventfd]
lrwx------. 1 kai kai 64 May 14 07:55 18 -> /run/user/1000/wayland-cursor-shared-LB6ilw (deleted)
lrwx------. 1 kai kai 64 May 14 07:55 19 -> socket:[116119]
```

```
lrwx------. 1 kai kai 64 May 14 07:55 2 -> socket:[42097]

--snip--

lr-x------. 1 kai kai 64 May 14 07:55 5 -> pipe:[113474]
l-wx------. 1 kai kai 64 May 14 07:55 6 -> pipe:[113474]
lr-x------. 1 kai kai 64 May 14 07:55 7 -> pipe:[118086]
l-wx------. 1 kai kai 64 May 14 07:55 8 -> pipe:[118086]
lr-x------. 1 kai kai 64 May 14 07:55 9 -> /usr/lib64/libreoffice/program/types.rdb
$
```

>>

The File Table - the file structure

- Every open file on the system has a corresponding struct file structure in kernel memory.
- The equivalent of the **FTE** (File Table Entry) in UNIX
- Created by the kernel in the *open(2)* system call
- Referenced from user-space via the file descriptor as an index in an array of file structures
  *struct file * fd_array[NR_OPEN_DEFAULT];*
  and by the dynamic entry "struct file ** fd"

This definition is shown in the struct files_struct (the OFDT of the process), shown above.

- Entries 0, 1 and 2 always represent the file descriptors representing standard input, standard output and standard error respectively of the process
- The file structure is kept in kernel memory until the last close(2) on the file (i.e. until *filp->f_count* == 0)
- In the kernel sources, a pointer to struct file is frequently referred to as either **file** or **filp** ("file pointer")
- File objects exist only in kernel RAM- they have no corresponding objects on disk (no backing store) and hence no "dirty" field to specify that the object has been modified.

The members of the **file** structure, representing an open file are shown below[1] (from *include/linux/fs.h*):

Data Type	Field Name	Description
struct list_head	f_list	pointers for generic file object list
struct inode *	f_inode	(cached) file's inode pointer
~~struct dentry *~~	~~f_dentry~~	~~dentry object associated with the open file~~
struct vfsmount *	f_vfsmnt	mounted filesystem containing the file
const struct file_operations *f_op		pointer to file operation table
atomic_t	f_count	file object's reference counter
unsigned int	f_flags	flags specified when opening the file (like O_CREAT|O_RDWR|_NONBLOCK...)
mode_t	f_mode	process access mode
int	f_error	error code for network write operation
loff_t	f_pos	current file offset (file pointer position)
struct fown_struct	f_owner	data for I/O event notification via signals
unsigned int	f_uid	user's UID
unsigned int	f_gid	user's GID
struct file_ra_state	f_ra	file read-ahead state
size_t	f_maxcount	maximum number of bytes that can be read or written with a single operation (currently set to $2^{31} - 1$)
unsigned long	f_version	version number, automatically increased after each use
void *	f_security	pointer to file object's security structure
void *	private_data	pointer to data specific for a filesystem or device driver
struct address_space *	f_mapping	pointer to the file's address space object

[1] *"Understanding the Linux Kernel" by Bover & Cesati, 3rd Ed, O'Reilly & Associates*

struct dentry *f_dentry :
The dirent (directory entry) structure associated with this open file.
The dentry's are all cached - they are an optimization introduced in the 2.1 development series; for our purposes, we might use it to look up the inode structure:

filp→f_inode *[Newer; tested on a 4.2 Linux OS]*

[OLDER: filp→f_dentry→d_inode]

On more recent kernels:

```
...
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
};
…
struct path     f_path;
#define f_dentry f_path.dentry  << not there on recent
```

kernels >>

…

loff_t f_pos :
The current read/write pointer position (seek position).
loff_t is a 64-bit value (long long in gcc terminology); this value should be treated as read-only by a driver - the read/write routines should not update this value directly but rather the pointer they are passed as the last parameter (seen later).

unsigned int f_flags :
These are the flags passed to the file from open(2) - **O_RDONLY, O_WRONLY, O_RDWR, O_NONBLOCK, O_SYNC,** etc.
A driver would particularly need to check **O_NONBLOCK** for nonblocking I/O (seen later); other flags like read/write permissions should be checked using **f_mode** (above).

struct file_operations *f_op :
Described in some detail in the next section below.

void *private_data :
This field can be used by a device driver to store some state data across system calls. The driver is responsible for allocating memory; also, it must free this memory when the file structure is finally destroyed.

The File Operations pointer - the file_operations structure

- These are the FOPs - the file operations to be defined and carried out by a module/device driver as appropriate.
- The structure essentially consists of **function pointers** representing the functions – **registered methods** - that will ultimately act upon the open file. These function pointers are the functions that implement the file-related system calls like open, close, read, write, lseek, etc.
- A pointer to the FOPs structure is found in the file structure 'file'.
- A device or filesystem is free to implement just as many functions as it requires available in user-space; the rest are set to NULL. These (the 'null' ones) revert to a VFS-default or generic action.
- The FOPs play an essential role in implementing the "*every device is a file*" philosophy.
  << The UNIX design philosophy can perhaps be summed up by the phrase: *"On UNIX, everything is a process; if it's not a process, it's a file".*
  >>

Here is the file operations structure (defined in *include/linux/fs.h*):

```
struct file_operations {
        struct module *owner;                   << set to THIS_MODULE >>
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
<< 4.1 : aio_read() and aio_write() removed; should use read_iter() and write_iter() >>
       ~~ssize_t (*aio_read) (struct kiocb *, const struct iovec *,~~
~~                      unsigned long, loff_t);~~
~~        ssize_t (*aio_write) (struct kiocb *, const struct iovec *,~~
~~                      unsigned long, loff_t);~~
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        int (*readdir) (struct file *, void *, filldir_t);
         unsigned int (*poll) (struct file *, struct poll_table_struct *);

<< From 2.6.36, the 'ioctl' method changed to these two: >>

       ~~int (*ioctl) (struct inode *, struct file *, unsigned int,~~
~~                    unsigned long);~~
       long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
       long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
<< Reason: The intention behind removing the .ioctl operation is to be able to remove the big
```

kernel lock (BKL) from the ioctl system call. Use the 'unlocked_ioctl' method now.. >>

```
	int (*mmap) (struct file *, struct vm_area_struct *);
	int (*mremap)(struct file *, struct vm_area_struct *);
	int (*open) (struct inode *, struct file *);
	int (*flush) (struct file *, fl_owner_t id);
	int (*release) (struct inode *, struct file *);
	int (*fsync) (struct file *, struct dentry *, int datasync);
	int (*aio_fsync) (struct kiocb *, int datasync);
	int (*fasync) (int, struct file *, int);
	int (*lock) (struct file *, int, struct file_lock *);
	ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
	unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
unsigned long, unsigned long);
	int (*check_flags)(int);
	int (*flock) (struct file *, int, struct file_lock *);
	ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
unsigned int);
	ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
unsigned int);
	int (*setlease)(struct file *, long, struct file_lock **, void **);
	long (*fallocate)(struct file *file, int mode, loff_t offset,
			  loff_t len);
	void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
	unsigned (*mmap_capabilities)(struct file *);
#endif
};
```

- In essence, all that the VFS does on receiving a file-related system call “foo()” from userpsace is invoke:

  filp->f_op->foo(...)

  as long as it's not NULL (where *filp* is the pointer to the struct file (FTE)).
  See the Sidebar (and code) below (on Opening a File and 'Read/Write on a Open File' for details).

In effect, the design idea is very much **an Object Oriented (OO) one**, though of course, the code isn't. We're essentially using the notion of **virtual methods**. The Linux kernel makes heavy use of virtual methods. They are implemented, of course, as function pointers in data structures.

This methodology is very commonly used in the Linux kernel and is in fact based on a *design pattern* called the '**strategy pattern**' (from the wikipedia article: “... Strategy lets the algorithm vary independently from clients that use it. ... “).

VFS

- Each filesystem includes it's own set of file operations that perform activities such as read, write, lseek, etc on a file. When the kernel loads an inode into memory from disk, it stores a pointer to these file operations in a file_operations structure (above) whose address is contained in the i_fop field of the inode object.

- When a process opens the file, the VFS initializes the f_op field of the new file object with the address stored in the inode so that subsequent calls to file operations (any file-related system calls, in effect) will use these functions.

- If necessary, the VFS (or a device driver), may later modify the set of file operations by storing a new value in f_op.

SIDEBAR : Opening a File

Files must be opened before reading or writing. In the view of the application, this is done by the open function of the standard library, which returns a file descriptor. The function uses the identically named open system call, which invokes the *sys_open* function in *fs/open.c*. The associated code flow diagram is shown in Figure 8-11.

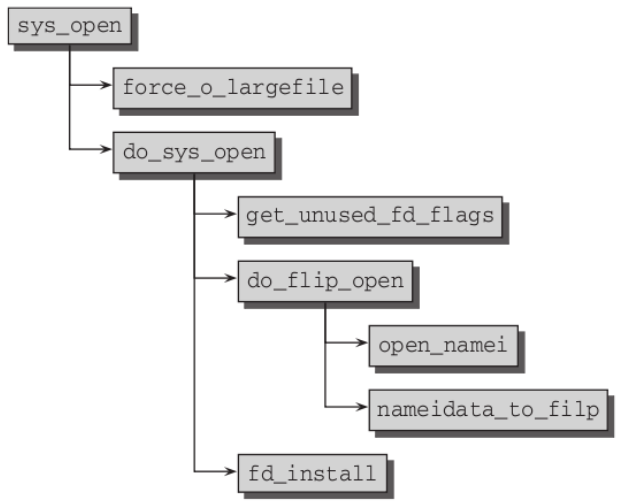

Figure 8-11: Code flow diagram for **sys_open**.

As a first step, force_o_largefile checks if the flag O_LARGEFILE should always be set irregardless of which flags were passed from userland. This is the case if the word size of the underlying processor is not 32 bits, that is, a 64-bit system. Such systems use 64-bit indexing, and large files are thus the only sensible default on them. The proper work of opening the file is then delegated to do_sys_open.

In the kernel, each opened file is represented by a file descriptor that acts as a position index

for
a process-specific array (task_struct->files->fd_array). This array contains an instance of the above-mentioned file structure with all necessary file information for each opened file. For this reason, get_unused_fd_flags is first invoked to find a used file descriptor.

Because a string with the name of the file is used as a system call parameter, **the main problem is to find** the matching inode. The procedure described below does this.

do_filp_open finds the file inode with the support of two helper functions.

1. *open_namei* invokes the *path_lookup* function to find the inode and performs several additional checks (e.g., to ascertain whether the application is trying to open a directory as if it were a regular file). If a new filesystem entry needs to be created, the function also applies the current default settings for the permission bits as stored in the process's umask (current->fs->umask).
2. *nameidata_to_filp* initializes the readahead structure, places the newly generated file instance on the s_files list of the superblock (see Section 8.4.1), and invokes the open function in the file_operations structure of the underlying filesystem.

fd_install must then install the file instance in files->fd from the task structure of the process before control is transferred back to the user process to which the file descriptor is returned.

Regular File I/O : Reading & Writing

How regular file i/o is done: the VFS open file (represented by 'filp'), filp->f_op is set to the inode->i_fop pointer for the underlying filesystem.

When? During the open...:

```
 sys_open* --> do_sys_open --> do_filp_open -->
                   nameidata_to_filp --> __dentry_open
```

In fs/open.c:__dentry_open

```
...
 f->f_pos = 0;
 f->f_op = fops_get(inode->i_fop);
...
```

*In recent kernels, sys_open shows up as:
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)

The inode i_fop has already been setup by the filesystem code. For eg on ext2:
(fs/ext2/namei.c)

```
...
     inode->i_fop = &ext2_file_operations;
...
```

When userspace issues a file-related system call, the VFS acts in an object-oriented fashion, merely invoking the open file's registerd method (via the f_op).
When the VFS detects that the file being opened is a device file, it invokes the registered

driver's open method which typically hooks filp->f_op pointers to the driver's own f_op data structure(s).

Userspace 'read' system call becomes 'sys_read' in the kernel (implemented as SYSCALL_DEFINE3 in recent kernels):

In fs/read_write.c
<code>
```
...
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t,
count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    file = fget_light(fd, &fput_needed);
    if (file) {
        loff_t pos = file_pos_read(file);
        ret = vfs_read(file, buf, count, &pos);
        file_pos_write(file, pos);
        fput_light(file, fput_needed);
    }

    return ret;
}

…

ssize_t vfs_read(struct file *file, char __user *buf, size_t count,
loff_t *pos)
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_READ))
        return -EBADF;
…
…
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
...
```
</code>

As seen above, if for some reason the underlying filesystem (or driver) has not initialized the open file's f_op pointer, the VFS resorts to calling a "generic" read (or write) – the 'do_sync_read' function (or write):

$$\begin{tikzcd} & & & & \text{filp->f\_op->read} \\ \text{read} \arrow[r] & \text{sys\_read*} \arrow[r] & \text{vfs\_read} \arrow[r] & \text{filp->f\_op->read exists?} \arrow[ur, "Y"] & \end{tikzcd}$$

N ↘ do_sync_read

do_sync_[read|write] ultimately become *filp->fop->aio_[read|write]* (with wait semantics).

*In recent kernels, sys_read shows up as:
```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
```