The Virtual File System (VFS)

The `proc` File System

The Linux kernel has two primary functions: to control access to physical devices on the computer and to schedule when and how processes interact with these devices. The `/proc/` directory contains a hierarchy of special files which represent the current state of the kernel — allowing applications and users to peer into the kernel's view of the system.

Within the `/proc/` directory, one can find a wealth of information about the system hardware and any processes currently running. In addition, some of the files within the `/proc/` directory tree can be manipulated by users and applications to communicate configuration changes to the kernel.

Under Linux, all data are stored as files. Most users are familiar with the two primary types of files: text and binary. But the `/proc/` directory contains another type of file called a *virtual file*. It is for this reason that `/proc/` is often referred to as a *virtual file system*.

These virtual files have unique qualities. Most of them are listed as zero bytes in size and yet when one is viewed, it can contain a large amount of information. In addition, most of the time and date settings on virtual files reflect the current time and date, indicative of the fact they constantly changing.

Virtual files such as `interrupts`, `/proc/meminfo`, `/proc/mounts`, and `/proc/partitions` provide an up-to-the-moment glimpse of the system's hardware. Others, like `/proc/filesystems` and the `/proc/sys/` directory provide system configuration information and interfaces.

For organizational purposes, files containing information on a similar topic are grouped into virtual directories and sub-directories. For instance, `/proc/ide/` contains information for all physical IDE devices. Likewise, process directories contain information about each running process on the system.

Viewing Virtual Files

By using the `cat`, `more`, or `less` commands on files within the `/proc/` directory, you can immediately access an enormous amount of information about the system. For example, if you want to see what sort of CPU your computer has, type `cat /proc/cpuinfo` and you will see something similar to the following:

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 5
model           : 9
model name      : AMD-K6(tm) 3D+ Processor
stepping        : 1
cpu MHz         : 400.919
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
```

```
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr mce cx8 pge mmx syscall 3dnow
k6_mtrr
bogomips        : 799.53
```

When viewing different virtual files in the `/proc/` file system, you will notice some of the information is easily understandable while some is not human-readable. This is in part why utilities exist to pull data from virtual files and display it in a useful way. Some examples of such applications are `lspci`, `apm`, `free`, and `top`.

Note

Some of the virtual files in the `/proc/` directory are only readable by the root user.

Changing Virtual Files

As a general rule, most virtual files within the `/proc/` directory are read only. However, some can be used to adjust settings in the kernel. This is especially true for files in the `/proc/sys/` subdirectory.

To change the value of a virtual file, use the `echo` command and a **>** symbol to redirect the new value to the file. For instance, to change your hostname on the fly, you can type:

```
echo bob.subgenius.com > /proc/sys/kernel/hostname
```

Other files act as binary or boolean switches. For instance, if you type `cat /proc/sys/net/ipv4/ip_forward`, you will see either a `0` or a `1`. A `0` indicates the kernel is not forwarding network packets. By using the `echo` command to change the value of the `ip_forward` file to `1`, you can immediately turn packet forwarding on.

Tip

Another command used to alter settings in the `/proc/sys/` subdirectory is `/sbin/sysctl`. For more information on this command, see the Section called *Using `sysctl`*

For a listing of some of the kernel configuration files available in the `/proc/sys/`, see the Section called *`/proc/sys/`*.

The Virtual File System

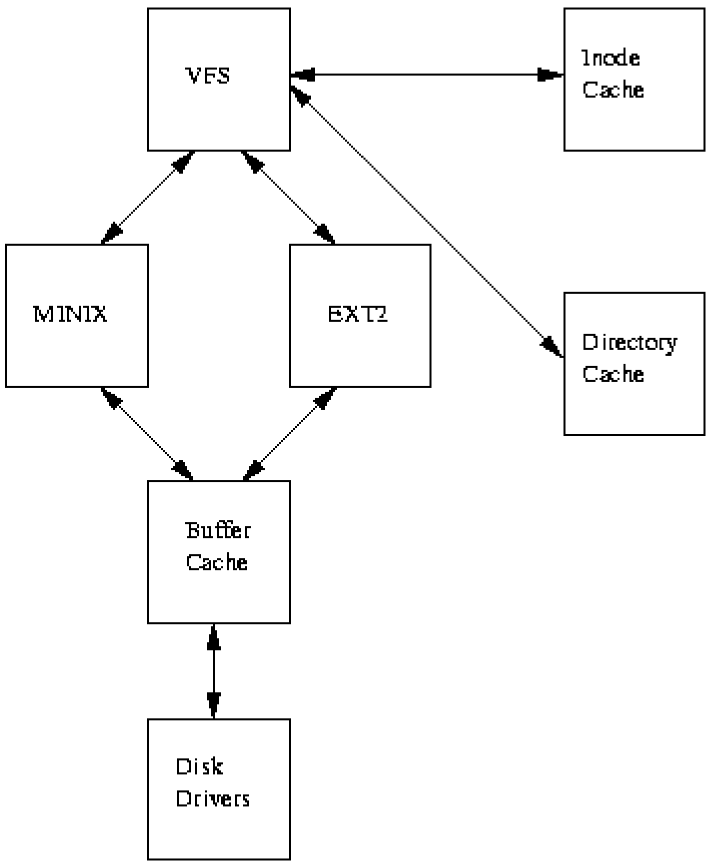

Figure: A Logical Diagram of the Virtual File System

Figure shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinquish them from EXT2 inodes and superblocks.

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the VFAT file system is implemented as a kernel module, then it is only loaded when a VFAT file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the ``/'' directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

As the system's processes access directories and files, system routines are called that traverse the VFS inodes in the system.

For example, typing ls for a directory or cat for a file cause the the Virtual File System to search through the VFS inodes that represent the file system. As every file and directory on the system is represented by a VFS inode, then a number of inodes will be being repeatedly accessed. These inodes are kept in the inode cache which makes access to them quicker. If an inode is not in the inode cache, then a file system specific routine must be called in order to read the appropriate inode. The action of reading the inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache. The less used VFS inodes get removed from the cache.

All of the Linux file systems use a common buffer cache to cache data buffers from the underlying devices to help speed up access by all of the file systems to the physical devices holding the file systems.

This buffer cache is independent of the file systems and is integrated into the mechanisms that the Linux kernel uses to allocate and read and write data buffers. It has the distinct advantage of making the Linux file systems independent from the underlying media and from the device drivers that support them. All block structured devices register themselves with the Linux kernel and present a uniform, block based, usually asynchronous interface. Even relatively complex block devices such as SCSI devices do this. As the real file systems read data from the underlying physical disks, this results in requests to the block device drivers to read physical blocks from the device that they control. Integrated into this block device interface is the buffer cache. As blocks are read by the file systems they are saved in the global buffer cache shared by all of the file systems and the Linux kernel. Buffers within it are identified by their block number and a unique identifier for the device that read it. So, if the same data is needed often, it will be retrieved from the buffer cache rather than read from the disk, which would take somewhat longer. Some devices support read ahead where data blocks are speculatively read just in case they are needed.

The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be quickly found.

As an experiment, try listing a directory that you have not listed recently. The first time you list it, you may notice a slight pause but the second time you list its contents the result is immediate. The directory cache does not store the inodes for the directories itself; these should be in the inode cache, the directory cache simply stores the mapping between the full directory names and their inode numbers.

The VFS Superblock

Every mounted file system is represented by a VFS superblock; amongst other information, the VFS superblock contains the:

Device
: This is the device identifier for the block device that this file system is contained in. For example, `/dev/hda1`, the first IDE hard disk in the system has a device identifier of *0x301*,

Inode pointers
: The `mounted` inode pointer points at the first inode in this file system. The `covered` inode pointer points at the inode representing the directory that this file system is mounted on. The root file system's VFS superblock does not have a `covered` pointer,

Blocksize
: The block size in bytes of this file system, for example 1024 bytes,

Superblock operations
: A pointer to a set of superblock routines for this file system. Amongst other things, these routines are used by the VFS to read and write inodes and superblocks.

File System type
: A pointer to the mounted file system's `file_system_type` data structure,

File System specific
: A pointer to information needed by this file system,

The VFS Inode

Like the EXT2 file system, every file, directory and so on in the VFS is represented by one and only one VFS inode.

The information in each VFS inode is built from information in the underlying file system by file system specific routines. VFS inodes exist only in the kernel's memory and are kept in the VFS inode cache as long as they are useful to the system. Amongst other information, VFS inodes contain the following fields:

device
: This is the device identifer of the device holding the file or whatever that this VFS inode represents,

inode number
: This is the number of the inode and is unique within this file system. The combination of `device` and `inode number` is unique within the Virtual File System,

mode
: Like EXT2 this field describes what this VFS inode represents as well as access rights to it,

user ids
: The owner identifiers,

times
: The creation, modification and write times,

block size
: The size of a block for this file in bytes, for example 1024 bytes,

inode operations

A pointer to a block of routine addresses. These routines are specific to the file system and they perform operations for this inode, for example, truncate the file that is represented by this inode.

count

The number of system components currently using this VFS inode. A count of zero means that the inode is free to be discarded or reused,

lock

This field is used to lock the VFS inode, for example, when it is being read from the file system,

dirty

Indicates whether this VFS inode has been written to, if so the underlying file system will need modifying,

file system specific information

Registering the File Systems

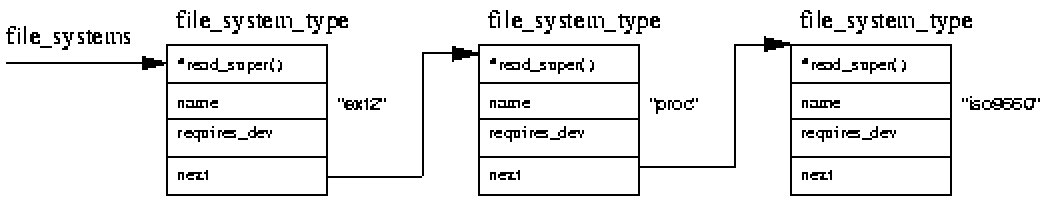

Figure 2: Registered File Systems

When you build the Linux kernel you are asked if you want each of the supported file systems. When the kernel is built, the file system startup code contains calls to the initialisation routines of all of the built in file systems.

Linux file systems may also be built as modules and, in this case, they may be demand loaded as they are needed or loaded by hand using insmod. Whenever a file system module is loaded it registers itself with the kernel and unregisters itself when it is unloaded. Each file system's initialisation routine registers itself with the Virtual File System and is represented by a `file_system_type` data structure which contains the name of the file system and a pointer to its VFS superblock read routine. Figure 2 shows that the `file_system_type` data structures are put into a list pointed at by the `file_systems` pointer. Each `file_system_type` data structure contains the following information:

Superblock read routine

This routine is called by the VFS when an instance of the file system is mounted,

File System name

The name of this file system, for example `ext2`,

Device needed

Does this file system need a device to support? Not all file system need a device to hold them. The `/proc` file system, for example, does not require a block device,

You can see which file systems are registered by looking in at `/proc/filesystems`. For example:

```
      ext2
nodev proc
      iso9660
```

Mounting a File System

When the superuser attempts to mount a file system, the Linux kernel must first validate the arguments passed in the system call. Although mount does some basic checking, it does not know which file systems this kernel has been built to support or that the proposed mount point actually exists. Consider the following mount command:

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

This mount command will pass the kernel three pieces of information; the name of the file system, the physical block device that contains the file system and, thirdly, where in the existing file system topology the new file system is to be mounted.

The first thing that the Virtual File System must do is to find the file system.

To do this it searches through the list of known file systems by looking at each `file_system_type` data structure in the list pointed at by `file_systems`.

If it finds a matching name it now knows that this file system type is supported by this kernel and it has the address of the file system specific routine for reading this file system's superblock. If it cannot find a matching file system name then all is not lost if the kernel is built to demand load kernel modules. In this case the kernel will request that the kernel daemon loads the appropriate file system module before continuing as before.

Next if the physical device passed by mount is not already mounted, it must find the VFS inode of the directory that is to be the new file system's mount point. This VFS inode may be in the inode cache or it might have to be read from the block device supporting the file system of the mount point. Once the inode has been found it is checked to see that it is a directory and that there is not already some other file system mounted there. The same directory cannot be used as a mount point for more than one file system.

At this point the VFS mount code must allocate a VFS superblock and pass it the mount information to the superblock read routine for this file system. All of the system's VFS superblocks are kept in the `super_blocks` vector of `super_block` data structures and one must be allocated for this mount. The superblock read routine must fill out the VFS superblock fields based on information that it reads from the physical device. For the EXT2 file system this mapping or translation of information is quite easy, it simply reads the EXT2 superblock and fills out the VFS superblock from there. For other file systems, such as the MS DOS file system, it is not quite such an easy task. Whatever the file system, filling out the VFS superblock means that the file system must read whatever describes it from the block device that supports it. If the block device cannot be read from or if it does not contain this type of file system then the mount command will fail.

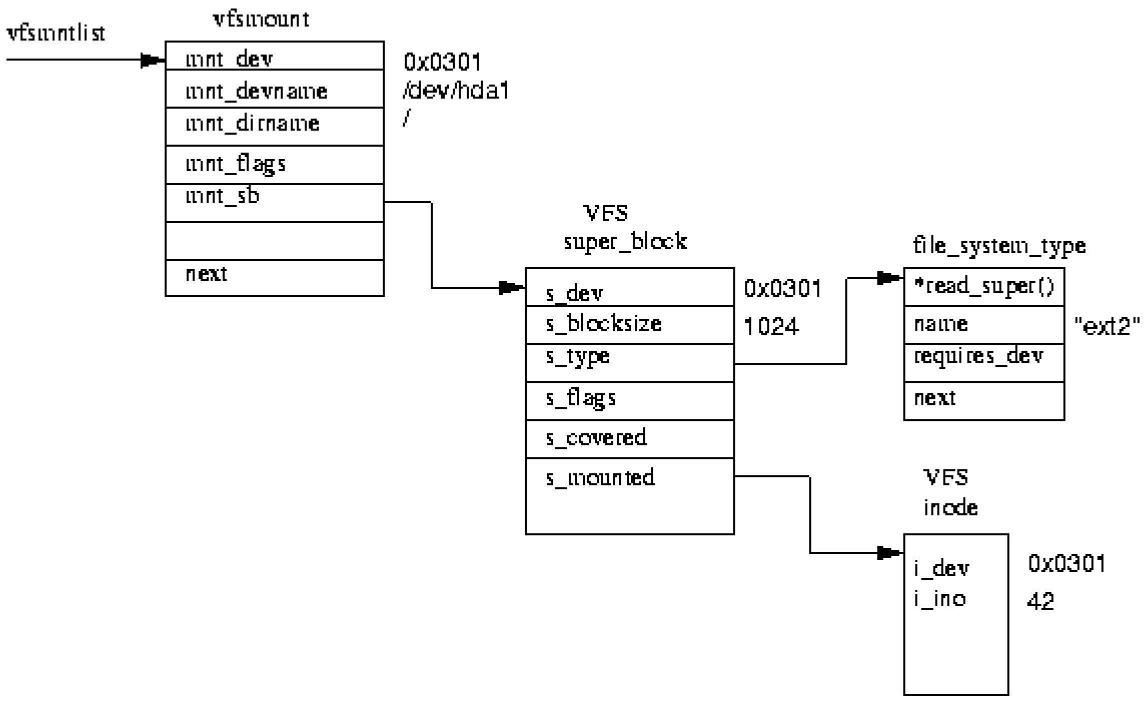

Figure 3: A Mounted File System

Each mounted file system is described by a `vfsmount` data structure; see figure 3. These are queued on a list pointed at by `vfsmntlist`.

Another pointer, `vfsmnttail` points at the last entry in the list and the `mru_vfsmnt` pointer points at the most recently used file system. Each `vfsmount` structure contains the device number of the block device holding the file system, the directory where this file system is mounted and a pointer to the VFS superblock allocated when this file system was mounted. In turn the VFS superblock points at the `file_system_type` data structure for this sort of file system and to the root inode for this file system. This inode is kept resident in the VFS inode cache all of the time that this file system is loaded.

Finding a File in the Virtual File System

To find the VFS inode of a file in the Virtual File System, VFS must resolve the name a directory at a time, looking up the VFS inode representing each of the intermediate directories in the name. Each directory lookup involves calling the file system specific lookup whose address is held in the VFS inode representing the parent directory. This works because we always have the VFS inode of the root of each file system available and pointed at by the VFS superblock for that system. Each time an inode is looked up by the real file system it checks the directory cache for the directory. If there is no entry in the directory cache, the real file system gets the VFS inode either from the underlying file system or from the inode cache.

Unmounting a File System

The workshop manual for my MG usually describes assembly as the reverse of disassembly and the reverse is more or less true for unmounting a file system.

A file system cannot be unmounted if something in the system is using one of its files. So, for example, you cannot umount `/mnt/cdrom` if a process is using that directory or any of its children. If anything is using the file system to be unmounted there may be VFS inodes from it in the VFS inode cache, and the code checks for this by looking through the list of inodes looking for inodes owned by the device that this file system occupies. If the VFS superblock for the mounted file system is dirty, that is it has been modified, then it must be written back to the file system on disk. Once it has been written to disk, the memory occupied by the VFS superblock is returned to the kernel's free pool of memory. Finally the `vfsmount` data structure for this mount is unlinked from `vfsmntlist` and freed.

The VFS Inode Cache

As the mounted file systems are navigated, their VFS inodes are being continually read and, in some cases, written. The Virtual File System maintains an inode cache to speed up accesses to all of the mounted file systems. Every time a VFS inode is read from the inode cache the system saves an access to a physical device.

The VFS inode cache is implmented as a hash table whose entries are pointers to lists of VFS inodes that have the same hash value. The hash value of an inode is calculated from its inode number and from the device identifier for the underlying physical device containing the file system. Whenever the Virtual File System needs to access an inode, it first looks in the VFS inode cache. To find an inode in the cache, the system first calculates its hash value and then uses it as an index into the inode hash table. This gives it a pointer to a list of inodes with the same hash value. It then reads each inode in turn until it finds one with both the same inode number and the same device identifier as the one that it is searching for.

If it can find the inode in the cache, its count is incremented to show that it has another user and the file system access continues. Otherwise a free VFS inode must be found so that the file system can read the inode from memory. VFS has a number of choices about how to get a free inode. If the system may allocate more VFS inodes then this is what it does; it allocates kernel pages and breaks them up into new, free, inodes and puts them into the inode list. All of the system's VFS inodes are in a list pointed at by `first_inode` as well as in the inode hash table. If the system already has all of the inodes that it is allowed to have, it must find an inode that is a good candidate to be reused. Good candidates are inodes with a usage count of zero; this indicates that the system is not currently using them. Really important VFS inodes, for example the root inodes of file systems always have a usage count greater than zero and so are never candidates for reuse. Once a candidate for reuse has been located it is cleaned up. The VFS inode might be dirty and in this case it needs to be written back to the file system or it might be locked and in this case the system must wait for it to be unlocked before continuing. The candidate VFS inode must be cleaned up before it can be reused.

However the new VFS inode is found, a file system specific routine must be called to fill it out from information read from the underlying real file system. Whilst it is being filled out, the new VFS inode has a usage count of one and is locked so that nothing else accesses it until it contains valid information.

To get the VFS inode that is actually needed, the file system may need to access several other inodes. This happens when you read a directory; only the inode for the final directory is needed but the inodes for the intermediate directories must also be read. As the VFS inode cache is used and filled up, the less used inodes will be discarded and the more used inodes will remain in the cache.

The Directory Cache

To speed up accesses to commonly used directories, the VFS maintains a cache of directory entries.

As directories are looked up by the real file systems their details are added into the directory cache. The next time the same directory is looked up, for example to list it or open a file within it, then it will be found in the directory cache. Only short directory entries (up to 15 characters long) are cached but this is reasonable as the shorter directory names are the most commonly used ones. For example, `/usr/X11R6/bin` is very commonly accessed when the X server is running.

The directory cache consists of a hash table, each entry of which points at a list of directory cache entries that have the same hash value. The hash function uses the device number of the device holding the file system and the directory's name to calculate the offset, or index, into the hash table. It allows cached directory entries to be quickly found. It is no use having a cache when lookups within the cache take too long to find entries, or even not to find them.

In an effort to keep the caches valid and up to date the VFS keeps lists of Least Recently Used (LRU) directory cache entries. When a directory entry is first put into the cache, which is when it is first looked up, it is added onto the end of the first level LRU list. In a full cache this will displace an existing entry from the front of the LRU list. As the directory entry is accessed again it is promoted to the back of the second LRU cache list. Again, this may displace a cached level two directory entry at the front of the level two LRU cache list. This displacing of entries at the front of the level one and level two LRU lists is fine. The only reason that entries are at the front of the lists is that they have not been recently accessed. If they had, they would be nearer the back of the lists. The entries in the second level LRU cache list are safer than entries in the level one LRU cache list. This is the intention as these entries have not only been looked up but also they have been repeatedly referenced.

List of Virtual File System Operations

The following entry points are specified by the virtual file system interface for performing operations on **vfs** structures:

vfs_cntl	Issues control operations for a file system.
vfs_init	Initializes a virtual file system.
vfs_mount	Mounts a virtual file system.
vfs_root	Finds the root v-node of a virtual file system.
vfs_statfs	Obtains virtual file system statistics.
vfs_sync	Forces file system updates to permanent storage.
vfs_umount	Unmounts a virtual file system.
vfs_vget	Gets the v-node corresponding to a file identifier.

The following entry points are specified by the Virtual File System interface for performing operations on v-node structures:

vn_access	Tests a user's permission to access a file.
vn_close	Releases the resources associated with a v-node.
vn_create	Creates and opens a new file.
vn_fclear	Releases portions of a file (by zeroing bytes).
vn_fid	Builds a file identifier for a v-node.
vn_fsync	Flushes in-memory information and data to permanent storage.
vn_ftrunc	Decreases the size of a file.
vn_getacl	Gets information about access control, by retrieving the access control list.
vn_getattr	Gets the attributes of a file.
vn_hold	Assures that a v-node is not destroyed, by incrementing the v-node's use count.
vn_ioctl	Performs miscellaneous operations on devices.
vn_link	Creates a new directory entry for a file.
vn_lockctl	Sets, removes, and queries file locks.
vn_lookup	Finds an object by name in a directory.
vn_map	Associates a file with a memory segment.
vn_mkdir	Creates a directory.
vn_mknod	Creates a file of arbitrary type.
vn_open	Gets read and/or write access to a file.
vn_rdwr	Reads or writes a file.
vn_readdir	Reads directory entries in standard format.
vn_readlink	Reads the contents of a symbolic link.
vn_rele	Releases a reference to a virtual node (v-node).
vn_remove	Unlinks a file or directory.
vn_rename	Renames a file or directory.
vn_revoke	Revokes access to an object.
vn_rmdir	Removes a directory.
vn_select	Polls a v-node for pending I/O.
vn_setacl	Sets information about access control for a file.
vn_setattr	Sets attributes of a file.
vn_strategy	Reads or writes blocks of a file.
vn_symlink	Creates a symbolic link.
vn_unmap	Destroys a file or memory association.

References

http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/ref-guide/ch-proc.html
http://csit1cwe.fsu.edu/doc_link/en_US/a_doc_lib/libs/ktechrf1/List.htm
http://www.tldp.org/LDP/tlk/fs/filesystem.html