

The VFS/VNODE interface in the FreeBSD kernel

Attilio Rao <attilio@FreeBSD.org>
The FreeBSD project

Summary

- History (just a little bit)
- The VFS/VNODE interface
- The VFS/VNODE interface via the OOP
- The VFS specific interface
- The VNODE specific interface
- The namei() interface
- The VFS objects lifecycle
- The VNODE objects lifecycle
- FreeBSD TODO list

Terminology

- **Partition:** Linear array of disk blocks, contiguous
- **Superblock:** disk block, within the partition, that identify *metadata* for the filesystem itself
- **Inode:** structure handling *metadatas* for any specific file
- **Directory:** special file containing lists of files (or other directories)

History

- Unix SystemV and BSD just supported the S5FS (1978)
- Unix BSD 4.2 introduced a new filesystem: FFS ('83-'84)
- There were other interesting filesystems out of there: NFS, FAT, etc.
- Problem: Just one filesystem per time could be supported by the kernel
- Solutions: several by several vendors.
- The vfs/vnode interface, by Sun, overtakes the others ('86-'87)

The VFS/VNODE interface

- Problem: the kernel subsystems needing to access the filesystem layout need intimate knowledge of inodes and filesystem structures
- Solution: outlining a common interface for all the (supported) filesystems that the kernel subsystems can use without having knowledge of the underlying objects

The VFS/VNODE interface

- There are several operations common to all the type of filesystems (**mount, unmount, etc**) and inodes (**open, close, read, write, etc**)
- The consumers usually just needs to access a small subset of filesystem facilities and do repetitive jobs
- **Idea:** split the filesystem/inode code in a filesystem specific part and filesystem indipendent part. Offer a **standard interface** that the latter can use and the former can implement

The VFS/VNODE interface

- The **interface** is implemented via OOP paradigms
- The **VFS** is a physical structure, used as base class for filesystems.
- The **VNODE** is a physical structure, used as base class for inodes.
- They both include virtual functions and useful, generic, supporting members
- New filesystem support adds derived classes for the **VFS** and **VNODE**, implementing filesystem specific support

The VFS interface

```
struct mount {  
    TAILQ_ENTRY(mount) mnt_list;    /* mount list */  
    struct vfsops      *mnt_op;      /* operations on fs */  
    struct vnode       *mnt_vnodecovered; /* vnode we mounted on */  
    int                mnt_kern_flag; /* kernel only flags */  
    u_int              mnt_flag;     /* flags shared with user */  
    void               *mnt_data;    /* private data */  
  
    ...  
};
```

```
struct vfsops {  
    vfs_mount_t      *vfs_mount;  
    vfs_unmount_t    *vfs_unmount;  
    vfs_root_t       *vfs_root;  
    vfs_sync_t       *vfs_sync;  
    vfs_vget_t       *vfs_vget;  
  
    ...  
};
```

The VFS interface

The polymorphism is implemented via the **mnt_data** and the **mnt_op** members:

- **mnt_data** holds a link to the derived vfs object (instance of the derived class)
- **mnt_op** represents the virtual functions that derived class (filesystem dependent specification) must implement
- Some of the **mnt_op** virtual functions represents pure virtual functions while other have a standard implementation already

*When a new filesystem is defined, the coder forges a derived filesystem class and a set of **struct vfsops** that partially override the default one. When the filesystem is loaded in the kernel, thanks to the **VFS_SET()** help, the new filesystem is added to a list called **Filesystem switch** (for a complete reference: `vfscnf tailqueue` in `sys/kern/vfs_init.c`).*

The VFS interface

(A typical mounting operation)

The **mount()** syscall accesses the **vfs_domount()** kernel function and performs the following activities:

- Looks up the **vfscnf** specific object from the filesystem switch
- Looks up the **vnode** for the mounting directory and does preliminary checks
- Allocate a new instance of the **struct mount**, assigning the right **struct vfsops** retrieved by the previous lookup, via the **vfs_mount_alloc()** function
- Calls the filesystem dependent virtual function **VFS_MOUNT()**
- If **VFS_MOUNT()** returns successfully, adds the new mount instance to the mountlist list

VFS_MOUNT() has some key responsibilities like the **mnt_data** correct setup and prepare the ground for the root **vnode** in a way that **VFS_ROOT()** can return consistent datas.

The VNODE interface

```
struct vnode {
    enum vtype      v_type;           /* vnode type */
    struct vop_vector *v_op;         /* vnode operations vector */
    void            *v_data;         /* private data for fs */
    struct mount    *v_mount;        /* ptr to vfs we are in */
    u_long          v_iflags;        /* vnode flags (see below) */
    u_long          v_vflags;        /* vnode flags */
    ...
};
```

(That is really generated in a special way by parsing a script in sys/kern/vnode_if.src)

```
struct vop_vector {
    vop_open_t      *vop_open;
    vop_close_t     *vop_close;
    vop_read_t      *vop_read;
    vop_write_t     *vop_write;
    vop_ioctl_t     *vop_ioctl;
    vop_lookup_t    *vop_lookup;
    ...
};
```

The VNODE interface

The polymorphism is implemented via the **v_data** and the **v_op** members:

- **v_data** holds a link to the derived inode object (instance of the derived class)
- **v_op** represents the virtual functions that derived class (filesystem dependent specification) must implement
- Some of the **v_op** virtual functions represents pure virtual functions while other have a standard implementation already

*When a new filesystem is defined, the coder forsee a derived inode class and a set of **struct vop_vector** that partially override the default one. He also forsee an implementation of **VFS_VGET()** that allocates new vnodes linked with the correct set of overridden virtual functions and usually a linked inode.*

The VNODE interface

(A typical open operation)

The **open()** syscall accesses the **kern_openat()** kernel function and performs the following activities:

- Allocate a filedescriptor (integer) and a “struct file” object using the **falloc()** interface
- Looks up the vnode for the provided path
- Calls the filesystem dependent virtual function **VOP_OPEN()**
- If **VOP_OPEN()** returns successfully, store the vnode un the file object
- Stores the file object in the per-process files table, indexed by the preallocated filedescriptor
- Returns the filedescriptor

The look up operation is usually responsible for generating new vnodes and link to the proper inodes on the **v_data** handle.

The `namei()` interface

The **`namei()`** interface is used for translating a path into a vnode. It generally relies on the low level **`lookup()`** kernel function to do the job.

The main `lookup()` activity is to identify the path tokens and do appropriate actions (examples: symbolic links, “..”, further mountpoints).

It relies on the filesystem dependent virtual function **`VOP_LOOKUP()`** in order to actual find the vnode, create a new one, returning a “not found” error or just processing the next token.

The moder Unixes implements a cache for speeding up the lookup operation called **`namecache`** (FreeBSD implementation in: `sys/kern/vfs_cache.c`).

The VFS objects lifecycle

- mount objects are allocated from **vfs_mount_alloc()** and destroyed by **vfs_mount_destroy()** using an UMA zone
- Mount objects are fully type-stable (UMA zone initialized with **UMA_ZONE_NOFREE**) because some activities rely on type-stability (**VFS_NEEDSGIANT()**)
- Type-stability is generally fine for them – there are few mount points generally

Synchronization support for mount objects comprehends:

- The mountlist lock
- A mutex (**MNT_ILOCK()/MNT_IUNLOCK()** interfaces)
- A “strong” refcount (**vfs_busy()/vfs_unbusy()** interfaces)
- A “light” refcount (**vfs_ref()/vfs_rel()** interfaces)
- Write/Suspension barriers

The VFS objects lifecycle

- The mountlist lock protects all the list movements (insertion, deletions, etc)
- The mount mutex interlock, usually protects flags and list of active vnodes
- The “strong” refcount protects against on-the-fly unmounts. It inhibits the possibility to have unmounts until it is released. In order to avoid busy-waiting loops, it can sleep on request, when failing
- The “light” refcount protects against complete structure destruction.
- The write/suspend barriers are used in order to synchronize snapshot creations (that may be just useful for FFS...)

The VFS objects lifecycle

A typical locking pattern for a mount object is:

- Acquire the **mountlist lock** and lookup the mount object
- Acquire a **vfs_ref()** on it
- Release the **mountlist lock** in order to avoid LOR with busying
- Try to **vfs_busy()** the mount object
- Do **vfs_rel()**
- If the busying fails, just quit the operation
- If the busying succeeds perform your own stuff
- When finished, do **vfs_unbusy()** and quit

Tip: Always check that there are initial conditions assuring the safety of the operation

The VNODE objects lifecycle

- mount objects are allocated from **getnewvnode()** and destroyed by **vdestroy()** using an UMA zone
- Generally, they are not destroyed but kept in a free-list for a fast re-allocation, when needed. This allocation happens via the **vbusy()** interface
- Vnodes are not type-stable
- The **vnlru** kernel daemon keeps under control the situation of the free-list (avoiding it to grow too much) and it can request **vnodes recycling**
- *The vnlru does a good job because it keeps a complete view of all the vnodes. Having recycling mechanism off the vnode interface would make this a lot less effective*
- Some type of filesystems, also, may force vnodes recycling at their will

The VNODE objects lifecycle

Synchronization support for the vnode objects comprehends:

- A mutex interlock (**VI_LOCK()/VI_UNLOCK()** interface)
 - A lockmgr lock (**vn_lock()/VOP_UNLOCK()** interface)
 - A “usecount” refcount (**vref()/vrele()** interface)
 - An “holdcount” refcount (**vhold()/vdrop()** interface)
-
- The mutex interlock is used mostly for handling vnode flags and avoiding opening of races
 - The lockmgr lock is used for all the other operations, in particular ones accessing to the I/O system (as it can sleep safely)
 - The vnode usecount is the main vnode refcount. It must be acquired for all the valid operations on the vnode. When the usecount drops to 0 the **VOP_INACTIVE()** function is called
 - The vnode holdcount does veto the recycling of the vnode. If it is different from 0 the vnode cannot be recycled.

The VNODE objects lifecycle

A typical locking pattern for a vnode object is:

- Get a vnode from any possible list by acquiring its lock
- Acquire the vnode interlock
- Release the initial list lock in order to avoid LOR
- Use the **vget()** interface that takes care of returning a correctly refcounted vnode
- Check for VI_DOOMED (vnode is being destroyed) if LK_RETRY was passed to the vget()
- Operate and in the end **vrele()/vput()** the vnode

Tip: Do not use vnodes that are not properly refcounted or without sanity checks

FreeBSD TODO list

Main items that are being worked right now:

- On-going researches on improving the namecache
- Implementation of byte-range locking at the vnode level
- VREG I/O to be performed via the vnode pager and not via the buffer cache (which remains only for metadata)
- Multithreaded and faster syncer
- Unmapped I/O for device drivers

Questions?