Virtual filesystem concepts

The VFS

Implements the filesystem namespace
- Hides filesystem types
- Hides filesystem boundaries

Handles I/O operations

Manages the page cache

...

It's complicated!

VFS concepts

Superblock
- Structure describing a mounted filesystem

inode
- Structure describing a file
- On-disk and in-core formats

dentry
- “Directory entry”
- The mapping from a name to an inode
- in-core only

inodes and dentries

Consider a multiply-linked file:

```
cd /usr/bin; ls -li c++ g++
252752 -rwxr-xr-x 4 root root 240K Sep 24 14:08 c++*
252752 -rwxr-xr-x 4 root root 240K Sep 24 14:08 g++*
```

This file has (at least) two dentries:

- One for c++
- One for g++

...but only one inode

Filesystem registration

```
struct file_system_type {
        const char *name;
        int fs_flags;
        int (*get_sb) (struct file_system_type *fst,
                       int flags,
                       const char *dev_name,
                       void *raw_data,
                       struct vfsmount *mnt);
        struct dentry *(*mount) (struct file_system_type *fst,
                                 int flags,
                                 const char *dev_name,
                                 void *data);
        void (*kill_sb)(struct super_block *sb);
        struct module *owner;
        /* ... */
};
```

To register a filesystem

Fill in the file_system_type structure

Then make a call to:

```
int register_filesystem(struct file_system_type *);
```

struct file_system_type

```
struct file_system_type {
        const char *name;
        int fs_flags;
        int (*get_sb) (struct file_system_type *fst,
                       int flags,
                       const char *dev_name,
                       void *raw_data,
                       struct vfsmount *mnt);
        struct dentry *(*mount) (struct file_system_type *fst,
                                 int flags,
                                 const char *dev_name,
                                 void *data);
        void (*kill_sb)(struct super_block *sb);
        struct module *owner;
        /* ... */
};
```

Example: ext4

An ext4 filesystem is mounted with:

```
static struct dentry *ext4_mount(
            struct file_system_type *fs_type,
            int flags,
            const char *dev_name,
            void *data)
{
    return mount_bdev(fs_type, flags, dev_name,
                      data, ext4_fill_super);
}
```

mount_bdev()

Opens the block device
Creates a superblock
Calls the fill_super() function to fill in the superblock
Returns the dentry for the root

struct super_block

```
struct super_block {
	struct list_head	s_list;
	dev_t			s_dev;
	unsigned char		s_dirt;
	unsigned char		s_blocksize_bits;
	unsigned long		s_blocksize;
	loff_t			s_maxbytes;
	struct file_system_type	*s_type;
	const struct super_operations *s_op;
	unsigned long		s_flags;
	struct dentry		*s_root;
	struct list_head	s_inodes;
	struct list_head	s_dentry_lru;
	int			s_nr_dentry_unused;
	/* ... */
};
```

struct super_block

```
struct super_block {
     struct list_head  s_list;
     dev_t             s_dev;
     unsigned char     s_dirt;
     unsigned char     s_blocksize_bits;
     unsigned long     s_blocksize;
     loff_t            s_maxbytes;
     struct file_system_type  *s_type;
     const struct super_operations *s_op;
     unsigned long     s_flags;
     struct dentry     *s_root;
     struct list_head  s_inodes;
     struct list_head  s_dentry_lru;
     int               s_nr_dentry_unused;
     /* ... */
};
```

struct super_operations

```
struct super_operations {
      struct inode *(*alloc_inode)(struct super_block *sb);
      void (*destroy_inode)(struct inode *);
      void (*dirty_inode) (struct inode *);
      int (*write_inode) (struct inode *,
                  struct writeback_control *wbc);
      int (*drop_inode) (struct inode *);
      void (*evict_inode) (struct inode *);
      void (*put_super) (struct super_block *);
      void (*write_super) (struct super_block *);
      int (*sync_fs)(struct super_block *sb, int wait);
      int (*freeze_fs) (struct super_block *);
      int (*unfreeze_fs) (struct super_block *);
      int (*statfs) (struct dentry *, struct kstatfs *);
      int (*remount_fs) (struct super_block *, int *, char *);
      void (*umount_begin) (struct super_block *);
      /* ... */
};
```

struct inode

```
struct inode {
        struct hlist_node i_hash;
        struct list_head  i_wb_list;
        struct list_head  i_lru;
        struct list_head  i_sb_list;
        struct list_head  i_dentry;
        unsigned long     i_ino;
        atomic_t          i_count;
        unsigned int      i_nlink;
        uid_t             i_uid;
        gid_t             i_gid;
        dev_t             i_rdev;
        unsigned int      i_blkbits;
        u64               i_version;
        loff_t            i_size;
```

struct inode (continued)

```
struct timespec     i_atime;
struct timespec     i_mtime;
struct timespec     i_ctime;
blkcnt_t     i_blocks;
unsigned short          i_bytes;
umode_t      i_mode;
const struct inode_operations *i_op;
const struct file_operations  *i_fop;
struct super_block  *i_sb;
struct address_space   *i_mapping;
struct address_space   i_data;
```

struct inode (continued)

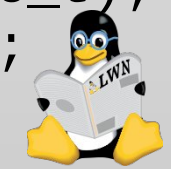
```
        union {
                struct pipe_inode_info *i_pipe;
                struct block_device *i_bdev;
                struct cdev    *i_cdev;
        };
        __u32           i_generation;
        unsigned long     i_state;
        unsigned int    i_flags;
        void            *i_private;
        /* ... */
};
```

struct inode_operations

```
struct inode_operations {
      int (*create) (struct inode *, struct dentry *,
                     int, struct nameidata *);
      struct dentry *(*lookup) (struct inode *,
                      struct dentry *, struct nameidata *);
      int (*link) (struct dentry *, struct inode *,
                    struct dentry *);
      int (*unlink) (struct inode *, struct dentry *);
      int (*symlink) (struct inode *, struct dentry *,
                      const char *);
      int (*mkdir) (struct inode *, struct dentry *, int);
      int (*rmdir) (struct inode *, struct dentry *);
      int (*mknod) (struct inode *, struct dentry *, int,
                     dev_t);
      int (*rename) (struct inode *, struct dentry *,
                     struct inode *, struct dentry *);
```

struct inode_operations

```
struct inode_operations {
        int (*readlink) (struct dentry *, char __user *, int);
        void *(*follow_link) (struct dentry *,
                              struct nameidata *);
        void (*put_link) (struct dentry *,
                          struct nameidata *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*check_acl)(struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt,
                        struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,
                         const void *, size_t, int);
        ssize_t (*getxattr) (struct dentry *, const char *,
                             void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
```



struct inode_operations

```
struct inode_operations {
      void (*truncate_range)(struct inode *, loff_t, loff_t);
      long (*fallocate)(struct inode *inode, int mode,
                        loff_t offset, loff_t len);
      int (*fiemap)(struct inode *,
                    struct fiemap_extent_info *,
                    u64 start, u64 len);
};
```

struct dentry

How the kernel caches name lookups

- No on-disk version

struct dentry

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;
        int d_mounted;
        struct inode *d_inode;
        struct hlist_node d_hash;
        struct dentry *d_parent;
        struct qstr d_name;
        struct list_head d_lru;
        struct list_head d_subdirs;
        struct list_head d_alias;
        const struct dentry_operations *d_op;
        struct super_block *d_sb;
        void *d_fsdata;
        /* ... */
};
```

struct dentry_operations

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *,
                            struct nameidata *);
        int (*d_hash) (struct dentry *,
                       struct qstr *);
        int (*d_compare) (struct dentry *,
                          struct qstr *,
                          struct qstr *);
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *,
                       struct inode *);
        char *(*d_dname)(struct dentry *,
                         char *, int);
};
```

Finding a file

Start at the root (or cwd) dentry

    Both found in the task_struct

while (not done)

    d_lookup(dentry, next component)

    if (not found)

      dentry->inode->i_op->lookup(next component)

      store in dentry cache

return final dentry

Negative dentries

Dentries can indicate that the name does not exist

- d_inode set to NULL

Lookups of nonexistent files are common

- Storing negative results is an important optimization.

struct file

Represents an open file

- Internal form of a file descriptor

If two processes open the same file

- The result is two file structures

struct file

```
struct file {
        struct path             f_path;
#define f_dentry        f_path.dentry
#define f_vfsmnt        f_path.mnt
        const struct file_operations    *f_op;
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        u64                     f_version;
        void                    *private_data;
        struct list_head        f_ep_links;
        struct address_space    *f_mapping;
};
```

struct file_operations

```
struct file_operations {
        struct module *owner;
        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 *);
        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);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *,
                              struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *,
                                unsigned int, unsigned long);
```

struct file_operations

```
        long (*compat_ioctl) (struct file *, unsigned int,
                              unsigned long);
        int (*mmap) (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 *, 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);
        int (*check_flags)(int);
        /* ... */
};
```

file_operations notes

Filesystems don't usually implement file_operations directly

- Generic VFS versions are used

Who does implement them?

- Device drivers

Implementing open()

- Lookup file name
  - Checking permissions on the way
- Create a file structure
  - Point to dentry
- Call f_op->open()
  - Might change f_op
- Allocate a file descriptor number
- Store in file descriptor table
  - Accessible from the task_struct

Most other VFS system calls

Locate file structure in descriptor table
Check permissions
Call associated file_operations function

read() and write()

These calls do not go straight to filesystems

Why?

- Byte stream -> blocks mapping
- I/O performance

Thus:

- The Linux page cache

struct address_space

How the kernel tracks a file's blocks
+ how it manipulates those blocks

Pointed to by:
- inode->i_mapping
- file->f_mapping
- struct page->mapping

struct address_space

```
struct address_space {
        struct inode            *host;
        struct radix_tree_root  page_tree;
        unsigned int            i_mmap_writable;
        struct prio_tree_root   i_mmap;
        struct list_head        i_mmap_nonlinear;
        spinlock_t              i_mmap_lock;
        unsigned long           nrpages;
        pgoff_t                 writeback_index;
        struct address_space_operations *a_ops;
        unsigned long           flags;
        struct backing_dev_info *backing_dev_info;
        /* ... */
}
```

struct address_space_operations

```
struct address_space_operations {
        int (*writepage)(struct page *page,
                         struct writeback_control *wbc);
        int (*readpage)(struct file *, struct page *);
        void (*sync_page)(struct page *);
        int (*writepages)(struct address_space *,
                          struct writeback_control *);
        int (*set_page_dirty)(struct page *page);
        int (*readpages)(struct file *filp,
                         struct address_space *mapping,
                         struct list_head *pages,
                         unsigned nr_pages);
        int (*releasepage) (struct page *, gfp_t);
        /* ... ... */
};
```

Is a page in the cache?

Call:

```
struct page *find_get_page(struct address_space *mapping,
                           pgoff_t offset);
```

So read() works like:

```
page = find_get_page(...);
if (page == NULL) {
        allocate a page
        fill it with mapping->readpage()
        add_to_page_cache()
}
copy_to_user(...);
```

read() is a bit more complicated

- read() is performance-critical
  - Applications are almost always waiting for it
- The kernel tries to help
  - File access patterns are tracked
  - If sequential access is detected
    - The kernel will read ahead of the application

read() is a bit more complicated

read() is performance-critical
- Applications are almost always waiting for it

The kernel tries to help
- File access patterns are tracked
- If sequential access is detected
  - The kernel will read ahead of the application

(it's complicated)

write()

Writes do not go directly to disk

- Multiple writes to one page
- Combining I/O operations
- File might be deleted first!

Thus:

- Written pages stay in the page cache
- Marked “dirty”

Writeback

Done by the [flush-*] kernel threads

Simplified algorithm

- Pick a file with dirty pages
- Write back a bunch of them

This is the preferred writeback mechanism

- File pages should be contiguous on disk
- Better I/O bandwidth will result

The LRU lists

The kernel maintains two lists
- Pagecache pages in active use
- Pages thought not to be in active use

Active pages can be shifted to the inactive list
- They move back to active if referenced

Whenever memory gets tight
- Clean pages are reclaimed from the inactive list

Direct reclaim

When memory gets really tight
- Allocating process must do some writeback work

This policy serves two goals
- It causes some memory to be freed
- It throttles heavy memory users

Unfortunately
- The resulting I/O pattern is awful

Anonymous pages

Pages with no backing store
    (i.e. program data)

These pages may be backed by swap

/proc/sys/vm/swappiness
    0: anonymous pages will not be reclaimed
    60: default (reclaim mostly pagecache pages)

To make things more complicated

We like to talk about:

- **the** filesystem
- **the** process hierarchy
- **the** system time

But what if there were more than one?

Namespaces

Every task has an nsproxy structure

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns;
        struct net           *net_ns;
};
```

The mount namespace

Controls the visible filesystems

```
struct mnt_namespace {
        atomic_t                count;
        struct vfsmount *       root;
        struct list_head        list;
        wait_queue_head_t poll;
        int event;
};
```

Namespaces

Filename lookups start at the local root
- Anything outside the local tree is invisible

Other namespaces exist
- Process IDs
- System time
- Network environment
- IPC resources

Why?

Process isolation

- “containers”
- Like chroot() but better

How to get a new namespace

- At clone() time

Other VFS concepts

- Direct I/O
- Asynchronous I/O
- Special files
  - Char devices
  - Block devices
- Virtual filesystems
  - /proc, debugfs, sysfs, ...

VFS questions?