

# Virtual File system Switch

A brief overview of the Linux storage stack

## Why add layers?

Originally, operating systems supported just one file system. What changed?

- introduction of removable media (floppy, CD-ROM, usb drives, etc.)
- demand for network file systems (nfs, cifs, etc.)

Design decisions

- Dedicated libraries for each file system type?
  - hard on application programmers
- What about common operations, like path lookup?
  - a shared layer between application and FS allows for code reuse

## Core Data Structures

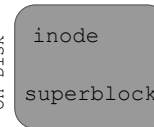
Only Exist  
In Memory



everything a process requires to interact with an open file

created for every component of a pathname - a specialized cache to aid in lookup

Mirrored  
On Disk



all information needed by a file system to handle a file

data pertaining to a mounted file system

## Impact on Designing a FS

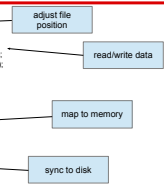
- Designers define a subset of common functions
  - register each file system with the kernel
  - when a file system is mounted, the kernel finds its functions in the table of registered file systems
- VFS calls FS functions at known locations

**Question:** How much flexibility does the VFS give us?

## File Operations

```

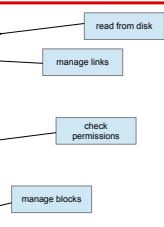
struct file_operations {
    loff_t (*bseek)(struct file *, loff_t, int);
    ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read)(struct iocb *, const struct iovec *, unsigned long, loff_t);
    int (*readahead)(struct file *, void *, size_t);
    ssize_t (*aio_write)(struct iocb *, const struct iovec *, unsigned long, loff_t);
    unsigned int (*poll)(struct file *, struct poll_table_struct *);
    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);
    int (*mmap)(struct file *, struct vm_area_struct *);
    int (*open)(struct inode *, struct file *);
    int (*flush)(struct file *, int, struct file *);
    int (*fsync)(struct inode *, struct file *);
    int (*fdatasync)(struct inode *, struct file *);
    int (*fsync_range)(struct inode *, struct file *, int, int);
    int (*lock)(struct file *, int, struct file *);
    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);
    int (*check_flags)(int);
    int (*bdev)(struct file *, int, struct file *);
    int (*fsync_file_range)(struct file *, int, struct file *, int, int);
    int (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    int (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file *);
};
    
```



## Inode Operations

```

struct inode_operations {
    int (*page)(struct inode *, struct dentry *, int, struct nameidata *);
    struct dentry *(*lookup)(struct inode *, struct dentry *, struct nameidata *);
    int (*path)(struct dentry *, struct inode *, struct nameidata *);
    int (*setattr)(struct inode *, struct dentry *, const char *);
    int (*setxattr)(struct inode *, struct dentry *, const char *, void *, size_t, int);
    int (*getxattr)(struct inode *, struct dentry *, const char *, void *, size_t);
    int (*removexattr)(struct inode *, struct dentry *, const char *);
    void (*truncate)(struct inode *, struct dentry *, struct nameidata *, int);
    void (*lock)(struct inode *, struct dentry *, struct nameidata *, void *);
    int (*permission)(struct inode *, int);
    int (*check_acl)(struct inode *, int);
    int (*set_acl)(struct dentry *, struct_acl *, int);
    int (*get_acl)(struct inode *, struct dentry *, struct nameidata *, struct inode *, struct dentry *, struct nameidata *);
    ssize_t (*readpage)(struct inode *, struct page *, int, void *, size_t);
    ssize_t (*writepage)(struct inode *, struct page *, int, void *, size_t);
    int (*move_pages)(struct inode *, struct dentry *, const char *);
    void (*launder)(struct inode *);
    void (*update_page_cache)(struct inode *, loff_t, loff_t, int);
    long (*allocate)(struct inode *, int, mode_t, loff_t, off_t, loff_t, loff_t);
    int (*bmap)(struct inode *, struct bmap_extents_info *, u64 start, u64 len);
};
    
```



## Super Operations

```

struct super_operations {
    struct inode *alloc_inode(struct super_block *sb);
    void (*cleanup_inode)(struct inode *);
    void (*dirty_inode)(struct inode *);
    int (*write_inode)(struct inode *, struct writeback_control *wbc);
    void (*dquot_initialize)(struct inode *);
    void (*write_super)(struct super_block *);
    void (*fsync)(struct super_block *);
    void (*fsync_file)(struct super_block *, int, int, int);
    int (*freeze_b)(struct super_block *);
    int (*unfreeze_b)(struct super_block *);
    int (*remount_fs)(struct super_block *, int *, char *);
    void (*clear_inode)(struct inode *);
    void (*umount_begin)(struct super_block *);
    int (*statfs)(struct super_block *, struct statfs *);
    int (*fsync_dentry)(struct super_block *, struct inode *, struct dentry *);
    int (*show_options)(struct super_block *, struct dentry *);
    int (*show_devname)(struct super_block *, struct dentry *);
    int (*show_devname)(struct super_block *, struct dentry *);
};
    
```

create/destroy/commit inodes

synchronize file system state

summarize file system state

```

#ifdef CONFIG_QUOTA
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
#endif
    int (*bdev_try_to_free_page)(struct super_block *, struct page *, gfp_t);
};
    
```

## Dentry Operations

```

struct dentry_operations {
    int (*d_revaldata)(struct dentry *, struct nameidata *);
    int (*d_hash)(struct dentry *, struct qstr *);
    int (*d_compare)(struct dentry *, struct qstr *, struct inode *);
    int (*d_delete)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_invalidate)(struct dentry *, struct inode *);
    char *(*d_name)(struct dentry *, char *, int);
};
    
```

generally use generic hash function

hooks called before deleting/freesing

Inode and dentry caches coupled

"dcache" is just a hashtable

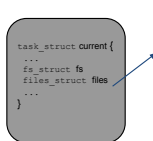
## Interacting with the VFS

- Root file system mounted at '/' and defines a tree
- Additional file systems can be mounted over existing directories, filling out the tree
- System calls like `open(2)` that take a name as input perform pathname lookup
  - Pathname lookup can start at '/' or in the current working directory (cwd specified in process desc.)
  - Performs permission checks, creates `struct dentry`s, reads on-disk inodes and instantiates in-memory `struct inode`s

## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

- The filename starts with a '/'
  - consult the process descriptor to find the root dentry



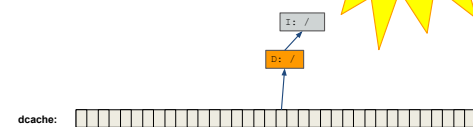
Each dentry points to a corresponding inode, which we need to continue our traversal

## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

- check the exec permission of the inode for '/'
  - `inode->i_op->permission(inode, MAY_EXEC)`

Each inode has an `i_op` field, which stores the set of file system specific inode functions



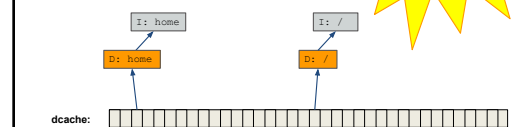
## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

- now look to see if a dentry for 'home' exists in hashtable
  - `dentry = d_lookup(parent_dentry, "home")`

- It exists in our dcache!
  - we don't need to go to disk to look up the inode

++refcounts on both the "home" dentry and its corresponding dentry->d\_inode

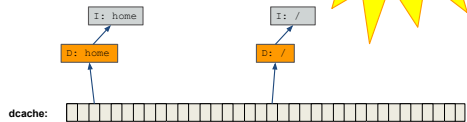


## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

- Next check the exec permission of the inode for "home"
  - `inode->i_op->permission(inode, MAY_EXEC)`

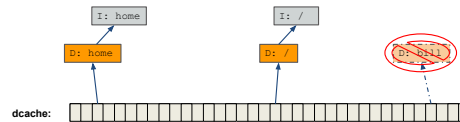
each inode has a mode field which stores permissions, including 9 "rwx" bits for "user|group|all"



## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

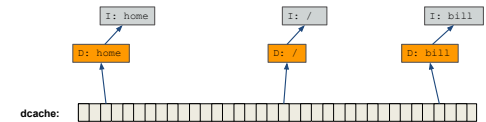
- look to see if a dentry for "bill" exists
  - `dentry = d_lookup(parent_dentry, "bill")`
- no dentry for "bill" exists!!!!



## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

- ... so we create a new dentry
  - `new_dentry = d_alloc(parent_dentry, "bill")`
- then look up the inode for "bill" on disk
  - `parent_dir->i_op->lookup(parent_dir, new_dentry, ...)`

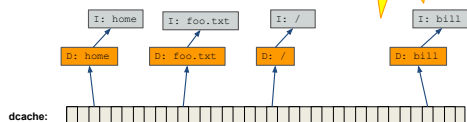


## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

- Next we look to see if a dentry for "foo.txt" exists
  - `dentry = d_lookup(parent_dentry, "foo.txt")`
- It exists?!?
  - how is it possible that "bill" wasn't cached, but "foo.txt" was?

Remember, the OS kernel (including the VFS data structs) is shared by all of your system's processes.



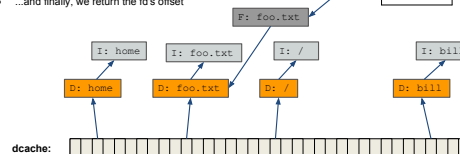
## Example (pathname lookup)

```
int fd = open("/home/bill/foo.txt", O_RDWR);
```

We finally have the inode of our file. Just a little more work...

- Check permissions (but this time, not just exec...)
  - `inode->i_op->permission(inode, acc_perms)`
- create a struct file and insert it into the process' file descriptor table
  - this is the reason that there is a limit to the # of open files
- ...and finally, we return the fd's offset

File Descriptor Table



## Notes

- What happens if part of a pathname cannot be found?
  - The VFS creates a *negative* dentry
    - Nothing more than a placeholder to represent the fact that a path component does **NOT** exist
- struct inode and struct superblock reflect on-disk data structures
  - a dirty bit tracks when they are out of sync with disk
  - should be written back frequently to avoid corruption
- What about struct dentry?
  - the dcache is just an in memory cache of names
    - a dentry does not have a dirty bit
    - deleting a dentry does not disrupt correctness