Virtual
File system
Switch

A brief overview of the Linux storage stack

Why add layers?

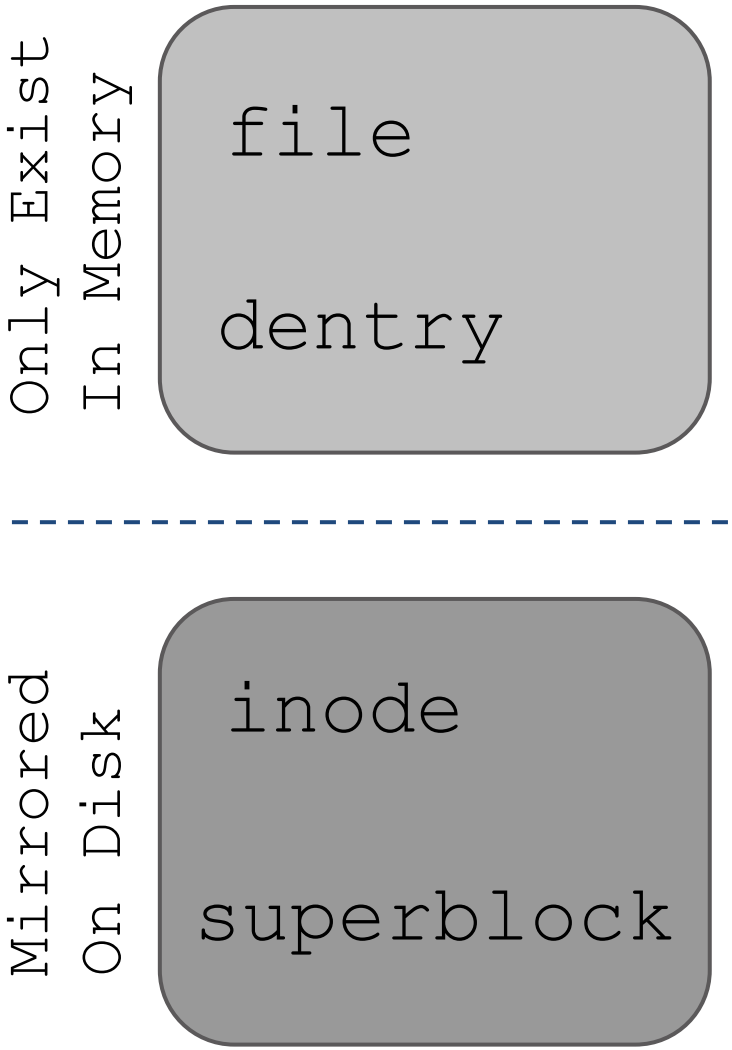
Originally, operating systems supported just one file system. What changed?

- 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



everything a process requires to interact with an open file

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

all information needed by a file system to handle a file

data pertaining to a mounted file system

Impact on Designing a FS

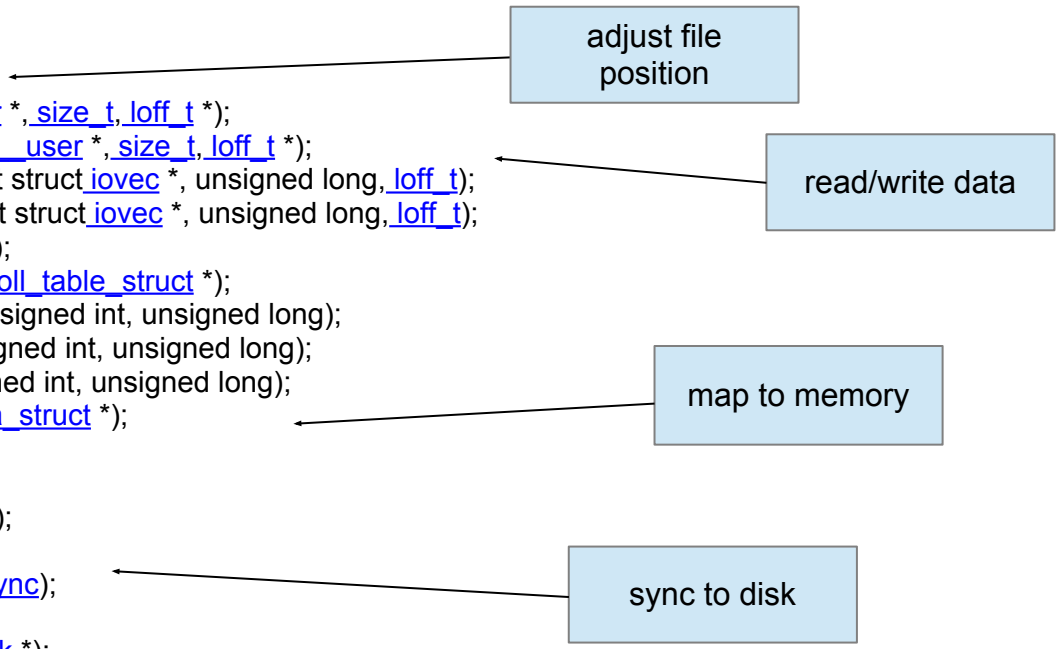
- Designers define a subset of common functions
  - register each file system with the kernel
  - when a file system is mounted, the kernel finds its functions in the table of registered file systems

- VFS calls FS functions at known locations

Question: How much flexibility does the VFS give us?

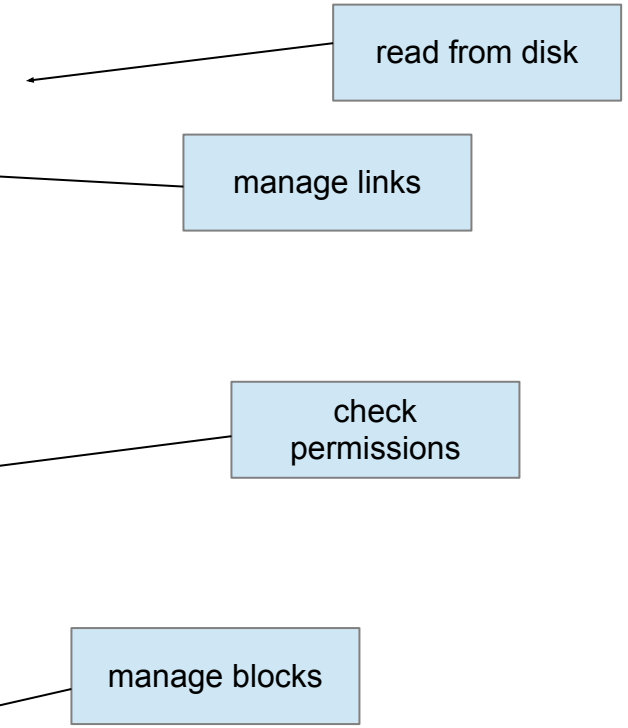
File Operations

```
struct file_operations {
        loff_t (*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 *);
        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 *, 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);
        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 **);
};
```



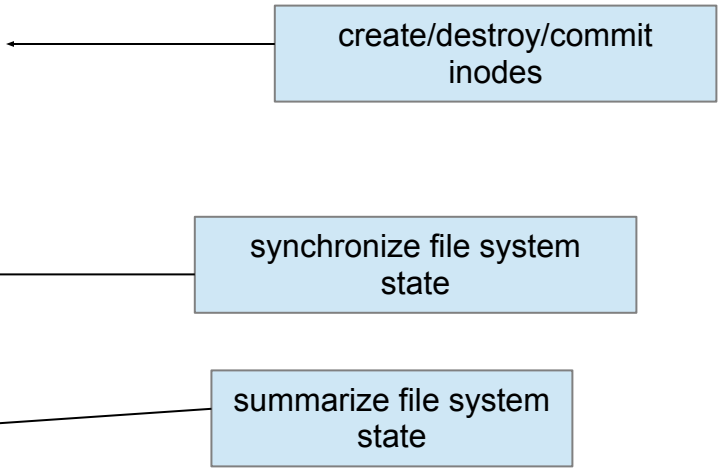
Inode Operations

```
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 *);
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        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 *);
        void (*truncate) (struct inode *);
        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);
}
```



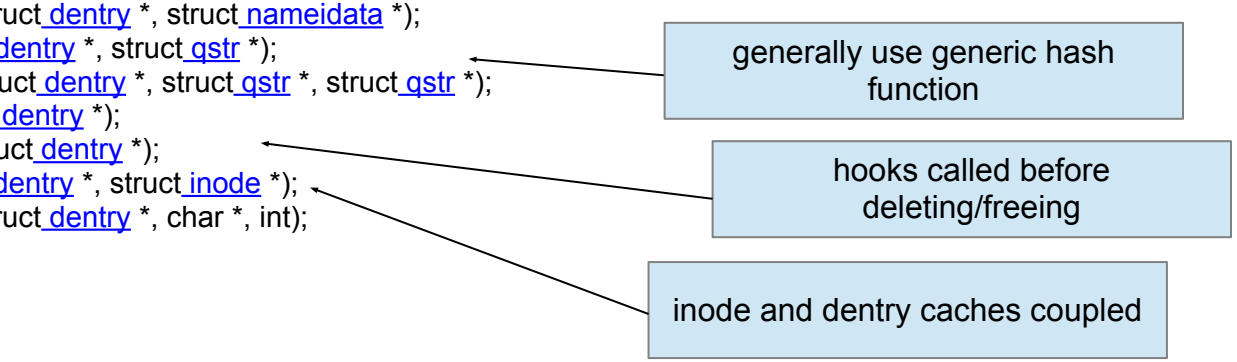
Super Operations

```
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);
        void (*drop_inode) (struct inode *);
        void (*delete_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 (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*statfs) (struct dentry *, struct kstatfs *);
        int (*show_options)(struct seq_file *, struct vfsmount *);
        int (*show_stats)(struct seq_file *, struct vfsmount *);
#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_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);
};
```



"**dcache**" is just a hashtable

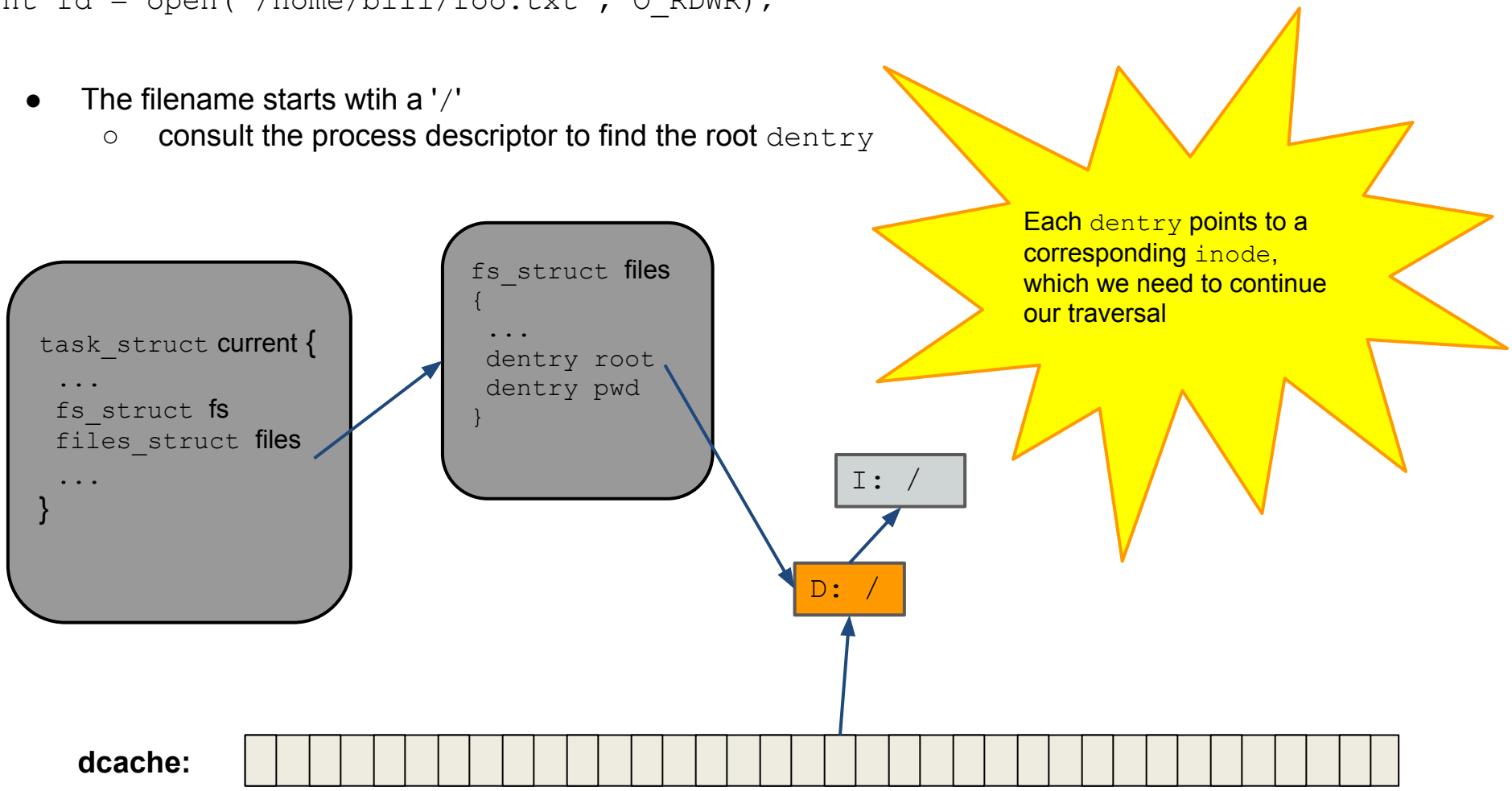
Interacting with the VFS

- 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);
```

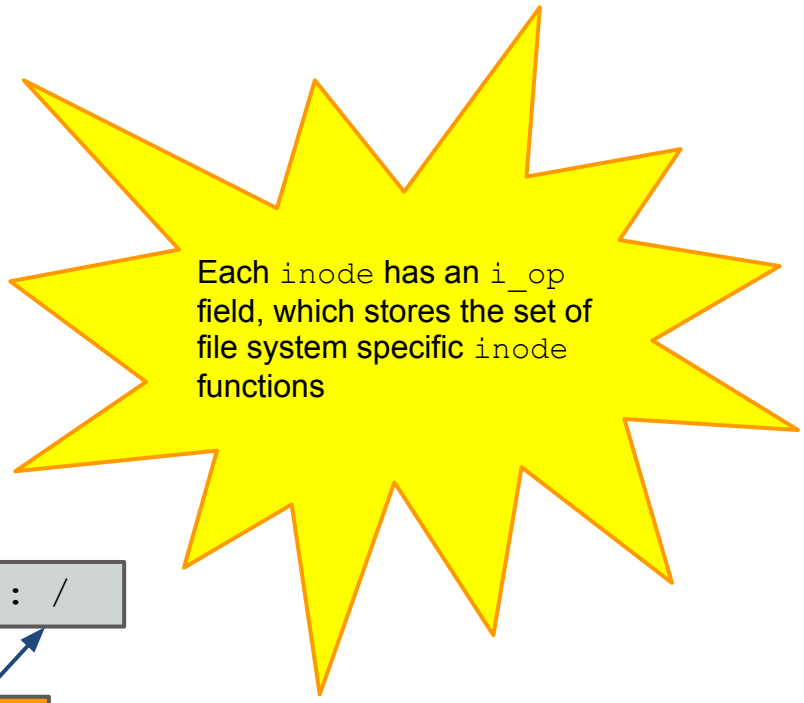
- The filename starts wtih a '/'
  - consult the process descriptor to find the root `dentry`

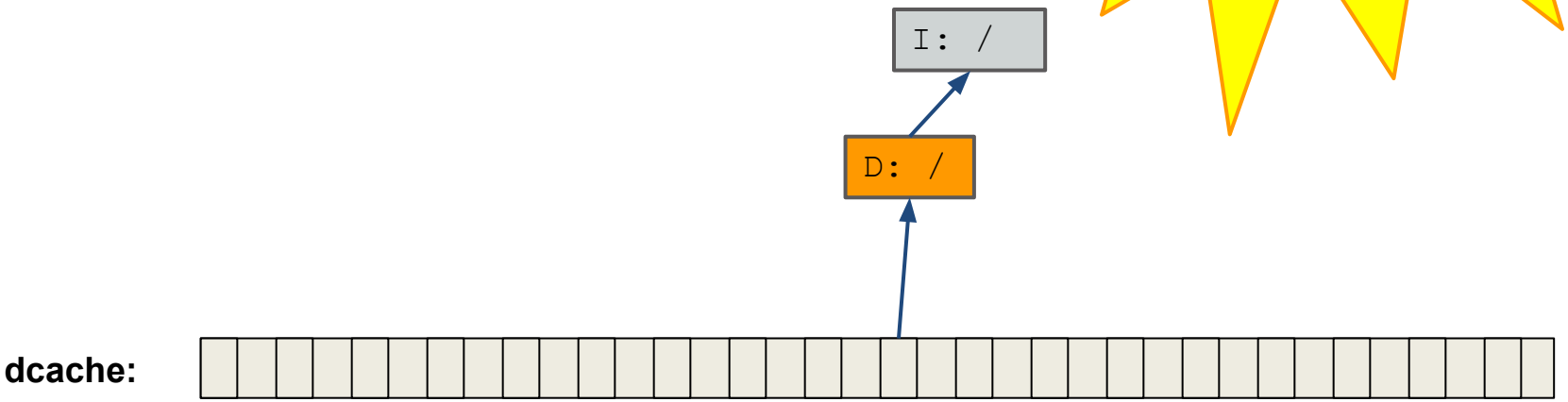


Example (pathname lookup)

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

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





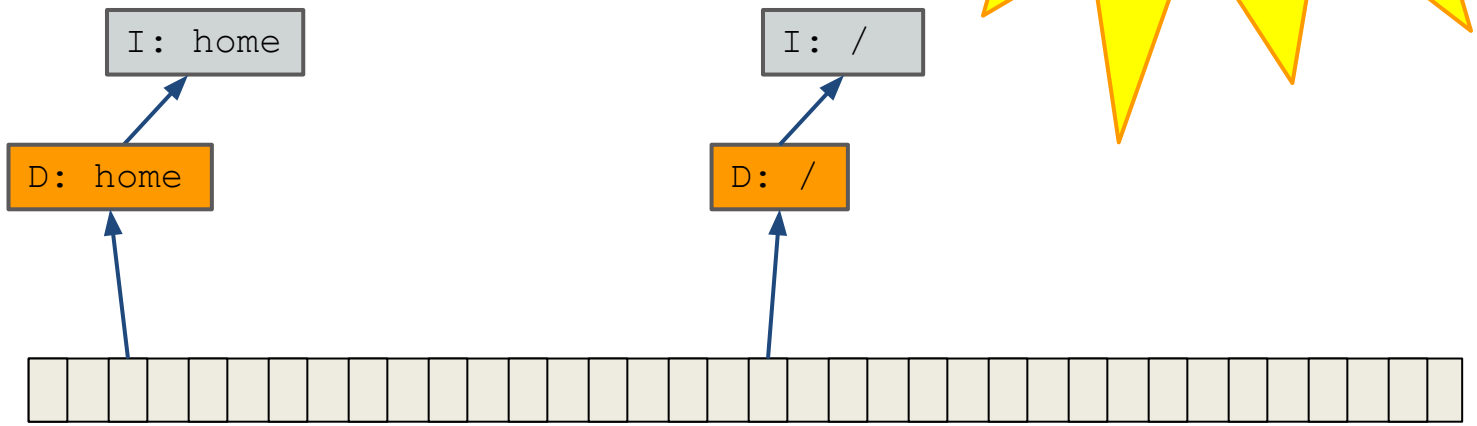
Example (pathname lookup)

```
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`

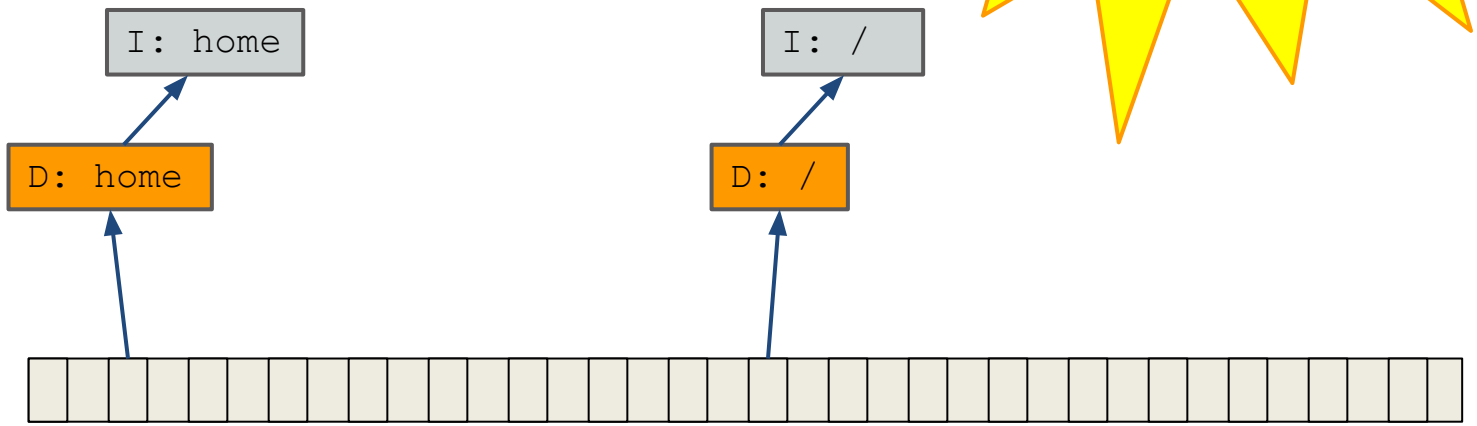


Example (pathname lookup)

```
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)`

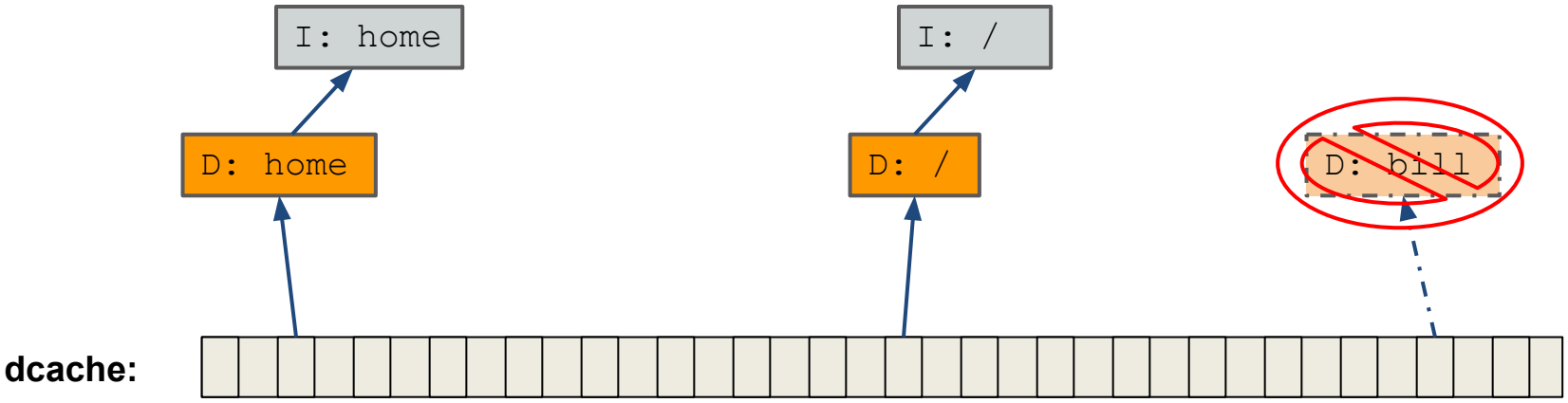
each `inode` has a `mode` field which stores permissions, including 9 '`rwx|rwx|rwx`' bits for '`user|group|all`'



Example (pathname lookup)

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

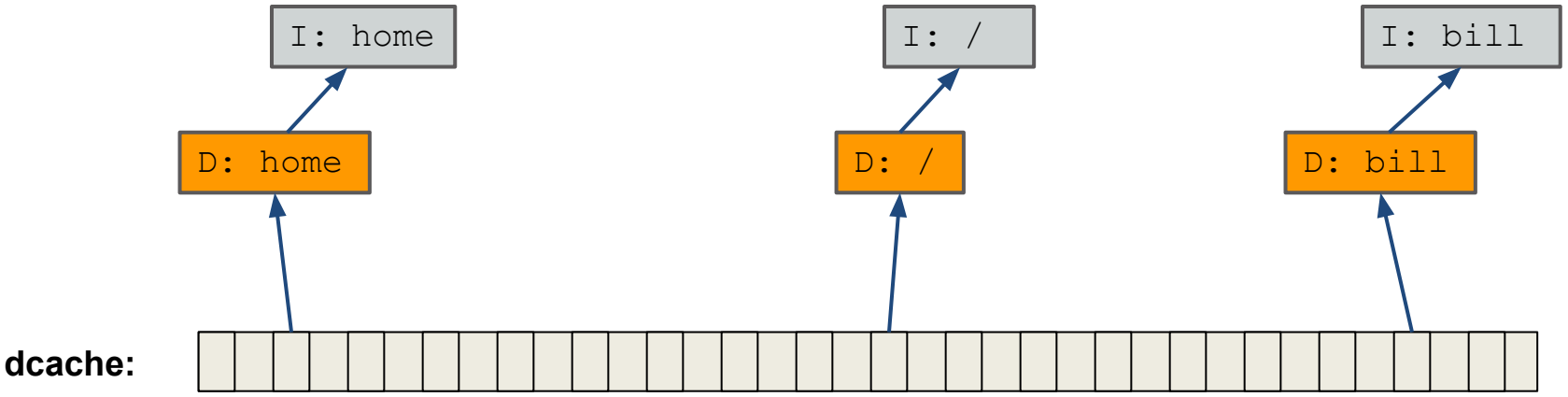
- look to see if a dentry for "`bill`" exists
  - `dentry = d_lookup(parent_dentry, "bill")`
- no dentry for "`bill`" exists!!!!



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, ...)`

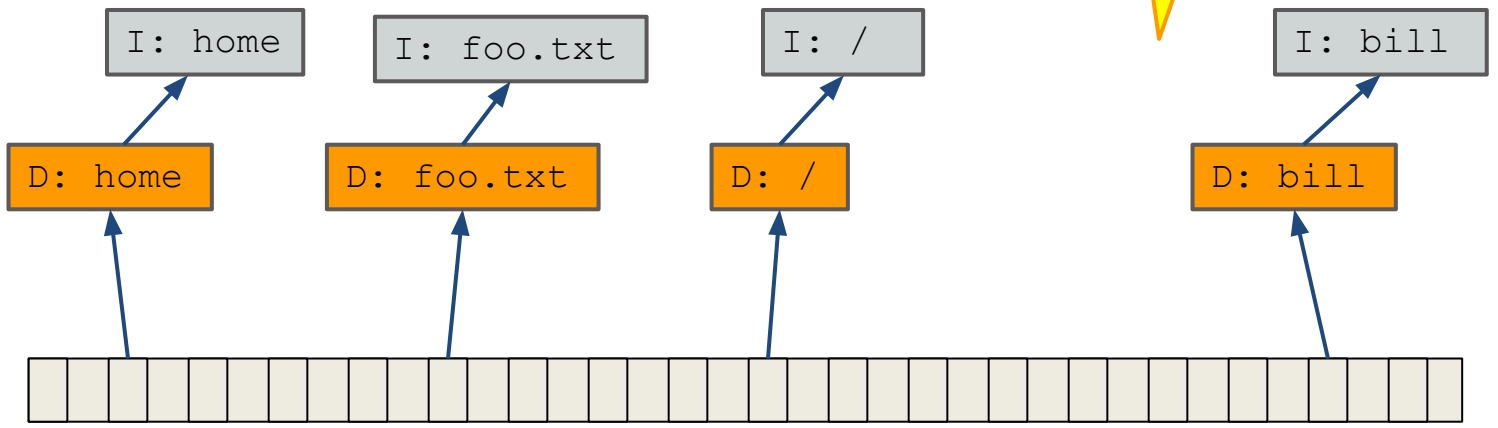


Example (pathname lookup)

```
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.

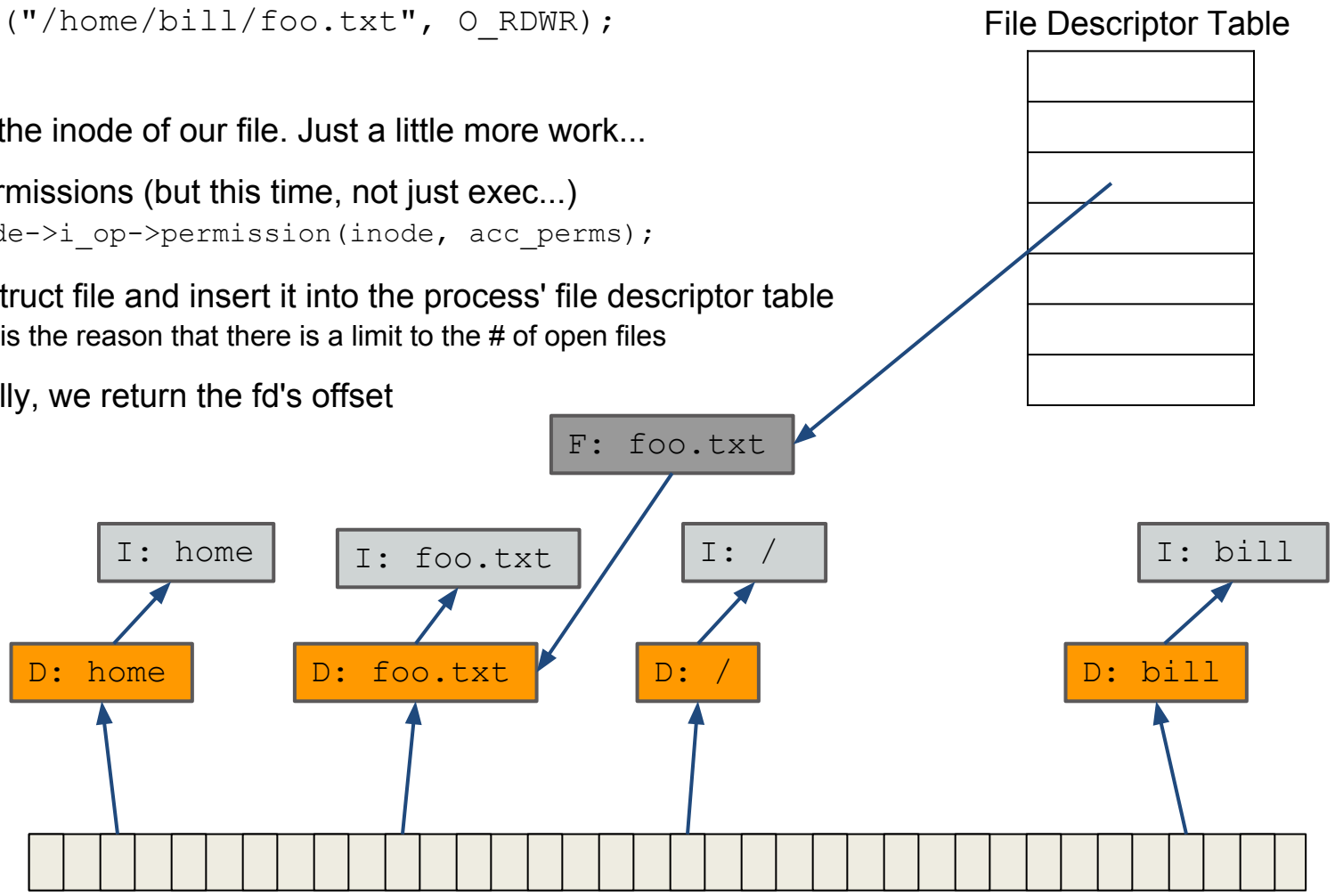


Example (pathname lookup)

```
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



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