* struct file, file_operations

struct path: contains dentry and 'vfsmount'
- vfsmount is a virtualized version of struct super_block
- allows a mount pt to have several instances in the namespace
- allows for private namespaces, per user/process, etc. (poly-instantiated namespaces)
- also supports chroot(2)

struct file_operations:
- the usual things corresponding to syscalls that create or take an 'fd'
- fallocate: ask f/s to pre-reserve space for a file, to minimize ENOSPC errors in middle of ->write.
- splice: allow copying of file data inside kernel b/t two fds
- sendpage: allows copying file data (by fd) to/from a socket fd
  useful for web servers to read HTML/etc files and send them back to web browser over a socket.

* struct address_space, address_space_operations

- handles interactions b/t f/s and the virtual memory (VM) subsystem, including caches, like pagecache.
- file->mmap can map a file to a user virt addr space
- supports mmap, swap, and also other mappings of file data to memory (e.g., compressed, encrypted file pages)

->readpage: get 4KB data into struct page, VFS will cache
- page struct includes "page index" (index of 4KB page in file)

->writepage: pagecache asks f/s to flush dirty page to disk, no file (may be closed already)
- struct writeback_control: VFS/VM tells f/s how to treat this dirty page write request. e.g., slower async write, vs. must-write-now sync write.

- direct_io: bypass page cache. useful if you read data once, and don't want to cache it; or write data directly to persistent media (e.g., database transaction log).

* stackable f/s

F/s has to translate POSIX syscalls, through VFS ops, to specific media:
- disk based f/s (ext4): figure out which sector/LBA number to access
- network based f/s (nfs, cifs): package request into a packet, send over some protocol
- CDROM/DVD: readonly block based
- in-memory f/s (ramfs): access individual bytes directly

Always want to add/change functionality to f/s: where?
- e.g., transparent file encryption
1. user level tools: manual, cumbersome
2. modify block-based f/s, to add support
- problem: you have to maintain and backport changes to the base f/s into yours.
- say you got f/s maintainers to take your changes into mainline. (ext4 supports file encryption as of ~2 years ago).
- adding support to each f/s is very cumbersome and time consuming.
3. let's add it to the VFS.
- but, may add too much overhead to all f/s, plus code stability
4. stacking: access another f/s that's already mounted
- not touching vfs or any one f/s
- but can mount on top of any other f/s and intercept VFS calls to that f/s.

VFS Structures:

Any f/s has:

```
VFS: sits above wrapfs (the stackable f/s)
upper (wrapfs): F -> D -> I
                |    |    |
lower (ext4):   F -> D -> I
below: a scsi disk
```

each stackable f/s object has a ptr to its corresponding lower f/s object, stored in the 'void*' in that struct.

A stackable f/s is called by the VFS: it looks to the VFS like a "regular" f/s. A stackable f/s calls a lower f/s AS IF it's the VFS that's calling it.

->iget: inode op to create a new inode, and populate it with the inode_ops,
  and other ops.
also sb->mount, initializes which ops vectors will be used for files,
  dentries, inodes, etc.