```
* struct inode_operations

almost 1-to-1 correspondence with syscalls. e.g.,

a fs doesn't have to implement all inode ops: only those that matter for
that f/s.  Note: some ops are mandatory (e.g., lookup).
- VFS will return -ENOSUPP for any op not supported by the f/s

        int (*unlink) (struct inode *,struct dentry *);

is the method the vfs calls from sys_unlink to the file system's own
->unlink method.

Note: ->FOO or X->FOO is jargon for "the FOO method of object X".  Examples
of ->unlink methods are nfs_unlink, ext4_unlink, msdos_unlink, etc.

->unlink takes an inode (I), and a dentry (D), and returns an int.
1. you return 0 on success, or >0 if the syscall permits it
        e.g., read/write return "number of bytes successfully read/written"
2. you return -errno to indicate error.
3. the "D", is the dentry that holds the name of the object to unlink
4. the "I", is the inode of the parent directory in which the name is to be
   deleted, namely the "." directory (not "..").  "I" is mutex locked by the
   vfs.

Many inode ops take I+D: same behavior.  "I" is the locked inode of the dir
in which "D" resides.  Same for unlink, create, or rename.

symlink takes a char*: what the symlink should point to

rename takes two inodes and two dentries: the I's represent the src and dst
dirs to rename D1 to D2.  But I1 and I2 must be locked exactly once!  If
I1==I2 and you lock both -> self deadlock.
- if I have two "struct inode *" ptrs, lock them in order
- we can lock the inode with the smallest inode number first (or highest
  first). Ensure that unlocking works in the SAME order.
- if I1==I2: lock just one; else lock in some hard-coded order.
- the above is just for renaming FILES.
- when renaming dirs: careful b/c you can move whole subtrees from one part
  of the namespace to another part of the namespace, while another thread is
  trying to move other parts into parts you're moving.
- simple solution: serialize all dir renames using sb->s_vfs_rename_mutex

->rename2 allows for atomic renames including name swaps, only if f/s
  supports it.

Most inode ops return an int, other than ->lookup

struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
- takes an I, D, some flags; returns a struct dentry* (D')
- looking up in a locked directory "I", for object whose name is "D"
1. if error: returns an encoded ptr, use IS_ERR and PTR_ERR to find error
   (EIO, ENOMEM)
2. if object not found: returns D', w/o an inode inside -- a "negative"
   dentry. VFS will cache this neg. dentry D', and return -ENOENT to userland.
3. if object found: return a positive dentry D' with valid inode inside.
   VFS will cache this positive dentry and return 0 to userland (if need to
   return).

How does a lookup operation happens from userland to actual f/s:
1. suppose user runs "/bin/rm /home/jdoe/src/hw1/foo.c"
2. translates to unlink("/home/jdoe/src/hw1/foo.c")
3. translates to sys_unlink w/ same pathname
...
```

```
...
...


X-2: ->lookup(I, D)
- lookup "hw1" (D) inside "src" (I)
X-1: ->lookup(I, D)
- lookup "foo.c" (D) inside directory "hw1/" (I)
X. the f/s will get an ->unlink(I, D)
- I: is for "hw1/" dir ("I" had to have been found using ->lookup)
- D: is for "foo.c"

Recursively: To find any dir entry, has to look it up in its parent
- at top we have to lookup "home" in "/"
- how do we find "/"?! that is sb->s_root, and gets created at mount(2) time
  of the f/s.

General alg when performing syscall such as unlink(2) on a pathname
- this procedure is traditionally called "namei" resolution, or
"pathname-lookup", or lookup_pn, etc.

Parse pathname one component at a time
foreach component in pathname
        lookup component in its parent dir
                if entry is in dcache, return it
                else call ->lookup on specific f/s
        if lookup failed: return err or ENOENT
        if ENOENT: cache entry in dcache
#       if found, go to next component in loop
        if found, call ->permission on inode with read/write/execute perm
        checks, so f/s can compare obj perms on disk w/ "struct task
        *current" executing process's permissions.
if final component of pathname is found
        check ->permission again
        call ->unlink (or any other ->op as per the syscall)
        return status of ->op to vfs, which is returned to userland

So if you're a f/s, what you'll see for EVERY syscall that takes a "char *pathname" is:
- a series of lookup+permission pairs, for everything that's NOT cached in
  cached
- followed by the actual ->op (if lookups+permissions succeeded)

The namei procedure is actually more complicated.  Has to handle
1. symlinks:
- if obj you found is a symlink, then have to call ->readlink to retrieve
  the content of the symlink.  Then parse it on each "/".  Note: symlinks
  can point to absolute ("/foo/bar") or relative ("foo/bar" or
  "../foo/bar") pathnames.
- have to detect symlink loops: namei fxn keeps a counter "how many times a
  symlink was crossed", if over some max, abort and return ELOOP.
2. mount points:
- each time you get a dentry, find out if it's a mount point
- if so, find out WHAT other f/s is mounted at this point (and get its SB)
- then, continue namei resolution from the NEW sb->s_root

Note: relative pathname resolutions (e.g., "stat ../foo") use struct task
*current's "cwd" (a dentry that's changed bu chdir/cd).


* dentry_operations

most are optional

->d_revalidate: useful esp for network f/s, to verify if a dcached dentry is
  still valid:
```

```
- if f/s implements d_revalidate, then VFS calls it. ->d_revalidate returns
1. -errno: "bad" error happened (e.g., EIO, ENOMEM):
- remove obj from cache
- abort namei lookup process at this component
- return -errno to user
2. 0: obj is valid, can use what's cached
3. 1: obj is stale (invalid).  VFS removes the obj from dcache, and then
  issues a new ->lookup.
```