```
* Virtual File System (VFS)

1. a repository of common/useful code (like libc) for f/s developers.
2. acts as an abstraction layer, a broker/dispatcher from syscalls to actual
   f/s implementations.

Sources:
- include/fs.h -- main VFS header file
- include/dcache.h -- for directory cache
- fs/*.c: VFS sources
- fs/XXX/*.c: sources for file system XXX

Data structures:

0. Things common to all (or most) VFS d-s

- structures to connect multiple instances using linked lists, hash tables
  (e.g., list_head)
- a ptr to a function ptr array/vector (methods to operate on the object)
- various locks, several, different types.
- reference counts: to track object liveness.

1. struct inode [I]

VFS structure.  Encodes information about a single object (e.g., file,
directory, symlink, etc.) that is stored on some persistent media (e.g.,
hard disk).  Also includes #links to inode, in case of hardlinks.

Contains: inode number, permissions, owner/group, timestamps(m/a/ctime),
etc.  stat(2) info.  Object type (file, dir, symlink, etc.).  Pointers to
where the data blocks live.

2. struct dentry [D]

VFS structure.  Encodes the name of the object/file.   It's used to cache
dentry objects in a large "directory entry cache" or "dcache".  (In other
OSs it's called a Directory Name Lookup Cache, or DNLC.)

3. struct file [F]

VFS structure.  Contain info about an object/file that's been opened:
read/write offset, open mode (can open a file as readonly, even if inode
permissions are readwrite).

4. struct super_block [SB]

Similar to inode, but records info about a whole f/s on "disk".  Total
no. of inodes/blocks (how many are free/used), unique ID, block size, f/s
type.  statfs(2)/statvfs(2).

F points to D; D points to I; I points to SB; F points to SB; etc.

* reference counts (RC)

a number inside object X, saying how many others point to this object.
- if RC==0: object has no pointers to it (or "users of the object").  Can free.
- if RC > 0: it's used by someone, cannot free object.

If someone looks up a name "foo.c" and finds a corresponding inode on disk,
then we have both D and I.  A lookup can happen with simple stat(2).

D -> I

I's rc=1
```

```
D1 -> I <- D2 (hard link example)

I's rc=2.  D1's and D2's RC=0, so they can be removed if/when needed to make
room.  If someone wants removes D2, I's rc is decremented by 1, rc=1.

Opening a file:

F -> D -> I
I's rc=1
D's rc=1
F's rc=1 (someone else points to this 'F')

Struct file in kernel, is seen as an int fd in userland.  Per process/task
list of open files.   In linux, use 'struct task':

struct task {
  struct file **open_files; // up to some MAX open-fd allowed per process
}
task->open_files[0] -> "struct file *" for fd=0 (stdin)
task->open_files[1] -> "struct file *" for fd=1 (stdout)
task->open_files[2] -> "struct file *" for fd=2 (stderr)
task->open_files[3] -> "struct file *" for fd=3 (first opened file in a new process)

Refcounts use an atomic variable (atomic_t) in kernel, and methods
atomic_inc, atomic_dec, atomic_read, etc.  See <linux/atomic.h>.

The one who adds a ptr to an object X, must increment X->rc by 1.
The one who removes a ptr to an object X, must decrements X->rc by 1.
- If you decrement an RC too much, you can have dangling pointers -- bug
- if you inc an rc too much, you can "leak" an object, like a memleak.

Processes 1 and 2 open file "foo.c".
process 1: F1 -> D -> I
process 2: F2 -> D -> I
RCs: I's rc=1, D's rc=2, F1's rc=1, F2's rc=1.

Sometimes you see a struct file with rc=2 or more.  This can happen when you
use dup(2)/dup2(2).

* dcache

F -> D -> I
D -> I

D -> NULL  (a "negative dentry")

Represents a negative cache entry, useful to store info that an object DOES
NOT exist in the backend store, to save I/O.  If you find neg. dentry in
dcache, quickly return ENOENT.

A positive dentry can turn negative after you "unlink" or delete the file
(inode).  A negative dentry can turn positive if you create a file by same
name.  Can also remove neg. dentries from memory to save space.
```