* VFS data structures, cont.

prefix for fields of d-s:
i_* for inode
d_* for dentry
f_* for file
etc.

foo->i_* -- easy to tell what type 'foo' is (inode)

Fields in inode and other structs are close to each other, to ensure CPU cache line locality: if you access one field, you're likely to access the others. e.g., i_[acm]time, size fields, uid/gid/mode (for permissions).

struct list_head: an embedded linked list inside the struct
see <linux/list.h>
- i_sb_list: all inodes in this SB, unordered (for general sweep)
- i_lru: inodes in SB, in rough LRU order (for cache mgmt)
- i_io_list: inodes under i/o

struct hlist_node: a hash table built on top of list_head's
- i_hash: to lookup an inode efficiently

atomic_t i_count: the refcount for inodes

* locking in inode

some fields are readonly, no need to grab a lock (e.g., i_blkbits, i_ino)

Some fields require grabbing a mutex (a blocking lock, may go into WAIT state): e.g., when performing I/O on the file.

Some fields are self locking: e.g., atomic_t i_count (refcount)

Some fields use a non-blocking spinlock (for quick in-ram updates only):

```
        spinlock_t i_lock; /* i_blocks, i_bytes, maybe i_size */
```

e.g., in O_APPEND, then update file size in inode in ram, flush it later (need to grab mutex for that)

* inode extensibility

always have a void* for f/s to put their own extra inf into

In past (e.g., 2.4.18) had a big union at end of struct inode
- con: had add new field for every new f/s added into mainline
- con: size of inode limited by size of largest union member (wasted mem on avg)
- pro: per fs info is localized (in CPU cache) with rest of inode (a ptr would most likely cause a CPU flush)

In newer kernels, no more union, only void*. So how to get benefit of locality? A: an out of band (or variable length) data structure.

e.g., ext3 to alloc a new inode (inside ext3 f/s code)

```
1. void *ptr = kmalloc(sizeof(struct inode) + sizeof(struct ext3_inode_info))
2. struct inode *ip = ptr;
3. struct ext3_inode_info *ext3ip = &ptr[sizeof(struct inode)]
                                    or ptr + sizeof(struct inode)
4. return ptr back to VFS
(and not using inode->i_private)
```

Pros: locality in CPU caches, one buf to kmalloc/kfree, size is just large to hold what's needed and no more.

Note: can put ext3_inode_info bytes before struct inode, if wanted.
UPDATE: linux puts *_inode_info bytes before struct inode in container

The above OOB d-s is called a "container" in linux.

e.g.,

```
struct foo {
        int i;
        float f;
        char buf[0]; // may need char buf[1];
}; // variable length null terminated string goes at end of struct, field 'buf'
```

* linux/dcache.h (struct dentry)

usual stuff: locks, ops, locality of fields, void* (extensibility),

also list_head's d_child and d_subdirs: for recursive programs like find, chmod -R, etc.

```
struct dentry *d_parent;        /* parent directory */
```

for "cd .." and parsing paths like "../../../foo/bar"

```
struct qstr d_name;
unsigned char d_iname[DNAME_INLINE_LEN];        /* small names */
```

- DNAME_INLINE_LEN is usually 32-40 bytes long
- note: max pathname in POSIX is 4096, max file name is 256B

Locality: most file names are short, so can fit inside dentry (cache locality). Larger file names need a kmalloc of the full str, and put ptr inside struct qstr (embedded inside dentry).