File management
Virtual file system: mounting, dentry cache, pathname lookup, inodes in memory

Table of contents

- Registering file systems
- Mounting file systems
- Dentry structure
- Dentry cache
- Pathname lookup
- Managing inodes in memory
  - function ilookup
  - function iput
- Relations between VFS data structures

Registering file systems

The file system is **registered** at system startup or when the kernel module supporting the file system is being loaded.

Registered file systems are represented by **file_system_type** structures, combined into a linear list, pointed to by the variable **file_systems**. New objects are placed at the end of the list.

```
static struct file_system_type *file_systems;
```

Some fields of the **file_system_type** structure:

type	field	description
const char *	name	File system name
int	fs_flags	Mount flags
struct module *	owner	Pointer to the module implementing the file system
struct file_system_type *	next	Pointer to the next item in the list
hlist_head	fs_supers	List of superblocks of this file system

Registering file systems

File system registration is done by **register_filesystem()**.

The **unregister_filesystem()** removes the given file system from the list of registered systems.

```
int  register_filesystem(struct file_system_type * fs)
Int  unregister_filesystem(struct file_system_type * fs)
```

You can see all registered filesystems in the file **/proc/filesystems**.

The screen to the right shows the filesystems on ‘students’. If a filesystem is marked with "**nodev**", this means that it does not require a block device to be mounted (e.g. virtual filesystem, network filesystem).

```
jmd@students:~$ cat /proc/filesystems
nodev   sysfs
nodev   tmpfs
nodev   bdev
nodev   proc
nodev   cgroup
nodev   cgroup2
nodev   cpuset
nodev   devtmpfs
nodev   debugfs
nodev   tracefs
nodev   securityfs
nodev   sockfs
nodev   bpf
nodev   pipefs
nodev   ramfs
nodev   hugetlbfs
nodev   devpts
nodev   mqueue
nodev   selinuxfs
nodev   pstore
        ext3
        ext2
        ext4
nodev   autofs
nodev   efivarfs
nodev   configfs
nodev   rpc_pipefs
        fuseblk
nodev   fuse
nodev   fusectl
nodev   nfsd
        vfat
nodev   binfmt_misc
jmd@students:~$
```

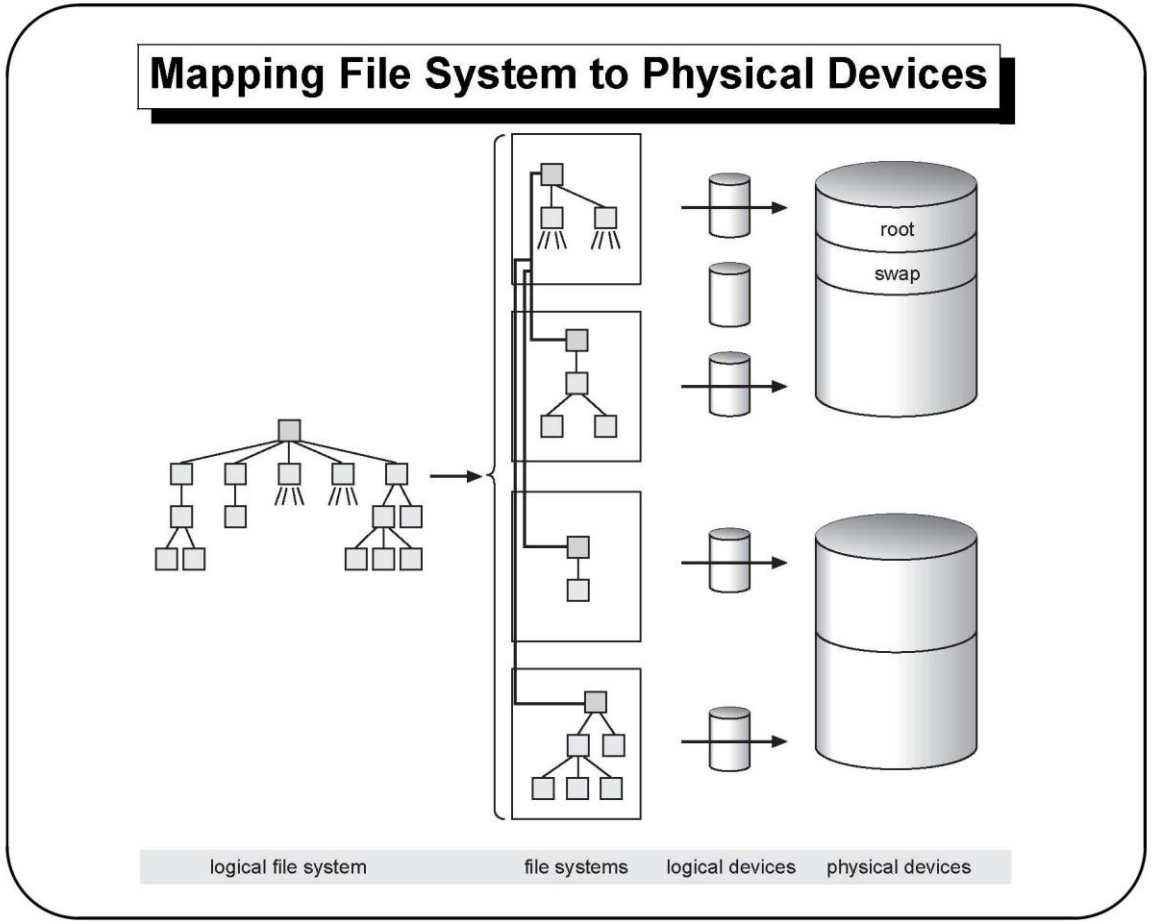
Mounting file systems

To mount the file system in an existing directory structure, use the **sys_mount()** system call (calling **do_mount**).

The reverse action is performed by the **sys_umount()** function (calling **do_umount**).

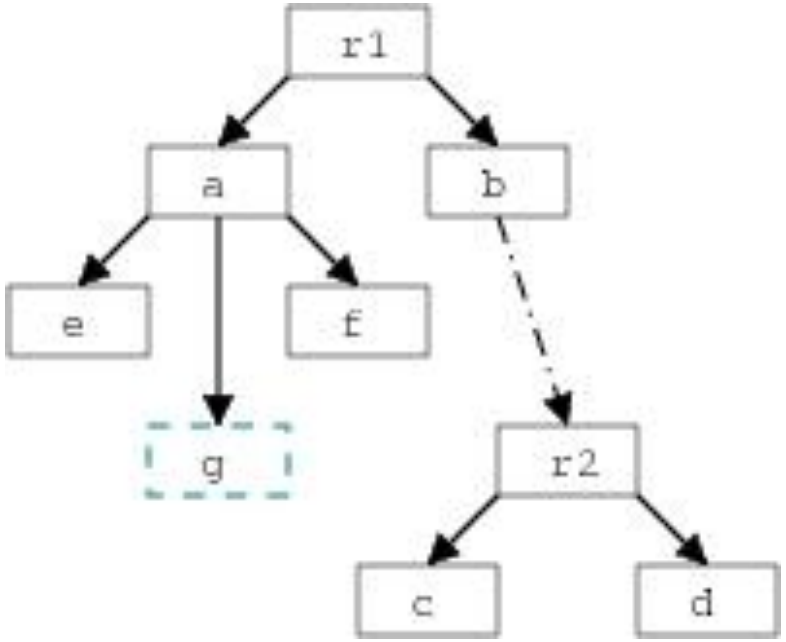
```
asmlinkage long
sys_mount(char * dev_name, char * dir_name, char *
type, unsigned long flags, void * data)

asmlinkage  long  sys_umount(char * name, int flags)
```

Since Linux 2.4 a **single filesystem** can be mounted at **multiple mount points**, and **multiple mounts** can be stacked on the same **mount point** (see man pages for mount()).



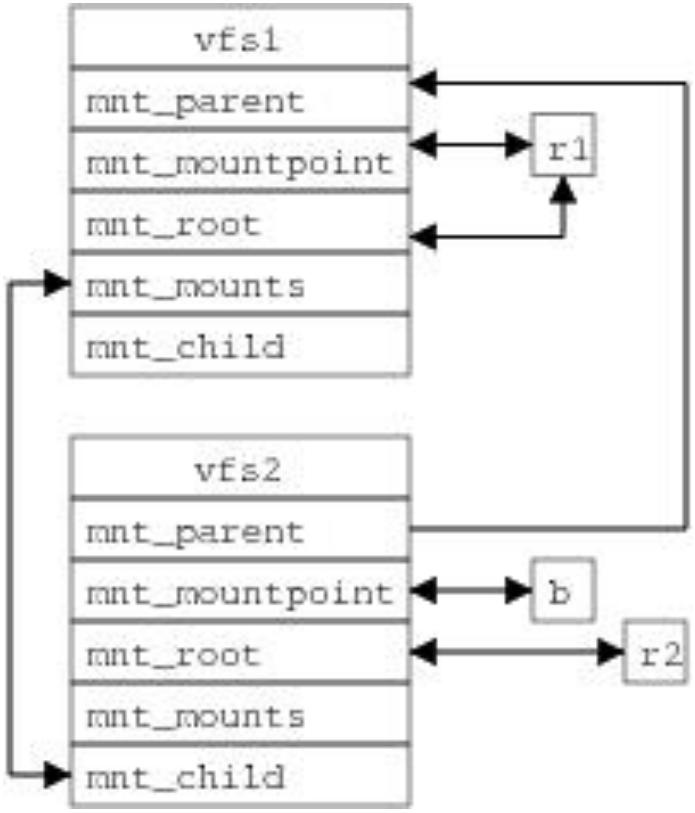
Hierarchical directory structure (source: Silberschatz)

Mounted filesystems and dentry cache



The **example filesystem tree** has two mounted filesystems, with roots **r1** and **r2**, respectively. The filesystem **r2** is mounted on directory **b**. The file **g** has not been referenced recently and therefore is not present in **dcache**.

VFS Mount tree



Mounted filesystems

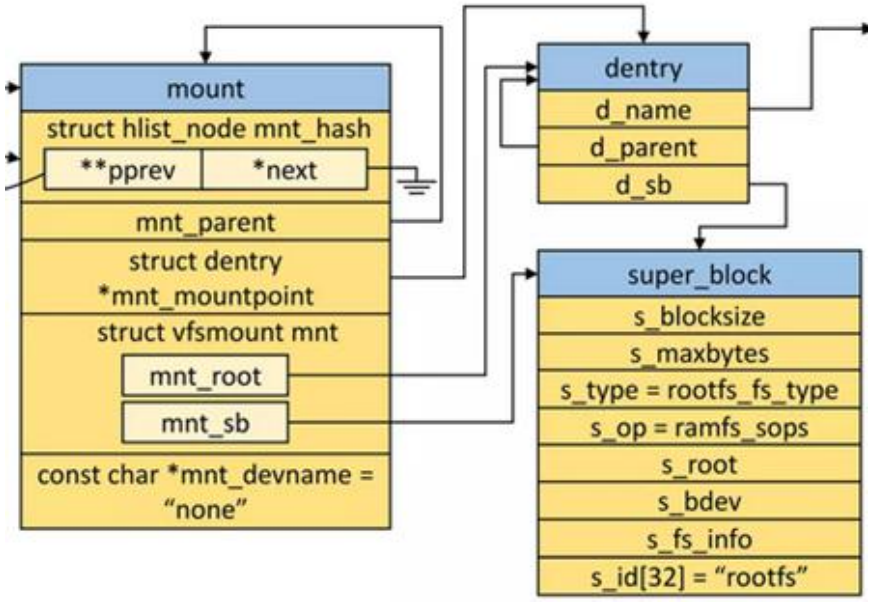
The **vfs1** structure references the **root dentry r1** both as the **mnt_mountpoint** and the **mnt_root**, because this filesystem is the ultimate root of the filesystem tree.

The **vfs2** structure references **dentry b** as its **mnt_mountpoint** and **r2** as its **mnt_root**.

When the **mount hash table lookup** returns a pointer to **vfs2**, the **mnt_root** field quickly locates the root of the mounted filesystem.

(source: Scaling dcache with RCU, Paul E. McKenney, 2004)

Mounted filesystems



The mounted file system is represented by **struct vfsmount**.

Once **struct mountpoint** used to be a part of **struct dentry** – the list of all mounts on given mountpoint. Since it doesn't make sense to bloat every dentry for the sake of a very small fraction that will ever be anyone's mountpoints, that thing got separated.

```
struct vfsmount {
    struct dentry  *mnt_root;        /* root of the mounted tree */
    struct super_block  *mnt_sb;     /* pointer to superblock */
    int  mnt_flags;
} __randomize_layout;

struct mountpoint {
    struct hlist_node   m_hash;
    struct dentry   *m_dentry;
    struct hlist_head   m_list;      /* mounts for the same mountpoint */
    int   m_count;
};

struct mount {
    struct hlist_node    mnt_hash;   /* a node in the mount_hashtable */
    struct mount    *mnt_parent;     /* parent mount */
    struct dentry    *mnt_mountpoint;  /* mount point */
    struct vfsmount   mnt;           /* filesystem */
    struct list_head  mnt_mounts;    /* list of children, anchored here */
    struct list_head  mnt_child;     /* and going through their mnt_child */
    struct list_head  mnt_list;
    struct mountpoint  *mnt_mp;      /* where is it mounted */
    struct hlist_node  mnt_mp_list;  /* list mounts with */
    ...                              /* the same mountpoint */
};
```

Why does mount structure has two mountpoint fields? (Al Viro, 2019)

m->mnt_mp->m_dentry == m->mnt_mountpoint

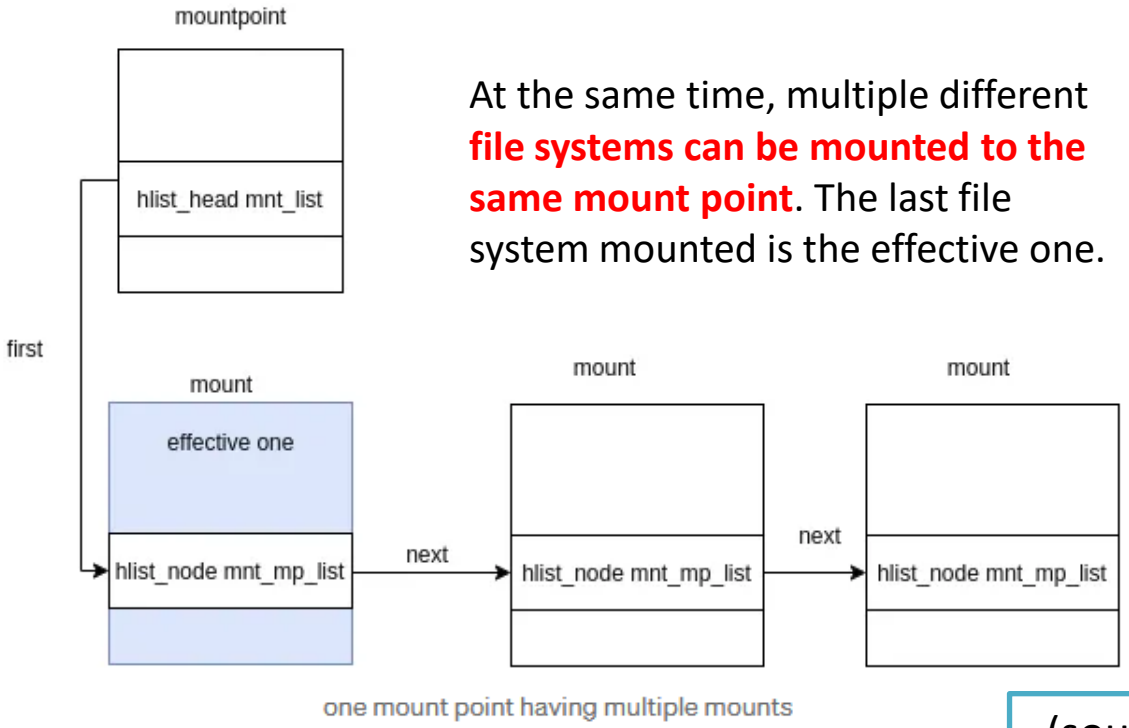
Simplified version

Mounted filesystems

The mounts in Linux are in a tree structure. The **parent** mount can have multiple **children**.

mnt_mounts – head of a list including all filesystem descriptors mounted on directories of this filesystem.

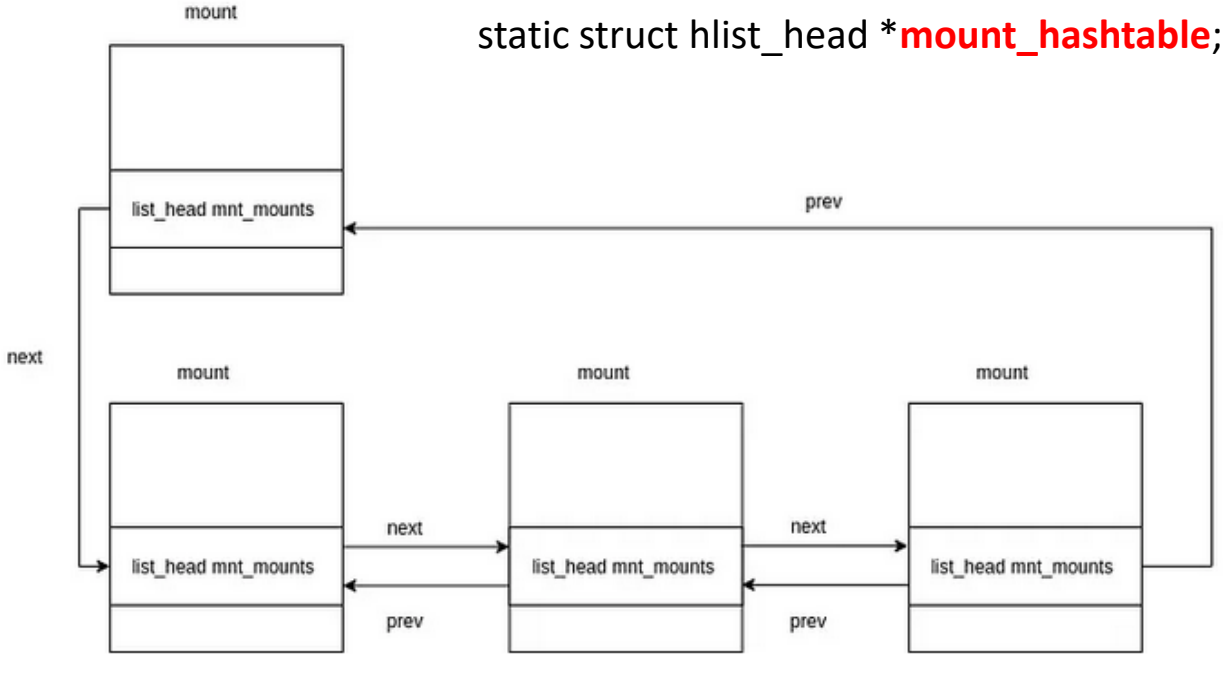
static struct hlist_head ***mountpoint_hashtable**;



one mount point having multiple mounts

At the same time, multiple different **file systems can be mounted to the same mount point**. The last file system mounted is the effective one.

static struct hlist_head ***mount_hashtable**;



mount tree

- dentry which is a mountpoint is marked with **DCACHE_MOUNTED** flag
- each such **dentry** has a struct **mountpoint** instance (exactly one)
- struct **mountpoint** has a pointer to its **dentry** (->m_dentry)
- struct **mountpoint** instances are **hashed**, using ->m_dentry as search key
- struct **mount** has reference to struct **mountpoint** (->mnt_mp)
- when ->mnt_mp is non-NULL we are guaranteed that
  m->mnt_mp->m_dentry == **m->mnt_mountpoint**

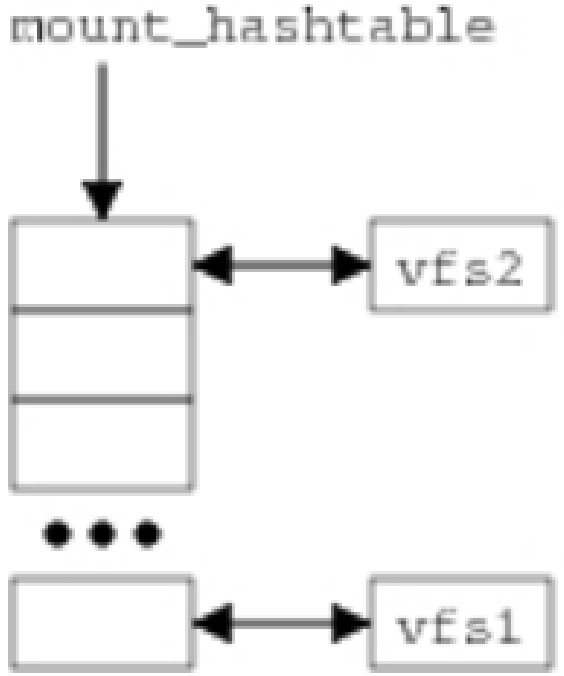
(source: Linux Namespaces (Container Technology), Ryan Zeng, 2021)

Mounted filesystems and dentry cache

All mounted file systems are placed in a **hash table** pointed to by the variable **mount_hashtable** defined in **/fs/namespace.c**.

The **mount_hashtable** data structure is used to map the **mountpoint dentry** to the struct **vfsmount** of the **mounted filesystem**.

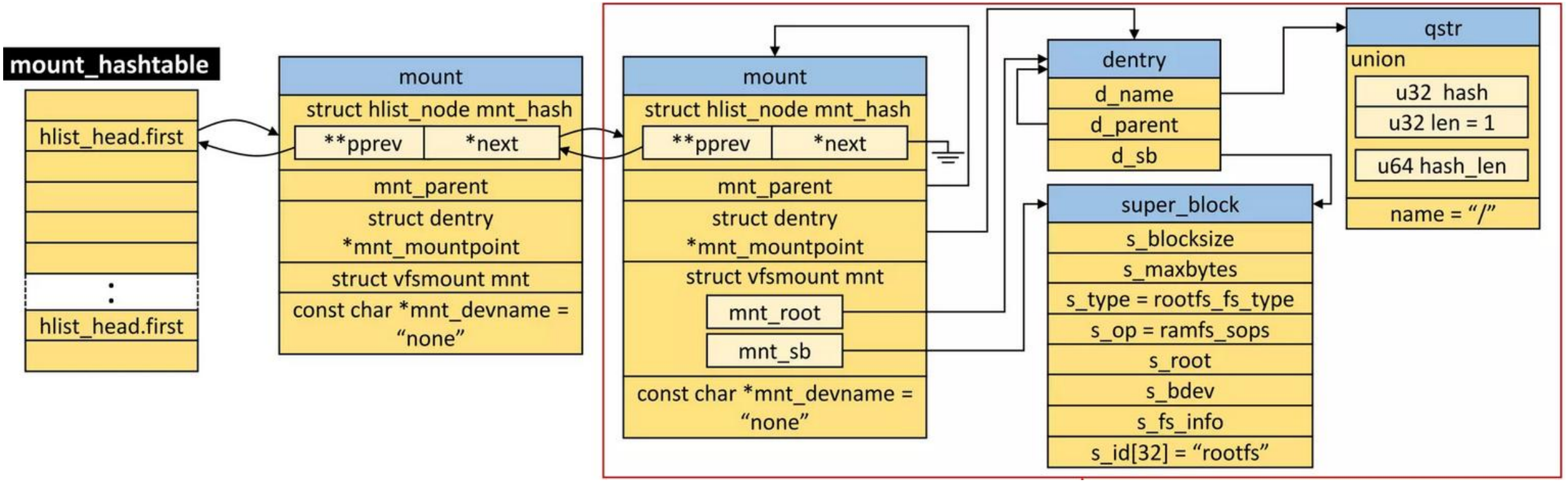
This mapping hashes the **pointer** to the **mountpoint dentry** and the **pointer** to the struct **vfsmount** for the **filesystem** containing the **mountpoint**.



Instead of **struct dentry** having a list pointer, all the **mountpoints** are stored in a **hash table**. The **m_dentry** field is used to distinguish different **mountpoints** that fall into the same hash bucket. **Struct mount** holds a reference to **struct mountpoint** mostly for **cleanup**.

(source: Scaling dcache with RCU, Paul E. McKenney, 2004)

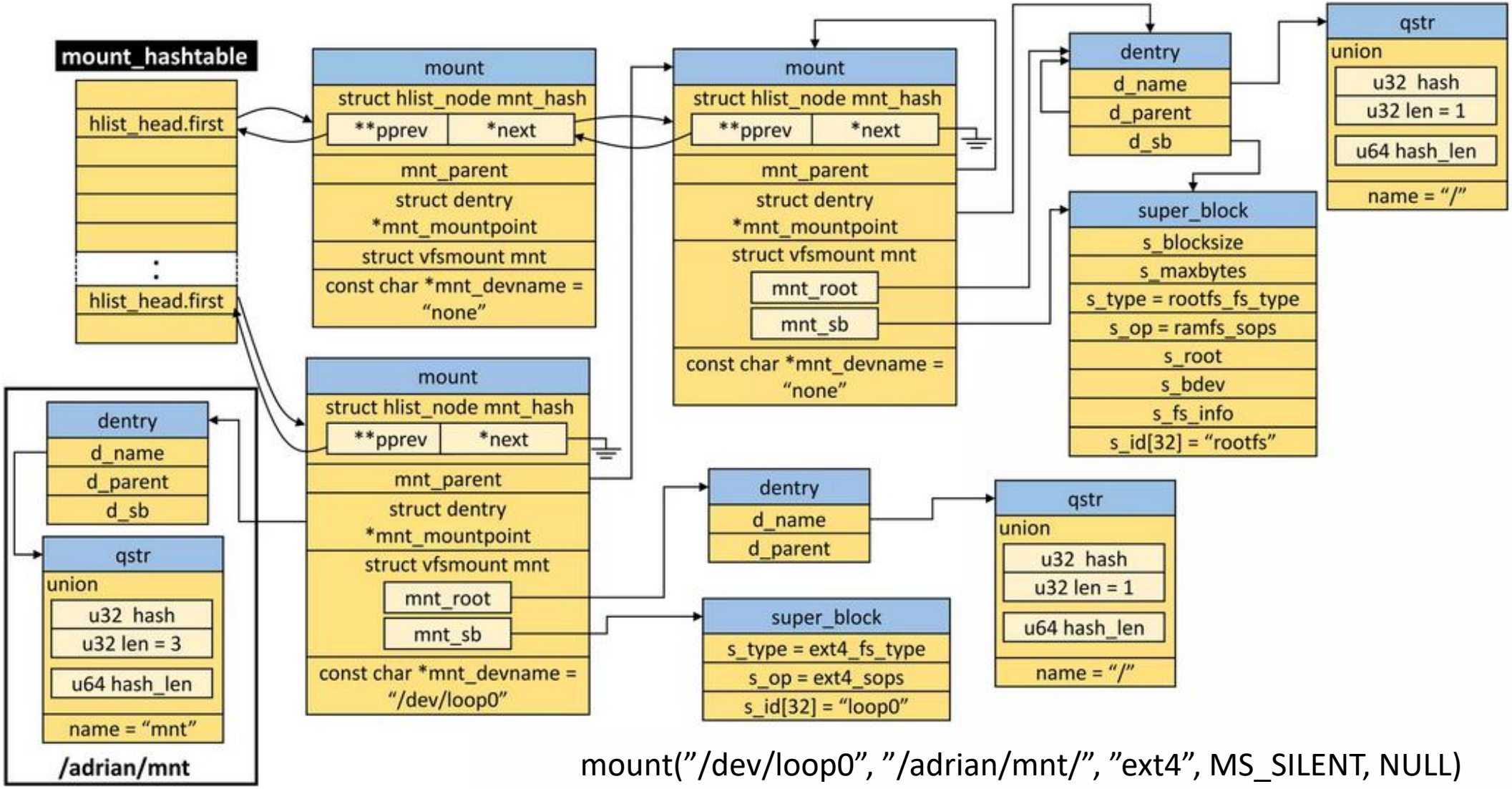
Mount before mounting a new ext4 file system



Information about mounted file systems: **cat /proc/mounts**
List all the mounted filesystems in the system: **findmnt**

(source: Adrian Huang, Virtual File System, 2022)

Mount after mounting a new ext4 file system



mount("/dev/loop0", "/adrian/mnt/", "ext4", MS_SILENT, NULL)

(source: Adrian Huang, Virtual File System, 2022)

Dentry structure

Users identify files by **path names** (one file can have multiple names), the system identifies files by **inode numbers**.

The path name components (relative names) and their corresponding inode numbers are stored in **directories**.

A **directory** is treated as a file whose contents are a list of directory entries.

The operation on the file requires **translating** the human-readable name to the system-readable inode number.

This, in turn, requires **reading** the appropriate directory file.

To speed up access to directory entries, some of them are stored in memory as **dentry objects**.

Dentry objects are created for each component of the path searched by the process, e.g. if we use the **/etc/passwd** file, the kernel will create three objects, for **/**, **etc**, **passwd**, respectively.

The directory entry is represented by the **dentry structure**.

SKIP

Lockref allows mostly-lockless manipulation of a reference count while still respecting an associated lock.
u64 cmpxchg(u64 *location, u64 old, u64 new)

```
struct qstr {                /* quick string - slightly simplified */
    u32 hash;
    u32 len;                  /* name length*/
    const unsigned char * name;    /* name*/
struct dentry {              /* slightly simplified*/
    unsigned int  d_flags;
    seqcount_t  d_seq;               /* per dentry seqlock */
    struct hlist_bl_node  d_hash;   /* hash table */
    struct dentry  * d_parent;       /* parent directory dentry object */
    struct qstr  d_name;             /* name */
    struct inode   * d_inode;        /* inode */
    unsigned char  d_iname[DNAME_INLINE_LEN];  /* short name*/
    struct lockref  d_lockref;       /* per-dentry lock and refcount */
    const struct  dentry_operations  *d_op; /* dentry operations */
    struct super_block  * d_sb;     / * dentry tree root */
    struct list_head  d_lru;         /* LRU list of currently unused items */
    struct list_head  d_child;      /* list of children from the parent directory (our siblings) */
    struct list_head  d_subdirs;     /* list of our children (files and subdirectories) */
    union {
        struct list_head  d_alias;     /* list of inode aliases */
        struct rcu_head d_rcu;
  } d_u;
```

Simplified version

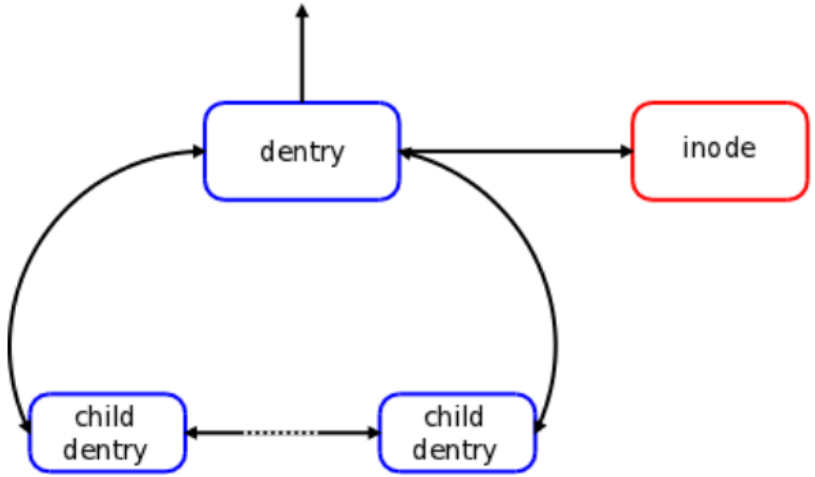
```
struct lockref { /* eight bytes */
    union {
#if USE_CMPXCHG_LOCKREF
            aligned_u64 lock_count;
#endif
            struct {
                spinlock_t lock;
                int count;
            };
    };
};
```

From 3.12

lockref, by combining the spinlock and the reference count into a single eight-byte quantity, is able to considerably reduce that cost of cache-line bouncing when tracking the lifecycle of the **reference count**.

cmpxchg(location, old, new);

Dentry structure



Dentry objects are linked into a directory tree using fields:
d_parent, **d_child**, **d_subdirs**
Each directory entry is associated with a certain inode (**d_inode** field).

The **dentry** object has no equivalent on disk, it doesn't need any field to indicate that the object has been modified.

Each object can be in one of four states:

- **free** – it does not contain any important information, it is not used by VFS (memory is managed by a slab allocator).
- **unused** – unused by the kernel, **d_lockref.count** pointer is zero, but the **d_inode** field still points to the inode.
- **used** – used by the kernel, the **d_lockref.count** pointer is positive, and the **d_inode** field points to the inode.
- **empty** (**negative**) – no inode exists for this **dentry** entry because either the inode on the disk was deleted or a **dentry** entry was created for the nonexistent file. The **d_inode** field is NULL, but the **dentry** object still exists, accelerating future **lookup** operations.

SKIP

Dentry structure

From Pathname lookup in Linux about **dentry->d_lockref**.

*The association between a **dentry** and its **inode** is fairly permanent. For example, when a file is renamed, the **dentry** and **inode** move together to the new location. When a file is **created** the dentry will initially be **negative** (i.e. **d_inode** is NULL), and will be assigned to the new inode as part of the act of creation.*

*When a file is **deleted**, this can be reflected in the cache either by setting **d_inode** to NULL, or by removing it from the hash table used to look up the name in the parent directory. If the dentry is still in use, the second option is used, as it is perfectly legal to keep using an open file after it has been deleted; having the dentry around helps. If the dentry is not otherwise in use (i.e. if the **refcount** in **d_lockref** is one), only then will **d_inode** be set to NULL. Doing it this way is more efficient for a very common case.*

So as long as a counted reference is held to a dentry, a non-NULL →d_inode value will never be changed.

SKIP

Dentry operations

The methods associated with the directory entry object are described by the **dentry_operations** structure whose address is stored in the **d_op** field. Some dentry operations:

d_revalidate – validates the directory entry object before using it when translating the file path; for most local file systems the function is not provided (the pointer is NULL), but for network file systems it is needed because the file may change on the server and the client will not know about it;

d_delete – called when deleting the last reference to a dentry object;

d_release – called when the dentry object is to be passed to the slab allocator;

d_iput – called when the dentry object becomes **empty**, meaning it loses its inode;

d_hash – hash function (called when **adding** a dentry to the **hash table**);

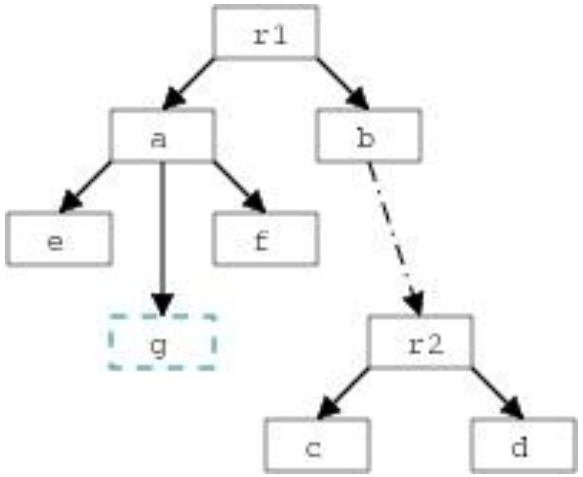
d_compare – compares the name of dentry with the name provided by qstr.

Most filesystems treat **names** as **uninterpreted strings** of **bytes** so the **default compare** and **hash** functions are the common case. A few filesystems define these to handle **case-insensitive names** but that is not the norm.

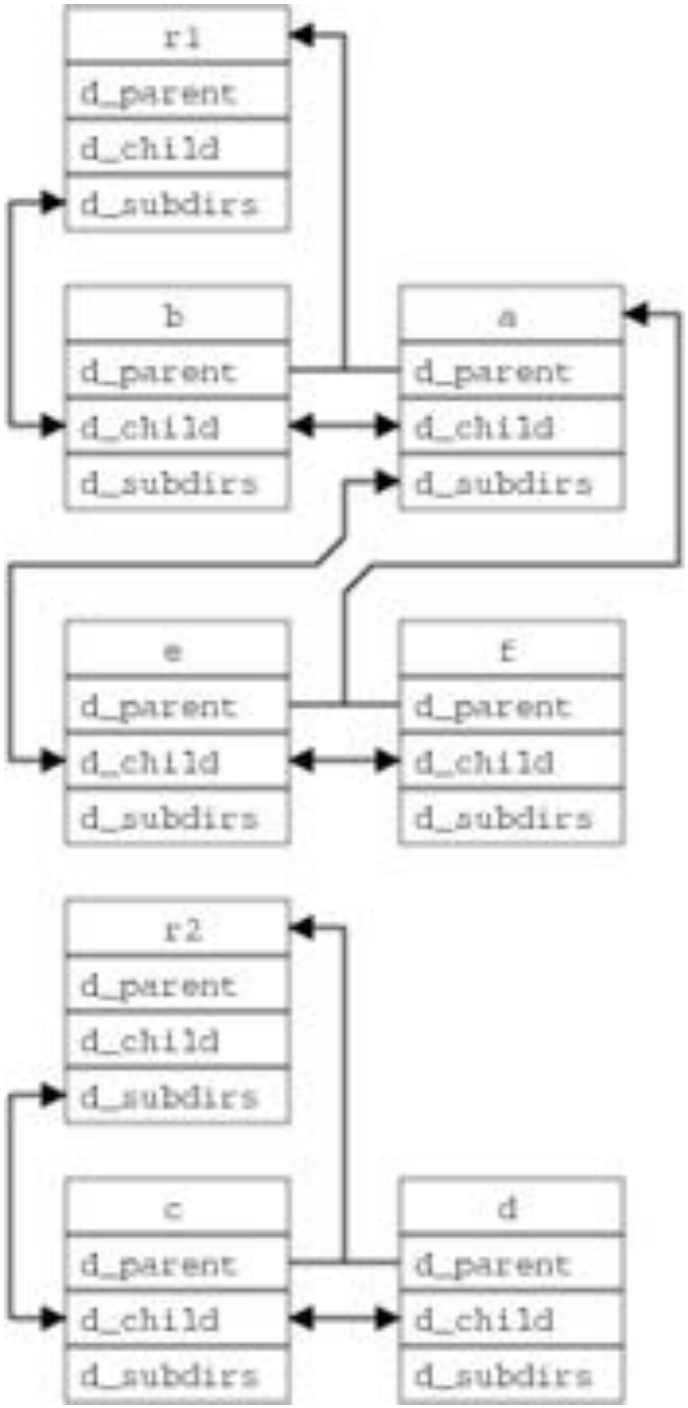
Dentry cache

The mountpoint (dentry **b**) does not reference the mounted filesystem directly. Instead, the mountpoint's **DCACHE_MOUNTED** flag is set, which influences **dcache** look up.

```
#define DCACHE_MOUNTED   0x00010000 /* is a mountpoint */

static inline bool  d_mountpoint(const struct dentry *dentry)
{
   return dentry->d_flags & DCACHE_MOUNTED;
}
```



Dcache representation of the example filesystem (source: Scaling dcache with RCU, Paul E. McKenney, 2004)



Dentry cache

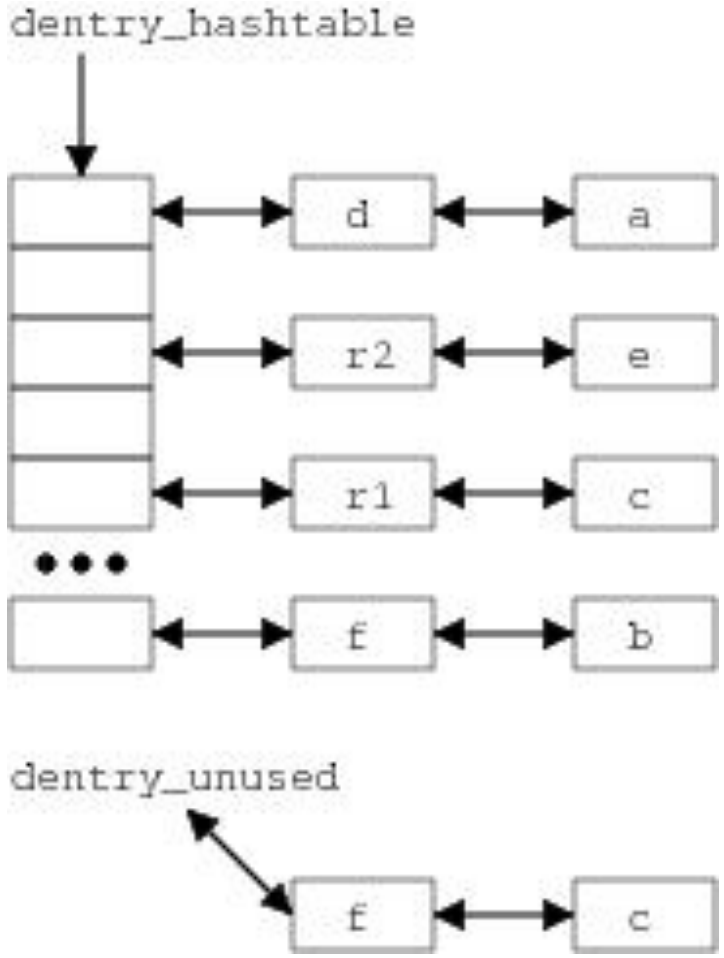
The **dentry** objects are placed in the **dentry_hashtable** (defined in **/fs/dcache.c**), whose elements are doubly-linked cyclic lists (**d_hash** field). This mechanism works for all file systems used in Linux.

__d_lookup() hashes the **parent directory's dentry pointer** and the **child's name**, searching the **dentry hash table** for the corresponding dentry.

All **unused dentries** are included in the **doubly-linked LRU list**, (**d_lru** field) the last released object is placed at the head of the list, the least recently used objects are placed near the end. When the list is to be shortened, the objects are removed from the end.

Each **used** object is inserted into the doubly-linked list (**d_alias** field) indicated by the **i_dentry** field of the **inode** (a list is needed, because the inode can be associated with several hard links).

Any dentry in the LRU list usually is in the hash table as well.



Dentry hash table
(source: Scaling dcache with RCU, Paul E. McKenney, 2004)

SKIP

Dentry cache – main functions

- **d_add** – add a dentry to its parents hash list and then calls **d_instantiate()**.
- **d_instantiate** – add a dentry to the alias hash list for the inode and update the d_inode member. The i_count member in the inode structure should be set/incremented.
- **d_delete** – delete a dentry. If there are no other open references to the dentry then the dentry is turned into a negative dentry (the **d_iput()** method is called). If there are other references, then **d_drop()** is called instead.
- **d_drop** – unhash a dentry from its parents hash list. A subsequent call to **dput()** will deallocate the dentry if its usage count drops to 0.
- **d_lookup** – look up a dentry given its parent and path name component. It looks up the child of that given name from the **dcache hash table**. If it is found, the reference count is incremented and the dentry is returned. The caller must use **dput()** to free the dentry when it finishes using it.
- **dget** – open a new handle for an existing dentry (this just increments the usage count).
- **dput** – close a handle for a dentry (decrements the usage count). If the usage count drops to 0, and the dentry is still in its parent's hash, the **d_delete** method is called to check whether it should be cached. If it should not be cached, or if the dentry is not hashed, it is deleted. Otherwise cached dentries are put into an LRU list to be reclaimed on memory shortage.

Pathname lookup

To convert the **path name** of a file to an **inode number**, the system must navigate to all intermediate directories.

The **link_path_walk()** function does it (from the file **fs/namei.c**). It is used by **open()**, **stat()** , **mkdir()**.

The input parameters :

- **name** – path name,
- **nameidata** object.

Starting point:

- **current→fs→pwd** or
- **current→fs→root**

The result of the function:

- 0 for success or a number different from 0 for error,
- completed **dentry** field in the structure **nd**.

Simplified version

```
struct path {
   struct vfsmount *mnt;
   struct dentry *dentry;
};

struct nameidata {
    struct path    path;
    struct qstr    last;
    struct path    root;
    struct inode   *inode; /* path.dentry.d_inode */
    unsigned int   flags;
    unsigned       depth;
    int            total_link_count;
    struct saved {
        struct path   link;
        const char   *name;
        unsigned     seq;
        ...
   } *stack, internal[EMBEDDED_LEVELS];
   ...
};
static int link_path_walk(const char *name,
                          struct nameidata *nd)
```

Pathname lookup

The function uses the **nameidata** data structure passed as a parameter according to the following scheme:

- (**START**) get the **inode** address corresponding to the last considered element of the path name;
- check if the process has the **permission** to execute;
- get the **next element** from the path name;
- consider specific cases of names **.** and **..**;
- search the cache directory entries for the **dentry** object corresponding to the last one (if it is not in the cache, the directory has to be loaded from the disk and a new **dentry** object created);
- consider specific cases of **mount points** and **symbolic links**;
- if it's not over then go back to (**START**).

If the current inode is the point where a **file system** was **mounted**, the **current inode** changes to the **root inode** of the mounted file system.

Symbolic links (files containing path names of other files) require special treatment. To prevent a function from looping, symbolic links are **counted** when searching names and signals an error when their upper limit is exceeded.

Path walking synchronisation

While one process is looking up a pathname, another might be making changes that affect that lookup. E.g. if "a/b" were renamed to "a/c/b" while another process were looking up "a/b/..", that process might successfully resolve on "a/c". Most races are much more subtle, and a big part of the task of pathname lookup is to prevent them from having damaging effects.

Prior to 2.5.10, **dcache_lock** was acquired in **d_lookup** (dcache hash lookup) and thus in every component during path look-up.

Since 2.5.10 onwards, **fast-walk algorithm** changed this by holding the **dcache_lock** at the beginning and walking as many cached path component dentries as possible. This significantly decreases the number of acquisition of **dcache_lock**. However it also increases the lock hold time significantly and affects performance in large SMP systems.

Since 2.5.62 , dcache has been using a new locking model that uses **RCU** to make dcache look-up lock-free.

All the above algorithms required **taking a lock and reference count on the dentry** that was looked up, so that may be used as the basis for walking the next path element. This is inefficient and unscalable.

Since 2.6.38, **RCU** is used to make a significant part of the entire path walk (including dcache look-up) completely "store-free" (no locks, atomics, or even stores into cachelines of common dentries). This is known as **RCU-walk** path walking (in opposite to **REF-walk**).

Additional reading

- Dentry negativity, Jonathan Corbet, March2020.
- Documentation/filesystems/path-lookup.rst – Pathname lookup in Linux, **Neil Brown** with help from Al Viro and Jon Corbet. Based on:
  - Pathname lookup in Linux, Neil Brown, June 2015.
  - RCU-walk: faster pathname lookup in Linux, Neil Brown, July 2015.
  - A walk among the symlinks, Neil Brown, July 2015.
- Documentation/filesystems/path-lookup.txt – Path walking and name lookup locking.
- Case-insensitive filesystem lookups, Jake Edge, May 2018.
- Dcache scalability and RCU-walk, Jonathan Corbet, December 2010.
- Scaling dcache with RCU, Paul E. McKenney, 2004.

SKIP

Inodes in memory – ilookup() function

The **ilookup()** function is responsible for providing the process with an **inode** in memory. It searches for the inode **ino** in the **inode cache**, and if the inode is in the cache, the inode is returned with an incremented reference count. The inode comes from the file system represented by the super block **sb**.

```
struct inode  *ilookup(struct super_block *sb, unsigned long ino)
{
        struct hlist_head *head = inode_hashtable + hash(sb, ino);
        struct inode *inode;

        spin_lock(&inode_hash_lock);
        inode = find_inode_fast(sb, head, ino);
        spin_unlock(&inode_hash_lock);

        if (inode)
                wait_on_inode(inode);
                        ...
        return inode;
}
```

Simplified version

SKIP

Inodes in memory – iput() function

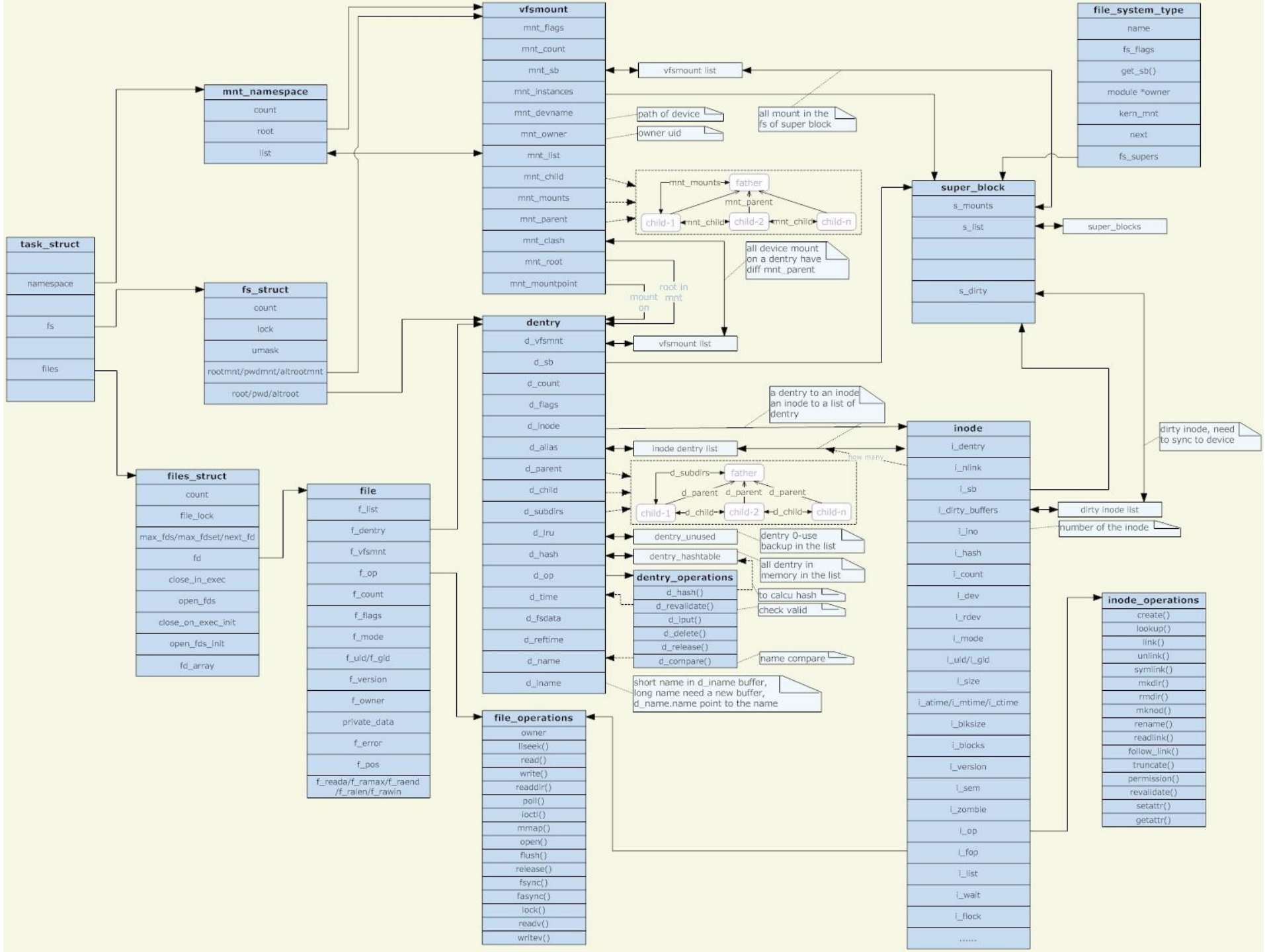
The **iput()** function is used to **free** an **inode** in memory. The file open count (**i_count**) **decreases**, and if it is still greater than zero, the function ends.

Otherwise:

- release processes waiting for a free inode,
- if the inode represents a **pipe** – release the associated memory pages,
- if the inode had the **dirty** attribute set – save it to disk,
- increase the free inode count.

```
void   iput(struct inode *inode) {
   if (!inode)
       return;
   BUG_ON(inode->i_state & I_CLEAR);
retry:
   if (atomic_dec_and_lock(&inode->i_count, &inode->i_lock)) {
       if (inode->i_nlink && (inode->i_state & I_DIRTY_TIME)) {
           atomic_inc(&inode->i_count);
           spin_unlock(&inode->i_lock);
           trace_writeback_lazytime_iput(inode);
           mark_inode_dirty_sync(inode);
           goto retry;
       }
       iput_final(inode);
   }
}
```

Simplified version


vfsmount
mnt_flags
mnt_count
mnt_sb
mnt_instances
mnt_devname
mnt_owner
mnt_list
mnt_child
mnt_mounts
mnt_parent
mnt_clash
mnt_root
mnt_mountpoint
vfsmount list
path of device
owner uid
all mount in the fs of super block
mnt_mounts
father
mnt_parent
child-1
mnt_child
child-2
mnt_child
child-n
all device mount on a dentry have diff mnt_parent
root in mount mnt on
file_system_type
name
fs_flags
get_sb()
module *owner
kern_mnt
next
fs_supers
super_block
s_mounts
s_list
super_blocks
s_dirty
mnt_namespace
count
root
list
task_struct
namespace
fs
files
fs_struct
count
lock
umask
rootmnt/pwdmnt/altrootmnt
root/pwd/altroot
dentry
d_vfsmnt
d_sb
d_count
d_flags
d_inode
d_alias
d_parent
d_child
d_subdirs
d_lru
d_hash
d_op
d_time
d_fsdata
d_reftime
d_name
d_iname
vfsmount list
a dentry to an inode an inode to a list of dentry
inode dentry list
how many
d_subdirs
father
d_parent
d_parent
d_parent
child-1
d_child
child-2
d_child
child-n
dentry_unused
dentry 0-use backup in the list
dentry_hashtable
all dentry in memory in the list
dentry_operations
d_hash()
d_revalidate()
d_iput()
d_delete()
d_release()
d_compare()
to calcu hash
check valid
name compare
short name in d_iname buffer, long name need a new buffer, d_name.name point to the name
inode
i_dentry
i_nlink
i_sb
i_dirty_buffers
i_ino
i_hash
i_count
i_dev
i_rdev
i_mode
i_uid/i_gid
i_size
i_atime/i_mtime/i_ctime
i_blksize
i_blocks
i_version
i_sem
i_zombie
i_op
i_fop
i_list
i_wait
i_flock
.....
dirty inode, need to sync to device
dirty inode list
number of the inode
inode_operations
create()
lookup()
link()
unlink()
symlink()
mkdir()
rmdir()
mknod()
rename()
readlink()
follow_link()
truncate()
permission()
revalidate()
setattr()
getattr()
files_struct
count
file_lock
max_fds/max_fdset/next_fd
fd
close_in_exec
open_fds
close_on_exec_init
open_fds_init
fd_array
file
f_list
f_dentry
f_vfsmnt
f_op
f_count
f_flags
f_mode
f_uid/f_gid
f_version
f_owner
private_data
f_error
f_pos
f_reada/f_ramax/f_raend /f_ralen/f_rawin
file_operations
owner
llseek()
read()
write()
readdir()
poll()
ioctl()
mmap()
open()
flush()
release()
fsync()
fasync()
lock()
readv()
writev()

Additional reading

- Documentation/filesystems/vfs.rst – VFS data structures.
- Introducing lockrefs, Jonathan Corbet, September 2013.
- Linux VFS, Kaustubh R. Joshi, 2013.
- Anatomy of the Linux virtual file system switch, M. Tim Jones.