The Linux Kernel API

The Linux Kernel API

Table of Contents

Chapter 1. The Linux VFS

The Directory Cache

d_invalidate

Name `d_invalidate` — invalidate a dentry

Synopsis

```
int d_invalidate (struct dentry * dentry);
```

Arguments

`dentry`

dentry to invalidate

Description

Try to invalidate the dentry if it turns out to be possible. If there are other dentries that can be reached through this one we can’t delete it and we return -EBUSY. On success we return 0.

d_find_alias

Name `d_find_alias` — grab a hashed alias of inode

Synopsis

```
struct dentry * d_find_alias (struct inode * inode);
```

Arguments

`inode`

inode in question

Description

If inode has a hashed alias - acquire the reference to alias and return it. Otherwise return NULL. Notice that if inode is a directory there can be only one alias and it can be unhashed only if it has no children.

prune_dcache

Name `prune_dcache` — shrink the dcache

Synopsis

```
void prune_dcache (int count);
```

Arguments

`count`

number of entries to try and free

Description

Shrink the dcache. This is done when we need more memory, or simply when we need to unmount something (at which point we need to unuse all dentries).

This function may fail to free any resources if all the dentries are in use.

shrink_dcache_sb

Name `shrink_dcache_sb` — shrink dcache for a superblock

Synopsis

```
void shrink_dcache_sb (struct super_block * sb);
```

Arguments

`sb`

superblock

Description

Shrink the dcache for the specified super block. This is used to free the dcache before unmounting a file system

have_submounts

Name `have_submounts` — check for mounts over a dentry

Synopsis

```
int have_submounts (struct dentry * parent);
```

Arguments

`parent`

dentry to check.

Description

Return true if the parent or its subdirectories contain a mount point

shrink_dcache_parent

Name `shrink_dcache_parent` — prune dcache

Synopsis

```
void shrink_dcache_parent (struct dentry * parent);
```

Arguments

`parent`

parent of entries to prune

Description

Prune the dcache to remove unused children of the parent dentry.

d_alloc

Name `d_alloc` — allocate a dcache entry

Synopsis

```
struct dentry * d_alloc (struct dentry * parent, const struct qstr * name);
```

Arguments

`parent`
: parent of entry to allocate

`name`
: qstr of the name

Description

Allocates a dentry. It returns `NULL` if there is insufficient memory available. On a success the dentry is returned. The name passed in is copied and the copy passed in may be reused after this call.

d_instantiate

Name `d_instantiate` — fill in inode information for a dentry

Synopsis

```
void d_instantiate (struct dentry * entry, struct inode * inode);
```

Arguments

`entry`

dentry to complete

`inode`

inode to attach to this dentry

Description

Fill in inode information in the entry.

This turns negative dentries into productive full members of society.

NOTE! This assumes that the inode count has been incremented (or otherwise set) by the caller to indicate that it is now in use by the dcache.

d_alloc_root

Name `d_alloc_root` — allocate root dentry

Synopsis

```
struct dentry * d_alloc_root (struct inode * root_inode);
```

Arguments

`root_inode`

inode to allocate the root for

Description

Allocate a root (“/”) dentry for the inode given. The inode is instantiated and returned. `NULL` is returned if there is insufficient memory or the inode passed is `NULL`.

d_lookup

Name `d_lookup` — search for a dentry

Synopsis

```
struct dentry * d_lookup (struct dentry * parent, struct qstr * name);
```

Arguments

`parent`

parent dentry

`name`

qstr of name we wish to find

Description

Searches the children of the parent dentry for the name in question. If the dentry is found its reference count is incremented and the dentry is returned. The caller must use d_put to free the entry when it has finished using it. `NULL` is returned on failure.

d_validate

Name `d_validate` — verify dentry provided from insecure source

Synopsis

`int` **`d_validate`** `(struct dentry *` *`dentry`*`, struct dentry *` *`dparent`*`, unsigned int` *`hash`*`, unsigned int` *`len`*`);`

Arguments

`dentry`

The dentry alleged to be valid

`dparent`

The parent dentry

`hash`

Hash of the dentry

`len`

Length of the name

Description

An insecure source has sent us a dentry, here we verify it. This is used by ncpfs in its readdir implementation. Zero is returned in the dentry is invalid.

NOTE

This function does _not_ dereference the pointers before we have validated them. We can test the pointer values, but we must not actually use them until we have found a valid copy of the pointer in kernel space..

d_delete

Name `d_delete` — delete a dentry

Synopsis

```
void d_delete (struct dentry * dentry);
```

Arguments

dentry

The dentry to delete

Description

Turn the dentry into a negative dentry if possible, otherwise remove it from the hash queues so it can be deleted later

d_rehash

Name `d_rehash` — add an entry back to the hash

Synopsis

```
void d_rehash (struct dentry * entry);
```

Arguments

entry

dentry to add to the hash

Description

Adds a dentry to the hash according to its name.

d_move

Name `d_move` — move a dentry

Synopsis

```
void d_move (struct dentry * dentry, struct dentry * target);
```

Arguments

`dentry`
: entry to move

`target`
: new dentry

Description

Update the dcache to reflect the move of a file name. Negative dcache entries should not be moved in this way.

__d_path

Name `__d_path` — return the path of a dentry

Synopsis

```
char * __d_path (struct dentry * dentry, struct vfsmount * vfsmnt, struct
dentry * root, struct vfsmount * rootmnt, char * buffer, int buflen);
```

Arguments

dentry

dentry to report

vfsmnt

– undescribed –

root

– undescribed –

rootmnt

– undescribed –

buffer

buffer to return value in

buflen

buffer length

Description

Convert a dentry into an ASCII path name. If the entry has been deleted the string “ (deleted)” is appended. Note that this is ambiguous. Returns the buffer.

“buflen” should be `PAGE_SIZE` or more.

is_subdir

Name

`is_subdir` — is new dentry a subdirectory of old_dentry

Synopsis

```
int is_subdir (struct dentry * new_dentry, struct dentry * old_dentry);
```

Arguments

`new_dentry`
: new dentry

`old_dentry`
: old dentry

Description

Returns 1 if new_dentry is a subdirectory of the parent (at any depth). Returns 0 otherwise.

find_inode_number

Name `find_inode_number` — check for dentry with name

Synopsis

```
ino_t find_inode_number (struct dentry * dir, struct qstr * name);
```

Arguments

`dir`
: directory to check

`name`
: Name to find.

Description

Check whether a dentry already exists for the given name, and return the inode number if it has an inode. Otherwise 0 is returned.

This routine is used to post-process directory listings for filesystems using synthetic inode numbers, and is necessary to keep `getcwd` working.

d_drop

Name `d_drop` — drop a dentry

Synopsis

```
void d_drop (struct dentry * dentry);
```

Arguments

`dentry`

dentry to drop

Description

`d_drop` unhashes the entry from the parent dentry hashes, so that it won't be found through a VFS lookup any more. Note that this is different from deleting the dentry - d_delete will try to mark the dentry negative if possible, giving a successful _negative_ lookup, while d_drop will just make the cache lookup fail.

`d_drop` is used mainly for stuff that wants to invalidate a dentry for some reason (NFS timeouts or autofs deletes).

d_add

Name `d_add` — add dentry to hash queues

Synopsis

```
void d_add (struct dentry * entry, struct inode * inode);
```

Arguments

`entry`

dentry to add

`inode`

The inode to attach to this dentry

Description

This adds the entry to the hash queues and initializes *`inode`*. The entry was actually filled in earlier during `d_alloc`.

dget

Name `dget` — get a reference to a dentry

Synopsis

```
struct dentry * dget (struct dentry * dentry);
```

Arguments

`dentry`

dentry to get a reference to

Description

Given a dentry or `NULL` pointer increment the reference count if appropriate and return the dentry. A dentry will not be destroyed when it has references.

d_unhashed

Name `d_unhashed` — is dentry hashed

Synopsis

```
int d_unhashed (struct dentry * dentry);
```

Arguments

`dentry`

entry to check

Description

Returns true if the dentry passed is not currently hashed.

Inode Handling

__mark_inode_dirty

Name `__mark_inode_dirty` — internal function

Synopsis

```
void __mark_inode_dirty (struct inode * inode);
```

Arguments

`inode`

inode to mark

Description

Mark an inode as dirty. Callers should use mark_inode_dirty.

write_inode_now

Name `write_inode_now` — write an inode to disk

Synopsis

```
void write_inode_now (struct inode * inode);
```

Arguments

`inode`

inode to write to disk

Description

This function commits an inode to disk immediately if it is dirty. This is primarily needed by knfsd.

clear_inode

Name `clear_inode` — clear an inode

Synopsis

```
void clear_inode (struct inode * inode);
```

Arguments

`inode`

inode to clear

Description

This is called by the filesystem to tell us that the inode is no longer useful. We just terminate it with extreme prejudice.

invalidate_inodes

Name `invalidate_inodes` — discard the inodes on a device

Synopsis

```
int invalidate_inodes (struct super_block * sb);
```

Arguments

sb

superblock

Description

Discard all of the inodes for a given superblock. If the discard fails because there are busy inodes then a non zero value is returned. If the discard is successful all the inodes have been discarded.

get_empty_inode

Name `get_empty_inode` — obtain an inode

Synopsis

```
struct inode * get_empty_inode ( void);
```

Arguments

`void`

no arguments

Description

This is called by things like the networking layer etc that want to get an inode without any inode number, or filesystems that allocate new inodes with no pre-existing information.

On a successful return the inode pointer is returned. On a failure a `NULL` pointer is returned. The returned inode is not on any superblock lists.

iunique

Name `iunique` — get a unique inode number

Synopsis

```
ino_t iunique (struct super_block * sb, ino_t max_reserved);
```

Arguments

`sb`

superblock

`max_reserved`

highest reserved inode number

Description

Obtain an inode number that is unique on the system for a given superblock. This is used by file systems that have no natural permanent inode numbering system. An inode number is returned that is higher than the reserved limit but unique.

BUGS

With a large number of inodes live on the file system this function currently becomes quite slow.

insert_inode_hash

Name `insert_inode_hash` — hash an inode

Synopsis

```
void insert_inode_hash (struct inode * inode);
```

Arguments

`inode`

unhashed inode

Description

Add an inode to the inode hash for this superblock. If the inode has no superblock it is added to a separate anonymous chain.

remove_inode_hash

Name `remove_inode_hash` — remove an inode from the hash

Synopsis

```
void remove_inode_hash (struct inode * inode);
```

Arguments

`inode`

inode to unhash

Description

Remove an inode from the superblock or anonymous hash.

iput

Name `iput` — put an inode

Synopsis

```
void iput (struct inode * inode);
```

Arguments

`inode`

inode to put

Description

Puts an inode, dropping its usage count. If the inode use count hits zero the inode is also then freed and may be destroyed.

bmap

Name `bmap` — find a block number in a file

Synopsis

```
int bmap (struct inode * inode, int block);
```

Arguments

`inode`

inode of file

`block`

block to find

Description

Returns the block number on the device holding the inode that is the disk block number for the block of the file requested. That is, asked for block 4 of inode 1 the function will return the disk block relative to the disk start that holds that block of the file.

update_atime

Name `update_atime` — update the access time

Synopsis

```
void update_atime (struct inode * inode);
```

Arguments

`inode`

inode accessed

Description

Update the accessed time on an inode and mark it for writeback. This function automatically handles read only file systems and media, as well as the “noatime” flag and inode specific “noatime” markers.

make_bad_inode

Name `make_bad_inode` — mark an inode bad due to an I/O error

Synopsis

```
void make_bad_inode (struct inode * inode);
```

Arguments

`inode`

Inode to mark bad

Description

When an inode cannot be read due to a media or remote network failure this function makes the inode “bad” and causes I/O operations on it to fail from this point on.

is_bad_inode

Name `is_bad_inode` — is an inode errored

Synopsis

```
int is_bad_inode (struct inode * inode);
```

Arguments

`inode`

inode to test

Description

Returns true if the inode in question has been marked as bad.

Registration and Superblocks

register_filesystem

Name `register_filesystem` — register a new filesystem

Synopsis

```
int register_filesystem (struct file_system_type * fs);
```

Arguments

fs

the file system structure

Description

Adds the file system passed to the list of file systems the kernel is aware of for mount and other syscalls. Returns 0 on success, or a negative errno code on an error.

The &struct file_system_type that is passed is linked into the kernel structures and must not be freed until the file system has been unregistered.

unregister_filesystem

Name

`unregister_filesystem` — unregister a file system

Synopsis

```
int unregister_filesystem (struct file_system_type * fs);
```

Arguments

fs

filesystem to unregister

Description

Remove a file system that was previously successfully registered with the kernel. An error is returned if the file system is not found. Zero is returned on a success.

Once this function has returned the &struct file_system_type structure may be freed or reused.

__wait_on_super

Name `__wait_on_super` — wait on a superblock

Synopsis

```
void __wait_on_super (struct super_block * sb);
```

Arguments

`sb`

superblock to wait on

Description

Waits for a superblock to become unlocked and then returns. It does not take the lock. This is an internal function. See `wait_on_super`.

get_super

Name `get_super` — get the superblock of a device

Synopsis

```
struct super_block * get_super (kdev_t dev);
```

Arguments

`dev`

device to get the superblock for

Description

Scans the superblock list and finds the superblock of the file system mounted on the device given. `NULL` is returned if no match is found.

get_empty_super

Name `get_empty_super` — find empty superblocks

Synopsis

```
struct super_block * get_empty_super ( void);
```

Arguments

`void`

no arguments

Description

Find a superblock with no device assigned. A free superblock is found and returned. If neccessary new superblocks are allocated. `NULL` is returned if there are insufficient resources to complete the request.

Chapter 2. Linux Networking

Socket Buffer Functions

skb_queue_empty

Name `skb_queue_empty` — check if a queue is empty

Synopsis

```
int skb_queue_empty (struct sk_buff_head * list);
```

Arguments

`list`

queue head

Description

Returns true if the queue is empty, false otherwise.

skb_get

Name `skb_get` — reference buffer

Synopsis

```
struct sk_buff * skb_get (struct sk_buff * skb);
```

Arguments

`skb`

buffer to reference

Description

Makes another reference to a socket buffer and returns a pointer to the buffer.

kfree_skb

Name `kfree_skb` — free an sk_buff

Synopsis

```
void kfree_skb (struct sk_buff * skb);
```

Arguments

`skb`

buffer to free

Description

Drop a reference to the buffer and free it if the usage count has hit zero.

skb_cloned

Name `skb_cloned` — is the buffer a clone

Synopsis

```
int skb_cloned (struct sk_buff * skb);
```

Arguments

`skb`

buffer to check

Description

Returns true if the buffer was generated with `skb_clone` and is one of multiple shared copies of the buffer. Cloned buffers are shared data so must not be written to under normal circumstances.

skb_shared

Name `skb_shared` — is the buffer shared

Synopsis

```
int skb_shared (struct sk_buff * skb);
```

Arguments

skb

buffer to check

Description

Returns true if more than one person has a reference to this buffer.

skb_unshare

Name

`skb_unshare` — make a copy of a shared buffer

Synopsis

```
struct sk_buff * skb_unshare (struct sk_buff * skb, int pri);
```

Arguments

skb

buffer to check

pri

priority for memory allocation

Description

If the socket buffer is a clone then this function creates a new copy of the data, drops a reference count on the old copy and returns the new copy with the reference count at 1. If the buffer is not a clone the original buffer is returned. When called with a spinlock held or from interrupt state *pri* must be `GFP_ATOMIC`

`NULL` is returned on a memory allocation failure.

skb_peek

Name `skb_peek` —

Synopsis

```
struct sk_buff * skb_peek (struct sk_buff_head * list_);
```

Arguments

`list_`

list to peek at

Description

Peek an &sk_buff. Unlike most other operations you _MUST_ be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns `NULL` for an empty list or a pointer to the head element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

skb_peek_tail

Name `skb_peek_tail` —

Synopsis

```
struct sk_buff * skb_peek_tail (struct sk_buff_head * list_);
```

Arguments

`list_`

list to peek at

Description

Peek an &sk_buff. Unlike most other operations you _MUST_ be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns `NULL` for an empty list or a pointer to the tail element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

skb_queue_len

Name `skb_queue_len` — get queue length

Synopsis

```
__u32 skb_queue_len (struct sk_buff_head * list_);
```

Arguments

`list_`

list to measure

Description

Return the length of an &sk_buff queue.

__skb_queue_head

Name `__skb_queue_head` — queue a buffer at the list head

Synopsis

```
void __skb_queue_head (struct sk_buff_head * list, struct sk_buff * newsk);
```

Arguments

`list`

list to use

`newsk`

buffer to queue

Description

Queue a buffer at the start of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

skb_queue_head

Name `skb_queue_head` — queue a buffer at the list head

Synopsis

```
void skb_queue_head (struct sk_buff_head * list, struct sk_buff * newsk);
```

Arguments

`list`

list to use

`newsk`

buffer to queue

Description

Queue a buffer at the start of the list. This function takes the list lock and can be used safely with other locking &sk_buff functions safely.

A buffer cannot be placed on two lists at the same time.

__skb_queue_tail

Name `__skb_queue_tail` — queue a buffer at the list tail

Synopsis

```
void __skb_queue_tail (struct sk_buff_head * list, struct sk_buff * newsk);
```

Arguments

`list`

list to use

`newsk`

buffer to queue

Description

Queue a buffer at the end of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

skb_queue_tail

Name `skb_queue_tail` — queue a buffer at the list tail

Synopsis

```
void skb_queue_tail (struct sk_buff_head * list, struct sk_buff * newsk);
```

Arguments

`list`

list to use

`newsk`

buffer to queue

Description

Queue a buffer at the tail of the list. This function takes the list lock and can be used safely with other locking &sk_buff functions safely.

A buffer cannot be placed on two lists at the same time.

__skb_dequeue

Name `__skb_dequeue` — remove from the head of the queue

Synopsis

```
struct sk_buff * __skb_dequeue (struct sk_buff_head * list);
```

Arguments

`list`

list to dequeue from

Description

Remove the head of the list. This function does not take any locks so must be used with appropriate locks held only. The head item is returned or `NULL` if the list is empty.

skb_dequeue

Name `skb_dequeue` — remove from the head of the queue

Synopsis

```
struct sk_buff * skb_dequeue (struct sk_buff_head * list);
```

Arguments

list

list to dequeue from

Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The head item is returned or `NULL` if the list is empty.

skb_insert

Name `skb_insert` — insert a buffer

Synopsis

```
void skb_insert (struct sk_buff * old, struct sk_buff * newsk);
```

Arguments

old

buffer to insert before

newsk

buffer to insert

Description

Place a packet before a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls A buffer cannot be placed on two lists at the same time.

skb_append

Name `skb_append` — append a buffer

Synopsis

void **skb_append** (struct sk_buff * *old*, struct sk_buff * *newsk*);

Arguments

old

buffer to insert after

newsk

buffer to insert

Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

skb_unlink

Name `skb_unlink` — remove a buffer from a list

Synopsis

void **skb_unlink** (struct sk_buff * *skb*);

Arguments

`skb`

buffer to remove

Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls

Works even without knowing the list it is sitting on, which can be handy at times. It also means that THE LIST MUST EXIST when you unlink. Thus a list must have its contents unlinked before it is destroyed.

__skb_dequeue_tail

Name `__skb_dequeue_tail` — remove from the tail of the queue

Synopsis

```
struct sk_buff * __skb_dequeue_tail (struct sk_buff_head * list);
```

Arguments

`list`

list to dequeue from

Description

Remove the tail of the list. This function does not take any locks so must be used with appropriate locks held only. The tail item is returned or `NULL` if the list is empty.

skb_dequeue_tail

Name `skb_dequeue_tail` — remove from the head of the queue

Synopsis

```
struct sk_buff * skb_dequeue_tail (struct sk_buff_head * list);
```

Arguments

`list`

list to dequeue from

Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The tail item is returned or `NULL` if the list is empty.

skb_put

Name `skb_put` — add data to a buffer

Synopsis

```
unsigned char * skb_put (struct sk_buff * skb, unsigned int len);
```

Arguments

`skb`

buffer to use

`len`

amount of data to add

Description

This function extends the used data area of the buffer. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

skb_push

Name

`skb_push` — add data to the start of a buffer

Synopsis

```
unsigned char * skb_push (struct sk_buff * skb, unsigned int len);
```

Arguments

`skb`

buffer to use

`len`

amount of data to add

Description

This function extends the used data area of the buffer at the buffer start. If this would exceed the total buffer headroom the kernel will panic. A pointer to the first byte of the extra data is returned.

skb_pull

Name `skb_pull` — remove data from the start of a buffer

Synopsis

```
unsigned char * skb_pull (struct sk_buff * skb, unsigned int len);
```

Arguments

`skb`

buffer to use

`len`

amount of data to remove

Description

This function removes data from the start of a buffer, returning the memory to the headroom. A pointer to the next data in the buffer is returned. Once the data has been pulled future pushes will overwrite the old data.

skb_headroom

Name `skb_headroom` — bytes at buffer head

Synopsis

```
int skb_headroom (const struct sk_buff * skb);
```

Arguments

`skb`

buffer to check

Description

Return the number of bytes of free space at the head of an &sk_buff.

skb_tailroom

Name `skb_tailroom` — bytes at buffer end

Synopsis

```
int skb_tailroom (const struct sk_buff * skb);
```

Arguments

`skb`

buffer to check

Description

Return the number of bytes of free space at the tail of an sk_buff

skb_reserve

Name `skb_reserve` — adjust headroom

Synopsis

```
void skb_reserve (struct sk_buff * skb, unsigned int len);
```

Arguments

`skb`

buffer to alter

`len`

bytes to move

Description

Increase the headroom of an empty &sk_buff by reducing the tail room. This is only allowed for an empty buffer.

skb_trim

Name `skb_trim` — remove end from a buffer

Synopsis

```
void skb_trim (struct sk_buff * skb, unsigned int len);
```

Arguments

`skb`

buffer to alter

`len`

new length

Description

Cut the length of a buffer down by removing data from the tail. If the buffer is already under the length specified it is not modified.

skb_orphan

Name `skb_orphan` — orphan a buffer

Synopsis

```
void skb_orphan (struct sk_buff * skb);
```

Arguments

`skb`

buffer to orphan

Description

If a buffer currently has an owner then we call the owner's destructor function and make the *`skb`* unowned. The buffer continues to exist but is no longer charged to its former owner.

skb_queue_purge

Name `skb_queue_purge` — empty a list

Synopsis

```
void skb_queue_purge (struct sk_buff_head * list);
```

Arguments

`list`

list to empty

Description

Delete all buffers on an &sk_buff list. Each buffer is removed from the list and one reference dropped. This function takes the list lock and is atomic with respect to other list locking functions.

__skb_queue_purge

Name `__skb_queue_purge` — empty a list

Synopsis

```
void __skb_queue_purge (struct sk_buff_head * list);
```

Arguments

`list`

list to empty

Description

Delete all buffers on an &sk_buff list. Each buffer is removed from the list and one reference dropped. This function does not take the list lock and the caller must hold the relevant locks to use it.

dev_alloc_skb

Name `dev_alloc_skb` — allocate an skbuff for sending

Synopsis

```
struct sk_buff * dev_alloc_skb (unsigned int length);
```

Arguments

`length`

length to allocate

Description

Allocate a new &sk_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

`NULL` is returned in there is no free memory. Although this function allocates memory it can be called from an interrupt.

skb_cow

Name `skb_cow` — copy a buffer if need be

Synopsis

```
struct sk_buff * skb_cow (struct sk_buff * skb, unsigned int headroom);
```

Arguments

`skb`
: buffer to copy

`headroom`
: needed headroom

Description

If the buffer passed lacks sufficient headroom or is a clone then it is copied and the additional headroom made available. If there is no free memory `NULL` is returned. The new buffer is returned if a copy was made (and the old one dropped a reference). The existing buffer is returned otherwise.

This function primarily exists to avoid making two copies when making a writable copy of a buffer and then growing the headroom.

skb_over_panic

Name `skb_over_panic` — private function

Synopsis

```
void skb_over_panic (struct sk_buff * skb, int sz, void * here);
```

Arguments

`skb`

buffer

`sz`

size

`here`

address

Description

Out of line support code for `skb_put`. Not user callable.

skb_under_panic

Name `skb_under_panic` — private function

Synopsis

```
void skb_under_panic (struct sk_buff * skb, int sz, void * here);
```

Arguments

`skb`

buffer

`sz`

size

`here`

address

Description

Out of line support code for `skb_push`. Not user callable.

alloc_skb

Name `alloc_skb` — allocate a network buffer

Synopsis

```
struct sk_buff * alloc_skb (unsigned int size, int gfp_mask);
```

Arguments

`size`

size to allocate

`gfp_mask`

allocation mask

Description

Allocate a new &sk_buff. The returned buffer has no headroom and a tail room of size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is `NULL`.

Buffers may only be allocated from interrupts using a *`gfp_mask`* of `GFP_ATOMIC`.

__kfree_skb

Name `__kfree_skb` — private function

Synopsis

```
void __kfree_skb (struct sk_buff * skb);
```

Arguments

`skb`

buffer

Description

Free an sk_buff. Release anything attached to the buffer. Clean the state. This is an internal helper function. Users should always call kfree_skb

skb_clone

Name `skb_clone` — duplicate an sk_buff

Synopsis

```
struct sk_buff * skb_clone (struct sk_buff * skb, int gfp_mask);
```

Arguments

skb

buffer to clone

gfp_mask

allocation priority

Description

Duplicate an &sk_buff. The new one is not owned by a socket. Both copies share the same packet data but not structure. The new buffer has a reference count of 1. If the allocation fails the function returns `NULL` otherwise the new buffer is returned.

If this function is called from an interrupt `gfp_mask` must be `GFP_ATOMIC`.

skb_copy

Name `skb_copy` — copy an sk_buff

Synopsis

```
struct sk_buff * skb_copy (const struct sk_buff * skb, int gfp_mask);
```

Arguments

`skb`

buffer to copy

`gfp_mask`

allocation priority

Description

Make a copy of both an &sk_buff and its data. This is used when the caller wishes to modify the data and needs a private copy of the data to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass `GFP_ATOMIC` as the allocation priority if this function is called from an interrupt.

skb_copy_expand

Name `skb_copy_expand` — copy and expand sk_buff

Synopsis

```
struct sk_buff * skb_copy_expand (const struct sk_buff * skb, int
newheadroom, int newtailroom, int gfp_mask);
```

Arguments

`skb`

buffer to copy

`newheadroom`

new free bytes at head

`newtailroom`

new free bytes at tail

`gfp_mask`

allocation priority

Description

Make a copy of both an &sk_buff and its data and while doing so allocate additional space.

This is used when the caller wishes to modify the data and needs a private copy of the data to alter as well as more space for new fields. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass `GFP_ATOMIC` as the allocation priority if this function is called from an interrupt.

Socket Filter

sk_run_filter

Name `sk_run_filter` — run a filter on a socket

Synopsis

```
int sk_run_filter (struct sk_buff * skb, struct sock_filter * filter, int
flen);
```

Arguments

`skb`

buffer to run the filter on

`filter`

filter to apply

`flen`

length of filter

Description

Decode and apply filter instructions to the skb->data. Return length to keep, 0 for none. skb is the data we are filtering, filter is the array of filter instructions, and len is the number of filter blocks in the array.

Chapter 3. Network device support

Driver Support

init_etherdev

Name `init_etherdev` — Register ethernet device

Synopsis

```
struct net_device * init_etherdev (struct net_device * dev, int sizeof_priv);
```

Arguments

`dev`

An ethernet device structure to be filled in, or `NULL` if a new struct should be allocated.

`sizeof_priv`

Size of additional driver-private structure to be allocated for this ethernet device

Description

Fill in the fields of the device structure with ethernet-generic values.

If no device structure is passed, a new one is constructed, complete with a private data area of size *`sizeof_priv`*. A 32-byte (not bit) alignment is enforced for this private data area.

If an empty string area is passed as dev->name, or a new structure is made, a new name string is constructed.

dev_add_pack

Name `dev_add_pack` — add packet handler

Synopsis

```
void dev_add_pack (struct packet_type * pt);
```

Arguments

`pt`
: packet type declaration

Description

Add a protocol handler to the networking stack. The passed &packet_type is linked into kernel lists and may not be freed until it has been removed from the kernel lists.

dev_remove_pack

Name `dev_remove_pack` — remove packet handler

Synopsis

```
void dev_remove_pack (struct packet_type * pt);
```

Arguments

`pt`

packet type declaration

Description

Remove a protocol handler that was previously added to the kernel protocol handlers by `dev_add_pack`. The passed &packet_type is removed from the kernel lists and can be freed or reused once this function returns.

__dev_get_by_name

Name `__dev_get_by_name` — find a device by its name

Synopsis

```
struct net_device * __dev_get_by_name (const char * name);
```

Arguments

`name`

name to find

Description

Find an interface by name. Must be called under RTNL semaphore or *`dev_base_lock`*. If the name is found a pointer to the device is returned. If the name is not found then `NULL` is returned. The reference counters are not incremented so the caller must be careful with locks.

dev_get_by_name

Name `dev_get_by_name` — find a device by its name

Synopsis

```
struct net_device * dev_get_by_name (const char * name);
```

Arguments

`name`

name to find

Description

Find an interface by name. This can be called from any context and does its own locking. The returned handle has the usage count incremented and the caller must use `dev_put` to release it when it is no longer needed. `NULL` is returned if no matching device is found.

dev_get

Name `dev_get` — test if a device exists

Synopsis

```
int dev_get (const char * name);
```

Arguments

`name`

name to test for

Description

Test if a name exists. Returns true if the name is found. In order to be sure the name is not allocated or removed during the test the caller must hold the rtnl semaphore.

This function primarily exists for back compatibility with older drivers.

__dev_get_by_index

Name `__dev_get_by_index` — find a device by its ifindex

Synopsis

```
struct net_device * __dev_get_by_index (int ifindex);
```

Arguments

`ifindex`

index of device

Description

Search for an interface by index. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold either the RTNL semaphore or *`dev_base_lock`*.

dev_get_by_index

Name `dev_get_by_index` — find a device by its ifindex

Synopsis

```
struct net_device * dev_get_by_index (int ifindex);
```

Arguments

`ifindex`

index of device

Description

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device returned has had a reference added and the pointer is safe until the user calls dev_put to indicate they have finished with it.

dev_alloc_name

Name `dev_alloc_name` — allocate a name for a device

Synopsis

```
int dev_alloc_name (struct net_device * dev, const char * name);
```

Arguments

`dev`

device

`name`

name format string

Description

Passed a format string - eg “`lt%d`” it will try and find a suitable id. Not efficient for many devices, not called a lot. The caller must hold the dev_base or rtnl lock while allocating the name and adding the device in order to avoid duplicates. Returns the number of the unit assigned or a negative errno code.

dev_alloc

Name `dev_alloc` — allocate a network device and name

Synopsis

```
struct net_device * dev_alloc (const char * name, int * err);
```

Arguments

`name`

name format string

`err`

error return pointer

Description

Passed a format string, eg. “`ltd`”, it will allocate a network device and space for the name. `NULL` is returned if no memory is available. If the allocation succeeds then the name is assigned and the device pointer returned. `NULL` is returned if the name allocation failed. The cause of an error is returned as a negative errno code in the variable *`err`* points to.

The caller must hold the *`dev_base`* or RTNL locks when doing this in order to avoid duplicate name allocations.

netdev_state_change

Name `netdev_state_change` — device changes state

Synopsis

```
void netdev_state_change (struct net_device * dev);
```

Arguments

`dev`

  device to cause notification

Description

Called to indicate a device has changed state. This function calls the notifier chains for netdev_chain and sends a NEWLINK message to the routing socket.

dev_load

Name `dev_load` — load a network module

Synopsis

```
void dev_load (const char * name);
```

Arguments

`name`

name of interface

Description

If a network interface is not present and the process has suitable privileges this function loads the module. If module loading is not available in this kernel then it becomes a nop.

dev_open

Name `dev_open` — prepare an interface for use.

Synopsis

```
int dev_open (struct net_device * dev);
```

Arguments

`dev`

device to open

Description

Takes a device from down to up state. The device's private open function is invoked and then the multicast lists are loaded. Finally the device is moved into the up state and a `NETDEV_UP` message is sent to the netdev notifier chain.

Calling this function on an active interface is a nop. On a failure a negative errno code is returned.

dev_close

Name `dev_close` — shutdown an interface.

Synopsis

```
int dev_close (struct net_device * dev);
```

Arguments

`dev`

device to shutdown

Description

This function moves an active device into down state. A `NETDEV_GOING_DOWN` is sent to the netdev notifier chain. The device is then deactivated and finally a `NETDEV_DOWN` is sent to the notifier chain.

register_netdevice_notifier

Name `register_netdevice_notifier` — register a network notifier block

Synopsis

```
int register_netdevice_notifier (struct notifier_block * nb);
```

Arguments

`nb`

notifier

Description

Register a notifier to be called when network device events occur. The notifier passed is linked into the kernel structures and must not be reused until it has been unregistered. A negative errno code is returned on a failure.

unregister_netdevice_notifier

Name `unregister_netdevice_notifier` — unregister a network notifier block

Synopsis

```
int unregister_netdevice_notifier (struct notifier_block * nb);
```

Arguments

`nb`

notifier

Description

Unregister a notifier previously registered by `register_netdevice_notifier`. The notifier is unlinked into the kernel structures and may then be reused. A negative errno code is returned on a failure.

dev_queue_xmit

Name `dev_queue_xmit` — transmit a buffer

Synopsis

```
int dev_queue_xmit (struct sk_buff * skb);
```

Arguments

`skb`

buffer to transmit

Description

Queue a buffer for transmission to a network device. The caller must have set the device and priority and built the buffer before calling this function. The function can be called from an interrupt.

A negative errno code is returned on a failure. A success does not guarantee the frame will be transmitted as it may be dropped due to congestion or traffic shaping.

netif_rx

Name `netif_rx` — post buffer to the network code

Synopsis

`void **netif_rx** (struct sk_buff * *skb*);`

Arguments

`skb`

buffer to post

Description

This function receives a packet from a device driver and queues it for the upper (protocol) levels to process. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

net_call_rx_atomic

Name `net_call_rx_atomic` —

Synopsis

`void **net_call_rx_atomic** (void (*fn) (void));`

Arguments

`fn`

function to call

Description

Make a function call that is atomic with respect to the protocol layers.

register_gifconf

Name `register_gifconf` — register a SIOCGIF handler

Synopsis

```
int register_gifconf (unsigned int family, gifconf_func_t * gifconf);
```

Arguments

`family`

Address family

`gifconf`

Function handler

Description

Register protocol dependent address dumping routines. The handler that is passed must not be freed or reused until it has been replaced by another handler.

netdev_set_master

Name `netdev_set_master` — set up master/slave pair

Synopsis

```
int netdev_set_master (struct net_device * slave, struct net_device *
master);
```

Arguments

`slave`

slave device

`master`

new master device

Description

Changes the master device of the slave. Pass `NULL` to break the bonding. The caller must hold the RTNL semaphore. On a failure a negative errno code is returned. On success the reference counts are adjusted, `RTM_NEWLINK` is sent to the routing socket and the function returns zero.

dev_set_promiscuity

Name `dev_set_promiscuity` — update promiscuity count on a device

Synopsis

```
void dev_set_promiscuity (struct net_device * dev, int inc);
```

Arguments

`dev`

device

`inc`

modifier

Description

Add or remove promsicuity from a device. While the count in the device remains above zero the interface remains promiscuous. Once it hits zero the device reverts back to normal filtering operation. A negative inc value is used to drop promiscuity on the device.

dev_set_allmulti

Name `dev_set_allmulti` — update allmulti count on a device

Synopsis

```
void dev_set_allmulti (struct net_device * dev, int inc);
```

Arguments

`dev`

device

`inc`

modifier

Description

Add or remove reception of all multicast frames to a device. While the count in the device remains above zero the interface remains listening to all interfaces. Once it hits zero the device reverts back to normal filtering operation. A negative *`inc`* value is used to drop the counter when releasing a resource needing all multicasts.

dev_ioctl

Name `dev_ioctl` — network device ioctl

Synopsis

```
int dev_ioctl (unsigned int cmd, void * arg);
```

Arguments

`cmd`

command to issue

`arg`

pointer to a struct ifreq in user space

Description

Issue ioctl functions to devices. This is normally called by the user space syscall interfaces but can sometimes be useful for other purposes. The return value is the return from the syscall if positive or a negative errno code on error.

dev_new_index

Name `dev_new_index` — allocate an ifindex

Synopsis

```
int dev_new_index ( void);
```

Arguments

`void`

no arguments

Description

Returns a suitable unique value for a new device interface number. The caller must hold the rtnl semaphore to be sure it remains unique.

register_netdevice

Name `register_netdevice` — register a network device

Synopsis

```
int register_netdevice (struct net_device * dev);
```

Arguments

`dev`

device to register

Description

Take a completed network device structure and add it to the kernel interfaces. A `NETDEV_REGISTER` message is sent to the netdev notifier chain. 0 is returned on success. A negative errno code is returned on a failure to set up the device, or if the name is a duplicate.

BUGS

The locking appears insufficient to guarantee two parallel registers will not get the same name.

netdev_finish_unregister

Name

`netdev_finish_unregister` — complete unregistration

Synopsis

```
int netdev_finish_unregister (struct net_device * dev);
```

Arguments

`dev`

device

Description

Destroy and free a dead device. A value of zero is returned on success.

unregister_netdevice

Name

`unregister_netdevice` — remove device from the kernel

Synopsis

```
int unregister_netdevice (struct net_device * dev);
```

Arguments

`dev`

device

Description

This function shuts down a device interface and removes it from the kernel tables. On success 0 is returned, on a failure a negative errno code is returned.

8390 Based Network Cards

ei_open

Name `ei_open` — Open/initialize the board.

Synopsis

```
int ei_open (struct net_device * dev);
```

Arguments

`dev`

network device to initialize

Description

This routine goes all-out, setting everything up anew at each open, even though many of these registers should only need to be set once at boot.

ei_close

Name `ei_close` — shut down network device

Synopsis

```
int ei_close (struct net_device * dev);
```

Arguments

`dev`

network device to close

Description

Opposite of `ei_open`. Only used when “ifconfig <devname> down” is done.

ei_interrupt

Name `ei_interrupt` — handle the interrupts from an 8390

Synopsis

```
void ei_interrupt (int irq, void * dev_id, struct pt_regs * regs);
```

Arguments

irq

interrupt number

dev_id

a pointer to the net_device

regs

unused

Description

Handle the ether interface interrupts. We pull packets from the 8390 via the card specific functions and fire them at the networking stack. We also handle transmit completions and wake the transmit path if neccessary. We also update the counters and do other housekeeping as needed.

ethdev_init

Name

`ethdev_init` — init rest of 8390 device struct

Synopsis

```
int ethdev_init (struct net_device * dev);
```

Arguments

dev

network device structure to init

Description

Initialize the rest of the 8390 device structure. Do NOT __init this, as it is used by 8390 based modular drivers too.

NS8390_init

Name `NS8390_init` — initialize 8390 hardware

Synopsis

```
void NS8390_init (struct net_device * dev, int startp);
```

Arguments

`dev`

network device to initialize

`startp`

boolean. non-zero value to initiate chip processing

Description

Must be called with lock held.

Synchronous PPP

sppp_input

Name `sppp_input` — receive and process a WAN PPP frame

Synopsis

```
void sppp_input (struct net_device * dev, struct sk_buff * skb);
```

Arguments

dev

The device it arrived on

skb

The buffer to process

Description

This can be called directly by cards that do not have timing constraints but is normally called from the network layer after interrupt servicing to process frames queued via `netif_rx`.

We process the options in the card. If the frame is destined for the protocol stacks then it requeues the frame for the upper level protocol. If it is a control from it is processed and discarded here.

sppp_close

Name `sppp_close` — close down a synchronous PPP or Cisco HDLC link

Synopsis

```
int sppp_close (struct net_device * dev);
```

Arguments

dev

The network device to drop the link of

Description

This drops the logical interface to the channel. It is not done politely as we assume we will also be dropping DTR. Any timeouts are killed.

sppp_open

Name `sppp_open` — open a synchronous PPP or Cisco HDLC link

Synopsis

```
int sppp_open (struct net_device * dev);
```

Arguments

dev

Network device to activate

Description

Close down any existing synchronous session and commence from scratch. In the PPP case this means negotiating LCP/IPCP and friends, while for Cisco HDLC we simply need to staet sending keepalives

sppp_reopen

Name `sppp_reopen` — notify of physical link loss

Synopsis

```
int sppp_reopen (struct net_device * dev);
```

Arguments

dev

Device that lost the link

Description

This function informs the synchronous protocol code that the underlying link died (for example a carrier drop on X.21)

We increment the magic numbers to ensure that if the other end failed to notice we will correctly start a new session. It happens do to the nature of telco circuits is that you can lose carrier on one endonly.

Having done this we go back to negotiating. This function may be called from an interrupt context.

sppp_change_mtu

Name `sppp_change_mtu` — Change the link MTU

Synopsis

```
int sppp_change_mtu (struct net_device * dev, int new_mtu);
```

Arguments

dev

Device to change MTU on

`new_mtu`

New MTU

Description

Change the MTU on the link. This can only be called with the link down. It returns an error if the link is up or the mtu is out of range.

sppp_do_ioctl

Name `sppp_do_ioctl` — Ioctl handler for ppp/hdlc

Synopsis

```
int sppp_do_ioctl (struct net_device * dev, struct ifreq * ifr, int cmd);
```

Arguments

`dev`

Device subject to ioctl

`ifr`

Interface request block from the user

`cmd`

Command that is being issued

Description

This function handles the ioctls that may be issued by the user to control the settings of a PPP/HDLC link. It does both busy and security checks. This function is intended to be wrapped by callers who wish to add additional ioctl calls of their own.

sppp_attach

Name `sppp_attach` — attach synchronous PPP/HDLC to a device

Synopsis

```
void sppp_attach (struct ppp_device * pd);
```

Arguments

`pd`

PPP device to initialise

Description

This initialises the PPP/HDLC support on an interface. At the time of calling the dev element must point to the network device that this interface is attached to. The interface should not yet be registered.

sppp_detach

Name `sppp_detach` — release PPP resources from a device

Synopsis

```
void sppp_detach (struct net_device * dev);
```

Arguments

`dev`

Network device to release

Description

Stop and free up any PPP/HDLC resources used by this interface. This must be called before the device is freed.

Chapter 4. Module Loading

request_module

Name `request_module` — try to load a kernel module

Synopsis

```
int request_module (const char * module_name);
```

Arguments

`module_name`

Name of module

Description

Load a module using the user mode module loader. The function returns zero on success or a negative errno code on failure. Note that a successful module load does not mean the module did not then unload and exit on an error of its own. Callers must check that the service they requested is now available not blindly invoke it.

If module auto-loading support is disabled then this function becomes a no-operation.

Chapter 5. Hardware Interfaces

Interrupt Handling

disable_irq_nosync

Name `disable_irq_nosync` — disable an irq without waiting

Synopsis

```
void inline disable_irq_nosync (unsigned int irq);
```

Arguments

`irq`

Interrupt to disable

Description

Disable the selected interrupt line. Disables of an interrupt stack. Unlike `disable_irq`, this function does not ensure existing instances of the IRQ handler have completed before returning.

This function may be called from IRQ context.

disable_irq

Name `disable_irq` — disable an irq and wait for completion

Synopsis

```
void disable_irq (unsigned int irq);
```

Arguments

`irq`

Interrupt to disable

Description

Disable the selected interrupt line. Disables of an interrupt stack. That is for two disables you need two enables. This function waits for any pending IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock.

This function may be called - with care - from IRQ context.

enable_irq

Name `enable_irq` — enable interrupt handling on an irq

Synopsis

```
void enable_irq (unsigned int irq);
```

Arguments

`irq`

Interrupt to enable

Description

Re-enables the processing of interrupts on this IRQ line providing no disable_irq calls are now in effect.

This function may be called from IRQ context.

probe_irq_mask

Name `probe_irq_mask` — scan a bitmap of interrupt lines

Synopsis

```
unsigned int probe_irq_mask (unsigned long val);
```

Arguments

`val`

  mask of interrupts to consider

Description

Scan the ISA bus interrupt lines and return a bitmap of active interrupts. The interrupt probe logic state is then returned to its previous value.

MTRR Handling

mtrr_add

Name `mtrr_add` — Add a memory type region

Synopsis

int **mtrr_add** (unsigned long *base*, unsigned long *size*, unsigned int *type*, char *increment*);

Arguments

base

Physical base address of region

size

Physical size of region

type

Type of MTRR desired

increment

If this is true do usage counting on the region

Description

Memory type region registers control the caching on newer Intel and non Intel processors. This function allows drivers to request an MTRR is added. The details and hardware specifics of each processor's implementation are hidden from the caller, but nevertheless the caller should expect to need to provide a power of two size on an equivalent power of two boundary.

If the region cannot be added either because all regions are in use or the CPU cannot support it a negative value is returned. On success the register number for this entry is returned, but should be treated as a cookie only.

On a multiprocessor machine the changes are made to all processors. This is required on x86 by the Intel processors.

The available types are

`MTRR_TYPE_UNCACHEABLE` - No caching

`MTRR_TYPE_WRITEBACK` - Write data back in bursts whenever

`MTRR_TYPE_WRCOMB` - Write data back soon but allow bursts

`MTRR_TYPE_WRTHROUGH` - Cache reads but not writes

BUGS

Needs a quiet flag for the cases where drivers do not mind failures and do not wish system log messages to be sent.

mtrr_del

Name `mtrr_del` — delete a memory type region

Synopsis

```
int mtrr_del (int reg, unsigned long base, unsigned long size);
```

Arguments

`reg`

Register returned by mtrr_add

`base`

Physical base address

`size`

Size of region

Description

If register is supplied then base and size are ignored. This is how drivers should call it.

Releases an MTRR region. If the usage count drops to zero the register is freed and the region returns to default state. On success the register is returned, on failure a negative error code.