Linux Filesystems API

Table of Contents

Chapter 1. The Linux VFS

Table of Contents

The Filesystem types

Name

enum positive_aop_returns — aop return codes with specific semantics

Synopsis

```
enum positive_aop_returns {
  AOP_WRITEPAGE_ACTIVATE,
  AOP_TRUNCATED_PAGE
};
```

Constants

AOP_WRITEPAGE_ACTIVATE

> Informs the caller that page writeback has completed, that the page is still locked, and should be considered active. The VM uses this hint to return the page to the active list -- it won't be a candidate for writeback again in the near future. Other callers must be careful to unlock the page if they get this return. Returned by `writepage`;

AOP_TRUNCATED_PAGE

> The AOP method that was handed a locked page has unlocked it and the page might have been truncated. The caller should back up to acquiring a new page and trying again. The aop will be taking reasonable precautions not to livelock. If the caller held a page reference, it should drop it before retrying. Returned by `readpage`.

Description

address_space_operation functions return these large constants to indicate special semantics to the caller. These are much larger than the bytes in a page to allow for functions that return the number of bytes operated on in a given page.

Name

inc_nlink — directly increment an inode's link count

Synopsis

void **inc_nlink** (*inode*);

```
struct inode * inode;
```

Arguments

inode

inode

Description

This is a low-level filesystem helper to replace any direct filesystem manipulation of i_nlink. Currently, it is only here for parity with `dec_nlink`.

Name

drop_nlink — directly drop an inode's link count

Synopsis

```
void drop_nlink (inode);

struct inode * inode;
```

Arguments

inode

inode

Description

This is a low-level filesystem helper to replace any direct filesystem manipulation of i_nlink. In cases where we are attempting to track writes to the filesystem, a decrement to zero means an imminent write when the file is truncated and actually unlinked on the filesystem.

Name

clear_nlink — directly zero an inode's link count

Synopsis

```
void clear_nlink (inode);

struct inode * inode;
```

Arguments

`inode`

inode

Description

This is a low-level filesystem helper to replace any direct filesystem manipulation of i_nlink. See `drop_nlink` for why we care about i_nlink hitting zero.

Name

inode_inc_iversion — increments i_version

Synopsis

```
void inode_inc_iversion (inode);

struct inode * inode;
```

Arguments

`inode`

inode that need to be updated

Description

Every time the inode is modified, the i_version field will be incremented. The filesystem has to be mounted with i_version flag

The Directory Cache

Name

d_invalidate — invalidate a dentry

Synopsis

```
int d_invalidate (dentry);

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.

no dcache lock.

Name

shrink_dcache_sb — shrink dcache for a superblock

Synopsis

```
void shrink_dcache_sb (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

---

Name

have_submounts — check for mounts over a dentry

Synopsis

```
int have_submounts (parent);

struct dentry *  parent;
```

Arguments

parent

dentry to check.

Description

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

---

Name

shrink_dcache_parent — prune dcache

Synopsis

```
void shrink_dcache_parent (parent);

struct dentry *  parent;
```

Arguments

parent

parent of entries to prune

Description

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

Name

d_alloc — allocate a dcache entry

Synopsis

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

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.

Name

d_instantiate — fill in inode information for a dentry

Synopsis

```
void d_instantiate (entry,
                    inode);

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.

Name

d_alloc_root — allocate root dentry

Synopsis

```
struct dentry * d_alloc_root (root_inode);

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`.

Name

d_obtain_alias — find or allocate a dentry for a given inode

Synopsis

```
struct dentry * d_obtain_alias (inode);

struct inode *  inode;
```

Arguments

inode

inode to allocate the dentry for

Description

Obtain a dentry for an inode resulting from NFS filehandle conversion or similar open by handle operations. The returned dentry may be anonymous, or may have a full name (if the inode was already in the cache).

When called on a directory inode, we must ensure that the inode only ever has one dentry. If a dentry is found, that is returned instead of allocating a new one.

On successful return, the reference to the inode has been transferred to the dentry. In case of an error the reference on the inode is released. To make it easier to use in export operations a `NULL` or IS_ERR inode may be passed in and will be the error will be propagate to the return value, with a `NULL` *`inode`* replaced by ERR_PTR(-ESTALE).

Name

d_splice_alias — splice a disconnected dentry into the tree if one exists

Synopsis

```
struct dentry * d_splice_alias (inode,
                                   dentry);

struct inode *   inode;
struct dentry *  dentry;
```

Arguments

`inode`

the inode which may have a disconnected dentry

`dentry`

a negative dentry which we want to point to the inode.

Description

If inode is a directory and has a 'disconnected' dentry (i.e. IS_ROOT and DCACHE_DISCONNECTED), then d_move that in place of the given dentry and return it, else simply d_add the inode to the dentry and return NULL.

This is needed in the lookup routine of any filesystem that is exportable (via knfsd) so that we can build dcache paths to directories effectively.

If a dentry was found and moved, then it is returned. Otherwise NULL is returned. This matches the expected return value of ->lookup.

Name

d_add_ci — lookup or allocate new dentry with case-exact name

Synopsis

```
struct dentry * d_add_ci (dentry,
                          inode,
                          name);
```

```
struct dentry *  dentry;
struct inode *   inode;
struct qstr *    name;
```

Arguments

dentry

the negative dentry that was passed to the parent's lookup func

inode

the inode case-insensitive lookup has found

name

the case-exact name to be associated with the returned dentry

Description

This is to avoid filling the dcache with case-insensitive names to the same inode, only the actual correct case is stored in the dcache for case-insensitive filesystems.

For a case-insensitive lookup match and if the the case-exact dentry already exists in in the dcache, use it and return it.

If no entry exists with the exact case name, allocate new dentry with the exact case, and return the spliced entry.

Name

d_lookup — search for a dentry

Synopsis

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

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 dput to free the entry when it has finished using it. `NULL` is returned on failure.

__d_lookup is dcache_lock free. The hash list is protected using RCU. Memory barriers are used while updating and doing lockless traversal. To avoid races with d_move while rename is happening, d_lock is used.

Overflows in `memcmp`, while d_move, are avoided by keeping the length and name pointer in one structure pointed by d_qstr.

`rcu_read_lock` and `rcu_read_unlock` are used to disable preemption while lookup is going on.

The dentry unused LRU is not updated even if lookup finds the required dentry in there. It is updated in places such as prune_dcache, shrink_dcache_sb, select_parent and __dget_locked. This laziness saves lookup from dcache_lock acquisition.

`d_lookup` is protected against the concurrent renames in some unrelated directory using the seqlockt_t rename_lock.

Name

d_validate — verify dentry provided from insecure source

Synopsis

```
int d_validate (dentry,
                dparent);

struct dentry *  dentry;
struct dentry *  dparent;
```

Arguments

dentry

The dentry alleged to be valid child of *dparent*

dparent

The parent dentry (known to be valid)

Description

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

Name

d_delete — delete a dentry

Synopsis

```
void d_delete (dentry);

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

---

Name

d_rehash — add an entry back to the hash

Synopsis

```
void d_rehash (entry);

struct dentry *  entry;
```

Arguments

entry

dentry to add to the hash

Description

Adds a dentry to the hash according to its name.

Name

d_move — move a dentry

Synopsis

```
void d_move (dentry,
             target);

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.

Name

d_materialise_unique — introduce an inode into the tree

Synopsis

```
struct dentry * d_materialise_unique (dentry,
                                      inode);

struct dentry *  dentry;
struct inode *   inode;
```

Arguments

dentry

candidate dentry

inode

inode to bind to the dentry, to which aliases may be attached

Description

Introduces an dentry into the tree, substituting an extant disconnected root directory alias in its place if there is one

Name

d_path — return the path of a dentry

Synopsis

```
char * d_path (path,
               buf,
               buflen);

const struct path *  path;
char *               buf;
int                  buflen;
```

Arguments

path

path to report

buf

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 a pointer into the buffer or an error code if the path was too long. Note: Callers should use the returned pointer, not the passed in buffer, to use the name! The implementation often starts at an offset into the buffer, and may leave 0 bytes at the start.

“buflen” should be positive.

Name

find_inode_number — check for dentry with name

Synopsis

```
ino_t find_inode_number (dir,
                         name);

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.

Name

__d_drop — drop a dentry

Synopsis

```
void __d_drop (dentry);

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_drop requires dentry->d_lock.

Name

d_add — add dentry to hash queues

Synopsis

```
void d_add (entry,
            inode);

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`.

Name

d_add_unique — add dentry to hash queues without aliasing

Synopsis

```
struct dentry * d_add_unique (entry,
                              inode);

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`.

Name

dget — get a reference to a dentry

Synopsis

```
struct dentry * dget (dentry);

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. `dget` should never be called for dentries with zero reference counter. For these cases (preferably none, functions in dcache.c are sufficient for normal needs and they take necessary precautions) you should hold dcache_lock and call `dget_locked` instead of `dget`.

Name

d_unhashed — is dentry hashed

Synopsis

```
int d_unhashed (dentry);

struct dentry *  dentry;
```

Arguments

dentry

entry to check

Description

Returns true if the dentry passed is not currently hashed.

Inode Handling

Name

inode_init_always — perform inode structure intialisation

Synopsis

```
int inode_init_always (sb,
                       inode);

struct super_block *  sb;
struct inode *        inode;
```

Arguments

sb

superblock inode belongs to

inode

inode to initialise

Description

These are initializations that need to be done on every inode allocation as the fields are not initialised by slab allocation.

Name

clear_inode — clear an inode

Synopsis

```
void clear_inode (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.

Name

invalidate_inodes — discard the inodes on a device

Synopsis

```
int invalidate_inodes (sb);

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.

Name

inode_add_to_lists — add a new inode to relevant lists

Synopsis

```
void inode_add_to_lists (sb,
                         inode);

struct super_block *  sb;
struct inode *        inode;
```

Arguments

sb

superblock inode belongs to

inode

inode to mark in use

Description

When an inode is allocated it needs to be accounted for, added to the in use list, the owning superblock and the inode hash. This needs to be done under the inode_lock, so export a function to do this rather than the inode lock itself. We calculate the hash list to add to here so it is all internal which requires the caller to have already set up the inode number in the inode to add.

Name

new_inode — obtain an inode

Synopsis

```
struct inode * new_inode (sb);

struct super_block *  sb;
```

Arguments

sb

superblock

Description

Allocates a new inode for given superblock. The default gfp_mask for allocations related to inode->i_mapping is GFP_HIGHUSER_MOVABLE. If HIGHMEM pages are unsuitable or it is known that pages allocated for the page cache are not reclaimable or migratable, `mapping_set_gfp_mask` must be called with suitable flags on the newly created inode's mapping

---

Name

iunique — get a unique inode number

Synopsis

```
ino_t iunique (sb,
               max_reserved);

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.

Name

ilookup5_nowait — search for an inode in the inode cache

Synopsis

```
struct inode * ilookup5_nowait (sb,
                                hashval,
                                test,
                                data);

struct super_block *  sb;
unsigned long         hashval;
int (*                test(struct inode *, void *);
void *                data;
```

Arguments

sb

super block of file system to search

hashval

hash value (usually inode number) to search for

test

callback used for comparisons between inodes

data

opaque data pointer to pass to *test*

Description

ilookup5 uses ifind to search for the inode specified by *hashval* and *data* in the inode cache. This is a generalized version of ilookup for file systems where the inode number is not sufficient for unique identification of an inode.

If the inode is in the cache, the inode is returned with an incremented reference count. Note, the inode lock is not waited upon so you have to be very careful what you do with the returned inode. You probably should be using ilookup5 instead.

Otherwise NULL is returned.

Note, *test* is called with the inode_lock held, so can't sleep.

Name

ilookup5 — search for an inode in the inode cache

Synopsis

```
struct inode * ilookup5 (sb,
                         hashval,
                         test,
                         data);

struct super_block *  sb;
unsigned long         hashval;
int (*                test(struct inode *, void *);
void *                data;
```

Arguments

sb

super block of file system to search

hashval

hash value (usually inode number) to search for

test

callback used for comparisons between inodes

data

opaque data pointer to pass to *test*

Description

ilookup5 uses ifind to search for the inode specified by *hashval* and *data* in the inode cache. This is a generalized version of ilookup for file systems where the inode number is not sufficient for unique identification of an inode.

If the inode is in the cache, the inode lock is waited upon and the inode is returned with an incremented reference count.

Otherwise NULL is returned.

Note, *test* is called with the inode_lock held, so can't sleep.

Name

ilookup — search for an inode in the inode cache

Synopsis

```
struct inode * ilookup (sb,
                        ino);

struct super_block *  sb;
unsigned long         ino;
```

Arguments

sb

super block of file system to search

ino

inode number to search for

Description

`ilookup` uses `ifind_fast` to search for the inode *`ino`* in the inode cache. This is for file systems where the inode number is sufficient for unique identification of an inode.

If the inode is in the cache, the inode is returned with an incremented reference count.

Otherwise NULL is returned.

Name

iget5_locked — obtain an inode from a mounted file system

Synopsis

```
struct inode * iget5_locked (sb,
                             hashval,
                             test,
                             set,
                             data);

struct super_block *  sb;
unsigned long         hashval;
int (*                test(struct inode *, void *);
int (*                set(struct inode *, void *);
void *                data;
```

Arguments

sb

super block of file system

hashval

hash value (usually inode number) to get

test

callback used for comparisons between inodes

set

callback used to initialize a new struct inode

data

opaque data pointer to pass to *test* and *set*

Description

`iget5_locked` uses `ifind` to search for the inode specified by *hashval* and *data* in the inode cache and if present it is returned with an increased reference count. This is a generalized version of `iget_locked` for file systems where the inode number is not sufficient for unique identification of an inode.

If the inode is not in cache, `get_new_inode` is called to allocate a new inode and this is returned locked, hashed, and with the I_NEW flag set. The file system gets to fill it in before unlocking it via `unlock_new_inode`.

Note both *test* and *set* are called with the inode_lock held, so can't sleep.

Name

iget_locked — obtain an inode from a mounted file system

Synopsis

```
struct inode * iget_locked (sb,
                            ino);

struct super_block *  sb;
unsigned long         ino;
```

Arguments

sb

super block of file system

ino

inode number to get

Description

`iget_locked` uses `ifind_fast` to search for the inode specified by *`ino`* in the inode cache and if present it is returned with an increased reference count. This is for file systems where the inode number is sufficient for unique identification of an inode.

If the inode is not in cache, `get_new_inode_fast` is called to allocate a new inode and this is returned locked, hashed, and with the I_NEW flag set. The file system gets to fill it in before unlocking it via `unlock_new_inode`.

Name

__insert_inode_hash — hash an inode

Synopsis

```
void __insert_inode_hash (inode,
                          hashval);

struct inode *  inode;
unsigned long   hashval;
```

Arguments

`inode`

unhashed inode

`hashval`

unsigned long value used to locate this object in the inode_hashtable.

Description

Add an inode to the inode hash for this superblock.

Name

remove_inode_hash — remove an inode from the hash

Synopsis

```
void remove_inode_hash (inode);
```

`struct inode * inode;`

Arguments

inode

inode to unhash

Description

Remove an inode from the superblock.

Name

generic_detach_inode — remove inode from inode lists

Synopsis

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

Arguments

inode

inode to remove

Description

Remove inode from inode lists, write it if it's dirty. This is just an internal VFS helper exported for hugetlbfs. Do not use!

Returns 1 if inode should be completely destroyed.

Name

iput — put an inode

Synopsis

```
void iput (inode);
```

```
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 then freed and may also be destroyed.

Consequently, `iput` can sleep.

Name

bmap — find a block number in a file

Synopsis

```
sector_t bmap (inode,
               block);

struct inode *  inode;
sector_t        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.

Name

touch_atime — update the access time

Synopsis

```
void touch_atime (mnt,
                  dentry);

struct vfsmount *  mnt;
struct dentry *    dentry;
```

Arguments

mnt

mount the inode is accessed on

dentry

dentry 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.

Name

file_update_time — update mtime and ctime time

Synopsis

```
void file_update_time (file);

struct file *  file;
```

Arguments

file

file accessed

Description

Update the mtime and ctime members of an inode and mark the inode for writeback. Note that this function is meant exclusively for usage in the file write path of filesystems, and filesystems may choose to explicitly ignore update via this function with the S_NOCMTIME inode flag, e.g. for network filesystem where these timestamps are handled by the server.

Name

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

Synopsis

```
void make_bad_inode (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.

Name

is_bad_inode — is an inode errored

Synopsis

```
int is_bad_inode (inode);

struct inode *  inode;
```

Arguments

inode

inode to test

Description

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

Name

iget_failed — Mark an under-construction inode as dead and release it

Synopsis

```
void iget_failed (inode);

struct inode *  inode;
```

Arguments

inode

> The inode to discard

Description

Mark an under-construction inode as dead and release it.

Registration and Superblocks

Name

deactivate_super — drop an active reference to superblock

Synopsis

```
void deactivate_super (s);

struct super_block *  s;
```

Arguments

s

> superblock to deactivate

Description

Drops an active reference to superblock, acquiring a temprory one if there is no active references left. In that case we lock superblock, tell fs driver to shut it down and drop the temporary reference we had just acquired.

Name

deactivate_locked_super — drop an active reference to superblock

Synopsis

```
void deactivate_locked_super (s);

struct super_block *  s;
```

Arguments

s

superblock to deactivate

Description

Equivalent of up_write(s->s_umount); deactivate_super(s);, except that it does not unlock it until it's all over. As the result, it's safe to use to dispose of new superblock on ->`get_sb` failure exits - nobody will see the sucker until it's all over. Equivalent using up_write + deactivate_super is safe for that purpose only if superblock is either safe to use or has NULL ->s_root when we unlock.

Name

generic_shutdown_super — common helper for ->`kill_sb`

Synopsis

```
void generic_shutdown_super (sb);

struct super_block *  sb;
```

Arguments

sb

superblock to kill

Description

`generic_shutdown_super` does all fs-independent work on superblock shutdown. Typical ->`kill_sb` should pick all fs-specific objects that need destruction out of superblock, call `generic_shutdown_super` and release aforementioned objects. Note: dentries and inodes _are_ taken care of and do not need specific handling.

Upon calling this function, the filesystem may no longer alter or rearrange the set of dentries belonging to this super_block, nor may it change the attachments of dentries to inodes.

Name

sget — find or create a superblock

Synopsis

```
struct super_block * sget (type,
                           test,
                           set,
                           data);

struct file_system_type *  type;
int (*                     test(struct super_block *,void *);
int (*                     set(struct super_block *,void *);
void *                     data;
```

Arguments

type

filesystem type superblock should belong to

test

comparison callback

set

setup callback

data

argument to each of them

Name

get_super — get the superblock of a device

Synopsis

```
struct super_block * get_super (bdev);

struct block_device *  bdev;
```

Arguments

bdev

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.

File Locks

Name

posix_lock_file — Apply a POSIX-style lock to a file

Synopsis

```
int posix_lock_file (filp,
                     fl,
                     conflock);

struct file *       filp;
struct file_lock *  fl;
struct file_lock *  conflock;
```

Arguments

filp

The file to apply the lock to

fl

The lock to be applied

conflock

Place to return a copy of the conflicting lock, if found.

Description

Add a POSIX style lock to a file. We merge adjacent & overlapping locks whenever possible. POSIX locks are sorted by owner task, then by starting address

Note that if called with an FL_EXISTS argument, the caller may determine whether or not a lock was successfully freed by testing the return value for -ENOENT.

Name

posix_lock_file_wait — Apply a POSIX-style lock to a file

Synopsis

```
int posix_lock_file_wait (filp,
                          fl);

struct file *      filp;
struct file_lock * fl;
```

Arguments

filp

The file to apply the lock to

fl

The lock to be applied

Description

Add a POSIX style lock to a file. We merge adjacent & overlapping locks whenever possible. POSIX locks are sorted by owner task, then by starting address

Name

locks_mandatory_area — Check for a conflicting lock

Synopsis

```
int locks_mandatory_area (read_write,
                          inode,
                          filp,
                          offset,
                          count);

int            read_write;
struct inode * inode;
struct file *  filp;
loff_t         offset;
size_t         count;
```

Arguments

read_write

`FLOCK_VERIFY_WRITE` for exclusive access, `FLOCK_VERIFY_READ` for shared

inode

the file to check

filp

how the file was opened (if it was)

offset

start of area to check

count

length of area to check

Description

Searches the inode's list of locks to find any POSIX locks which conflict. This function is called from `rw_verify_area` and `locks_verify_truncate`.

Name

__break_lease — revoke all outstanding leases on file

Synopsis

```
int __break_lease (inode,
                   mode);

struct inode *  inode;
unsigned int    mode;
```

Arguments

inode

the inode of the file to return

mode

the open mode (read or write)

Description

break_lease (inlined for speed) has checked there already is at least some kind of lock (maybe a lease) on this file. Leases are broken on a call to `open` or `truncate`. This function can sleep unless you specified `O_NONBLOCK` to your `open`.

Name

lease_get_mtime — get the last modified time of an inode

Synopsis

```
void lease_get_mtime (inode,
                      time);

struct inode *     inode;
struct timespec *  time;
```

Arguments

inode

the inode

time

pointer to a timespec which will contain the last modified time

Description

This is to force NFS clients to flush their caches for files with exclusive leases. The justification is that if someone has an exclusive lease, then they could be modifying it.

Name

generic_setlease — sets a lease on an open file

Synopsis

```
int generic_setlease (filp,
                      arg,
                      flp);

struct file *        filp;
long                 arg;
```

```
struct file_lock **  flp;
```

Arguments

filp

file pointer

arg

type of lease to obtain

flp

input - file_lock to use, output - file_lock inserted

Description

The (input) flp->fl_lmops->fl_break function is required by `break_lease`.

Called with kernel lock held.

Name

flock_lock_file_wait — Apply a FLOCK-style lock to a file

Synopsis

```
int flock_lock_file_wait (filp,
                          fl);

struct file *       filp;
struct file_lock *  fl;
```

Arguments

filp

The file to apply the lock to

fl

The lock to be applied

Description

Add a FLOCK style lock to a file.

Name

vfs_test_lock — test file byte range lock

Synopsis

```
int vfs_test_lock (filp,
                   fl);

struct file *      filp;
struct file_lock * fl;
```

Arguments

filp

The file to test lock for

fl

The lock to test; also used to hold result

Description

Returns -ERRNO on failure. Indicates presence of conflicting lock by setting conf->fl_type to something other than F_UNLCK.

Name

vfs_lock_file — file byte range lock

Synopsis

```
int vfs_lock_file (filp,
                   cmd,
                   fl,
                   conf);

struct file *      filp;
unsigned int       cmd;
struct file_lock * fl;
struct file_lock * conf;
```

Arguments

filp

The file to apply the lock to

cmd

type of locking operation (F_SETLK, F_GETLK, etc.)

fl

The lock to be applied

conf

Place to return a copy of the conflicting lock, if found.

Description

A caller that doesn't care about the conflicting lock may pass NULL as the final argument.

If the filesystem defines a private ->lock method, then *conf* will be left unchanged; so a caller that cares should initialize it to some acceptable default.

To avoid blocking kernel daemons, such as lockd, that need to acquire POSIX locks, the ->lock interface may return asynchronously, before the lock has been granted or denied by the underlying filesystem, if (and only if) fl_grant is set. Callers expecting ->lock to return asynchronously will only use F_SETLK, not F_SETLKW; they will set FL_SLEEP if (and only if) the request is for a blocking lock. When ->lock does return asynchronously, it must return FILE_LOCK_DEFERRED, and call ->fl_grant when the lock request completes. If the request is for non-blocking lock the file system should return FILE_LOCK_DEFERRED then try to get the lock and call the callback routine with the result. If the request timed out the callback routine will return a nonzero return code and the file system should release the lock. The file system is also responsible to keep a corresponding posix lock when it grants a lock so the VFS can find out which locks are locally held and do the correct lock cleanup when required. The underlying filesystem must not drop the kernel lock or call ->fl_grant before returning to the caller with a FILE_LOCK_DEFERRED return code.

Name

posix_unblock_lock — stop waiting for a file lock

Synopsis

```
int posix_unblock_lock (filp,
                        waiter);

struct file *       filp;
struct file_lock *  waiter;
```

Arguments

filp

how the file was opened

waiter

the lock which was waiting

Description

lockd needs to block waiting for locks.

Name

vfs_cancel_lock — file byte range unblock lock

Synopsis

```
int vfs_cancel_lock (filp,
                     fl);

struct file *      filp;
struct file_lock * fl;
```

Arguments

filp

The file to apply the unblock to

fl

The lock to be unblocked

Description

Used by lock managers to cancel blocked requests

Name

lock_may_read — checks that the region is free of locks

Synopsis

```
int lock_may_read (inode,
                        start,
                        len);

struct inode *  inode;
loff_t          start;
unsigned long   len;
```

Arguments

inode

the inode that is being read

start

the first byte to read

len

the number of bytes to read

Description

Emulates Windows locking requirements. Whole-file mandatory locks (share modes) can prohibit a read and byte-range POSIX locks can prohibit a read if they overlap.

N.B. this function is only ever called from knfsd and ownership of locks is never checked.

Name

lock_may_write — checks that the region is free of locks

Synopsis

```
int lock_may_write (inode,
                    start,
                    len);

struct inode *  inode;
loff_t          start;
unsigned long   len;
```

Arguments

inode

the inode that is being written

start

the first byte to write

len

the number of bytes to write

Description

Emulates Windows locking requirements. Whole-file mandatory locks (share modes) can prohibit a write and byte-range POSIX locks can prohibit a write if they overlap.

N.B. this function is only ever called from knfsd and ownership of locks is never checked.

Name

locks_mandatory_locked — Check for an active lock

Synopsis

`int` **`locks_mandatory_locked`** `(`*inode*`);`

`struct inode *` *`inode`*`;`

Arguments

inode

the file to check

Description

Searches the inode's list of locks to find any POSIX locks which conflict. This function is called from `locks_verify_locked` only.

Name

fcntl_getlease — Enquire what lease is currently active

Synopsis

`int` **`fcntl_getlease`** `(`*filp*`);`

`struct file *` *`filp`*`;`

Arguments

filp

the file

Description

The value returned by this function will be one of (if no lease break is pending):

`F_RDLCK` to indicate a shared lease is held.

`F_WRLCK` to indicate an exclusive lease is held.

`F_UNLCK` to indicate no lease is held.

(if a lease break is pending):

`F_RDLCK` to indicate an exclusive lease needs to be changed to a shared lease (or removed).

`F_UNLCK` to indicate the lease needs to be removed.

XXX

sfr & willy disagree over whether F_INPROGRESS should be returned to userspace.

Name

fcntl_setlease — sets a lease on an open file

Synopsis

```
int fcntl_setlease (fd,
                    filp,
                    arg);

unsigned int   fd;
struct file *  filp;
long           arg;
```

Arguments

fd

open file descriptor

filp

file pointer

arg

type of lease to obtain

Description

Call this fcntl to establish a lease on the file. Note that you also need to call `F_SETSIG` to receive a signal when the lease is broken.

Name

sys_flock — `flock` system call.

Synopsis

```
long sys_flock (fd,
                cmd);

unsigned int  fd;
unsigned int  cmd;
```

Arguments

fd

the file descriptor to lock.

cmd

the type of lock to apply.

Description

Apply a `FL_FLOCK` style lock to an open file descriptor. The *cmd* can be one of

`LOCK_SH` -- a shared lock.

`LOCK_EX` -- an exclusive lock.

`LOCK_UN` -- remove an existing lock.

`LOCK_MAND` -- a `mandatory' flock. This exists to emulate Windows Share Modes.

`LOCK_MAND` can be combined with `LOCK_READ` or `LOCK_WRITE` to allow other processes read and write access respectively.

Other Functions

Name

mpage_readpages — populate an address space with some pages & start reads against them

Synopsis

```
int mpage_readpages (mapping,
                     pages,
                     nr_pages,
                     get_block);

struct address_space *  mapping;
struct list_head *      pages;
unsigned                nr_pages;
get_block_t             get_block;
```

Arguments

mapping

the address_space

pages

The address of a list_head which contains the target pages. These pages have their ->index populated and are otherwise uninitialised. The page at *pages*->prev has the lowest file offset, and reads should be issued in *pages*->prev to *pages*->next order.

nr_pages

The number of pages at **pages*

get_block

The filesystem's block mapper function.

Description

This function walks the pages and the blocks within each page, building and emitting large BIOs.

If anything unusual happens, such as:

- encountering a page which has buffers - encountering a page which has a non-hole after a hole - encountering a page with non-contiguous blocks

then this code just gives up and calls the buffer_head-based read function. It does handle a page which has holes at the end - that is a common case: the end-of-file on blocksize < PAGE_CACHE_SIZE setups.

BH_Boundary explanation

There is a problem. The mpage read code assembles several pages, gets all their disk mappings, and then submits them all. That's fine, but obtaining the disk mappings may require I/O. Reads of indirect blocks, for example.

So an mpage read of the first 16 blocks of an ext2 file will cause I/O to be

submitted in the following order

12 0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16

because the indirect block has to be read to get the mappings of blocks 13,14,15,16. Obviously, this impacts performance.

So what we do it to allow the filesystem's `get_block` function to set BH_Boundary when it maps block 11. BH_Boundary says: mapping of the block after this one will require I/O against a block which is probably close to this one. So you should push what I/O you have currently accumulated.

This all causes the disk requests to be issued in the correct order.

Name

mpage_writepages — walk the list of dirty pages of the given address space & `writepage` all of them

Synopsis

```
int mpage_writepages (mapping,
                      wbc,
                      get_block);

struct address_space *      mapping;
struct writeback_control *  wbc;
get_block_t                 get_block;
```

Arguments

mapping

address space structure to write

wbc

subtract the number of written pages from **wbc*->nr_to_write

get_block

the filesystem's block mapper function. If this is NULL then use a_ops->writepage. Otherwise, go direct-to-BIO.

Description

This is a library function, which implements the `writepages` address_space_operation.

If a page is already under I/O, `generic_writepages` skips it, even if it's dirty. This is desirable behaviour for memory-cleaning writeback, but it is INCORRECT for data-integrity system calls such as `fsync`. `fsync` and `msync` need to guarantee that all the data which was dirty at the time the call was made get new I/O started against them. If wbc->sync_mode is WB_SYNC_ALL then we were called for data integrity and we must wait for existing IO to complete.

Name

generic_permission — check for access rights on a Posix-like filesystem

Synopsis

```
int generic_permission (inode,
                        mask,
                        check_acl);

struct inode *  inode;
int             mask;
int (*          check_acl(struct inode *inode, int mask);
```

Arguments

inode

inode to check access rights for

mask

right to check for (`MAY_READ`, `MAY_WRITE`, `MAY_EXEC`)

check_acl

optional callback to check for Posix ACLs

Description

Used to check for read/write/execute permissions on a file. We use “fsuid” for this, letting us set arbitrary permissions for filesystem access without changing the “normal” uids which are used for other things..

Name

inode_permission — check for access rights to a given inode

Synopsis

```
int inode_permission (inode,
                      mask);

struct inode *  inode;
int             mask;
```

Arguments

inode

inode to check permission on

mask

right to check for (`MAY_READ`, `MAY_WRITE`, `MAY_EXEC`)

Description

Used to check for read/write/execute permissions on an inode. We use “fsuid” for this, letting us set arbitrary permissions for filesystem access without changing the “normal” uids which are used for other things.

Name

file_permission — check for additional access rights to a given file

Synopsis

```
int file_permission (file,
                     mask);

struct file *  file;
int            mask;
```

Arguments

file

file to check access rights for

mask

right to check for (`MAY_READ`, `MAY_WRITE`, `MAY_EXEC`)

Description

Used to check for read/write/execute permissions on an already opened file.

Note

Do not use this function in new code. All access checks should be done using `inode_permission`.

Name

path_get — get a reference to a path

Synopsis

```
void path_get (path);
```

```
struct path * path;
```

Arguments

path

path to get the reference to

Description

Given a path increment the reference count to the dentry and the vfsmount.

Name

path_put — put a reference to a path

Synopsis

```
void path_put (path);
```

```
struct path * path;
```

Arguments

path

path to put the reference to

Description

Given a path decrement the reference count to the dentry and the vfsmount.

Name

vfs_path_lookup — lookup a file path relative to a dentry-vfsmount pair

Synopsis

```
int vfs_path_lookup (dentry,
                     mnt,
                     name,
                     flags,
                     nd);

struct dentry *     dentry;
struct vfsmount *   mnt;
const char *        name;
unsigned int        flags;
struct nameidata *  nd;
```

Arguments

dentry

pointer to dentry of the base directory

mnt

pointer to vfs mount of the base directory

name

pointer to file name

flags

lookup flags

nd

pointer to nameidata

Name

lookup_one_len — filesystem helper to lookup single pathname component

Synopsis

```
struct dentry * lookup_one_len (name,
                                base,
                                len);

const char *    name;
struct dentry * base;
int             len;
```

Arguments

name

pathname component to lookup

base

base directory to lookup from

len

maximum length *len* should be interpreted to

Description

Note that this routine is purely a helper for filesystem usage and should not be called by generic code. Also note that by using this function the nameidata argument is passed to the filesystem methods and a filesystem using this helper needs to be prepared for that.

Name

filp_open — open file and return file pointer

Synopsis

```
struct file * filp_open (filename,
                         flags,
                         mode);

const char * filename;
int          flags;
int          mode;
```

Arguments

filename

path to open

flags

open flags as per the open(2) second argument

mode

mode for the new file if O_CREAT is set, else ignored

Description

This is the helper to open a file from kernelspace if you really have to. But in generally you should not do this, so please move along, nothing to see here..

Name

lookup_create — lookup a dentry, creating it if it doesn't exist

Synopsis

```
struct dentry * lookup_create (nd,
                               is_dir);

struct nameidata *  nd;
int                 is_dir;
```

Arguments

nd

nameidata info

is_dir

directory flag

Description

Simple function to lookup and return a dentry and create it if it doesn't exist. Is SMP-safe.

Returns with nd->path.dentry->d_inode->i_mutex locked.

Name

sync_mapping_buffers — write out & wait upon a mapping's “associated” buffers

Synopsis

```
int sync_mapping_buffers (mapping);

struct address_space * mapping;
```

Arguments

mapping

the mapping which wants those buffers written

Description

Starts I/O against the buffers at mapping->private_list, and waits upon that I/O.

Basically, this is a convenience function for `fsync`. *mapping* is a file or directory which needs those buffers to be written for a successful `fsync`.

Name

mark_buffer_dirty — mark a buffer_head as needing writeout

Synopsis

```
void mark_buffer_dirty (bh);

struct buffer_head * bh;
```

Arguments

bh

the buffer_head to mark dirty

Description

`mark_buffer_dirty` will set the dirty bit against the buffer, then set its backing page dirty, then tag the page as dirty in its address_space's radix tree and then attach the address_space's inode to its superblock's dirty inode list.

`mark_buffer_dirty` is atomic. It takes bh->b_page->mapping->private_lock, mapping->tree_lock and the global inode_lock.

Name

__bread — reads a specified block and returns the bh

Synopsis

```
struct buffer_head * __bread (bdev,
                              block,
                              size);

struct block_device *  bdev;
sector_t               block;
unsigned               size;
```

Arguments

bdev

the block_device to read from

block

number of block

size

size (in bytes) to read

Description

Reads a specified block, and returns buffer head that contains it. It returns NULL if the block was unreadable.

Name

block_invalidatepage — invalidate part of all of a buffer-backed page

Synopsis

```
void block_invalidatepage (page,
                           offset);

struct page *  page;
```

```
unsigned long  offset;
```

Arguments

page

the page which is affected

offset

the index of the truncation point

Description

`block_invalidatepage` is called when all or part of the page has become invalidatedby a truncate operation.

`block_invalidatepage` does not have to release all buffers, but it must ensure that no dirty buffer is left outside *offset* and that no I/O is underway against any of the blocks which are outside the truncation point. Because the caller is about to free (and possibly reuse) those blocks on-disk.

Name

ll_rw_block — level access to block devices (DEPRECATED)

Synopsis

```
void ll_rw_block (rw,
                  nr,
                  bhs[]);

int                    rw;
int                    nr;
struct buffer_head *  bhs[];
```

Arguments

rw

whether to `READ` or `WRITE` or `SWRITE` or maybe `READA` (readahead)

nr

number of struct buffer_heads in the array

bhs[]

array of pointers to struct buffer_head

Description

`ll_rw_block` takes an array of pointers to struct buffer_heads, and requests an I/O operation on them, either a `READ` or a `WRITE`. The third `SWRITE` is like `WRITE` only we make sure that the *current* data in buffers are sent to disk. The fourth `READA` option is described in the documentation for `generic_make_request` which `ll_rw_block` calls.

This function drops any buffer that it cannot get a lock on (with the BH_Lock state bit) unless SWRITE is required, any buffer that appears to be clean when doing a write request, and any buffer that appears to be up-to-date when doing read request. Further it marks as clean buffers that are processed for writing (the buffer cache won't assume that they are actually clean until the buffer gets unlocked).

ll_rw_block sets b_end_io to simple completion handler that marks the buffer up-to-date (if approriate), unlocks the buffer and wakes any waiters.

All of the buffers must be for the same device, and must also be a multiple of the current approved size for the device.

Name

bh_uptodate_or_lock — Test whether the buffer is uptodate

Synopsis

```
int bh_uptodate_or_lock (bh);

struct buffer_head *  bh;
```

Arguments

bh

struct buffer_head

Description

Return true if the buffer is up-to-date and false, with the buffer locked, if not.

Name

bh_submit_read — Submit a locked buffer for reading

Synopsis

```
int bh_submit_read (bh);

struct buffer_head *  bh;
```

Arguments

bh

struct buffer_head

Description

Returns zero on success and -EIO on error.

Name

bio_alloc_bioset — allocate a bio for I/O

Synopsis

```
struct bio * bio_alloc_bioset (gfp_mask,
                               nr_iovecs,
                               bs);

gfp_t            gfp_mask;
int              nr_iovecs;
struct bio_set * bs;
```

Arguments

gfp_mask

the GFP_ mask given to the slab allocator

nr_iovecs

number of iovecs to pre-allocate

bs

the bio_set to allocate from. If `NULL`, just use kmalloc

Description

bio_alloc_bioset will first try its own mempool to satisfy the allocation. If `__GFP_WAIT` is set then we will block on the internal pool waiting for a struct bio to become free. If a `NULL` *bs* is passed in, we will fall back to just using *kmalloc* to allocate the required memory.

Note that the caller must set ->bi_destructor on succesful return of a bio, to do the appropriate freeing of the bio once the reference count drops to zero.

Name

bio_alloc — allocate a new bio, memory pool backed

Synopsis

```
struct bio * bio_alloc (gfp_mask,
                        nr_iovecs);

gfp_t  gfp_mask;
int    nr_iovecs;
```

Arguments

gfp_mask

allocation mask to use

nr_iovecs

number of iovecs

Description

Allocate a new bio with *nr_iovecs* bvecs. If *gfp_mask* contains __GFP_WAIT, the allocation is guaranteed to succeed.

RETURNS

Pointer to new bio on success, NULL on failure.

Name

bio_kmalloc — allocate a bio for I/O

Synopsis

```
struct bio * bio_kmalloc (gfp_mask,
                          nr_iovecs);

gfp_t  gfp_mask;
int    nr_iovecs;
```

Arguments

gfp_mask

the GFP_ mask given to the slab allocator

nr_iovecs

number of iovecs to pre-allocate

Description

bio_alloc will allocate a bio and associated bio_vec array that can hold at least *nr_iovecs* entries. Allocations will be done from the fs_bio_set. Also see *bio_alloc_bioset*.

If __GFP_WAIT is set, then bio_alloc will always be able to allocate a bio. This is due to the mempool guarantees. To make this work, callers must never allocate more than 1 bio at a time from this pool. Callers that need to allocate more than 1 bio must always submit the previously allocated bio for IO before attempting to allocate a new one. Failure to do so can cause livelocks under memory pressure.

Name

bio_put — release a reference to a bio

Synopsis

```
void bio_put (bio);

struct bio * bio;
```

Arguments

bio

bio to release reference to

Description

Put a reference to a struct bio, either one you have gotten with bio_alloc or bio_get. The last put of a bio will free it.

Name

__bio_clone — clone a bio

Synopsis

```
void __bio_clone (bio,
                  bio_src);

struct bio *  bio;
struct bio *  bio_src;
```

Arguments

bio

destination bio

bio_src

bio to clone

Description

Clone a bio. Caller will own the returned bio, but not the actual data it points to. Reference count of returned bio will be one.

Name

bio_clone — clone a bio

Synopsis

```
struct bio * bio_clone (bio,
                        gfp_mask);

struct bio *  bio;
gfp_t         gfp_mask;
```

Arguments

bio

bio to clone

gfp_mask

allocation priority

Description

Like __bio_clone, only also allocates the returned bio

Name

bio_get_nr_vecs — return approx number of vecs

Synopsis

```
int bio_get_nr_vecs (bdev);

struct block_device *  bdev;
```

Arguments

bdev

I/O target

Description

Return the approximate number of pages we can send to this target. There's no guarantee that you will be able to fit this number of pages into a bio, it does not account for dynamic restrictions that vary on offset.

Name

bio_add_pc_page — attempt to add page to bio

Synopsis

```
int bio_add_pc_page (q,
                     bio,
                     page,
                     len,
                     offset);

struct request_queue *  q;
struct bio *            bio;
struct page *           page;
unsigned int            len;
unsigned int            offset;
```

Arguments

q

the target queue

bio

destination bio

page

page to add

len

vec entry length

offset

vec entry offset

Description

Attempt to add a page to the bio_vec maplist. This can fail for a number of reasons, such as the bio being full or target block device limitations. The target block device must allow bio's smaller than PAGE_SIZE, so it is always possible to add a single page to an empty bio. This should only be used by REQ_PC bios.

Name

bio_add_page — attempt to add page to bio

Synopsis

```
int bio_add_page (bio,
                  page,
                  len,
                  offset);

struct bio *   bio;
struct page *  page;
unsigned int   len;
unsigned int   offset;
```

Arguments

bio

destination bio

`page`

page to add

`len`

vec entry length

`offset`

vec entry offset

Description

Attempt to add a page to the bio_vec maplist. This can fail for a number of reasons, such as the bio being full or target block device limitations. The target block device must allow bio's smaller than PAGE_SIZE, so it is always possible to add a single page to an empty bio.

Name

bio_uncopy_user — finish previously mapped bio

Synopsis

```
int bio_uncopy_user (bio);

struct bio * bio;
```

Arguments

`bio`

bio being terminated

Description

Free pages allocated from `bio_copy_user` and write back data to user space in case of a read.

Name

bio_copy_user — copy user data to bio

Synopsis

```
struct bio * bio_copy_user (q,
                                 map_data,
                                 uaddr,
                                 len,
                                 write_to_vm,
                                 gfp_mask);

struct request_queue *  q;
struct rq_map_data *    map_data;
unsigned long           uaddr;
unsigned int            len;
int                     write_to_vm;
gfp_t                   gfp_mask;
```

Arguments

q

destination block queue

map_data

pointer to the rq_map_data holding pages (if necessary)

uaddr

start of user address

len

length in bytes

write_to_vm

bool indicating writing to pages or not

gfp_mask

memory allocation flags

Description

Prepares and returns a bio for indirect user io, bouncing data to/from kernel pages as necessary. Must be paired with call `bio_uncopy_user` on io completion.

Name

bio_map_user — map user address into bio

Synopsis

```
struct bio * bio_map_user (q,
                           bdev,
                           uaddr,
                           len,
                           write_to_vm,
                           gfp_mask);

struct request_queue *  q;
struct block_device *   bdev;
unsigned long           uaddr;
unsigned int            len;
int                     write_to_vm;
gfp_t                   gfp_mask;
```

Arguments

q

the struct request_queue for the bio

bdev

destination block device

uaddr

start of user address

len

length in bytes

write_to_vm

bool indicating writing to pages or not

gfp_mask

memory allocation flags

Description

Map the user space address into a bio suitable for io to a block device. Returns an error pointer in case of error.

Name

bio_unmap_user — unmap a bio

Synopsis

```
void bio_unmap_user (bio);

struct bio *  bio;
```

Arguments

bio

the bio being unmapped

Description

Unmap a bio previously mapped by `bio_map_user`. Must be called with a process context.

`bio_unmap_user` may sleep.

Name

bio_map_kern — map kernel address into bio

Synopsis

```
struct bio * bio_map_kern (q,
                           data,
                           len,
                           gfp_mask);

struct request_queue *  q;
void *                  data;
unsigned int            len;
gfp_t                   gfp_mask;
```

Arguments

q

the struct request_queue for the bio

data

pointer to buffer to map

len

length in bytes

gfp_mask

allocation flags for bio allocation

Description

Map the kernel address into a bio suitable for io to a block device. Returns an error pointer in case of error.

Name

bio_copy_kern — copy kernel address into bio

Synopsis

```
struct bio * bio_copy_kern (q,
                            data,
                            len,
                            gfp_mask,
                            reading);

struct request_queue *  q;
void *                  data;
unsigned int            len;
gfp_t                   gfp_mask;
int                     reading;
```

Arguments

q

the struct request_queue for the bio

data

pointer to buffer to copy

len

length in bytes

gfp_mask

allocation flags for bio and page allocation

reading

data direction is READ

Description

copy the kernel address into a bio suitable for io to a block device. Returns an error pointer in case of error.

Name

bio_endio — end I/O on a bio

Synopsis

```
void bio_endio (bio,
                error);

struct bio *  bio;
int           error;
```

Arguments

bio

bio

error

error, if any

Description

`bio_endio` will end I/O on the whole bio. `bio_endio` is the preferred way to end I/O on a bio, it takes care of clearing BIO_UPTODATE on error. *error* is 0 on success, and and one of the established -Exxxx (-EIO, for instance) error values in case something went wrong. Noone should call `bi_end_io` directly on a bio unless they own it and thus know that it has an end_io function.

Name

bio_sector_offset — Find hardware sector offset in bio

Synopsis

```
sector_t bio_sector_offset (bio,
                            index,
                            offset);

struct bio *    bio;
```

```
unsigned short  index;
unsigned int    offset;
```

Arguments

bio

bio to inspect

index

bio_vec index

offset

offset in bv_page

Description

Return the number of hardware sectors between beginning of bio and an end point indicated by a bio_vec index and an offset within that vector's page.

Name

bioset_create — Create a bio_set

Synopsis

```
struct bio_set * bioset_create (pool_size,
                                front_pad);

unsigned int  pool_size;
unsigned int  front_pad;
```

Arguments

pool_size

Number of bio and bio_vecs to cache in the mempool

front_pad

Number of bytes to allocate in front of the returned bio

Description

Set up a bio_set to be used with *bio_alloc_bioset*. Allows the caller to ask for a number of bytes to be allocated in front of the bio. Front pad allocation is useful for embedding the bio inside another structure, to avoid allocating extra data to go with the bio. Note that the bio must be embedded at the END of that structure always, or things will break badly.

Name

seq_open — initialize sequential file

Synopsis

```
int seq_open (file,
              op);

struct file *                   file;
const struct seq_operations *  op;
```

Arguments

file

file we initialize

op

method table describing the sequence

Description

seq_open sets *file*, associating it with a sequence described by *op*. *op*->start sets the iterator up and returns the first element of sequence. *op*->stop shuts it down. *op*->next returns the next element of sequence. *op*->show prints element into the buffer. In case of error ->start and ->next return ERR_PTR(error). In the end of sequence they return NULL. ->show returns 0 in case of success and negative number in case of error. Returning SEQ_SKIP means “discard this element and move on”.

Name

seq_read — ->read method for sequential files.

Synopsis

```
ssize_t seq_read (file,
                  buf,
                  size,
                  ppos);
```

```
struct file *  file;
char __user *  buf;
size_t         size;
loff_t *       ppos;
```

Arguments

file

the file to read from

buf

the buffer to read to

size

the maximum number of bytes to read

ppos

the current position in the file

Description

Ready-made ->f_op->`read`

Name

seq_lseek — ->`llseek` method for sequential files.

Synopsis

```
loff_t seq_lseek (file,
                  offset,
                  origin);
```

```
struct file *  file;
loff_t         offset;
int            origin;
```

Arguments

file

the file in question

offset

new position

origin

0 for absolute, 1 for relative position

Description

Ready-made ->f_op->`llseek`

Name

seq_release — free the structures associated with sequential file.

Synopsis

```
int seq_release (inode,
                 file);

struct inode *  inode;
struct file *   file;
```

Arguments

inode

file->f_path.dentry->d_inode

file

file in question

Description

Frees the structures associated with sequential file; can be used as ->f_op->`release` if you don't have private data to destroy.

Name

seq_escape — print string into buffer, escaping some characters

Synopsis

```
int seq_escape (m,
                s,
                  esc);

struct seq_file *  m;
const char *       s;
const char *       esc;
```

Arguments

m

target buffer

s

string

esc

set of characters that need escaping

Description

Puts string into buffer, replacing each occurrence of character from *esc* with usual octal escape. Returns 0 in case of success, -1 - in case of overflow.

Name

mangle_path — mangle and copy path to buffer beginning

Synopsis

```
char * mangle_path (s,
                    p,
                    esc);

char *  s;
char *  p;
char *  esc;
```

Arguments

s

buffer start

p

beginning of path in above buffer

esc

set of characters that need escaping

Description

Copy the path from *p* to *s*, replacing each occurrence of character from *esc* with usual octal escape. Returns pointer past last written character in *s*, or NULL in case of failure.

Name

seq_path — seq_file interface to print a pathname

Synopsis

```
int seq_path (m,
              path,
              esc);

struct seq_file *  m;
struct path *      path;
char *             esc;
```

Arguments

m

the seq_file handle

path

the struct path to print

esc

set of characters to escape in the output

Description

return the absolute path of 'path', as represented by the dentry / mnt pair in the path parameter.

Name

seq_write — write arbitrary data to buffer

Synopsis

```
int seq_write (seq,
               data,
               len);

struct seq_file *  seq;
const void *       data;
size_t             len;
```

Arguments

seq

seq_file identifying the buffer to which data should be written

data

data address

len

number of bytes

Description

Return 0 on success, non-zero otherwise.

Name

register_filesystem — register a new filesystem

Synopsis

```
int register_filesystem (fs);

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.

Name

unregister_filesystem — unregister a file system

Synopsis

```
int unregister_filesystem (fs);

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.

Name

__mark_inode_dirty — internal function

Synopsis

```
void __mark_inode_dirty (inode,
                         flags);

struct inode *  inode;
int             flags;
```

Arguments

inode

> inode to mark

flags

> what kind of dirty (i.e. I_DIRTY_SYNC) Mark an inode as dirty. Callers should use mark_inode_dirty or mark_inode_dirty_sync.

Description

Put the inode on the super block's dirty list.

CAREFUL! We mark it dirty unconditionally, but move it onto the dirty list only if it is hashed or if it refers to a blockdev. If it was not hashed, it will never be added to the dirty list even if it is later hashed, as it will have been marked dirty already.

In short, make sure you hash any inodes _before_ you start marking them dirty.

This function *must* be atomic for the I_DIRTY_PAGES case - `set_page_dirty` is called under spinlock in several places.

Note that for blockdevs, inode->dirtied_when represents the dirtying time of the block-special inode (/dev/hda1) itself. And the ->dirtied_when field of the kernel-internal blockdev inode represents the dirtying time of the blockdev's pages. This is why for I_DIRTY_PAGES we always use page->mapping->host, so the page-dirtying time is recorded in the internal blockdev inode.

Name

writeback_inodes_sb — writeback dirty inodes from given super_block

Synopsis

```
void writeback_inodes_sb (sb);

struct super_block *  sb;
```

Arguments

sb

> the superblock

Description

Start writeback on some inodes on this super_block. No guarantees are made on how many (if any) will be written, and this function does not wait for IO completion of submitted IO. The number of pages submitted is returned.

Name

sync_inodes_sb — sync sb inode pages

Synopsis

```
void sync_inodes_sb (sb);

struct super_block *  sb;
```

Arguments

sb

the superblock

Description

This function writes and waits on any dirty inode belonging to this super_block. The number of pages synced is returned.

Name

write_inode_now — write an inode to disk

Synopsis

```
int write_inode_now (inode,
                     sync);

struct inode *  inode;
int             sync;
```

Arguments

inode

inode to write to disk

sync

whether the write should be synchronous or not

Description

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

The caller must either have a ref on the inode or must have set I_WILL_FREE.

Name

sync_inode — write an inode and its pages to disk.

Synopsis

```
int sync_inode (inode,
                wbc);

struct inode *               inode;
struct writeback_control *   wbc;
```

Arguments

inode

the inode to sync

wbc

controls the writeback mode

Description

`sync_inode` will write an inode and its pages to disk. It will also correctly update the inode on its superblock's dirty inode lists and will update inode->i_state.

The caller must have a ref on the inode.

Name

freeze_bdev — - lock a filesystem and force it into a consistent state

Synopsis

```
struct super_block * freeze_bdev (bdev);

struct block_device *  bdev;
```

Arguments

bdev

blockdevice to lock

Description

If a superblock is found on this device, we take the s_umount semaphore on it to make sure nobody unmounts until the snapshot creation is done. The reference counter (bd_fsfreeze_count) guarantees that only the last unfreeze process can unfreeze the frozen filesystem actually when multiple freeze requests arrive simultaneously. It counts up in `freeze_bdev` and count down in `thaw_bdev`. When it becomes 0, `thaw_bdev` will unfreeze actually.

Name

thaw_bdev — - unlock filesystem

Synopsis

```
int thaw_bdev (bdev,
               sb);

struct block_device *  bdev;
struct super_block *   sb;
```

Arguments

bdev

blockdevice to unlock

sb

associated superblock

Description

Unlocks the filesystem and marks it writeable again after `freeze_bdev`.

Name

bd_claim_by_disk — wrapper function for `bd_claim_by_kobject`

Synopsis

```
int bd_claim_by_disk (bdev,
                                 holder,
                                 disk);

struct block_device *  bdev;
void *                 holder;
struct gendisk *       disk;
```

Arguments

bdev

block device to be claimed

holder

holder's signature

disk

holder's gendisk

Description

Call `bd_claim_by_kobject` with getting *disk*->slave_dir.

Name

bd_release_from_disk — wrapper function for `bd_release_from_kobject`

Synopsis

```
void bd_release_from_disk (bdev,
                           disk);

struct block_device *  bdev;
struct gendisk *       disk;
```

Arguments

bdev

block device to be claimed

disk

holder's gendisk

Description

Call `bd_release_from_kobject` and put *`disk`*->slave_dir.

Name

check_disk_size_change — checks for disk size change and adjusts bdev size.

Synopsis

```
void check_disk_size_change (disk,
                             bdev);

struct gendisk *       disk;
struct block_device *  bdev;
```

Arguments

`disk`

struct gendisk to check

`bdev`

struct bdev to adjust.

Description

This routine checks to see if the bdev size does not match the disk size and adjusts it if it differs.

Name

revalidate_disk — wrapper for lower-level driver's revalidate_disk call-back

Synopsis

```
int revalidate_disk (disk);

struct gendisk *  disk;
```

Arguments

`disk`

struct gendisk to be revalidated

Description

This routine is a wrapper for lower-level driver's revalidate_disk call-backs. It is used to do common pre and post operations needed for all revalidate_disk operations.

Name

lookup_bdev — lookup a struct block_device by name

Synopsis

```
struct block_device * lookup_bdev (pathname);

const char *  pathname;
```

Arguments

pathname

special file representing the block device