Linux Filesystems API

Linux Filesystems API

Table of Contents

Chapter 1. The Linux VFS

1.1. The Filesystem types

enum positive_aop_returns

LINUX

Kernel Hackers ManualJuly 2010

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.

inc_nlink

LINUX

Kernel Hackers ManualJuly 2010

Name

`inc_nlink` — directly increment an inode's link count

Synopsis

```
void inc_nlink (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`.

drop_nlink

LINUX

Kernel Hackers ManualJuly 2010

Name

`drop_nlink` — directly drop an inode's link count

Synopsis

```
void drop_nlink (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.

clear_nlink

LINUX

Kernel Hackers ManualJuly 2010

Name

`clear_nlink` — directly zero an inode's link count

Synopsis

```
void clear_nlink (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.

inode_inc_iversion

LINUX

Kernel Hackers ManualJuly 2010

Name

`inode_inc_iversion` — increments i_version

Synopsis

```
void inode_inc_iversion (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

1.2. The Directory Cache

d_invalidate

LINUX

Kernel Hackers ManualJuly 2010

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.

no dcache lock.

shrink_dcache_sb

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

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

LINUX

Kernel Hackers ManualJuly 2010

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_obtain_alias

LINUX

Kernel Hackers ManualJuly 2010

Name

`d_obtain_alias` — find or allocate a dentry for a given inode

Synopsis

```
struct dentry * d_obtain_alias (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).

d_splice_alias

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
struct dentry * d_splice_alias (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.

d_add_ci

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

`struct dentry * d_add_ci (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.

d_lookup

LINUX

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 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.

d_validate

LINUX

Kernel Hackers ManualJuly 2010

Name

`d_validate` — verify dentry provided from insecure source

Synopsis

```
int d_validate (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.

d_delete

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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_materialise_unique

LINUX

Kernel Hackers ManualJuly 2010

Name

d_materialise_unique — introduce an inode into the tree

Synopsis

```
struct dentry * d_materialise_unique (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

d_path

LINUX

Kernel Hackers ManualJuly 2010

Name

`d_path` — return the path of a dentry

Synopsis

```
char * d_path (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.

find_inode_number

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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_drop requires dentry->d_lock.

d_add

LINUX

Kernel Hackers ManualJuly 2010

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`.

d_add_unique

LINUX

Kernel Hackers ManualJuly 2010

Name

`d_add_unique` — add dentry to hash queues without aliasing

Synopsis

```
struct dentry * d_add_unique (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

LINUX

Kernel Hackers ManualJuly 2010

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. 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.

d_unhashed

LINUX

Kernel Hackers ManualJuly 2010

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.

1.3. Inode Handling

inode_init_always

LINUX

Kernel Hackers ManualJuly 2010

Name

`inode_init_always` — perform inode structure intialisation

Synopsis

```
int inode_init_always (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.

clear_inode

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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.

inode_add_to_lists

LINUX

Name

`inode_add_to_lists` — add a new inode to relevant lists

Synopsis

```
void inode_add_to_lists (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.

new_inode

LINUX

Kernel Hackers ManualJuly 2010

Name

`new_inode` — obtain an inode

Synopsis

```
struct inode * new_inode (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

iunique

LINUX

Kernel Hackers ManualJuly 2010

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.

ilookup5_nowait

LINUX

Kernel Hackers ManualJuly 2010

Name

`ilookup5_nowait` — search for an inode in the inode cache

Synopsis

```
struct inode * ilookup5_nowait (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.

ilookup5

LINUX

Kernel Hackers ManualJuly 2010

Name

`ilookup5` — search for an inode in the inode cache

Synopsis

```
struct inode * ilookup5 (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.

ilookup

LINUX

Kernel Hackers ManualJuly 2010

Name

`ilookup` — search for an inode in the inode cache

Synopsis

```
struct inode * ilookup (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.

iget5_locked

LINUX

Kernel Hackers ManualJuly 2010

Name

`iget5_locked` — obtain an inode from a mounted file system

Synopsis

```
struct inode * iget5_locked (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.

iget_locked

LINUX

Kernel Hackers ManualJuly 2010

Name

`iget_locked` — obtain an inode from a mounted file system

Synopsis

```
struct inode * iget_locked (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`.

__insert_inode_hash

LINUX

Kernel Hackers ManualJuly 2010

Name

`__insert_inode_hash` — hash an inode

Synopsis

```
void __insert_inode_hash (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.

remove_inode_hash

LINUX

Kernel Hackers ManualJuly 2010

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.

generic_detach_inode

LINUX

Kernel Hackers ManualJuly 2010

Name

generic_detach_inode — remove inode from inode lists

Synopsis

```
int generic_detach_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.

iput

LINUX

Kernel Hackers ManualJuly 2010

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 then freed and may also be destroyed.

Consequently, `iput` can sleep.

bmap

LINUX

Kernel Hackers ManualJuly 2010

Name

`bmap` — find a block number in a file

Synopsis

```
sector_t bmap (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.

touch_atime

LINUX

Name

`touch_atime` — update the access time

Synopsis

```
void touch_atime (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.

file_update_time

LINUX

Kernel Hackers ManualJuly 2010

Name

`file_update_time` — update mtime and ctime time

Synopsis

```
void file_update_time (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.

make_bad_inode

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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.

iget_failed

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

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

Arguments

inode

The inode to discard

Description

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

1.4. Registration and Superblocks

deactivate_super

LINUX

Kernel Hackers ManualJuly 2010

Name

`deactivate_super` — drop an active reference to superblock

Synopsis

```
void deactivate_super (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.

deactivate_locked_super

LINUX

Kernel Hackers ManualJuly 2010

Name

`deactivate_locked_super` — drop an active reference to superblock

Synopsis

```
void deactivate_locked_super (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.

generic_shutdown_super

LINUX

Name

`generic_shutdown_super` — common helper for ->`kill_sb`

Synopsis

```
void generic_shutdown_super (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.

sget

LINUX

Name

`sget` — find or create a superblock

Synopsis

```
struct super_block * sget (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

get_super

LINUX

Name

get_super — get the superblock of a device

Synopsis

```
struct super_block * get_super (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.

1.5. File Locks

posix_lock_file

LINUX

Name

`posix_lock_file` — Apply a POSIX-style lock to a file

Synopsis

```
int posix_lock_file (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.

posix_lock_file_wait

LINUX

Name

`posix_lock_file_wait` — Apply a POSIX-style lock to a file

Synopsis

```
int posix_lock_file_wait (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

locks_mandatory_area

LINUX

Name

`locks_mandatory_area` — Check for a conflicting lock

Synopsis

```
int locks_mandatory_area (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`.

__break_lease

LINUX

Kernel Hackers ManualJuly 2010

Name

`__break_lease` — revoke all outstanding leases on file

Synopsis

```
int __break_lease (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`.

lease_get_mtime

LINUX

Name

`lease_get_mtime` — get the last modified time of an inode

Synopsis

```
void lease_get_mtime (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.

generic_setlease

LINUX

Name

`generic_setlease` — sets a lease on an open file

Synopsis

```
int generic_setlease (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.

flock_lock_file_wait

LINUX

Kernel Hackers ManualJuly 2010

Name

`flock_lock_file_wait` — Apply a FLOCK-style lock to a file

Synopsis

```
int flock_lock_file_wait (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.

vfs_test_lock

LINUX

Kernel Hackers ManualJuly 2010

Name

`vfs_test_lock` — test file byte range lock

Synopsis

`int vfs_test_lock (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.

vfs_lock_file

LINUX

Kernel Hackers ManualJuly 2010

Name

`vfs_lock_file` — file byte range lock

Synopsis

`int vfs_lock_file (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.

posix_unblock_lock

LINUX

Kernel Hackers ManualJuly 2010

Name

`posix_unblock_lock` — stop waiting for a file lock

Synopsis

`int posix_unblock_lock (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.

vfs_cancel_lock

LINUX

Kernel Hackers ManualJuly 2010

Name

`vfs_cancel_lock` — file byte range unblock lock

Synopsis

```
int vfs_cancel_lock (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

lock_may_read

LINUX

Kernel Hackers ManualJuly 2010

Name

`lock_may_read` — checks that the region is free of locks

Synopsis

```
int lock_may_read (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.

lock_may_write

LINUX

Kernel Hackers ManualJuly 2010

Name

`lock_may_write` — checks that the region is free of locks

Synopsis

```
int lock_may_write (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.

locks_mandatory_locked

LINUX

Kernel Hackers ManualJuly 2010

Name

`locks_mandatory_locked` — Check for an active lock

Synopsis

```
int locks_mandatory_locked (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.

fcntl_getlease

LINUX

Kernel Hackers ManualJuly 2010

Name

`fcntl_getlease` — Enquire what lease is currently active

Synopsis

```
int fcntl_getlease (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.

fcntl_setlease

LINUX

Kernel Hackers ManualJuly 2010

Name

`fcntl_setlease` — sets a lease on an open file

Synopsis

```
int fcntl_setlease (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.

sys_flock

LINUX

Kernel Hackers ManualJuly 2010

Name

`sys_flock` — `flock` system call.

Synopsis

```
long sys_flock (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.

1.6. Other Functions

mpage_readpages

LINUX

Name

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

Synopsis

```
int mpage_readpages (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.

mpage_writepages

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
int mpage_writepages (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.

generic_permission

LINUX

Name

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

Synopsis

```
int generic_permission (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..

inode_permission

LINUX

Name

`inode_permission` — check for access rights to a given inode

Synopsis

```
int inode_permission (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.

file_permission

LINUX

Name

`file_permission` — check for additional access rights to a given file

Synopsis

```
int file_permission (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`.

path_get

LINUX

Kernel Hackers ManualJuly 2010

Name

`path_get` — get a reference to a path

Synopsis

```
void path_get (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.

path_put

LINUX

Kernel Hackers ManualJuly 2010

Name

`path_put` — put a reference to a path

Synopsis

```
void path_put (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.

release_open_intent

LINUX

Kernel Hackers ManualJuly 2010

Name

`release_open_intent` — free up open intent resources

Synopsis

```
void release_open_intent (struct nameidata * nd);
```

Arguments

nd

pointer to nameidata

vfs_path_lookup

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
int vfs_path_lookup (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

lookup_one_len

LINUX

Kernel Hackers ManualJuly 2010

Name

`lookup_one_len` — filesystem helper to lookup single pathname component

Synopsis

```
struct dentry * lookup_one_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.

filp_open

LINUX

Kernel Hackers ManualJuly 2010

Name

`filp_open` — open file and return file pointer

Synopsis

```
struct file * filp_open (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..

lookup_create

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
struct dentry * lookup_create (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.

sync_mapping_buffers

LINUX

Kernel Hackers ManualJuly 2010

Name

`sync_mapping_buffers` — write out & wait upon a mapping's "associated" buffers

Synopsis

```
int sync_mapping_buffers (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`.

mark_buffer_dirty

LINUX

Kernel Hackers ManualJuly 2010

Name

`mark_buffer_dirty` — mark a buffer_head as needing writeout

Synopsis

```
void mark_buffer_dirty (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.

__bread

LINUX

Name

`__bread` — reads a specified block and returns the bh

Synopsis

```
struct buffer_head * __bread (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.

block_invalidatepage

LINUX

Name

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

Synopsis

```
void block_invalidatepage (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.

ll_rw_block

LINUX

Name

`ll_rw_block` — level access to block devices (DEPRECATED)

Synopsis

```
void ll_rw_block (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.

bh_uptodate_or_lock

LINUX

Kernel Hackers ManualJuly 2010

Name

`bh_uptodate_or_lock` — Test whether the buffer is uptodate

Synopsis

```
int bh_uptodate_or_lock (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.

bh_submit_read

LINUX

Kernel Hackers ManualJuly 2010

Name

`bh_submit_read` — Submit a locked buffer for reading

Synopsis

```
int bh_submit_read (struct buffer_head * bh);
```

Arguments

bh

struct buffer_head

Description

Returns zero on success and -EIO on error.

bio_alloc_bioset

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_alloc_bioset` — allocate a bio for I/O

Synopsis

```
struct bio * bio_alloc_bioset (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.

Description

bio_alloc_bioset will 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.

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

bio_alloc

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_alloc` — allocate a new bio, memory pool backed

Synopsis

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

Arguments

gfp_mask

allocation mask to use

nr_iovecs

number of iovecs

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* and *bio_kmalloc*.

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.

RETURNS

Pointer to new bio on success, NULL on failure.

bio_kmalloc

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_kmalloc` — allocate a bio for I/O using `kmalloc`

Synopsis

```
struct bio * bio_kmalloc (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

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

bio_put

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_put` — release a reference to a bio

Synopsis

`void bio_put (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, bio_get or bio_clone. The last put of a bio will free it.

__bio_clone

LINUX

Kernel Hackers ManualJuly 2010

Name

`__bio_clone` — clone a bio

Synopsis

`void __bio_clone (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.

bio_clone

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_clone` — clone a bio

Synopsis

```
struct bio * bio_clone (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

bio_get_nr_vecs

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_get_nr_vecs` — return approx number of vecs

Synopsis

```
int bio_get_nr_vecs (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.

bio_add_pc_page

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_add_pc_page` — attempt to add page to bio

Synopsis

```
int bio_add_pc_page (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.

bio_add_page

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_add_page` — attempt to add page to bio

Synopsis

```
int bio_add_page (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.

bio_uncopy_user

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_uncopy_user` — finish previously mapped bio

Synopsis

```
int bio_uncopy_user (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.

bio_copy_user

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_copy_user` — copy user data to bio

Synopsis

```
struct bio * bio_copy_user (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.

bio_map_user

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_map_user` — map user address into bio

Synopsis

```
struct bio * bio_map_user (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.

bio_unmap_user

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_unmap_user` — unmap a bio

Synopsis

```
void bio_unmap_user (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.

bio_map_kern

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_map_kern` — map kernel address into bio

Synopsis

```
struct bio * bio_map_kern (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.

bio_copy_kern

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_copy_kern` — copy kernel address into bio

Synopsis

```
struct bio * bio_copy_kern (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.

bio_endio

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_endio` — end I/O on a bio

Synopsis

```
void bio_endio (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.

bio_sector_offset

LINUX

Kernel Hackers ManualJuly 2010

Name

`bio_sector_offset` — Find hardware sector offset in bio

Synopsis

```
sector_t bio_sector_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.

bioset_create

LINUX

Kernel Hackers ManualJuly 2010

Name

bioset_create — Create a bio_set

Synopsis

```
struct bio_set * bioset_create (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.

seq_open

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_open` — initialize sequential file

Synopsis

```
int seq_open (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”.

seq_read

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_read` — ->`read` method for sequential files.

Synopsis

```
ssize_t seq_read (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`

seq_lseek

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
loff_t seq_lseek (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`

seq_release

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_release` — free the structures associated with sequential file.

Synopsis

```
int seq_release (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.

seq_escape

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_escape` — print string into buffer, escaping some characters

Synopsis

```
int seq_escape (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.

mangle_path

LINUX

Kernel Hackers ManualJuly 2010

Name

`mangle_path` — mangle and copy path to buffer beginning

Synopsis

```
char * mangle_path (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.

seq_path

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_path` — seq_file interface to print a pathname

Synopsis

```
int seq_path (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.

seq_write

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_write` — write arbitrary data to buffer

Synopsis

```
int seq_write (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.

seq_hlist_start

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_hlist_start` — start an iteration of a hlist

Synopsis

```
struct hlist_node * seq_hlist_start (struct hlist_head * head,
loff_t pos);
```

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

Called at seq_file->op->`start`.

seq_hlist_start_head

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_hlist_start_head` — start an iteration of a hlist

Synopsis

```
struct hlist_node * seq_hlist_start_head (struct hlist_head *
head, loff_t pos);
```

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

Called at seq_file->op->`start`. Call this function if you want to print a header at the top of the output.

seq_hlist_next

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_hlist_next` — move to the next position of the hlist

Synopsis

```
struct hlist_node * seq_hlist_next (void * v, struct
hlist_head * head, loff_t * ppos);
```

Arguments

v

the current iterator

head

the head of the hlist

ppos

the current position

Description

Called at seq_file->op->`next`.

seq_hlist_start_rcu

LINUX

Kernel Hackers ManualJuly 2010

Name

seq_hlist_start_rcu — start an iteration of a hlist protected by RCU

Synopsis

```
struct hlist_node * seq_hlist_start_rcu (struct hlist_head *
head, loff_t pos);
```

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

Called at seq_file->op->start.

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as hlist_add_head_rcu as long as the traversal is guarded by rcu_read_lock.

seq_hlist_start_head_rcu

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_hlist_start_head_rcu` — start an iteration of a hlist protected by RCU

Synopsis

```
struct hlist_node * seq_hlist_start_head_rcu (struct
hlist_head * head, loff_t pos);
```

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

Called at seq_file->op->`start`. Call this function if you want to print a header at the top of the output.

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as `hlist_add_head_rcu` as long as the traversal is guarded by `rcu_read_lock`.

seq_hlist_next_rcu

LINUX

Kernel Hackers ManualJuly 2010

Name

`seq_hlist_next_rcu` — move to the next position of the hlist protected by RCU

Synopsis

```
struct hlist_node * seq_hlist_next_rcu (void * v, struct
hlist_head * head, loff_t * ppos);
```

Arguments

v

the current iterator

head

the head of the hlist

ppos

the current position

Description

Called at seq_file->op->`next`.

This list-traversal primitive may safely run concurrently with the _rcu list-mutation primitives such as `hlist_add_head_rcu` as long as the traversal is guarded by `rcu_read_lock`.

register_filesystem

LINUX

Kernel Hackers ManualJuly 2010

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

LINUX

Kernel Hackers ManualJuly 2010

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.

__mark_inode_dirty

LINUX

Kernel Hackers ManualJuly 2010

Name

`__mark_inode_dirty` — internal function

Synopsis

```
void __mark_inode_dirty (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.

writeback_inodes_sb

LINUX

Kernel Hackers ManualJuly 2010

Name

`writeback_inodes_sb` — writeback dirty inodes from given super_block

Synopsis

```
void writeback_inodes_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.

writeback_inodes_sb_if_idle

LINUX

Kernel Hackers ManualJuly 2010

Name

`writeback_inodes_sb_if_idle` — start writeback if none underway

Synopsis

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

Arguments

sb

the superblock

Description

Invoke writeback_inodes_sb if no writeback is currently underway. Returns 1 if writeback was started, 0 if not.

sync_inodes_sb

LINUX

Kernel Hackers ManualJuly 2010

Name

sync_inodes_sb — sync sb inode pages

Synopsis

```
void sync_inodes_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.

write_inode_now

LINUX

Kernel Hackers ManualJuly 2010

Name

`write_inode_now` — write an inode to disk

Synopsis

```
int write_inode_now (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.

sync_inode

LINUX

Kernel Hackers ManualJuly 2010

Name

`sync_inode` — write an inode and its pages to disk.

Synopsis

```
int sync_inode (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.

freeze_bdev

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
struct super_block * freeze_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.

thaw_bdev

LINUX

Kernel Hackers ManualJuly 2010

Name

`thaw_bdev` — - unlock filesystem

Synopsis

```
int thaw_bdev (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`.

bd_claim_by_disk

LINUX

Name

`bd_claim_by_disk` — wrapper function for `bd_claim_by_kobject`

Synopsis

```
int bd_claim_by_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.

bd_release_from_disk

LINUX

Name

`bd_release_from_disk` — wrapper function for `bd_release_from_kobject`

Synopsis

```
void bd_release_from_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.

check_disk_size_change

LINUX

Name

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

Synopsis

```
void check_disk_size_change (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.

revalidate_disk

LINUX

Kernel Hackers ManualJuly 2010

Name

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

Synopsis

```
int revalidate_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.

lookup_bdev

LINUX

Kernel Hackers ManualJuly 2010

Name

`lookup_bdev` — lookup a struct block_device by name

Synopsis

```
struct block_device * lookup_bdev (const char * pathname);
```

Arguments

pathname

special file representing the block device

Description

Get a reference to the blockdevice at *pathname* in the current namespace if possible and return it. Return ERR_PTR(error) otherwise.

open_bdev_exclusive

LINUX

Kernel Hackers ManualJuly 2010

Name

open_bdev_exclusive — open a block device by name and set it up for use

Synopsis

```
struct block_device * open_bdev_exclusive (const char * path,
fmode_t mode, void * holder);
```

Arguments

path

special file representing the block device

mode

FMODE_... combination to pass be used

holder

owner for exclusion

Description

Open the blockdevice described by the special file at *path*, claim it for the *holder*.

close_bdev_exclusive

LINUX

Kernel Hackers ManualJuly 2010

Name

`close_bdev_exclusive` — close a blockdevice opened by `open_bdev_exclusive`

Synopsis

```
void close_bdev_exclusive (struct block_device * bdev, fmode_t
mode);
```

Arguments

bdev

blockdevice to close

mode

mode, must match that used to open.

Description

This is the counterpart to `open_bdev_exclusive`.

Chapter 2. The proc filesystem

2.1. sysctl interface

register_sysctl_paths

LINUX

Kernel Hackers ManualJuly 2010

Name

`register_sysctl_paths` — register a sysctl table hierarchy

Synopsis

```
struct ctl_table_header * register_sysctl_paths (const struct
ctl_path * path, struct ctl_table * table);
```

Arguments

path

The path to the directory the sysctl table is in.

table

the top-level table structure

Description

Register a sysctl table hierarchy. *table* should be a filled in ctl_table array. A completely 0 filled entry terminates the table.

See __register_sysctl_paths for more details.

register_sysctl_table

LINUX

Kernel Hackers ManualJuly 2010

Name

`register_sysctl_table` — register a sysctl table hierarchy

Synopsis

```
struct ctl_table_header * register_sysctl_table (struct
ctl_table * table);
```

Arguments

`table`

the top-level table structure

Description

Register a sysctl table hierarchy. *`table`* should be a filled in ctl_table array. A completely 0 filled entry terminates the table.

See register_sysctl_paths for more details.

unregister_sysctl_table

LINUX

Kernel Hackers ManualJuly 2010

Name

`unregister_sysctl_table` — unregister a sysctl table hierarchy

Synopsis

```
void unregister_sysctl_table (struct ctl_table_header *
header);
```

Arguments

header

the header returned from register_sysctl_table

Description

Unregisters the sysctl table and all children. proc entries may not actually be removed until they are no longer used by anyone.

proc_dostring

LINUX

Name

`proc_dostring` — read a string sysctl

Synopsis

```
int proc_dostring (struct ctl_table * table, int write, void
__user * buffer, size_t * lenp, loff_t * ppos);
```

Arguments

table

the sysctl table

write

`TRUE` if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

Reads/writes a string from/to the user buffer. If the kernel buffer provided is not large enough to hold the string, the string is truncated. The copied string is `NULL-terminated`. If the string is being read by the user process, it is copied and a newline '\n' is added. It is truncated if the buffer is not large enough.

Returns 0 on success.

proc_dointvec

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_dointvec` — read a vector of integers

Synopsis

`int` **`proc_dointvec`** `(struct ctl_table *` *`table`*`, int` *`write`*`, void __user *` *`buffer`*`, size_t *` *`lenp`*`, loff_t *` *`ppos`*`);`

Arguments

`table`

the sysctl table

`write`

`TRUE` if this is a write to the sysctl file

`buffer`

the user buffer

`lenp`

the size of the user buffer

`ppos`

file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned int) integer values from/to the user buffer, treated as an ASCII string.

Returns 0 on success.

proc_dointvec_minmax

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_dointvec_minmax` — read a vector of integers with min/max values

Synopsis

```
int proc_dointvec_minmax (struct ctl_table * table, int write,
void __user * buffer, size_t * lenp, loff_t * ppos);
```

Arguments

table

the sysctl table

write

`TRUE` if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned int) integer values from/to the user buffer, treated as an ASCII string.

This routine will ensure the values are within the range specified by table->extra1 (min) and table->extra2 (max).

Returns 0 on success.

proc_doulongvec_minmax

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_doulongvec_minmax` — read a vector of long integers with min/max values

Synopsis

```
int proc_doulongvec_minmax (struct ctl_table * table, int
write, void __user * buffer, size_t * lenp, loff_t * ppos);
```

Arguments

table

the sysctl table

write

`TRUE` if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned long) unsigned long values from/to the user buffer, treated as an ASCII string.

This routine will ensure the values are within the range specified by table->extra1 (min) and table->extra2 (max).

Returns 0 on success.

proc_doulongvec_ms_jiffies_minmax

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_doulongvec_ms_jiffies_minmax` — read a vector of millisecond values with min/max values

Synopsis

```
int proc_doulongvec_ms_jiffies_minmax (struct ctl_table *
table, int write, void __user * buffer, size_t * lenp, loff_t
* ppos);
```

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned long) unsigned long values from/to the user buffer, treated as an ASCII string. The values are treated as milliseconds, and converted to jiffies when they are stored.

This routine will ensure the values are within the range specified by table->extra1 (min) and table->extra2 (max).

Returns 0 on success.

proc_dointvec_jiffies

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_dointvec_jiffies` — read a vector of integers as seconds

Synopsis

```
int proc_dointvec_jiffies (struct ctl_table * table, int
write, void __user * buffer, size_t * lenp, loff_t * ppos);
```

Arguments

table

the sysctl table

write

`TRUE` if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned int) integer values from/to the user buffer, treated as an ASCII string. The values read are assumed to be in seconds, and are converted into jiffies.

Returns 0 on success.

proc_dointvec_userhz_jiffies

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_dointvec_userhz_jiffies` — read a vector of integers as 1/USER_HZ seconds

Synopsis

```
int proc_dointvec_userhz_jiffies (struct ctl_table * table,
int write, void __user * buffer, size_t * lenp, loff_t *
ppos);
```

Arguments

table

the sysctl table

write

`TRUE` if this is a write to the sysctl file *buffer*

the user buffer

lenp

the size of the user buffer

ppos

pointer to the file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned int) integer values from/to the user buffer, treated as an ASCII string. The values read are assumed to be in 1/USER_HZ seconds, and are converted into jiffies.

Returns 0 on success.

proc_dointvec_ms_jiffies

LINUX

Kernel Hackers ManualJuly 2010

Name

`proc_dointvec_ms_jiffies` — read a vector of integers as 1 milliseconds

Synopsis

```
int proc_dointvec_ms_jiffies (struct ctl_table * table, int
write, void __user * buffer, size_t * lenp, loff_t * ppos);
```

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

the current position in the file

Description

Reads/writes up to table->maxlen/sizeof(unsigned int) integer values from/to the user buffer, treated as an ASCII string. The values read are assumed to be in 1/1000 seconds, and are converted into jiffies.

Returns 0 on success.

2.2. proc filesystem interface

proc_flush_task

LINUX

Name

`proc_flush_task` — Remove dcache entries for *`task`* from the /proc dcache.

Synopsis

```
void proc_flush_task (struct task_struct * task);
```

Arguments

`task`

task that should be flushed.

Description

When flushing dentries from proc, one needs to flush them from global proc (proc_mnt) and from all the namespaces' procs this task was seen in. This call is supposed to do all of this job.

Looks in the dcache for /proc/*`pid`* /proc/*`tgid`*/task/*`pid`* if either directory is present flushes it and all of it'ts children from the dcache.

It is safe and reasonable to cache /proc entries for a task until that task exits. After that they just clog up the dcache with useless entries, possibly causing useful dcache entries to be flushed instead. This routine is proved to flush those useless dcache entries at process exit time.

NOTE

This routine is just an optimization so it does not guarantee that no dcache entries will exist at process exit time it just makes it very unlikely that any will persist.

Chapter 3. The Filesystem for Exporting Kernel Objects

sysfs_create_file

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_create_file` — create an attribute file for an object.

Synopsis

```
int sysfs_create_file (struct kobject * kobj, const struct
attribute * attr);
```

Arguments

kobj

object we're creating for.

attr

attribute descriptor.

sysfs_add_file_to_group

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_add_file_to_group` — add an attribute file to a pre-existing group.

Synopsis

```
int sysfs_add_file_to_group (struct kobject * kobj, const
struct attribute * attr, const char * group);
```

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

group

group name.

sysfs_chmod_file

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_chmod_file` — update the modified mode value on an object attribute.

Synopsis

int **sysfs_chmod_file** (struct kobject * *kobj*, struct attribute * *attr*, mode_t *mode*);

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

mode

file permissions.

sysfs_remove_file

LINUX

Kernel Hackers ManualJuly 2010

Name

sysfs_remove_file — remove an object attribute.

Synopsis

void **sysfs_remove_file** (struct kobject * *kobj*, const struct attribute * *attr*);

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

Description

Hash the attribute name and kill the victim.

sysfs_remove_file_from_group

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_remove_file_from_group` — remove an attribute file from a group.

Synopsis

```
void sysfs_remove_file_from_group (struct kobject * kobj,
const struct attribute * attr, const char * group);
```

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

group

group name.

sysfs_schedule_callback

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_schedule_callback` — helper to schedule a callback for a kobject

Synopsis

```
int sysfs_schedule_callback (struct kobject * kobj, void
(*func) (void *), void * data, struct module * owner);
```

Arguments

kobj

object we're acting for.

func

callback function to invoke later.

data

argument to pass to *func*.

owner

module owning the callback code

Description

sysfs attribute methods must not unregister themselves or their parent kobject (which would amount to the same thing). Attempts to do so will deadlock, since unregistration is mutually exclusive with driver callbacks.

Instead methods can call this routine, which will attempt to allocate and schedule a workqueue request to call back *func* with *data* as its argument in the workqueue's process context. *kobj* will be pinned until *func* returns.

Returns 0 if the request was submitted, -ENOMEM if storage could not be allocated, -ENODEV if a reference to *owner* isn't available, -EAGAIN if a callback has already been scheduled for *kobj*.

sysfs_create_link

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_create_link` — create symlink between two objects.

Synopsis

```
int sysfs_create_link (struct kobject * kobj, struct kobject *
target, const char * name);
```

Arguments

kobj

object whose directory we're creating the link in.

target

object we're pointing to.

name

name of the symlink.

sysfs_remove_link

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_remove_link` — remove symlink in object's directory.

Synopsis

```
void sysfs_remove_link (struct kobject * kobj, const char *
name);
```

Arguments

kobj

object we're acting for.

name

name of the symlink to remove.

sysfs_create_bin_file

LINUX

Kernel Hackers ManualJuly 2010

Name

`sysfs_create_bin_file` — create binary file for object.

Synopsis

```
int sysfs_create_bin_file (struct kobject * kobj, const struct
bin_attribute * attr);
```

Arguments

kobj

object.

attr

attribute descriptor.

sysfs_remove_bin_file

LINUX

Name

`sysfs_remove_bin_file` — remove binary file for object.

Synopsis

```
void sysfs_remove_bin_file (struct kobject * kobj, const
struct bin_attribute * attr);
```

Arguments

kobj

object.

attr

attribute descriptor.

Chapter 4. The debugfs filesystem

4.1. debugfs interface

debugfs_create_file

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_file` — create a file in the debugfs filesystem

Synopsis

```
struct dentry * debugfs_create_file (const char * name, mode_t
mode, struct dentry * parent, void * data, const struct
file_operations * fops);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have.

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this paramater is NULL, then the file will be created in the root of the debugfs filesystem.

data

a pointer to something that the caller will want to get to later on. The inode.i_private pointer will point to this value on the `open` call.

fops

a pointer to a struct file_operations that should be used for this file.

Description

This is the basic "create a file" function for debugfs. It allows for a wide range of flexibility in creating a file, or a directory (if you want to create a directory, the `debugfs_create_dir` function is recommended to be used instead.)

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned.

debugfs_create_dir

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_dir` — create a directory in the debugfs filesystem

Synopsis

```
struct dentry * debugfs_create_dir (const char * name, struct
dentry * parent);
```

Arguments

name

a pointer to a string containing the name of the directory to create.

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this paramater is NULL, then the directory will be created in the root of the debugfs filesystem.

Description

This function creates a directory in debugfs with the given name.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned.

debugfs_create_symlink

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_symlink` — create a symbolic link in the debugfs filesystem

Synopsis

```
struct dentry * debugfs_create_symlink (const char * name,
struct dentry * parent, const char * target);
```

Arguments

name

a pointer to a string containing the name of the symbolic link to create.

parent

a pointer to the parent dentry for this symbolic link. This should be a directory dentry if set. If this paramater is NULL, then the symbolic link will be created in the root of the debugfs filesystem.

target

a pointer to a string containing the path to the target of the symbolic link.

Description

This function creates a symbolic link with the given name in debugfs that links to the given target path.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the symbolic link is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned.

debugfs_remove

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_remove` — removes a file or directory from the debugfs filesystem

Synopsis

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

Arguments

dentry

a pointer to a the dentry of the file or directory to be removed.

Description

This function removes a file or directory in debugfs that was previously created with a call to another debugfs function (like `debugfs_create_file` or variants thereof.)

This function is required to be called in order for the file to be removed, no automatic cleanup of files will happen when a module is removed, you are responsible here.

debugfs_remove_recursive

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_remove_recursive` — recursively removes a directory

Synopsis

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

Arguments

dentry

a pointer to a the dentry of the directory to be removed.

Description

This function recursively removes a directory tree in debugfs that was previously created with a call to another debugfs function (like `debugfs_create_file` or variants thereof.)

This function is required to be called in order for the file to be removed, no automatic cleanup of files will happen when a module is removed, you are responsible here.

debugfs_rename

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_rename` — rename a file/directory in the debugfs filesystem

Synopsis

```
struct dentry * debugfs_rename (struct dentry * old_dir,
struct dentry * old_dentry, struct dentry * new_dir, const
char * new_name);
```

Arguments

old_dir

a pointer to the parent dentry for the renamed object. This should be a directory dentry.

old_dentry

dentry of an object to be renamed.

new_dir

a pointer to the parent dentry where the object should be moved. This should be a directory dentry.

new_name

a pointer to a string containing the target name.

Description

This function renames a file/directory in debugfs. The target must not exist for rename to succeed.

This function will return a pointer to old_dentry (which is updated to reflect renaming) if it succeeds. If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned.

debugfs_initialized

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_initialized` — Tells whether debugfs has been registered

Synopsis

```
bool debugfs_initialized ( void);
```

Arguments

void

no arguments

debugfs_create_u8

LINUX

Kernel Hackers ManualJuly 2010

Name

debugfs_create_u8 — create a debugfs file that is used to read and write an unsigned 8-bit value

Synopsis

```
struct dentry * debugfs_create_u8 (const char * name, mode_t
mode, struct dentry * parent, u8 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is NULL, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

Description

This function creates a file in debugfs with the given name that contains the value of the variable *value*. If the *mode* variable is so set, it can be read from, and written to.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the debugfs_remove function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, NULL will be returned.

If debugfs is not enabled in the kernel, the value -ENODEV will be returned. It is not wise to check for this value, but rather, check for NULL or !NULL instead as to eliminate the need for #ifdef in the calling code.

debugfs_create_u16

LINUX

Kernel Hackers ManualJuly 2010

Name

debugfs_create_u16 — create a debugfs file that is used to read and write an unsigned 16-bit value

Synopsis

```
struct dentry * debugfs_create_u16 (const char * name, mode_t
mode, struct dentry * parent, u16 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

Description

This function creates a file in debugfs with the given name that contains the value of the variable *value*. If the *mode* variable is so set, it can be read from, and written to.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned. It is not wise to check for this value, but rather, check for `NULL` or !`NULL` instead as to eliminate the need for #ifdef in the calling code.

debugfs_create_u32

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_u32` — create a debugfs file that is used to read and write an unsigned 32-bit value

Synopsis

```
struct dentry * debugfs_create_u32 (const char * name, mode_t
mode, struct dentry * parent, u32 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

Description

This function creates a file in debugfs with the given name that contains the value of the variable *value*. If the *mode* variable is so set, it can be read from, and written to.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned. It is not wise to check for this value, but rather, check for `NULL` or !`NULL` instead as to eliminate the need for #ifdef in the calling code.

debugfs_create_u64

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_u64` — create a debugfs file that is used to read and write an unsigned 64-bit value

Synopsis

```
struct dentry * debugfs_create_u64 (const char * name, mode_t
mode, struct dentry * parent, u64 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have *parent*

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is NULL, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

Description

This function creates a file in debugfs with the given name that contains the value of the variable *value*. If the *mode* variable is so set, it can be read from, and written to.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the debugfs_remove function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, NULL will be returned.

If debugfs is not enabled in the kernel, the value -ENODEV will be returned. It is not wise to check for this value, but rather, check for NULL or !NULL instead as to eliminate the need for #ifdef in the calling code.

debugfs_create_x8

LINUX

Kernel Hackers ManualJuly 2010

Name

debugfs_create_x8 — create a debugfs file that is used to read and write an unsigned 8-bit value

Synopsis

```
struct dentry * debugfs_create_x8 (const char * name, mode_t
mode, struct dentry * parent, u8 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

debugfs_create_x16

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_x16` — create a debugfs file that is used to read and write an unsigned 16-bit value

Synopsis

```
struct dentry * debugfs_create_x16 (const char * name, mode_t
mode, struct dentry * parent, u16 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

debugfs_create_x32

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_x32` — create a debugfs file that is used to read and write an unsigned 32-bit value

Synopsis

```
struct dentry * debugfs_create_x32 (const char * name, mode_t
mode, struct dentry * parent, u32 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

debugfs_create_size_t

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_size_t` — create a debugfs file that is used to read and write an size_t value

Synopsis

```
struct dentry * debugfs_create_size_t (const char * name,
mode_t mode, struct dentry * parent, size_t * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

debugfs_create_bool

LINUX

Kernel Hackers ManualJuly 2010

Name

`debugfs_create_bool` — create a debugfs file that is used to read and write a boolean value

Synopsis

```
struct dentry * debugfs_create_bool (const char * name, mode_t mode, struct dentry * parent, u32 * value);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

value

a pointer to the variable that the file should read to and write from.

Description

This function creates a file in debugfs with the given name that contains the value of the variable *value*. If the *mode* variable is so set, it can be read from, and written to.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned. It is not wise to check for this value, but rather, check for `NULL` or !`NULL` instead as to eliminate the need for #ifdef in the calling code.

debugfs_create_blob

LINUX

Name

`debugfs_create_blob` — create a debugfs file that is used to read a binary blob

Synopsis

```
struct dentry * debugfs_create_blob (const char * name, mode_t
mode, struct dentry * parent, struct debugfs_blob_wrapper *
blob);
```

Arguments

name

a pointer to a string containing the name of the file to create.

mode

the permission that the file should have

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this parameter is `NULL`, then the file will be created in the root of the debugfs filesystem.

blob

a pointer to a struct debugfs_blob_wrapper which contains a pointer to the blob data and the size of the data.

Description

This function creates a file in debugfs with the given name that exports *blob*->data as a binary blob. If the *mode* variable is so set it can be read from. Writing is not supported.

This function will return a pointer to a dentry if it succeeds. This pointer must be passed to the `debugfs_remove` function when the file is to be removed (no automatic cleanup happens if your module is unloaded, you are responsible here.) If an error occurs, `NULL` will be returned.

If debugfs is not enabled in the kernel, the value -`ENODEV` will be returned. It is not wise to check for this value, but rather, check for `NULL` or !`NULL` instead as to eliminate the need for #ifdef in the calling code.

Chapter 5. The Linux Journalling API

5.1. Overview

5.1.1. Details

The journalling layer is easy to use. You need to first of all create a journal_t data structure. There are two calls to do this dependent on how you decide to allocate the physical media on which the journal resides. The journal_init_inode() call is for journals stored in filesystem inodes, or the journal_init_dev() call can be use for journal stored on a raw device (in a continuous range of blocks). A journal_t is a typedef for a struct pointer, so when you are finally finished make sure you call journal_destroy() on it to free up any used kernel memory.

Once you have got your journal_t object you need to 'mount' or load the journal file, unless of course you haven't initialised it yet - in which case you need to call journal_create().

Most of the time however your journal file will already have been created, but before you load it you must call journal_wipe() to empty the journal file. Hang on, you say , what if the filesystem wasn't cleanly umount()'d . Well, it is the job of the client file system to detect this and skip the call to journal_wipe().

In either case the next call should be to journal_load() which prepares the journal file for use. Note that journal_wipe(..,0) calls journal_skip_recovery() for you if it detects any outstanding transactions in the journal and similarly journal_load() will call journal_recover() if necessary. I would advise reading fs/ext3/super.c for examples on this stage. [RGG: Why is the journal_wipe() call necessary - doesn't this needlessly complicate the API. Or isn't a good idea for the journal layer to hide dirty mounts from the client fs]

Now you can go ahead and start modifying the underlying filesystem. Almost.

You still need to actually journal your filesystem changes, this is done by wrapping them into transactions. Additionally you also need to wrap the modification of each of the buffers with calls to the journal layer, so it knows what the modifications you are actually making are. To do this use journal_start() which returns a transaction handle.

journal_start() and its counterpart journal_stop(), which indicates the end of a transaction are nestable calls, so you can reenter a transaction if necessary, but remember you must call journal_stop() the same number of times as journal_start() before the transaction is completed (or more accurately leaves the update phase). Ext3/VFS makes use of this feature to simplify quota support.

Inside each transaction you need to wrap the modifications to the individual buffers (blocks). Before you start to modify a buffer you need to call journal_get_{create,write,undo}_access() as appropriate, this allows the journalling layer to copy the unmodified data if it needs to. After all the buffer may be part of a previously uncommitted transaction. At this point you are at last ready to modify a buffer, and once you are have done so you need to call journal_dirty_{meta,}data(). Or if you've asked for access to a buffer you now know is now longer required to be pushed back on the device you can call journal_forget() in much the same way as you might have used bforget() in the past.

A journal_flush() may be called at any time to commit and checkpoint all your transactions.

Then at umount time , in your put_super() (2.4) or write_super() (2.5) you can then call journal_destroy() to clean up your in-core journal object.

Unfortunately there a couple of ways the journal layer can cause a deadlock. The first thing to note is that each task can only have a single outstanding transaction at any one time, remember nothing commits until the outermost journal_stop(). This means you must complete the transaction at the end of each file/inode/address etc. operation you perform, so that the journalling system isn't re-entered on another journal. Since transactions can't be nested/batched across differing journals, and another filesystem other than yours (say ext3) may be modified in a later syscall.

The second case to bear in mind is that journal_start() can block if there isn't enough space in the journal for your transaction (based on the passed nblocks param) - when it blocks it merely(!) needs to wait for transactions to complete and be committed from other tasks, so essentially we are waiting for journal_stop(). So to avoid deadlocks you must treat journal_start/stop() as if they were semaphores and include them in your semaphore ordering rules to prevent deadlocks. Note that journal_extend() has similar blocking behaviour to journal_start() so you can deadlock here just as easily as on journal_start().

Try to reserve the right number of blocks the first time. ;-). This will be the maximum number of blocks you are going to touch in this transaction. I advise having a look at at least ext3_jbd.h to see the basis on which ext3 uses to make these decisions.

Another wriggle to watch out for is your on-disk block allocation strategy. why? Because, if you undo a delete, you need to ensure you haven't reused any of the freed blocks in a later transaction. One simple way of doing this is make sure any blocks you allocate only have checkpointed transactions listed against them. Ext3 does this in ext3_test_allocatable().

Lock is also providing through journal_{un,}lock_updates(), ext3 uses this when it wants a window with a clean and stable fs for a moment. eg.

```
journal_lock_updates() //stop new stuff happening..
journal_flush()        // checkpoint everything.
..do stuff on stable fs
journal_unlock_updates() // carry on with filesystem use.
```

The opportunities for abuse and DOS attacks with this should be obvious, if you allow unprivileged userspace to trigger codepaths containing these calls.

A new feature of jbd since 2.5.25 is commit callbacks with the new journal_callback_set() function you can now ask the journalling layer to call you back when the transaction is finally committed to disk, so that you can do some of your own management. The key to this is the journal_callback struct, this maintains the internal callback information but you can extend it like this:-

```
struct  myfs_callback_s {
 //Data structure element required by jbd..
 struct journal_callback for_jbd;
 // Stuff for myfs allocated together.
 myfs_inode*    i_commited;

}
```

this would be useful if you needed to know when data was committed to a particular inode.

5.1.2. Summary

Using the journal is a matter of wrapping the different context changes, being each mount, each modification (transaction) and each changed buffer to tell the journalling layer about them.

Here is a some pseudo code to give you an idea of how it works, as an example.

```
 journal_t* my_jnrl = journal_create();
 journal_init_{dev,inode}(jnrl,...)
 if (clean) journal_wipe();
 journal_load();

  foreach(transaction) { /*transactions must be
                          completed before
                          a syscall returns to
                          userspace*/

         handle_t * xct=journal_start(my_jnrl);
         foreach(bh) {
               journal_get_{create,write,undo}_access(xact,bh);
               if ( myfs_modify(bh) ) { /* returns true
```

```
                                    if makes changes */
                        journal_dirty_{meta,}data(xact,bh);
                } else {
                        journal_forget(bh);
                }
        }
        journal_stop(xct);
}
journal_destroy(my_jrnl);
```

5.2. Data Types

The journalling layer uses typedefs to 'hide' the concrete definitions of the structures used. As a client of the JBD layer you can just rely on the using the pointer as a magic cookie of some sort. Obviously the hiding is not enforced as this is 'C'.

5.2.1. Structures

typedef handle_t

LINUX

Kernel Hackers ManualJuly 2010

Name

`typedef handle_t` — The handle_t type represents a single atomic update being performed by some process.

Synopsis

```
typedef handle_t;
```

Description

All filesystem modifications made by the process go through this handle. Recursive operations (such as quota operations) are gathered into a single update.

The buffer credits field is used to account for journaled buffers being modified by the running process. To ensure that there is enough log space for all outstanding operations, we need to limit the number of outstanding buffers possible at any time. When the operation completes, any buffer credits not used are credited back to the transaction, so that at all times we know how many buffers the outstanding updates on a transaction might possibly touch.

This is an opaque datatype.

typedef journal_t

LINUX

Kernel Hackers ManualJuly 2010

Name

`typedef journal_t` — The journal_t maintains all of the journaling state information for a single filesystem.

Synopsis

```
typedef journal_t;
```

Description

journal_t is linked to from the fs superblock structure.

We use the journal_t to keep track of all outstanding transaction activity on the filesystem, and to manage the state of the log writing process.

This is an opaque datatype.

struct handle_s

LINUX

Kernel Hackers ManualJuly 2010

Name

`struct handle_s` — this is the concrete type associated with handle_t.

Synopsis

```
struct handle_s {
  transaction_t * h_transaction;
  int h_buffer_credits;
  int h_ref;
  int h_err;
  unsigned int h_sync:1;
  unsigned int h_jdata:1;
  unsigned int h_aborted:1;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
  struct lockdep_map h_lockdep_map;
#endif
};
```

Members

h_transaction

Which compound transaction is this update a part of?

h_buffer_credits

Number of remaining buffers we are allowed to dirty.

h_ref

Reference count on this handle

h_err

Field for caller's use to track errors through large fs operations h_sync

flag for sync-on-close

h_jdata

flag to force data journaling

h_aborted

flag indicating fatal error on handle

h_lockdep_map

lockdep info for debugging lock problems

struct journal_s

LINUX

Kernel Hackers ManualJuly 2010

Name

`struct journal_s` — this is the concrete type associated with journal_t.

Synopsis

```
struct journal_s {
  unsigned long j_flags;
  int j_errno;
  struct buffer_head * j_sb_buffer;
  journal_superblock_t * j_superblock;
  int j_format_version;
  spinlock_t j_state_lock;
  int j_barrier_count;
  struct mutex j_barrier;
  transaction_t * j_running_transaction;
  transaction_t * j_committing_transaction;
  transaction_t * j_checkpoint_transactions;
  wait_queue_head_t j_wait_transaction_locked;
  wait_queue_head_t j_wait_logspace;
  wait_queue_head_t j_wait_done_commit;
```

```
    wait_queue_head_t j_wait_checkpoint;
    wait_queue_head_t j_wait_commit;
    wait_queue_head_t j_wait_updates;
    struct mutex j_checkpoint_mutex;
    unsigned int j_head;
    unsigned int j_tail;
    unsigned int j_free;
    unsigned int j_first;
    unsigned int j_last;
    struct block_device * j_dev;
    int j_blocksize;
    unsigned int j_blk_offset;
    struct block_device * j_fs_dev;
    unsigned int j_maxlen;
    spinlock_t j_list_lock;
    struct inode * j_inode;
    tid_t j_tail_sequence;
    tid_t j_transaction_sequence;
    tid_t j_commit_sequence;
    tid_t j_commit_request;
    __u8 j_uuid[16];
    struct task_struct * j_task;
    int j_max_transaction_buffers;
    unsigned long j_commit_interval;
    struct timer_list j_commit_timer;
    spinlock_t j_revoke_lock;
    struct jbd_revoke_table_s * j_revoke;
    struct jbd_revoke_table_s * j_revoke_table[2];
    struct buffer_head ** j_wbuf;
    int j_wbufsize;
    pid_t j_last_sync_writer;
    u64 j_average_commit_time;
    void * j_private;
};
```

Members

j_flags

General journaling state flags

j_errno

Is there an outstanding uncleared error on the journal (from a prior abort)?

j_sb_buffer

First part of superblock buffer

j_superblock

Second part of superblock buffer

j_format_version

Version of the superblock format

j_state_lock

Protect the various scalars in the journal

j_barrier_count

Number of processes waiting to create a barrier lock

j_barrier

The barrier lock itself

j_running_transaction

The current running transaction..

j_committing_transaction

the transaction we are pushing to disk

j_checkpoint_transactions

a linked circular list of all transactions waiting for checkpointing

j_wait_transaction_locked

Wait queue for waiting for a locked transaction to start committing, or for a barrier lock to be released

j_wait_logspace

Wait queue for waiting for checkpointing to complete

j_wait_done_commit

Wait queue for waiting for commit to complete

j_wait_checkpoint

Wait queue to trigger checkpointing

j_wait_commit

Wait queue to trigger commit

j_wait_updates

Wait queue to wait for updates to complete

j_checkpoint_mutex

Mutex for locking against concurrent checkpoints

j_head

Journal head - identifies the first unused block in the journal

j_tail

Journal tail - identifies the oldest still-used block in the journal.

j_free

Journal free - how many free blocks are there in the journal?

j_first

The block number of the first usable block

j_last

The block number one beyond the last usable block

j_dev

Device where we store the journal

j_blocksize

blocksize for the location where we store the journal.

j_blk_offset

starting block offset for into the device where we store the journal

j_fs_dev

Device which holds the client fs. For internal journal this will be equal to j_dev

j_maxlen

Total maximum capacity of the journal region on disk.

j_list_lock

Protects the buffer lists and internal buffer state.

j_inode

Optional inode where we store the journal. If present, all journal block numbers are mapped into this inode via `bmap`.

j_tail_sequence

Sequence number of the oldest transaction in the log

j_transaction_sequence

Sequence number of the next transaction to grant

j_commit_sequence

Sequence number of the most recently committed transaction

j_commit_request

Sequence number of the most recent transaction wanting commit

j_uuid[16]

Uuid of client object.

j_task

Pointer to the current commit thread for this journal

j_max_transaction_buffers

Maximum number of metadata buffers to allow in a single compound commit transaction

j_commit_interval

What is the maximum transaction lifetime before we begin a commit?

j_commit_timer

The timer used to wakeup the commit thread

j_revoke_lock

Protect the revoke table

j_revoke

The revoke table - maintains the list of revoked blocks in the current transaction.

j_revoke_table[2]

alternate revoke tables for j_revoke

j_wbuf

array of buffer_heads for journal_commit_transaction

j_wbufsize

maximum number of buffer_heads allowed in j_wbuf, the number that will fit in j_blocksize

j_last_sync_writer

most recent pid which did a synchronous write

j_average_commit_time

the average amount of time in nanoseconds it takes to commit a transaction to the disk.

j_private

An opaque pointer to fs-private information.

5.3. Functions

The functions here are split into two groups those that affect a journal as a whole, and those which are used to manage transactions

5.3.1. Journal Level

journal_init_dev

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_init_dev` — creates and initialises a journal structure

Synopsis

```
journal_t * journal_init_dev (struct block_device * bdev,
struct block_device * fs_dev, int start, int len, int
blocksize);
```

Arguments

bdev

Block device on which to create the journal

fs_dev

Device which hold journalled filesystem for this journal.

start

Block nr Start of journal.

len

Length of the journal in blocks.

blocksize

blocksize of journalling device

Returns

a newly created journal_t *

journal_init_dev creates a journal which maps a fixed contiguous range of blocks on an arbitrary block device.

journal_init_inode

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_init_inode` — creates a journal which maps to a inode.

Synopsis

```
journal_t * journal_init_inode (struct inode * inode);
```

Arguments

inode

An inode to create the journal in

Description

journal_init_inode creates a journal which maps an on-disk inode as the journal. The inode must exist already, must support `bmap` and must have all data blocks preallocated.

journal_create

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_create` — Initialise the new journal file

Synopsis

```
int journal_create (journal_t * journal);
```

Arguments

journal

Journal to create. This structure must have been initialised

Description

Given a journal_t structure which tells us which disk blocks we can use, create a new journal superblock and initialise all of the journal fields from scratch.

journal_load

LINUX

Kernel Hackers ManualJuly 2010

Name

journal_load — Read journal from disk.

Synopsis

```
int journal_load (journal_t * journal);
```

Arguments

journal

Journal to act on.

Description

Given a journal_t structure which tells us which disk blocks contain a journal, read the journal from disk to initialise the in-memory structures.

journal_destroy

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_destroy` — Release a journal_t structure.

Synopsis

```
int journal_destroy (journal_t * journal);
```

Arguments

journal

Journal to act on.

Description

Release a journal_t structure once it is no longer in use by the journaled object. Return <0 if we couldn't clean up the journal.

journal_check_used_features

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_check_used_features` — Check if features specified are used.

Synopsis

```
int journal_check_used_features (journal_t * journal, unsigned
long compat, unsigned long ro, unsigned long incompat);
```

Arguments

journal

Journal to check.

compat

bitmask of compatible features

ro

bitmask of features that force read-only mount

incompat

bitmask of incompatible features

Description

Check whether the journal uses all of a given set of features. Return true (non-zero) if it does.

journal_check_available_features

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_check_available_features` — Check feature set in journalling layer

Synopsis

int **journal_check_available_features** (journal_t * *journal*, unsigned long *compat*, unsigned long *ro*, unsigned long *incompat*);

Arguments

journal

Journal to check.

compat

bitmask of compatible features

ro

bitmask of features that force read-only mount *incompat*

bitmask of incompatible features

Description

Check whether the journaling code supports the use of all of a given set of features on this journal. Return true

journal_set_features

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_set_features` — Mark a given journal feature in the superblock

Synopsis

```
int journal_set_features (journal_t * journal, unsigned long
compat, unsigned long ro, unsigned long incompat);
```

Arguments

journal

Journal to act on.

compat

bitmask of compatible features

ro

bitmask of features that force read-only mount

incompat

bitmask of incompatible features

Description

Mark a given journal feature as present on the superblock. Returns true if the requested features could be set.

journal_update_format

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_update_format` — Update on-disk journal structure.

Synopsis

```
int journal_update_format (journal_t * journal);
```

Arguments

journal

Journal to act on.

Description

Given an initialised but unloaded journal struct, poke about in the on-disk structure to update it to the most recent supported version.

journal_flush

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_flush` — Flush journal

Synopsis

```
int journal_flush (journal_t * journal);
```

Arguments

`journal`

Journal to act on.

Description

Flush all data for a given journal to disk and empty the journal. Filesystems can use this when remounting readonly to ensure that recovery does not need to happen on remount.

journal_wipe

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_wipe` — Wipe journal contents

Synopsis

```
int journal_wipe (journal_t * journal, int write);
```

Arguments

journal

Journal to act on.

write

flag (see below)

Description

Wipe out all of the contents of a journal, safely. This will produce a warning if the journal contains any valid recovery information. Must be called between journal_init_*() and `journal_load`.

If 'write' is non-zero, then we wipe out the journal on disk; otherwise we merely suppress recovery.

journal_abort

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_abort` — Shutdown the journal immediately.

Synopsis

```
void journal_abort (journal_t * journal, int errno);
```

Arguments

journal

the journal to shutdown.

errno

an error number to record in the journal indicating the reason for the shutdown.

Description

Perform a complete, immediate shutdown of the ENTIRE journal (not of a single transaction). This operation cannot be undone without closing and reopening the journal.

The journal_abort function is intended to support higher level error recovery mechanisms such as the ext2/ext3 remount-readonly error mode.

Journal abort has very specific semantics. Any existing dirty, unjournaled buffers in the main filesystem will still be written to disk by bdflush, but the journaling mechanism will be suspended immediately and no further transaction commits will be honoured.

Any dirty, journaled buffers will be written back to disk without hitting the journal. Atomicity cannot be guaranteed on an aborted filesystem, but we _do_ attempt to leave as much data as possible behind for fsck to use for cleanup.

Any attempt to get a new transaction handle on a journal which is in ABORT state will just result in an -EROFS error return. A journal_stop on an existing handle will return -EIO if we have entered abort state during the update.

Recursive transactions are not disturbed by journal abort until the final journal_stop, which will receive the -EIO error.

Finally, the journal_abort call allows the caller to supply an errno which will be recorded (if possible) in the journal superblock. This allows a client to record failure conditions in the middle of a transaction without having to complete the transaction to record the failure to disk. ext3_error, for example, now uses this functionality.

Errors which originate from within the journaling layer will NOT supply an errno; a null errno implies that absolutely no further writes are done to the journal (unless there are any already in progress).

journal_errno

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_errno` — returns the journal's error state.

Synopsis

```
int journal_errno (journal_t * journal);
```

Arguments

journal

journal to examine.

Description

This is the errno numbet set with `journal_abort`, the last time the journal was mounted - if the journal was stopped without calling abort this will be 0.

If the journal has been aborted on this mount time -EROFS will be returned.

journal_clear_err

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_clear_err` — clears the journal's error state

Synopsis

```
int journal_clear_err (journal_t * journal);
```

Arguments

journal

journal to act on.

Description

An error must be cleared or Acked to take a FS out of readonly mode.

journal_ack_err

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_ack_err` — Ack journal err.

Synopsis

```
void journal_ack_err (journal_t * journal);
```

Arguments

journal

journal to act on.

Description

An error must be cleared or Acked to take a FS out of readonly mode.

journal_recover

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_recover` — recovers a on-disk journal

Synopsis

```
int journal_recover (journal_t * journal);
```

Arguments

journal

the journal to recover

Description

The primary function for recovering the log contents when mounting a journaled device.

Recovery is done in three passes. In the first pass, we look for the end of the log. In the second, we assemble the list of revoke blocks. In the third and final pass, we replay any un-revoked blocks in the log.

journal_skip_recovery

LINUX

Name

`journal_skip_recovery` — Start journal and wipe exiting records

Synopsis

```
int journal_skip_recovery (journal_t * journal);
```

Arguments

journal

journal to startup

Description

Locate any valid recovery information from the journal and set up the journal structures in memory to ignore it (presumably because the caller has evidence that it is out of date). This function does'nt appear to be exorted..

We perform one pass over the journal to allow us to tell the user how much recovery information is being erased, and to let us initialise the journal transaction sequence numbers to the next unused ID.

5.3.2. Transasction Level

journal_start

LINUX

Name

`journal_start` — Obtain a new handle.

Synopsis

```
handle_t * journal_start (journal_t * journal, int nblocks);
```

Arguments

journal

Journal to start transaction on.

nblocks

number of block buffer we might modify

Description

We make sure that the transaction can guarantee at least nblocks of modified buffers in the log. We block until the log can guarantee that much space.

This function is visible to journal users (like ext3fs), so is not called with the journal already locked.

Return a pointer to a newly allocated handle, or NULL on failure

journal_extend

LINUX

Name

`journal_extend` — extend buffer credits.

Synopsis

```
int journal_extend (handle_t * handle, int nblocks);
```

Arguments

handle

handle to 'extend'

nblocks

nr blocks to try to extend by.

Description

Some transactions, such as large extends and truncates, can be done atomically all at once or in several stages. The operation requests a credit for a number of buffer modications in advance, but can extend its credit if it needs more.

journal_extend tries to give the running handle more buffer credits. It does not guarantee that allocation - this is a best-effort only. The calling process MUST be able to deal cleanly with a failure to extend here.

Return 0 on success, non-zero on failure.

return code < 0 implies an error return code > 0 implies normal transaction-full status.

journal_restart

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_restart` — restart a handle.

Synopsis

```
int journal_restart (handle_t * handle, int nblocks);
```

Arguments

handle

handle to restart

nblocks

nr credits requested

Description

Restart a handle for a multi-transaction filesystem operation.

If the `journal_extend` call above fails to grant new buffer credits to a running handle, a call to journal_restart will commit the handle's transaction so far and reattach the handle to a new transaction capabable of guaranteeing the requested number of credits.

journal_lock_updates

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_lock_updates` — establish a transaction barrier.

Synopsis

```
void journal_lock_updates (journal_t * journal);
```

Arguments

journal

Journal to establish a barrier on.

Description

This locks out any further updates from being started, and blocks until all existing updates have completed, returning only once the journal is in a quiescent state with no updates running.

The journal lock should not be held on entry.

journal_unlock_updates

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_unlock_updates` — release barrier

Synopsis

```
void journal_unlock_updates (journal_t * journal);
```

Arguments

journal

Journal to release the barrier on.

Description

Release a transaction barrier obtained with `journal_lock_updates`.

Should be called without the journal lock held.

journal_get_write_access

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_get_write_access` — notify intent to modify a buffer for metadata (not data) update.

Synopsis

```
int journal_get_write_access (handle_t * handle, struct
buffer_head * bh);
```

Arguments

handle

transaction to add buffer modifications to

bh

bh to be used for metadata writes

Description

Returns an error code or 0 on success.

In full data journalling mode the buffer may be of type BJ_AsyncData, because we're `write`ing a buffer which is also part of a shared mapping.

journal_get_create_access

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_get_create_access` — notify intent to use newly created bh

Synopsis

```
int journal_get_create_access (handle_t * handle, struct
buffer_head * bh);
```

Arguments

handle

transaction to new buffer to

bh

new buffer.

Description

Call this if you create a new bh.

journal_get_undo_access

LINUX

Kernel Hackers ManualJuly 2010

Name

journal_get_undo_access — Notify intent to modify metadata with non-rewindable consequences

Synopsis

```
int journal_get_undo_access (handle_t * handle, struct
buffer_head * bh);
```

Arguments

handle

transaction

bh

buffer to undo

Description

Sometimes there is a need to distinguish between metadata which has been committed to disk and that which has not. The ext3fs code uses this for freeing and allocating space, we have to make sure that we do not reuse freed space until the deallocation has been committed, since if we overwrote that space we would make the delete un-rewindable in case of a crash.

To deal with that, journal_get_undo_access requests write access to a buffer for parts of non-rewindable operations such as delete operations on the bitmaps. The journaling code must keep a copy of the buffer's contents prior to the undo_access call until such time as we know that the buffer has definitely been committed to disk.

We never need to know which transaction the committed data is part of, buffers touched here are guaranteed to be dirtied later and so will be committed to a new transaction in due course, at which point we can discard the old committed data pointer.

Returns error number or 0 on success.

journal_dirty_data

LINUX

Name

`journal_dirty_data` — mark a buffer as containing dirty data to be flushed

Synopsis

```
int journal_dirty_data (handle_t * handle, struct buffer_head
* bh);
```

Arguments

handle

    transaction

bh

    bufferhead to mark

Description

Mark a buffer as containing dirty data which needs to be flushed before we can commit the current transaction.

The buffer is placed on the transaction's data list and is marked as belonging to the transaction.

Returns error number or 0 on success.

`journal_dirty_data` can be called via page_launder->ext3_writepage by kswapd.

journal_dirty_metadata

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_dirty_metadata` — mark a buffer as containing dirty metadata

Synopsis

```
int journal_dirty_metadata (handle_t * handle, struct
buffer_head * bh);
```

Arguments

handle

transaction to add buffer to.

bh

buffer to mark

Description

Mark dirty metadata which needs to be journaled as part of the current transaction.

The buffer is placed on the transaction's metadata list and is marked as belonging to the transaction.

Returns error number or 0 on success.

Special care needs to be taken if the buffer already belongs to the current committing transaction (in which case we should have frozen data present for that commit). In that case, we don't relink the

buffer

that only gets done when the old transaction finally completes its commit.

journal_forget

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_forget` — `bforget` for potentially-journaled buffers.

Synopsis

```
int journal_forget (handle_t * handle, struct buffer_head *
bh);
```

Arguments

handle

  transaction handle

bh

  bh to ’forget’

Description

We can only do the bforget if there are no commits pending against the buffer. If the buffer is dirty in the current running transaction we can safely unlink it.

bh may not be a journalled buffer at all - it may be a non-JBD buffer which came off the hashtable. Check for this.

Decrements bh->b_count by one.

Allow this call even if the handle has aborted --- it may be part of the caller's cleanup after an abort.

journal_stop

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_stop` — complete a transaction

Synopsis

```
int journal_stop (handle_t * handle);
```

Arguments

handle

transaction to complete.

Description

All done for a particular handle.

There is not much action needed here. We just return any remaining buffer credits to the transaction and remove the handle. The only complication is that we need to start a commit operation if the filesystem is marked for synchronous update.

journal_stop itself will not usually return an error, but it may do so in unusual circumstances. In particular, expect it to return -EIO if a journal_abort has been executed since the transaction began.

journal_force_commit

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_force_commit` — force any uncommitted transactions

Synopsis

```
int journal_force_commit (journal_t * journal);
```

Arguments

journal

journal to force

For synchronous operations

force any uncommitted transactions to disk. May seem kludgy, but it reuses all the handle batching code in a very simple manner.

journal_try_to_free_buffers

LINUX

Name

`journal_try_to_free_buffers` — try to free page buffers.

Synopsis

```
int journal_try_to_free_buffers (journal_t * journal, struct
page * page, gfp_t gfp_mask);
```

Arguments

journal

journal for operation

page

to try and free

gfp_mask

we use the mask to detect how hard should we try to release buffers. If __GFP_WAIT and __GFP_FS is set, we wait for commit code to release the buffers.

Description

For all the buffers on this page, if they are fully written out ordered data, move them onto BUF_CLEAN so `try_to_free_buffers` can reap them.

This function returns non-zero if we wish `try_to_free_buffers` to be called. We do this if the page is releasable by `try_to_free_buffers`. We also do it if the page has locked or dirty buffers and the caller wants us to perform sync or async writeout.

This complicates JBD locking somewhat. We aren't protected by the BKL here. We wish to remove the buffer from its committing or running transaction's ->t_datalist via __journal_unfile_buffer.

This may *change* the value of transaction_t->t_datalist, so anyone who looks at t_datalist needs to lock against this function.

Even worse, someone may be doing a journal_dirty_data on this buffer. So we need to lock against that. `journal_dirty_data` will come out of the lock with the buffer dirty, which makes it ineligible for release here.

Who else is affected by this? hmm... Really the only contender is `do_get_write_access` - it could be looking at the buffer while `journal_try_to_free_buffer` is changing its state. But that cannot happen because we never reallocate freed data as metadata while the data is part of a transaction. Yes?

Return 0 on failure, 1 on success

journal_invalidatepage

LINUX

Kernel Hackers ManualJuly 2010

Name

`journal_invalidatepage` — invalidate a journal page

Synopsis

```
void journal_invalidatepage (journal_t * journal, struct page
* page, unsigned long offset);
```

Arguments

journal

journal to use for flush

`page`

page to flush

`offset`

length of page to invalidate.

Description

Reap page buffers containing data after offset in page.

5.4. See also

[Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen Tweedie (ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz)]

[Ext3 Journalling FileSystem, OLS 2000, Dr. Stephen Tweedie (http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html)]

Chapter 6. splice API

splice is a method for moving blocks of data around inside the kernel, without continually transferring them between the kernel and user space.

splice_to_pipe

LINUX

Kernel Hackers ManualJuly 2010

Name

`splice_to_pipe` — fill passed data into a pipe

Synopsis

```
ssize_t splice_to_pipe (struct pipe_inode_info * pipe, struct
splice_pipe_desc * spd);
```

Arguments

pipe

pipe to fill

spd

data to fill

Description

spd contains a map of pages and len/offset tuples, along with the struct pipe_buf_operations associated with these pages. This function will link that data to the pipe.

generic_file_splice_read

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_file_splice_read` — splice data from file to a pipe

Synopsis

```
ssize_t generic_file_splice_read (struct file * in, loff_t *
ppos, struct pipe_inode_info * pipe, size_t len, unsigned int
flags);
```

Arguments

in

file to splice from

ppos

position in *in*

pipe

pipe to splice to

len

number of bytes to splice

flags

splice modifier flags

Description

Will read pages from given file and fill them into a pipe. Can be used as long as the address_space operations for the source implements a `readpage` hook.

splice_from_pipe_feed

LINUX

Kernel Hackers ManualJuly 2010

Name

`splice_from_pipe_feed` — feed available data from a pipe to a file

Synopsis

```
int splice_from_pipe_feed (struct pipe_inode_info * pipe,
struct splice_desc * sd, splice_actor * actor);
```

Arguments

pipe

pipe to splice from

sd

information to *actor*

actor

handler that splices the data

Description

This function loops over the pipe and calls *actor* to do the actual moving of a single struct pipe_buffer to the desired destination. It returns when there's no more buffers left in the pipe or if the requested number of bytes (*sd*->total_len) have been copied. It returns a positive number (one) if the pipe needs to be filled with more data, zero if the required number of bytes have been copied and -errno on error.

This, together with splice_from_pipe_{begin,end,next}, may be used to implement the functionality of `__splice_from_pipe` when locking is required around copying the pipe buffers to the destination.

splice_from_pipe_next

LINUX

Kernel Hackers ManualJuly 2010

Name

`splice_from_pipe_next` — wait for some data to splice from

Synopsis

```
int splice_from_pipe_next (struct pipe_inode_info * pipe,
struct splice_desc * sd);
```

Arguments

pipe

pipe to splice from

sd

information about the splice operation

Description

This function will wait for some data and return a positive value (one) if pipe buffers are available. It will return zero or -errno if no more data needs to be spliced.

splice_from_pipe_begin

LINUX

Kernel Hackers ManualJuly 2010

Name

`splice_from_pipe_begin` — start splicing from pipe

Synopsis

```
void splice_from_pipe_begin (struct splice_desc * sd);
```

Arguments

sd

information about the splice operation

Description

This function should be called before a loop containing `splice_from_pipe_next` and `splice_from_pipe_feed` to initialize the necessary fields of *sd*.

splice_from_pipe_end

LINUX

Kernel Hackers ManualJuly 2010

Name

`splice_from_pipe_end` — finish splicing from pipe

Synopsis

```
void splice_from_pipe_end (struct pipe_inode_info * pipe,
struct splice_desc * sd);
```

Arguments

pipe

pipe to splice from

sd

information about the splice operation

Description

This function will wake up pipe writers if necessary. It should be called after a loop containing `splice_from_pipe_next` and `splice_from_pipe_feed`.

__splice_from_pipe

LINUX

Name

`__splice_from_pipe` — splice data from a pipe to given actor

Synopsis

```
ssize_t __splice_from_pipe (struct pipe_inode_info * pipe,
struct splice_desc * sd, splice_actor * actor);
```

Arguments

pipe

pipe to splice from

sd

information to *actor*

actor

handler that splices the data

Description

This function does little more than loop over the pipe and call *actor* to do the actual moving of a single struct pipe_buffer to the desired destination. See pipe_to_file, pipe_to_sendpage, or pipe_to_user.

splice_from_pipe

LINUX

Name

`splice_from_pipe` — splice data from a pipe to a file

Synopsis

```
ssize_t splice_from_pipe (struct pipe_inode_info * pipe,
struct file * out, loff_t * ppos, size_t len, unsigned int
flags, splice_actor * actor);
```

Arguments

pipe

pipe to splice from

out

file to splice to

ppos

position in *out*

len

how many bytes to splice

flags

splice modifier flags

actor

handler that splices the data

Description

See __splice_from_pipe. This function locks the pipe inode, otherwise it's identical to `__splice_from_pipe`.

generic_file_splice_write

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_file_splice_write` — splice data from a pipe to a file

Synopsis

```
ssize_t generic_file_splice_write (struct pipe_inode_info *
pipe, struct file * out, loff_t * ppos, size_t len, unsigned
int flags);
```

Arguments

pipe

pipe info

out

file to write to

ppos

position in *out*

len

number of bytes to splice

flags

splice modifier flags

Description

Will either move or copy pages (determined by *flags* options) from the given pipe inode to the given file.

generic_splice_sendpage

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_splice_sendpage` — splice data from a pipe to a socket

Synopsis

```
ssize_t generic_splice_sendpage (struct pipe_inode_info *
pipe, struct file * out, loff_t * ppos, size_t len, unsigned
int flags);
```

Arguments

pipe

pipe to splice from

out

socket to write to

ppos

position in *out*

len

number of bytes to splice

flags

splice modifier flags

Description

Will send *len* bytes from the pipe to a network socket. No data copying is involved.

splice_direct_to_actor

LINUX

Kernel Hackers ManualJuly 2010

Name

`splice_direct_to_actor` — splices data directly between two non-pipes

Synopsis

```
ssize_t splice_direct_to_actor (struct file * in, struct
splice_desc * sd, splice_direct_actor * actor);
```

Arguments

in

file to splice from

sd

actor information on where to splice to

actor

handles the data splicing

Description

This is a special case helper to splice directly between two points, without requiring an explicit pipe. Internally an allocated pipe is cached in the process, and reused during the lifetime of that process.

do_splice_direct

LINUX

Kernel Hackers ManualJuly 2010

Name

`do_splice_direct` — splices data directly between two files

Synopsis

`long` **`do_splice_direct`** `(struct file *` *`in`*`, loff_t *` *`ppos`*`, struct file *` *`out`*`, size_t` *`len`*`, unsigned int` *`flags`*`);`

Arguments

`in`

file to splice from

`ppos`

input file offset

`out`

file to splice to

`len`

number of bytes to splice

flags

splice modifier flags

Description

For use by `do_sendfile`. splice can easily emulate sendfile, but doing it in the application would incur an extra system call (splice in + splice out, as compared to just `sendfile`). So this helper can splice directly through a process-private pipe.

Chapter 7. pipes API

Pipe interfaces are all for in-kernel (builtin image) use. They are not exported for use by modules.

struct pipe_buffer

LINUX

Kernel Hackers ManualJuly 2010

Name

`struct pipe_buffer` — a linux kernel pipe buffer

Synopsis

```
struct pipe_buffer {
  struct page * page;
  unsigned int offset;
  unsigned int len;
  const struct pipe_buf_operations * ops;
  unsigned int flags;
  unsigned long private;
};
```

Members

page

    the page containing the data for the pipe buffer

offset

    offset of data inside the *page*

len

    length of data inside the *page*

ops

operations associated with this buffer. See *pipe_buf_operations*.

flags

pipe buffer flags. See above.

private

private data owned by the ops.

struct pipe_inode_info

LINUX

Kernel Hackers ManualJuly 2010

Name

struct pipe_inode_info — a linux kernel pipe

Synopsis

```
struct pipe_inode_info {
  wait_queue_head_t wait;
  unsigned int nrbufs;
  unsigned int curbuf;
  struct page * tmp_page;
  unsigned int readers;
  unsigned int writers;
  unsigned int waiting_writers;
  unsigned int r_counter;
  unsigned int w_counter;
  struct fasync_struct * fasync_readers;
  struct fasync_struct * fasync_writers;
  struct inode * inode;
  struct pipe_buffer bufs[PIPE_BUFFERS];
};
```

Members

wait

reader/writer wait point in case of empty/full pipe

nrbufs

the number of non-empty pipe buffers in this pipe

curbuf

the current pipe buffer entry

tmp_page

cached released page

readers

number of current readers of this pipe

writers

number of current writers of this pipe

waiting_writers

number of writers blocked waiting for room

r_counter

reader counter

w_counter

writer counter

fasync_readers

reader side fasync

fasync_writers

writer side fasync

inode

inode this pipe is attached to

bufs[PIPE_BUFFERS]

the circular array of pipe buffers

generic_pipe_buf_map

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_pipe_buf_map` — virtually map a pipe buffer

Synopsis

```
void * generic_pipe_buf_map (struct pipe_inode_info * pipe,
struct pipe_buffer * buf, int atomic);
```

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer that should be mapped

atomic

whether to use an atomic map

Description

This function returns a kernel virtual address mapping for the pipe_buffer passed in *buf*. If *atomic* is set, an atomic map is provided and the caller has to be careful not to fault before calling the unmap function.

Note that this function occupies KM_USER0 if *atomic* != 0.

generic_pipe_buf_unmap

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_pipe_buf_unmap` — unmap a previously mapped pipe buffer

Synopsis

```
void generic_pipe_buf_unmap (struct pipe_inode_info * pipe,
struct pipe_buffer * buf, void * map_data);
```

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer that should be unmapped

map_data

the data that the mapping function returned

Description

This function undoes the mapping that ->`map` provided.

generic_pipe_buf_steal

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_pipe_buf_steal` — attempt to take ownership of a pipe_buffer

Synopsis

```
int generic_pipe_buf_steal (struct pipe_inode_info * pipe,
struct pipe_buffer * buf);
```

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer to attempt to steal

Description

This function attempts to steal the struct page attached to *buf*. If successful, this function returns 0 and returns with the page locked. The caller may then reuse the page for whatever he wishes; the typical use is insertion into a different file page cache.

generic_pipe_buf_get

LINUX

Kernel Hackers ManualJuly 2010

Name

`generic_pipe_buf_get` — get a reference to a struct pipe_buffer

Synopsis

```
void generic_pipe_buf_get (struct pipe_inode_info * pipe,
struct pipe_buffer * buf);
```

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer to get a reference to

Description

This function grabs an extra reference to *buf*. It's used in in the `tee` system call, when we duplicate the buffers in one pipe into another.

generic_pipe_buf_confirm

LINUX

Name

`generic_pipe_buf_confirm` — verify contents of the pipe buffer

Synopsis

```
int generic_pipe_buf_confirm (struct pipe_inode_info * info,
struct pipe_buffer * buf);
```

Arguments

info

    the pipe that the buffer belongs to

buf

    the buffer to confirm

Description

This function does nothing, because the generic pipe code uses pages that are always good when inserted into the pipe.

generic_pipe_buf_release

LINUX

Name

`generic_pipe_buf_release` — put a reference to a struct pipe_buffer

Synopsis

```
void generic_pipe_buf_release (struct pipe_inode_info * pipe,
struct pipe_buffer * buf);
```

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer to put a reference to

Description

This function releases a reference to *buf*.