Linux Filesystems API

Release 4.13.0-rc4+

The kernel development community

Sep 05, 2017

CHAPTER

ONE

THE LINUX VFS

1.1 The Filesystem types

enum **positive_aop_returns**
aop return codes with specific semantics

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.

void **sb_end_write**(struct super_block * *sb*)
drop write access to a superblock

Parameters

struct super_block * sb the super we wrote to

Description

Decrement number of writers to the filesystem. Wake up possible waiters wanting to freeze the filesystem.

void **sb_end_pagefault**(struct super_block * *sb*)
drop write access to a superblock from a page fault

Parameters

struct super_block * sb the super we wrote to

Description

Decrement number of processes handling write page fault to the filesystem. Wake up possible waiters wanting to freeze the filesystem.

void **sb_end_intwrite**(struct super_block * *sb*)
drop write access to a superblock for internal fs purposes

Parameters

struct super_block * sb the super we wrote to

Description

Decrement fs-internal number of writers to the filesystem. Wake up possible waiters wanting to freeze the filesystem.

void **sb_start_write**(struct super_block * *sb*)
get write access to a superblock

Parameters

struct super_block * sb the super we write to

Description

When a process wants to write data or metadata to a file system (i.e. dirty a page or an inode), it should embed the operation in a *sb_start_write()* - *sb_end_write()* pair to get exclusion against file system freezing. This function increments number of writers preventing freezing. If the file system is already frozen, the function waits until the file system is thawed.

Since freeze protection behaves as a lock, users have to preserve ordering of freeze protection and other filesystem locks. Generally, freeze protection should be the outermost lock. In particular, we have:

sb_start_write -> i_mutex (write path, truncate, directory ops, …) -> s_umount (freeze_super, thaw_super)

void **sb_start_pagefault**(struct super_block * *sb*)
get write access to a superblock from a page fault

Parameters

struct super_block * sb the super we write to

Description

When a process starts handling write page fault, it should embed the operation into *sb_start_pagefault()* - *sb_end_pagefault()* pair to get exclusion against file system freezing. This is needed since the page fault is going to dirty a page. This function increments number of running page faults preventing freezing. If the file system is already frozen, the function waits until the file system is thawed.

Since page fault freeze protection behaves as a lock, users have to preserve ordering of freeze protection and other filesystem locks. It is advised to put *sb_start_pagefault()* close to mmap_sem in lock ordering. Page fault handling code implies lock dependency:

mmap_sem -> sb_start_pagefault

void **inode_inc_iversion**(struct inode * *inode*)
increments i_version

Parameters

struct inode * 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

void **filemap_set_wb_err**(struct address_space * *mapping*, int *err*)
set a writeback error on an address_space

Parameters

struct address_space * mapping mapping in which to set writeback error

int err error to be set in mapping

Description

When writeback fails in some way, we must record that error so that userspace can be informed when fsync and the like are called. We endeavor to report errors on any file that was open at the time of the error. Some internal callers also need to know when writeback errors have occurred.

When a writeback error occurs, most filesystems will want to call filemap_set_wb_err to record the error in the mapping so that it will be automatically reported whenever fsync is called on the file.

FIXME: mention FS_* flag here?

int **`filemap_check_wb_err`**(struct address_space * *mapping*, errseq_t *since*)
    has an error occurred since the mark was sampled?

Parameters

`struct address_space * mapping` mapping to check for writeback errors

`errseq_t since` previously-sampled errseq_t

Description

Grab the errseq_t value from the mapping, and see if it has changed "since" the given value was sampled.

If it has then report the latest error set, otherwise return 0.

errseq_t **`filemap_sample_wb_err`**(struct address_space * *mapping*)
    sample the current errseq_t to test for later errors

Parameters

`struct address_space * mapping` mapping to be sampled

Description

Writeback errors are always reported relative to a particular sample point in the past. This function provides those sample points.

1.2 The Directory Cache

void **`__d_drop`**(struct dentry * *dentry*)
    drop a dentry

Parameters

`struct dentry * 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.

void **`shrink_dcache_sb`**(struct super_block * *sb*)
    shrink dcache for a superblock

Parameters

`struct super_block * sb` superblock

Description

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

int **`path_has_submounts`**(const struct path * *parent*)
check for mounts over a dentry in the current namespace.

Parameters

`const struct path * parent` path to check.

Description

Return true if the parent or its subdirectories contain a mount point in the current namespace.

void **`shrink_dcache_parent`**(struct dentry * *parent*)
prune dcache

Parameters

`struct dentry * parent` parent of entries to prune

Description

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

void **`d_invalidate`**(struct dentry * *dentry*)
detach submounts, prune dcache, and drop

Parameters

`struct dentry * dentry` dentry to invalidate (aka detach, prune and drop)

Description

no dcache lock.

The final d_drop is done as an atomic operation relative to rename_lock ensuring there are no races with d_set_mounted. This ensures there are no unhashed dentries on the path to a mountpoint.

struct dentry * **`d_alloc`**(struct dentry * *parent*, const struct qstr * *name*)
allocate a dcache entry

Parameters

`struct dentry * parent` parent of entry to allocate

`const struct qstr * 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.

struct dentry * **`d_alloc_pseudo`**(struct super_block * *sb*, const struct qstr * *name*)
allocate a dentry (for lookup-less filesystems)

Parameters

`struct super_block * sb` the superblock

`const struct qstr * name` qstr of the name

Description

For a filesystem that just pins its dentries in memory and never performs lookups at all, return an unhashed IS_ROOT dentry.

void **`d_instantiate`**(struct dentry * *entry*, struct inode * *inode*)
fill in inode information for a dentry

Parameters

`struct dentry * entry` dentry to complete

`struct inode * 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.

int **`d_instantiate_no_diralias`**(struct dentry * *entry*, struct inode * *inode*)
instantiate a non-aliased dentry

Parameters

`struct dentry * entry` dentry to complete

`struct inode * inode` inode to attach to this dentry

Description

Fill in inode information in the entry. If a directory alias is found, then return an error (and drop inode). Together with `d_materialise_unique()` this guarantees that a directory inode may never have more than one alias.

struct dentry * **`d_find_any_alias`**(struct inode * *inode*)
find any alias for a given inode

Parameters

`struct inode * inode` inode to find an alias for

Description

If any aliases exist for the given inode, take and return a reference for one of them. If no aliases exist, return `NULL`.

struct dentry * **`d_obtain_alias`**(struct inode * *inode*)
find or allocate a DISCONNECTED dentry for a given inode

Parameters

`struct inode * 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 the error will be propagated to the return value, with a `NULL` **inode** replaced by ERR_PTR(-ESTALE).

struct dentry * **`d_obtain_root`**(struct inode * *inode*)
find or allocate a dentry for a given inode

Parameters

`struct inode * inode` inode to allocate the dentry for

Description

Obtain an IS_ROOT dentry for the root of a filesystem.

We must ensure that directory inodes only ever have 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. 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).

struct dentry * **`d_add_ci`**(struct dentry * *dentry*, struct inode * *inode*, struct qstr * *name*)
lookup or allocate new dentry with case-exact name

Parameters

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

`struct inode * inode` the inode case-insensitive lookup has found

`struct qstr * 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.

struct dentry * **`d_lookup`**(const struct dentry * *parent*, const struct qstr * *name*)
search for a dentry

Parameters

`const struct dentry * parent` parent dentry

`const struct qstr * name` qstr of name we wish to find

Return

dentry, or NULL

d_lookup 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 if the dentry does not exist.

struct dentry * **`d_hash_and_lookup`**(struct dentry * *dir*, struct qstr * *name*)
hash the qstr then search for a dentry

Parameters

`struct dentry * dir` Directory to search in

`struct qstr * name` qstr of name we wish to find

Description

On lookup failure NULL is returned; on bad name - ERR_PTR(-error)

void **`d_delete`**(struct dentry * *dentry*)
delete a dentry

Parameters

`struct dentry * 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

void **`d_rehash`**(struct dentry * *entry*)
add an entry back to the hash

Parameters

struct dentry * entry dentry to add to the hash

Description

Adds a dentry to the hash according to its name.

void **d_add**(struct dentry * *entry*, struct inode * *inode*)
  add dentry to hash queues

Parameters

struct dentry * entry dentry to add

struct inode * 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()*.

struct dentry * **d_exact_alias**(struct dentry * *entry*, struct inode * *inode*)
  find and hash an exact unhashed alias

Parameters

struct dentry * entry dentry to add

struct inode * inode The inode to go with this dentry

Description

If an unhashed dentry with the same name/parent and desired inode already exists, hash and return it. Otherwise, return NULL.

Parent directory should be locked.

void **dentry_update_name_case**(struct dentry * *dentry*, const struct qstr * *name*)
  update case insensitive dentry with a new name

Parameters

struct dentry * dentry dentry to be updated

const struct qstr * name new name

Description

Update a case insensitive dentry with new case of name.

dentry must have been returned by d_lookup with name **name**. Old and new name lengths must match (ie. no d_compare which allows mismatched name lengths).

Parent inode i_mutex must be held over d_lookup and into this call (to keep renames and concurrent inserts, and readdir(2) away).

struct dentry * **d_splice_alias**(struct inode * *inode*, struct dentry * *dentry*)
  splice a disconnected dentry into the tree if one exists

Parameters

struct inode * inode the inode which may have a disconnected dentry

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

Description

If inode is a directory and has an IS_ROOT alias, 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.

If a non-IS_ROOT directory is found, the filesystem is corrupt, and we should error out: directories can't have multiple aliases.

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.

Cluster filesystems may call this function with a negative, hashed dentry. In that case, we know that the inode will be a regular file, and also this will only occur during atomic_open. So we need to check for the dentry being already hashed only in the final case.

char * **d_path**(const struct path * *path*, char * *buf*, int *buflen*)
    return the path of a dentry

Parameters

const struct path * path path to report

char * buf buffer to return value in

int 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.

struct dentry * **dget_dlock**(struct dentry * *dentry*)
    get a reference to a dentry

Parameters

struct dentry * 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.

int **d_unhashed**(const struct dentry * *dentry*)
    is dentry hashed

Parameters

const struct dentry * dentry entry to check

Description

Returns true if the dentry passed is not currently hashed.

bool **d_really_is_negative**(const struct dentry * *dentry*)
    Determine if a dentry is really negative (ignoring fallthroughs)

Parameters

const struct dentry * dentry The dentry in question

Description

Returns true if the dentry represents either an absent name or a name that doesn't map to an inode (ie. ->d_inode is NULL). The dentry could represent a true miss, a whiteout that isn't represented by a 0,0 chardev or a fallthrough marker in an opaque directory.

Note! (1) This should be used *only* by a filesystem to examine its own dentries. It should not be used to look at some other filesystem's dentries. (2) It should also be used in combination with *d_inode()* to get the inode. (3) The dentry may have something attached to ->d_lower and the type field of the flags may be set to something other than miss or whiteout.

bool **d_really_is_positive**(const struct dentry * *dentry*)
    Determine if a dentry is really positive (ignoring fallthroughs)

Parameters

const struct dentry * dentry The dentry in question

Description

Returns true if the dentry represents a name that maps to an inode (ie. ->d_inode is not NULL). The dentry might still represent a whiteout if that is represented on medium as a 0,0 chardev.

Note! (1) This should be used *only* by a filesystem to examine its own dentries. It should not be used to look at some other filesystem's dentries. (2) It should also be used in combination with `d_inode()` to get the inode.

struct inode * **d_inode**(const struct dentry * *dentry*)
Get the actual inode of this dentry

Parameters

const struct dentry * dentry The dentry to query

Description

This is the helper normal filesystems should use to get at their own inodes in their own dentries and ignore the layering superimposed upon them.

struct inode * **d_inode_rcu**(const struct dentry * *dentry*)
Get the actual inode of this dentry with `ACCESS_ONCE()`

Parameters

const struct dentry * dentry The dentry to query

Description

This is the helper normal filesystems should use to get at their own inodes in their own dentries and ignore the layering superimposed upon them.

struct inode * **d_backing_inode**(const struct dentry * *upper*)
Get upper or lower inode we should be using

Parameters

const struct dentry * upper The upper layer

Description

This is the helper that should be used to get at the inode that will be used if this dentry were to be opened as a file. The inode may be on the upper dentry or it may be on a lower dentry pinned by the upper.

Normal filesystems should not use this to access their own inodes.

struct dentry * **d_backing_dentry**(struct dentry * *upper*)
Get upper or lower dentry we should be using

Parameters

struct dentry * upper The upper layer

Description

This is the helper that should be used to get the dentry of the inode that will be used if this dentry were opened as a file. It may be the upper dentry or it may be a lower dentry pinned by the upper.

Normal filesystems should not use this to access their own dentries.

struct dentry * **d_real**(struct dentry * *dentry*, const struct inode * *inode*, unsigned int *flags*)
Return the real dentry

Parameters

struct dentry * dentry the dentry to query

const struct inode * inode inode to select the dentry from multiple layers (can be NULL)

unsigned int flags open flags to control copy-up behavior

Description

If dentry is on a union/overlay, then return the underlying, real dentry. Otherwise return the dentry itself.

See also: Documentation/filesystems/vfs.txt

struct inode * **d_real_inode**(const struct dentry * *dentry*)
Return the real inode

Parameters

const struct dentry * dentry The dentry to query

Description

If dentry is on a union/overlay, then return the underlying, real inode. Otherwise return *d_inode()*.

1.3 Inode Handling

int **inode_init_always**(struct super_block * *sb*, struct inode * *inode*)
perform inode structure initialisation

Parameters

struct super_block * sb superblock inode belongs to

struct inode * 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.

void **drop_nlink**(struct inode * *inode*)
directly drop an inode's link count

Parameters

struct inode * 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.

void **clear_nlink**(struct inode * *inode*)
directly zero an inode's link count

Parameters

struct inode * 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.

void **set_nlink**(struct inode * *inode*, unsigned int *nlink*)
directly set an inode's link count

Parameters

struct inode * inode inode

unsigned int nlink new nlink (should be non-zero)

Description

This is a low-level filesystem helper to replace any direct filesystem manipulation of i_nlink.

void **inc_nlink**(struct inode * *inode*)
directly increment an inode's link count

Parameters

struct inode * 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()`.

void **inode_sb_list_add**(struct inode * *inode*)
add inode to the superblock list of inodes

Parameters

struct inode * inode inode to add

void **__insert_inode_hash**(struct inode * *inode*, unsigned long *hashval*)
hash an inode

Parameters

struct inode * inode unhashed inode

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

Description

Add an inode to the inode hash for this superblock.

void **__remove_inode_hash**(struct inode * *inode*)
remove an inode from the hash

Parameters

struct inode * inode inode to unhash

Description

Remove an inode from the superblock.

struct inode * **new_inode**(struct super_block * *sb*)
obtain an inode

Parameters

struct super_block * 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

void **unlock_new_inode**(struct inode * *inode*)
clear the I_NEW state and wake up any waiters

Parameters

struct inode * inode new inode to unlock

Description

Called when the inode is fully initialised to clear the new state of the inode and wake up anyone waiting for the inode to finish initialisation.

void **lock_two_nondirectories**(struct inode * *inode1*, struct inode * *inode2*)
take two i_mutexes on non-directory objects

Parameters

struct inode * inode1 first inode to lock

struct inode * inode2 second inode to lock

Description

Lock any non-NULL argument that is not a directory. Zero, one or two objects may be locked by this function.

void **unlock_two_nondirectories**(struct inode * *inode1*, struct inode * *inode2*)
release locks from *lock_two_nondirectories()*

Parameters

struct inode * inode1 first inode to unlock

struct inode * inode2 second inode to unlock

struct inode * **iget5_locked**(struct super_block * *sb*, unsigned long *hashval*, int (*test) (struct inode *, void *, int (*set) (struct inode *, void *, void * *data*)
obtain an inode from a mounted file system

Parameters

struct super_block * sb super block of file system

unsigned long hashval hash value (usually inode number) to get

int (*)(struct inode *,void *) test callback used for comparisons between inodes

int (*)(struct inode *,void *) set callback used to initialize a new struct inode

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

Description

Search for the inode specified by **hashval** and **data** in the inode cache, and if present it is return it 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, allocate a new inode and return it 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_hash_lock held, so can't sleep.

struct inode * **iget_locked**(struct super_block * *sb*, unsigned long *ino*)
obtain an inode from a mounted file system

Parameters

struct super_block * sb super block of file system

unsigned long ino inode number to get

Description

Search for the inode specified by **ino** in the inode cache and if present return it 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, allocate a new inode and return it locked, hashed, and with the I_NEW flag set. The file system gets to fill it in before unlocking it via *unlock_new_inode()*.

ino_t **iunique**(struct super_block * *sb*, ino_t *max_reserved*)
get a unique inode number

Parameters

struct super_block * sb superblock

ino_t 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.

struct inode * **ilookup5_nowait**(struct super_block * *sb*, unsigned long *hashval*, int (*test) (struct inode *, void *, void * *data*)
search for an inode in the inode cache

Parameters

struct super_block * sb super block of file system to search

unsigned long hashval hash value (usually inode number) to search for

int (*)(struct inode *,void *) test callback used for comparisons between inodes

void * data opaque data pointer to pass to **test**

Description

Search for the inode specified by **hashval** and **data** in the inode cache. If the inode is in the cache, the inode is returned with an incremented reference count.

Note

I_NEW 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.

Note2: **test** is called with the inode_hash_lock held, so can't sleep.

struct inode * **ilookup5**(struct super_block * *sb*, unsigned long *hashval*, int (*test) (struct inode *, void *, void * *data*)
search for an inode in the inode cache

Parameters

struct super_block * sb super block of file system to search

unsigned long hashval hash value (usually inode number) to search for

int (*)(struct inode *,void *) test callback used for comparisons between inodes

void * data opaque data pointer to pass to **test**

Description

Search for the inode specified by **hashval** and **data** in the inode cache, and if the inode is in the cache, return the inode with an incremented reference count. Waits on I_NEW before returning the inode. returned with an incremented reference count.

This is a generalized version of *ilookup()* for file systems where the inode number is not sufficient for unique identification of an inode.

Note

test is called with the inode_hash_lock held, so can't sleep.

struct inode * **ilookup**(struct super_block * *sb*, unsigned long *ino*)
search for an inode in the inode cache

Parameters

struct super_block * sb super block of file system to search

unsigned long ino inode number to search for

Description

Search for the inode **ino** in the inode cache, and if the inode is in the cache, the inode is returned with an incremented reference count.

struct inode * **`find_inode_nowait`**(struct super_block * *sb*, unsigned long *hashval*, int (*match) (struct inode *, unsigned *long*, void *, void * *data*)
find an inode in the inode cache

Parameters

`struct super_block * sb` super block of file system to search

`unsigned long hashval` hash value (usually inode number) to search for

`int (*)(struct inode *,unsigned long,void *) match` callback used for comparisons between inodes

`void * data` opaque data pointer to pass to **match**

Description

Search for the inode specified by **hashval** and **data** in the inode cache, where the helper function **match** will return 0 if the inode does not match, 1 if the inode does match, and -1 if the search should be stopped. The **match** function must be responsible for taking the i_lock spin_lock and checking i_state for an inode being freed or being initialized, and incrementing the reference count before returning 1. It also must not sleep, since it is called with the inode_hash_lock spinlock held.

This is a even more generalized version of *`ilookup5()`* when the function must never block — `find_inode()` can block in `__wait_on_freeing_inode()` — or when the caller can not increment the reference count because the resulting *`iput()`* might cause an inode eviction. The tradeoff is that the **match** funtion must be very carefully implemented.

void **`iput`**(struct inode * *inode*)
put an inode

Parameters

`struct inode * 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.

sector_t **`bmap`**(struct inode * *inode*, sector_t *block*)
find a block number in a file

Parameters

`struct inode * inode` inode of file

`sector_t 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.

int **`file_update_time`**(struct file * *file*)
update mtime and ctime time

Parameters

`struct file * 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. This can return an error for file systems who need to allocate space in order to update an inode.

void **`inode_init_owner`**(struct inode * *inode*, const struct inode * *dir*, umode_t *mode*)
Init uid,gid,mode for new inode according to posix standards

Parameters

`struct inode * inode` New inode

`const struct inode * dir` Directory inode

`umode_t mode` mode of the new inode

bool **`inode_owner_or_capable`**(const struct inode * *inode*)
check current task permissions to inode

Parameters

`const struct inode * inode` inode being checked

Description

Return true if current either has CAP_FOWNER in a namespace with the inode owner uid mapped, or owns the file.

void **`inode_dio_wait`**(struct inode * *inode*)
wait for outstanding DIO requests to finish

Parameters

`struct inode * inode` inode to wait for

Description

Waits for all pending direct I/O requests to finish so that we can proceed with a truncate or equivalent operation.

Must be called under a lock that serializes taking new references to i_dio_count, usually by inode->i_mutex.

struct timespec **`current_time`**(struct inode * *inode*)
Return FS time

Parameters

`struct inode * inode` inode.

Description

Return the current time truncated to the time granularity supported by the fs.

Note that inode and inode->sb cannot be NULL. Otherwise, the function warns and returns time without truncation.

void **`make_bad_inode`**(struct inode * *inode*)
mark an inode bad due to an I/O error

Parameters

`struct inode * 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.

bool **`is_bad_inode`**(struct inode * *inode*)
is an inode errored

Parameters

`struct inode * inode` inode to test

Description

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

void **`iget_failed`**(struct inode * *inode*)
Mark an under-construction inode as dead and release it

Parameters

`struct inode * inode` The inode to discard

Description

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

1.4 Registration and Superblocks

void **`deactivate_locked_super`**(struct super_block * *s*)
drop an active reference to superblock

Parameters

`struct super_block * s` superblock to deactivate

Description

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

Caller holds exclusive lock on superblock; that lock is released.

void **`deactivate_super`**(struct super_block * *s*)
drop an active reference to superblock

Parameters

`struct super_block * s` superblock to deactivate

Description

Variant of *`deactivate_locked_super()`*, except that superblock is *not* locked by caller. If we are going to drop the final active reference, lock will be acquired prior to that.

void **`generic_shutdown_super`**(struct super_block * *sb*)
common helper for ->:c:func:*kill_sb()*

Parameters

`struct super_block * sb` superblock to kill

Description

`generic_shutdown_super()` does all fs-independent work on superblock shutdown. Typical ->:c:func:*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.

struct super_block * **`sget_userns`**(struct file_system_type * *type*, int (*test) (struct super_block *, void *, int (*set) (struct super_block *, void *, int *flags*, struct user_namespace * *user_ns*, void * *data*)
find or create a superblock

Parameters

`struct file_system_type * type` filesystem type superblock should belong to

`int (*)(struct super_block *,void *) test` comparison callback

`int (*)(struct super_block *,void *) set` setup callback

`int flags` mount flags

`struct user_namespace * user_ns` User namespace for the super_block

`void * data` argument to each of them

struct super_block * **`sget`**(struct file_system_type * *type*, int (*test) (struct super_block *, void *, int (*set) (struct super_block *, void *, int *flags*, void * *data*)
find or create a superblock

Parameters

`struct file_system_type * type` filesystem type superblock should belong to

`int (*)(struct super_block *,void *) test` comparison callback

`int (*)(struct super_block *,void *) set` setup callback

`int flags` mount flags

`void * data` argument to each of them

void **`iterate_supers_type`**(struct file_system_type * *type*, void (*f) (struct super_block *, void *, void * *arg*)
call function for superblocks of given type

Parameters

`struct file_system_type * type` fs type

`void (*)(struct super_block *,void *) f` function to call

`void * arg` argument to pass to it

Description

Scans the superblock list and calls given function, passing it locked superblock and given argument.

struct super_block * **`get_super`**(struct block_device * *bdev*)
get the superblock of a device

Parameters

`struct block_device * 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.

struct super_block * **`get_super_thawed`**(struct block_device * *bdev*)
get thawed superblock of a device

Parameters

`struct block_device * bdev` device to get the superblock for

Description

Scans the superblock list and finds the superblock of the file system mounted on the device. The superblock is returned once it is thawed (or immediately if it was not frozen). `NULL` is returned if no match is found.

struct super_block * **`get_super_exclusive_thawed`**(struct block_device * *bdev*)
get thawed superblock of a device

Parameters

`struct block_device * bdev` device to get the superblock for

Description

Scans the superblock list and finds the superblock of the file system mounted on the device. The superblock is returned once it is thawed (or immediately if it was not frozen) and s_umount semaphore is held in exclusive mode. `NULL` is returned if no match is found.

int **`freeze_super`**(struct super_block * *sb*)
lock the filesystem and force it into a consistent state

Parameters

`struct super_block * sb` the super to lock

Description

Syncs the super to make sure the filesystem is consistent and calls the fs's freeze_fs. Subsequent calls to this without first thawing the fs will return -EBUSY.

During this function, sb->s_writers.frozen goes through these values:

SB_UNFROZEN: File system is normal, all writes progress as usual.

SB_FREEZE_WRITE: The file system is in the process of being frozen. New writes should be blocked, though page faults are still allowed. We wait for all writes to complete and then proceed to the next stage.

SB_FREEZE_PAGEFAULT: Freezing continues. Now also page faults are blocked but internal fs threads can still modify the filesystem (although they should not dirty new pages or inodes), writeback can run etc. After waiting for all running page faults we sync the filesystem which will clean all dirty pages and inodes (no new dirty pages or inodes can be created when sync is running).

SB_FREEZE_FS: The file system is frozen. Now all internal sources of fs modification are blocked (e.g. XFS preallocation truncation on inode reclaim). This is usually implemented by blocking new transactions for filesystems that have them and need this additional guard. After all internal writers are finished we call ->:c:func:*freeze_fs()* to finish filesystem freezing. Then we transition to SB_FREEZE_COMPLETE state. This state is mostly auxiliary for filesystems to verify they do not modify frozen fs.

sb->s_writers.frozen is protected by sb->s_umount.

int **`thaw_super`**(struct super_block * *sb*)

- unlock filesystem

Parameters

`struct super_block * sb` the super to thaw

Description

Unlocks the filesystem and marks it writeable again after *`freeze_super()`*.

1.5 File Locks

int **`posix_lock_file`**(struct file * *filp*, struct file_lock * *fl*, struct file_lock * *conflock*)
Apply a POSIX-style lock to a file

Parameters

`struct file * filp` The file to apply the lock to

`struct file_lock * fl` The lock to be applied

`struct file_lock * 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.

int **`locks_mandatory_area`**(struct inode * *inode*, struct file * *filp*, loff_t *start*, loff_t *end*, unsigned char *type*)
Check for a conflicting lock

Parameters

`struct inode * inode` the file to check

`struct file * filp` how the file was opened (if it was)

`loff_t start` first byte in the file to check

`loff_t end` lastbyte in the file to check

`unsigned char type` `F_WRLCK` for a write lock, else `F_RDLCK`

Description

Searches the inode's list of locks to find any POSIX locks which conflict.

int **`__break_lease`**(struct inode * *inode*, unsigned int *mode*, unsigned int *type*)
revoke all outstanding leases on file

Parameters

`struct inode * inode` the inode of the file to return

`unsigned int mode` O_RDONLY: break only write leases; O_WRONLY or O_RDWR: break all leases

`unsigned int type` FL_LEASE: break leases and delegations; FL_DELEG: break only delegations

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()`.

void **`lease_get_mtime`**(struct inode * *inode*, struct timespec * *time*)
get the last modified time of an inode

Parameters

`struct inode * inode` the inode

`struct timespec * 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.

int **`generic_setlease`**(struct file * *filp*, long *arg*, struct file_lock ** *flp*, void ** *priv*)
sets a lease on an open file

Parameters

`struct file * filp` file pointer

`long arg` type of lease to obtain

`struct file_lock ** flp` input - file_lock to use, output - file_lock inserted

`void ** priv` private data for lm_setup (may be NULL if lm_setup doesn't require it)

Description

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

int **`vfs_setlease`**(struct file * *filp*, long *arg*, struct file_lock ** *lease*, void ** *priv*)
sets a lease on an open file

Parameters

`struct file * filp` file pointer

`long arg` type of lease to obtain

`struct file_lock ** lease` file_lock to use when adding a lease

`void ** priv` private info for lm_setup when adding a lease (may be NULL if lm_setup doesn't require it)

Description

Call this to establish a lease on the file. The "lease" argument is not used for F_UNLCK requests and may be NULL. For commands that set or alter an existing lease, the `(*lease)->fl_lmops->lm_break` operation must be set; if not, this function will return -ENOLCK (and generate a scary-looking stack trace).

The "priv" pointer is passed directly to the lm_setup function as-is. It may be NULL if the lm_setup operation doesn't require it.

int **`locks_lock_inode_wait`**(struct inode * *inode*, struct file_lock * *fl*)
Apply a lock to an inode

Parameters

`struct inode * inode` inode of the file to apply to

`struct file_lock * fl` The lock to be applied

Description

Apply a POSIX or FLOCK style lock request to an inode.

int **`vfs_test_lock`**(struct file * *filp*, struct file_lock * *fl*)
test file byte range lock

Parameters

`struct file * filp` The file to test lock for

`struct file_lock * 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.

int **`vfs_lock_file`**(struct file * *filp*, unsigned int *cmd*, struct file_lock * *fl*, struct file_lock * *conf*)
file byte range lock

Parameters

`struct file * filp` The file to apply the lock to

`unsigned int cmd` type of locking operation (F_SETLK, F_GETLK, etc.)

`struct file_lock * fl` The lock to be applied

`struct file_lock * 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 ->:c:func:*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 ->:c:func:*lock()* interface may return asynchronously, before the lock has been granted or denied by the underlying filesystem, if (and only if) lm_grant is set. Callers expecting ->:c:func:*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 ->:c:func:*lock()* does return asynchronously, it must return FILE_LOCK_DEFERRED, and call ->:c:func:*lm_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 ->:c:func:*lm_grant()* before returning to the caller with a FILE_LOCK_DEFERRED return code.

int **posix_unblock_lock**(struct file_lock * *waiter*)
stop waiting for a file lock

Parameters

struct file_lock * waiter the lock which was waiting

Description

lockd needs to block waiting for locks.

int **vfs_cancel_lock**(struct file * *filp*, struct file_lock * *fl*)
file byte range unblock lock

Parameters

struct file * filp The file to apply the unblock to

struct file_lock * fl The lock to be unblocked

Description

Used by lock managers to cancel blocked requests

int **posix_lock_inode_wait**(struct inode * *inode*, struct file_lock * *fl*)
Apply a POSIX-style lock to a file

Parameters

struct inode * inode inode of file to which lock request should be applied

struct file_lock * fl The lock to be applied

Description

Apply a POSIX style lock request to an inode.

int **locks_mandatory_locked**(struct file * *file*)
Check for an active lock

Parameters

struct file * file 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.

int **fcntl_getlease**(struct file * *filp*)
Enquire what lease is currently active

Parameters

struct file * 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.

int **check_conflicting_open**(const struct dentry * *dentry*, const long *arg*, int *flags*)
see if the given dentry points to a file that has an existing open that would conflict with the desired lease.

Parameters

const struct dentry * dentry dentry to check

const long arg type of lease that we're trying to acquire

int flags current lock flags

Description

Check to see if there's an existing open fd on this file that would conflict with the lease we're trying to set.

int **fcntl_setlease**(unsigned int *fd*, struct file * *filp*, long *arg*)
sets a lease on an open file

Parameters

unsigned int fd open file descriptor

struct file * filp file pointer

long 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.

int **flock_lock_inode_wait**(struct inode * *inode*, struct file_lock * *fl*)
Apply a FLOCK-style lock to a file

Parameters

struct inode * inode inode of the file to apply to

struct file_lock * fl The lock to be applied

Description

Apply a FLOCK style lock request to an inode.

long **sys_flock**(unsigned int *fd*, unsigned int *cmd*)
flock() system call.

Parameters

unsigned int fd the file descriptor to lock.

unsigned int 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

int **`mpage_readpages`**(struct address_space * *mapping*, struct list_head * *pages*, unsigned *nr_pages*, get_block_t *get_block*)

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

Parameters

`struct address_space * mapping` the address_space

`struct list_head * 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.

`unsigned nr_pages` The number of pages at ***pages**

`get_block_t 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_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.

int **`mpage_writepages`**(struct address_space * *mapping*, struct writeback_control * *wbc*, get_block_t *get_block*)

walk the list of dirty pages of the given address space & `writepage()` all of them

Parameters

`struct address_space * mapping` address space structure to write

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

`get_block_t 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.

int **generic_permission**(struct inode * *inode*, int *mask*)
check for access rights on a Posix-like filesystem

Parameters

struct inode * inode inode to check access rights for

int mask right to check for (`MAY_READ`, `MAY_WRITE`, `MAY_EXEC`, …)

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.

generic_permission is rcu-walk aware. It returns -ECHILD in case an rcu-walk request cannot be satisfied (eg. requires blocking or too much complexity). It would then be called again in ref-walk mode.

int **__inode_permission**(struct inode * *inode*, int *mask*)
Check for access rights to a given inode

Parameters

struct inode * inode Inode to check permission on

int mask Right to check for (`MAY_READ`, `MAY_WRITE`, `MAY_EXEC`)

Description

Check for read/write/execute permissions on an inode.

When checking for MAY_APPEND, MAY_WRITE must also be set in **mask**.

This does not check for a read-only file system. You probably want *`inode_permission()`*.

int **inode_permission**(struct inode * *inode*, int *mask*)
Check for access rights to a given inode

Parameters

struct inode * inode Inode to check permission on

int mask Right to check for (`MAY_READ`, `MAY_WRITE`, `MAY_EXEC`)

Description

Check for read/write/execute permissions on an inode. We use fs[ug]id for this, letting us set arbitrary permissions for filesystem access without changing the "normal" UIDs which are used for other things.

When checking for MAY_APPEND, MAY_WRITE must also be set in **mask**.

void **path_get**(const struct path * *path*)
get a reference to a path

Parameters

const struct path * path path to get the reference to

Description

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

void **path_put**(const struct path * *path*)
put a reference to a path

Parameters

`const struct path * path` path to put the reference to

Description

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

int **`vfs_path_lookup`**(struct dentry * *dentry*, struct vfsmount * *mnt*, const char * *name*, unsigned int *flags*, struct path * *path*)
lookup a file path relative to a dentry-vfsmount pair

Parameters

`struct dentry * dentry` pointer to dentry of the base directory

`struct vfsmount * mnt` pointer to vfs mount of the base directory

`const char * name` pointer to file name

`unsigned int flags` lookup flags

`struct path * path` pointer to struct path to fill

struct dentry * **`lookup_one_len`**(const char * *name*, struct dentry * *base*, int *len*)
filesystem helper to lookup single pathname component

Parameters

`const char * name` pathname component to lookup

`struct dentry * base` base directory to lookup from

`int 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.

The caller must hold base->i_mutex.

struct dentry * **`lookup_one_len_unlocked`**(const char * *name*, struct dentry * *base*, int *len*)
filesystem helper to lookup single pathname component

Parameters

`const char * name` pathname component to lookup

`struct dentry * base` base directory to lookup from

`int 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.

Unlike lookup_one_len, it should be called without the parent i_mutex held, and will take the i_mutex itself if necessary.

int **`vfs_unlink`**(struct inode * *dir*, struct dentry * *dentry*, struct inode ** *delegated_inode*)
unlink a filesystem object

Parameters

`struct inode * dir` parent directory

`struct dentry * dentry` victim

`struct inode ** delegated_inode` returns victim inode, if the inode is delegated.

Description

The caller must hold dir->i_mutex.

If vfs_unlink discovers a delegation, it will return -EWOULDBLOCK and return a reference to the inode in delegated_inode. The caller should then break the delegation on that inode and retry. Because breaking a delegation may take a long time, the caller should drop dir->i_mutex before doing so.

Alternatively, a caller may pass NULL for delegated_inode. This may be appropriate for callers that expect the underlying filesystem not to be NFS exported.

int **`vfs_link`**(struct dentry * *old_dentry*, struct inode * *dir*, struct dentry * *new_dentry*, struct inode ** *delegated_inode*)
create a new link

Parameters

`struct dentry * old_dentry` object to be linked

`struct inode * dir` new parent

`struct dentry * new_dentry` where to create the new link

`struct inode ** delegated_inode` returns inode needing a delegation break

Description

The caller must hold dir->i_mutex

If vfs_link discovers a delegation on the to-be-linked file in need of breaking, it will return -EWOULDBLOCK and return a reference to the inode in delegated_inode. The caller should then break the delegation and retry. Because breaking a delegation may take a long time, the caller should drop the i_mutex before doing so.

Alternatively, a caller may pass NULL for delegated_inode. This may be appropriate for callers that expect the underlying filesystem not to be NFS exported.

int **`vfs_rename`**(struct inode * *old_dir*, struct dentry * *old_dentry*, struct inode * *new_dir*, struct dentry * *new_dentry*, struct inode ** *delegated_inode*, unsigned int *flags*)
rename a filesystem object

Parameters

`struct inode * old_dir` parent of source

`struct dentry * old_dentry` source

`struct inode * new_dir` parent of destination

`struct dentry * new_dentry` destination

`struct inode ** delegated_inode` returns an inode needing a delegation break

`unsigned int flags` rename flags

Description

The caller must hold multiple mutexes–see `lock_rename()`).

If vfs_rename discovers a delegation in need of breaking at either the source or destination, it will return -EWOULDBLOCK and return a reference to the inode in delegated_inode. The caller should then break the delegation and retry. Because breaking a delegation may take a long time, the caller should drop all locks before doing so.

Alternatively, a caller may pass NULL for delegated_inode. This may be appropriate for callers that expect the underlying filesystem not to be NFS exported.

The worst of all namespace operations - renaming directory. “Perverted” doesn’t even start to describe it. Somebody in UCB had a heck of a trip... Problems:

1. we can get into loop creation.
2. race potential - two innocent renames can create a loop together. That’s where 4.4 screws up. Current fix: serialization on sb->s_vfs_rename_mutex. We might be more accurate, but that’s another story.

3. we have to lock _four_ objects - parents and victim (if it exists), and source (if it is not a directory). And that - after we got ->i_mutex on parents (until then we don't know whether the target exists). Solution: try to be smart with locking order for inodes. We rely on the fact that tree topology may change only under ->s_vfs_rename_mutex _and_ that parent of the object we move will be locked. Thus we can rank directories by the tree (ancestors first) and rank all non-directories after them. That works since everybody except rename does "lock parent, lookup, lock child" and rename is under ->s_vfs_rename_mutex. HOWEVER, it relies on the assumption that any object with ->:c:func:*lookup()* has no more than 1 dentry. If "hybrid" objects will ever appear, we'd better make sure that there's no link(2) for them.

4. conversion from fhandle to dentry may come in the wrong moment - when we are removing the target. Solution: we will have to grab ->i_mutex in the fhandle_to_dentry code. [FIXME - current nfsfh.c relies on ->i_mutex on parents, which works but leads to some truly excessive locking].

int **vfs_readlink**(struct dentry * *dentry*, char __user * *buffer*, int *buflen*)
copy symlink body into userspace buffer

Parameters

struct dentry * dentry dentry on which to get symbolic link

char __user * buffer user memory pointer

int buflen size of buffer

Description

Does not touch atime. That's up to the caller if necessary

Does not call security hook.

const char * **vfs_get_link**(struct dentry * *dentry*, struct delayed_call * *done*)
get symlink body

Parameters

struct dentry * dentry dentry on which to get symbolic link

struct delayed_call * done caller needs to free returned data with this

Description

Calls security hook and i_op->:c:func:*get_link()* on the supplied inode.

It does not touch atime. That's up to the caller if necessary.

Does not work on "special" symlinks like /proc/$$/fd/N

int **sync_mapping_buffers**(struct address_space * *mapping*)
write out & wait upon a mapping's "associated" buffers

Parameters

struct address_space * 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()`.

void **mark_buffer_dirty**(struct buffer_head * *bh*)
mark a buffer_head as needing writeout

Parameters

struct buffer_head * 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 mapping->host->i_lock.

struct buffer_head * **__bread_gfp**(struct block_device * *bdev*, sector_t *block*, unsigned *size*, gfp_t *gfp*)
reads a specified block and returns the bh

Parameters

struct block_device * bdev the block_device to read from

sector_t block number of block

unsigned size size (in bytes) to read

gfp_t gfp page allocation flag

Description

Reads a specified block, and returns buffer head that contains it. The page cache can be allocated from non-movable area not to prevent page migration if you set gfp to zero. It returns NULL if the block was unreadable.

void **block_invalidatepage**(struct page * *page*, unsigned int *offset*, unsigned int *length*)
invalidate part or all of a buffer-backed page

Parameters

struct page * page the page which is affected

unsigned int offset start of the range to invalidate

unsigned int length length of the range to invalidate

Description

block_invalidatepage() is called when all or part of the page has become invalidated by 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.

void **clean_bdev_aliases**(struct block_device * *bdev*, sector_t *block*, sector_t *len*)

Parameters

struct block_device * bdev Block device to clean buffers in

sector_t block Start of a range of blocks to clean

sector_t len Number of blocks to clean

Description

We are taking a range of blocks for data and we don't want writeback of any buffer-cache aliases starting from return from this function and until the moment when something will explicitly mark the buffer dirty (hopefully that will not happen until we will free that block ;-) We don't even need to mark it not-uptodate - nobody can expect anything from a newly allocated buffer anyway. We used to use `unmap_buffer()` for such invalidation, but that was wrong. We definitely don't want to mark the alias unmapped, for example - it would confuse anyone who might pick it with `bread()` afterwards...

Also.. Note that `bforget()` doesn't lock the buffer. So there can be writeout I/O going on against recently-freed buffers. We don't wait on that I/O in `bforget()` - it's more efficient to wait on the I/O only if we really need to. That happens here.

void **`ll_rw_block`**(int *op*, int *op_flags*, int *nr*, struct buffer_head * *bhs*)
level access to block devices (DEPRECATED)

Parameters

`int op` whether to READ or WRITE

`int op_flags` req_flag_bits

`int nr` number of `struct buffer_heads` in the array

`struct buffer_head * 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 `REQ_OP_READ` or a `REQ_OP_WRITE`. **op_flags** contains flags modifying the detailed I/O behavior, most notably `REQ_RAHEAD`.

This function drops any buffer that it cannot get a lock on (with the BH_Lock state bit), 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 appropriate), 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.

int **`bh_uptodate_or_lock`**(struct buffer_head * *bh*)
Test whether the buffer is uptodate

Parameters

`struct buffer_head * bh` struct buffer_head

Description

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

int **`bh_submit_read`**(struct buffer_head * *bh*)
Submit a locked buffer for reading

Parameters

`struct buffer_head * bh` struct buffer_head

Description

Returns zero on success and -EIO on error.

void **`bio_reset`**(struct bio * *bio*)
reinitialize a bio

Parameters

`struct bio * bio` bio to reset

Description

> After calling *`bio_reset()`*, **bio** will be in the same state as a freshly allocated bio returned bio *`bio_alloc_bioset()`* - the only fields that are preserved are the ones that are initialized by *`bio_alloc_bioset()`*. See comment in struct bio.

void **`bio_chain`**(struct bio * *bio*, struct bio * *parent*)
chain bio completions

Parameters

`struct bio * bio` the target bio

`struct bio * parent` the **bio**'s parent bio

Description

The caller won't have a bi_end_io called when **bio** completes - instead, **parent**'s bi_end_io won't be called until both **parent** and **bio** have completed; the chained bio will also be freed when it completes.

The caller must not set bi_private or bi_end_io in **bio**.

struct bio * **`bio_alloc_bioset`**(gfp_t *gfp_mask*, unsigned int *nr_iovecs*, struct bio_set * *bs*)
allocate a bio for I/O

Parameters

`gfp_t gfp_mask` the **GFP_** mask given to the slab allocator

`unsigned int nr_iovecs` number of iovecs to pre-allocate

`struct bio_set * bs` the bio_set to allocate from.

Description

If **bs** is NULL, uses `kmalloc()` to allocate the bio; else the allocation is backed by the **bs**'s mempool.

When **bs** is not NULL, if `__GFP_DIRECT_RECLAIM` 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 deadlocks under memory pressure.

Note that when running under `generic_make_request()` (i.e. any block driver), bios are not submitted until after you return - see the code in `generic_make_request()` that converts recursion into iteration, to prevent stack overflows.

This would normally mean allocating multiple bios under `generic_make_request()` would be susceptible to deadlocks, but we have deadlock avoidance code that resubmits any blocked bios from a rescuer thread.

However, we do not guarantee forward progress for allocations from other mempools. Doing multiple allocations from the same mempool under `generic_make_request()` should be avoided - instead, use bio_set's front_pad for per bio allocations.

Return

Pointer to new bio on success, NULL on failure.

void **`bio_put`**(struct bio * *bio*)
release a reference to a bio

Parameters

`struct bio * 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.

void **`__bio_clone_fast`**(struct bio * *bio*, struct bio * *bio_src*)
clone a bio that shares the original bio's biovec

Parameters

`struct bio * bio` destination bio

`struct 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.

Caller must ensure that **bio_src** is not freed before **bio**.

struct bio * **`bio_clone_fast`**(struct bio * *bio*, gfp_t *gfp_mask*, struct bio_set * *bs*)
clone a bio that shares the original bio's biovec

Parameters

`struct bio * bio` bio to clone

`gfp_t gfp_mask` allocation priority

`struct bio_set * bs` bio_set to allocate from

Description

Like __bio_clone_fast, only also allocates the returned bio

struct bio * **`bio_clone_bioset`**(struct bio * *bio_src*, gfp_t *gfp_mask*, struct bio_set * *bs*)
clone a bio

Parameters

`struct bio * bio_src` bio to clone

`gfp_t gfp_mask` allocation priority

`struct bio_set * bs` bio_set to allocate from

Description

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

int **`bio_add_pc_page`**(struct request_queue * *q*, struct bio * *bio*, struct page * *page*, unsigned int *len*, unsigned int *offset*)
attempt to add page to bio

Parameters

`struct request_queue * q` the target queue

`struct bio * bio` destination bio

`struct page * page` page to add

`unsigned int len` vec entry length

`unsigned int 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 up to 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.

int **`bio_add_page`**(struct bio * *bio*, struct page * *page*, unsigned int *len*, unsigned int *offset*)
attempt to add page to bio

Parameters

`struct bio * bio` destination bio

`struct page * page` page to add

`unsigned int len` vec entry length

`unsigned int offset` vec entry offset

Description

Attempt to add a page to the bio_vec maplist. This will only fail if either bio->bi_vcnt == bio->bi_max_vecs or it's a cloned bio.

int **`bio_iov_iter_get_pages`**(struct bio * *bio*, struct iov_iter * *iter*)
pin user or kernel pages and add them to a bio

Parameters

`struct bio * bio` bio to add pages to

`struct iov_iter * iter` iov iterator describing the region to be mapped

Description

Pins as many pages from *iter and appends them to **bio*'s bvec array. The pages will have to be released using `put_page()` when done.

int **`submit_bio_wait`**(struct bio * *bio*)
submit a bio, and wait until it completes

Parameters

`struct bio * bio` The `struct bio` which describes the I/O

Description

Simple wrapper around `submit_bio()`. Returns 0 on success, or the error from *`bio_endio()`* on failure.

void **`bio_advance`**(struct bio * *bio*, unsigned *bytes*)
increment/complete a bio by some number of bytes

Parameters

`struct bio * bio` bio to advance

`unsigned bytes` number of bytes to complete

Description

This updates bi_sector, bi_size and bi_idx; if the number of bytes to complete doesn't align with a bvec boundary, then bv_len and bv_offset will be updated on the last bvec as well.

bio will then represent the remaining, uncompleted portion of the io.

int **`bio_alloc_pages`**(struct bio * *bio*, gfp_t *gfp_mask*)
allocates a single page for each bvec in a bio

Parameters

`struct bio * bio` bio to allocate pages for

`gfp_t gfp_mask` flags for allocation

Description

Allocates pages up to **bio**->bi_vcnt.

Returns 0 on success, -ENOMEM on failure. On failure, any allocated pages are freed.

void **`bio_copy_data`**(struct bio * *dst*, struct bio * *src*)
copy contents of data buffers from one chain of bios to another

Parameters

`struct bio * dst` destination bio list

`struct bio * src` source bio list

Description

If **src** and **dst** are single bios, bi_next must be NULL - otherwise, treats **src** and **dst** as linked lists of bios.

Stops when it reaches the end of either **src** or **dst** - that is, copies min(src->bi_size, dst->bi_size) bytes (or the equivalent for lists of bios).

struct bio * **`bio_map_kern`**(struct request_queue * *q*, void * *data*, unsigned int *len*, gfp_t *gfp_mask*)
map kernel address into bio

Parameters

struct request_queue * q the struct request_queue for the bio

void * data pointer to buffer to map

unsigned int len length in bytes

gfp_t 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.

void **bio_endio**(struct bio * *bio*)
end I/O on a bio

Parameters

struct bio * bio bio

Description

bio_endio() will end I/O on the whole bio. *bio_endio()* is the preferred way to end I/O on a bio. No one should call `bi_end_io()` directly on a bio unless they own it and thus know that it has an end_io function.

bio_endio() can be called several times on a bio that has been chained using *bio_chain()*. The ->:c:func:*bi_end_io()* function will only be called the last time. At this point the BLK_TA_COMPLETE tracing event will be generated if BIO_TRACE_COMPLETION is set.

struct bio * **bio_split**(struct bio * *bio*, int *sectors*, gfp_t *gfp*, struct bio_set * *bs*)
split a bio

Parameters

struct bio * bio bio to split

int sectors number of sectors to split from the front of **bio**

gfp_t gfp gfp mask

struct bio_set * bs bio set to allocate from

Description

Allocates and returns a new bio which represents **sectors** from the start of **bio**, and updates **bio** to represent the remaining sectors.

Unless this is a discard request the newly allocated bio will point to **bio**'s bi_io_vec; it is the caller's responsibility to ensure that **bio** is not freed before the split.

void **bio_trim**(struct bio * *bio*, int *offset*, int *size*)
trim a bio

Parameters

struct bio * bio bio to trim

int offset number of sectors to trim from the front of **bio**

int size size we want to trim **bio** to, in sectors

struct bio_set * **bioset_create**(unsigned int *pool_size*, unsigned int *front_pad*, int *flags*)
Create a bio_set

Parameters

unsigned int pool_size Number of bio and bio_vecs to cache in the mempool

unsigned int front_pad Number of bytes to allocate in front of the returned bio

int flags Flags to modify behavior, currently `BIOSET_NEED_BVECS` and `BIOSET_NEED_RESCUER`

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. If `BIOSET_NEED_BVECS` is set in **flags**, a separate pool will be allocated for allocating iovecs. This pool is not needed e.g. for *`bio_clone_fast()`*. If `BIOSET_NEED_RESCUER` is set, a workqueue is created which can be used to dispatch queued requests when the mempool runs out of space.

int **`bio_associate_blkcg`**(struct bio * *bio*, struct cgroup_subsys_state * *blkcg_css*)
associate a bio with the specified blkcg

Parameters

`struct bio * bio` target bio

`struct cgroup_subsys_state * blkcg_css` css of the blkcg to associate

Description

Associate **bio** with the blkcg specified by **blkcg_css**. Block layer will treat **bio** as if it were issued by a task which belongs to the blkcg.

This function takes an extra reference of **blkcg_css** which will be put when **bio** is released. The caller must own **bio** and is responsible for synchronizing calls to this function.

int **`bio_associate_current`**(struct bio * *bio*)
associate a bio with `current`

Parameters

`struct bio * bio` target bio

Description

Associate **bio** with `current` if it hasn't been associated yet. Block layer will treat **bio** as if it were issued by `current` no matter which task actually issues it.

This function takes an extra reference of **task**'s io_context and blkcg which will be put when **bio** is released. The caller must own **bio**, ensure `current->`io_context exists, and is responsible for synchronizing calls to this function.

int **`seq_open`**(struct file * *file*, const struct seq_operations * *op*)
initialize sequential file

Parameters

`struct file * file` file we initialize

`const struct seq_operations * op` method table describing the sequence

Description

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

Note

`seq_open()` **will allocate a struct seq_file and store its** pointer in **file**->private_data. This pointer should not be modified.

ssize_t **`seq_read`**(struct file * *file*, char __user * *buf*, size_t *size*, loff_t * *ppos*)
->:c:func:*read()* method for sequential files.

Parameters

struct file * file the file to read from

char __user * buf the buffer to read to

size_t size the maximum number of bytes to read

loff_t * ppos the current position in the file

Description

Ready-made ->f_op->:c:func:*read()*

loff_t **seq_lseek**(struct file * *file*, loff_t *offset*, int *whence*)
->:c:func:*llseek()* method for sequential files.

Parameters

struct file * file the file in question

loff_t offset new position

int whence 0 for absolute, 1 for relative position

Description

Ready-made ->f_op->:c:func:*llseek()*

int **seq_release**(struct inode * *inode*, struct file * *file*)
free the structures associated with sequential file.

Parameters

struct inode * inode its inode

struct file * file file in question

Description

Frees the structures associated with sequential file; can be used as ->f_op->:c:func:*release()* if you don't have private data to destroy.

void **seq_escape**(struct seq_file * *m*, const char * *s*, const char * *esc*)
print string into buffer, escaping some characters

Parameters

struct seq_file * m target buffer

const char * s string

const char * esc set of characters that need escaping

Description

Puts string into buffer, replacing each occurrence of character from **esc** with usual octal escape. Use `seq_has_overflowed()` to check for errors.

char * **mangle_path**(char * *s*, const char * *p*, const char * *esc*)
mangle and copy path to buffer beginning

Parameters

char * s buffer start

const char * p beginning of path in above buffer

const char * 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.

int **seq_path**(struct seq_file * *m*, const struct path * *path*, const char * *esc*)
seq_file interface to print a pathname

Parameters

`struct seq_file * m` the seq_file handle

`const struct path * path` the struct path to print

`const char * 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.

int **`seq_file_path`**(struct seq_file * *m*, struct file * *file*, const char * *esc*)
seq_file interface to print a pathname of a file

Parameters

`struct seq_file * m` the seq_file handle

`struct file * file` the struct file to print

`const char * esc` set of characters to escape in the output

Description

return the absolute path to the file.

int **`seq_write`**(struct seq_file * *seq*, const void * *data*, size_t *len*)
write arbitrary data to buffer

Parameters

`struct seq_file * seq` seq_file identifying the buffer to which data should be written

`const void * data` data address

`size_t len` number of bytes

Description

Return 0 on success, non-zero otherwise.

void **`seq_pad`**(struct seq_file * *m*, char *c*)
write padding spaces to buffer

Parameters

`struct seq_file * m` seq_file identifying the buffer to which data should be written

`char c` the byte to append after padding if non-zero

struct hlist_node * **`seq_hlist_start`**(struct hlist_head * *head*, loff_t *pos*)
start an iteration of a hlist

Parameters

`struct hlist_head * head` the head of the hlist

`loff_t pos` the start position of the sequence

Description

Called at seq_file->op->:c:func:*start()*.

struct hlist_node * **`seq_hlist_start_head`**(struct hlist_head * *head*, loff_t *pos*)
start an iteration of a hlist

Parameters

`struct hlist_head * head` the head of the hlist

`loff_t pos` the start position of the sequence

Description

Called at seq_file->op->:c:func:*start()*. Call this function if you want to print a header at the top of the output.

struct hlist_node * **`seq_hlist_next`**(void * *v*, struct hlist_head * *head*, loff_t * *ppos*)
    move to the next position of the hlist

Parameters

`void * v` the current iterator

`struct hlist_head * head` the head of the hlist

`loff_t * ppos` the current position

Description

Called at seq_file->op->:c:func:*next()*.

struct hlist_node * **`seq_hlist_start_rcu`**(struct hlist_head * *head*, loff_t *pos*)
    start an iteration of a hlist protected by RCU

Parameters

`struct hlist_head * head` the head of the hlist

`loff_t pos` the start position of the sequence

Description

Called at seq_file->op->:c:func:*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()`.

struct hlist_node * **`seq_hlist_start_head_rcu`**(struct hlist_head * *head*, loff_t *pos*)
    start an iteration of a hlist protected by RCU

Parameters

`struct hlist_head * head` the head of the hlist

`loff_t pos` the start position of the sequence

Description

Called at seq_file->op->:c:func:*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()`.

struct hlist_node * **`seq_hlist_next_rcu`**(void * *v*, struct hlist_head * *head*, loff_t * *ppos*)
    move to the next position of the hlist protected by RCU

Parameters

`void * v` the current iterator

`struct hlist_head * head` the head of the hlist

`loff_t * ppos` the current position

Description

Called at seq_file->op->:c:func:*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()`.

struct hlist_node * **`seq_hlist_start_percpu`**(struct hlist_head __percpu * *head*, int * *cpu*, loff_t *pos*)
    start an iteration of a percpu hlist array

Parameters

struct hlist_head __percpu * head pointer to percpu array of struct hlist_heads

int * cpu pointer to cpu "cursor"

loff_t pos start position of sequence

Description

Called at seq_file->op->:c:func:*start()*.

struct hlist_node * **seq_hlist_next_percpu**(void * *v*, struct hlist_head __percpu * *head*, int * *cpu*, loff_t * *pos*)
move to the next position of the percpu hlist array

Parameters

void * v pointer to current hlist_node

struct hlist_head __percpu * head pointer to percpu array of struct hlist_heads

int * cpu pointer to cpu "cursor"

loff_t * pos start position of sequence

Description

Called at seq_file->op->:c:func:*next()*.

int **register_filesystem**(struct file_system_type * *fs*)
register a new filesystem

Parameters

struct file_system_type * 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.

int **unregister_filesystem**(struct file_system_type * *fs*)
unregister a file system

Parameters

struct file_system_type * 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.

void **wbc_account_io**(struct writeback_control * *wbc*, struct page * *page*, size_t *bytes*)
account IO issued during writeback

Parameters

struct writeback_control * wbc writeback_control of the writeback in progress

struct page * page page being written out

size_t bytes number of bytes being written out

Description

bytes from **page** are about to written out during the writeback controlled by **wbc**. Keep the book for foreign inode detection. See `wbc_detach_inode()`.

int **`inode_congested`**(struct inode * *inode*, int *cong_bits*)
    test whether an inode is congested

Parameters

`struct inode * inode` inode to test for congestion (may be NULL)

`int cong_bits` mask of WB_[a]sync_congested bits to test

Description

Tests whether **inode** is congested. **cong_bits** is the mask of congestion bits to test and the return value is the mask of set bits.

If cgroup writeback is enabled for **inode**, the congestion state is determined by whether the cgwb (cgroup bdi_writeback) for the blkcg associated with **inode** is congested; otherwise, the root wb's congestion state is used.

inode is allowed to be NULL as this function is often called on mapping->host which is NULL for the swapper space.

void **`__mark_inode_dirty`**(struct inode * *inode*, int *flags*)
    internal function

Parameters

`struct inode * inode` inode to mark

`int flags` what kind of dirty (i.e. I_DIRTY_SYNC)

Description

Mark an inode as dirty. Callers should use mark_inode_dirty or mark_inode_dirty_sync.

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.

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.

void **`writeback_inodes_sb_nr`**(struct super_block * *sb*, unsigned long *nr*, enum wb_reason *reason*)
    writeback dirty inodes from given super_block

Parameters

`struct super_block * sb` the superblock

`unsigned long nr` the number of pages to write

`enum wb_reason reason` reason why some writeback work initiated

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.

void **`writeback_inodes_sb`**(struct super_block * *sb*, enum wb_reason *reason*)
    writeback dirty inodes from given super_block

Parameters

struct super_block * sb the superblock

enum wb_reason reason reason why some writeback work was initiated

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.

bool **try_to_writeback_inodes_sb_nr**(struct super_block * *sb*, unsigned long *nr*, enum wb_reason *reason*)
try to start writeback if none underway

Parameters

struct super_block * sb the superblock

unsigned long nr the number of pages to write

enum wb_reason reason the reason of writeback

Description

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

bool **try_to_writeback_inodes_sb**(struct super_block * *sb*, enum wb_reason *reason*)
try to start writeback if none underway

Parameters

struct super_block * sb the superblock

enum wb_reason reason reason why some writeback work was initiated

Description

Implement by *try_to_writeback_inodes_sb_nr()* Returns 1 if writeback was started, 0 if not.

void **sync_inodes_sb**(struct super_block * *sb*)
sync sb inode pages

Parameters

struct super_block * sb the superblock

Description

This function writes and waits on any dirty inode belonging to this super_block.

int **write_inode_now**(struct inode * *inode*, int *sync*)
write an inode to disk

Parameters

struct inode * inode inode to write to disk

int 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.

int **sync_inode**(struct inode * *inode*, struct writeback_control * *wbc*)
write an inode and its pages to disk.

Parameters

struct inode * inode the inode to sync

struct writeback_control * 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.

int **sync_inode_metadata**(struct inode * *inode*, int *wait*)
write an inode to disk

Parameters

struct inode * inode the inode to sync

int wait wait for I/O to complete.

Description

Write an inode to disk and adjust its dirty state after completion.

Note

only writes the actual inode, no associated data or other metadata.

struct super_block * **freeze_bdev**(struct block_device * *bdev*)

- lock a filesystem and force it into a consistent state

Parameters

struct block_device * 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.

int **thaw_bdev**(struct block_device * *bdev*, struct super_block * *sb*)

- unlock filesystem

Parameters

struct block_device * bdev blockdevice to unlock

struct super_block * sb associated superblock

Description

Unlocks the filesystem and marks it writeable again after *freeze_bdev()*.

int **bdev_read_page**(struct block_device * *bdev*, sector_t *sector*, struct page * *page*)
Start reading a page from a block device

Parameters

struct block_device * bdev The device to read the page from

sector_t sector The offset on the device to read the page to (need not be aligned)

struct page * page The page to read

Description

On entry, the page should be locked. It will be unlocked when the page has been read. If the block driver implements rw_page synchronously, that will be true on exit from this function, but it need not be.

Errors returned by this function are usually "soft", eg out of memory, or queue full; callers should try a different route to read this page rather than propagate an error back up the stack.

Return

negative errno if an error occurs, 0 if submission was successful.

int **bdev_write_page**(struct block_device * *bdev*, sector_t *sector*, struct page * *page*, struct writeback_control * *wbc*)
Start writing a page to a block device

Parameters

struct block_device * bdev The device to write the page to

sector_t sector The offset on the device to write the page to (need not be aligned)

struct page * page The page to write

struct writeback_control * wbc The writeback_control for the write

Description

On entry, the page should be locked and not currently under writeback. On exit, if the write started successfully, the page will be unlocked and under writeback. If the write failed already (eg the driver failed to queue the page to the device), the page will still be locked. If the caller is a ->writepage implementation, it will need to unlock the page.

Errors returned by this function are usually "soft", eg out of memory, or queue full; callers should try a different route to write this page rather than propagate an error back up the stack.

Return

negative errno if an error occurs, 0 if submission was successful.

struct block_device * **bdgrab**(struct block_device * *bdev*)

•Grab a reference to an already referenced block device

Parameters

struct block_device * bdev Block device to grab a reference to.

int **bd_link_disk_holder**(struct block_device * *bdev*, struct gendisk * *disk*)
create symlinks between holding disk and slave bdev

Parameters

struct block_device * bdev the claimed slave bdev

struct gendisk * disk the holding disk

Description

DON'T USE THIS UNLESS YOU'RE ALREADY USING IT.

This functions creates the following sysfs symlinks.

- from "slaves" directory of the holder **disk** to the claimed **bdev**
- from "holders" directory of the **bdev** to the holder **disk**

For example, if /dev/dm-0 maps to /dev/sda and disk for dm-0 is passed to *bd_link_disk_holder()*, then:

/sys/block/dm-0/slaves/sda –> /sys/block/sda /sys/block/sda/holders/dm-0 –> /sys/block/dm-0

The caller must have claimed **bdev** before calling this function and ensure that both **bdev** and **disk** are valid during the creation and lifetime of these symlinks.

Context

Might sleep.

Return

0 on success, -errno on failure.

void **bd_unlink_disk_holder**(struct block_device * *bdev*, struct gendisk * *disk*)
destroy symlinks created by *bd_link_disk_holder()*

Parameters

`struct block_device * bdev` the calimed slave bdev

`struct gendisk * disk` the holding disk

Description

DON'T USE THIS UNLESS YOU'RE ALREADY USING IT.

Context

Might sleep.

void **`check_disk_size_change`**(struct gendisk * *disk*, struct block_device * *bdev*)
checks for disk size change and adjusts bdev size.

Parameters

`struct gendisk * disk` struct gendisk to check

`struct block_device * 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.

int **`revalidate_disk`**(struct gendisk * *disk*)
wrapper for lower-level driver's revalidate_disk call-back

Parameters

`struct gendisk * 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.

int **`blkdev_get`**(struct block_device * *bdev*, fmode_t *mode*, void * *holder*)
open a block device

Parameters

`struct block_device * bdev` block_device to open

`fmode_t mode` FMODE_* mask

`void * holder` exclusive holder identifier

Description

Open **bdev** with **mode**. If **mode** includes `FMODE_EXCL`, **bdev** is open with exclusive access. Specifying `FMODE_EXCL` with `NULL` **holder** is invalid. Exclusive opens may nest for the same **holder**.

On success, the reference count of **bdev** is unchanged. On failure, **bdev** is put.

Context

Might sleep.

Return

0 on success, -errno on failure.

struct block_device * **`blkdev_get_by_path`**(const char * *path*, fmode_t *mode*, void * *holder*)
open a block device by name

Parameters

`const char * path` path to the block device to open

`fmode_t mode` FMODE_* mask

`void * holder` exclusive holder identifier

Description

Open the blockdevice described by the device file at **path**. **mode** and **holder** are identical to *blkdev_get()*.

On success, the returned block_device has reference count of one.

Context

Might sleep.

Return

Pointer to block_device on success, ERR_PTR(-errno) on failure.

struct block_device * **blkdev_get_by_dev**(dev_t *dev*, fmode_t *mode*, void * *holder*)
open a block device by device number

Parameters

dev_t dev device number of block device to open

fmode_t mode FMODE_* mask

void * holder exclusive holder identifier

Description

Open the blockdevice described by device number **dev**. **mode** and **holder** are identical to *blkdev_get()*.

Use it ONLY if you really do not have anything better - i.e. when you are behind a truly sucky interface and all you are given is a device number. _Never_ to be used for internal purposes. If you ever need it - reconsider your API.

On success, the returned block_device has reference count of one.

Context

Might sleep.

Return

Pointer to block_device on success, ERR_PTR(-errno) on failure.

struct block_device * **lookup_bdev**(const char * *pathname*)
lookup a struct block_device by name

Parameters

const char * 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.

CHAPTER

TWO

THE PROC FILESYSTEM

2.1 sysctl interface

int **proc_dostring**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a string sysctl

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_dointvec**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of integers

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_douintvec**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of unsigned integers

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

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

Returns 0 on success.

int **proc_dointvec_minmax**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of integers with min/max values

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_douintvec_minmax**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of unsigned ints with min/max values

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

Reads/writes up to table->maxlen/sizeof(unsigned int) unsigned integer values from/to the user buffer, treated as an ASCII string. Negative strings are not allowed.

This routine will ensure the values are within the range specified by table->extra1 (min) and table->extra2 (max). There is a final sanity check for UINT_MAX to avoid having to support wrap around uses from userspace.

Returns 0 on success.

int **proc_doulongvec_minmax**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of long integers with min/max values

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_doulongvec_ms_jiffies_minmax**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of millisecond values with min/max values

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_dointvec_jiffies**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of integers as seconds

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_dointvec_userhz_jiffies**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of integers as 1/USER_HZ seconds

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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.

int **proc_dointvec_ms_jiffies**(struct ctl_table * *table*, int *write*, void __user * *buffer*, size_t * *lenp*, loff_t * *ppos*)
read a vector of integers as 1 milliseconds

Parameters

struct ctl_table * table the sysctl table

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

void __user * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * 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

void **proc_flush_task**(struct task_struct * *task*)
Remove dcache entries for **task** from the /proc dcache.

Parameters

struct task_struct * 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

THREE

EVENTS BASED ON FILE DESCRIPTORS

__u64 **eventfd_signal**(struct eventfd_ctx * *ctx*, __u64 *n*)
Adds **n** to the eventfd counter.

Parameters

struct eventfd_ctx * ctx [in] Pointer to the eventfd context.

__u64 n [in] Value of the counter to be added to the eventfd internal counter. The value cannot be negative.

Description

This function is supposed to be called by the kernel in paths that do not allow sleeping. In this function we allow the counter to reach the ULLONG_MAX value, and we signal this as overflow condition by returning a POLLERR to poll(2).

Returns the amount by which the counter was incremented. This will be less than **n** if the counter has overflowed.

struct eventfd_ctx * **eventfd_ctx_get**(struct eventfd_ctx * *ctx*)
Acquires a reference to the internal eventfd context.

Parameters

struct eventfd_ctx * ctx [in] Pointer to the eventfd context.

Return

In case of success, returns a pointer to the eventfd context.

void **eventfd_ctx_put**(struct eventfd_ctx * *ctx*)
Releases a reference to the internal eventfd context.

Parameters

struct eventfd_ctx * ctx [in] Pointer to eventfd context.

Description

The eventfd context reference must have been previously acquired either with *eventfd_ctx_get()* or *eventfd_ctx_fdget()*.

int **eventfd_ctx_remove_wait_queue**(struct eventfd_ctx * *ctx*, wait_queue_entry_t * *wait*, __u64 * *cnt*)
Read the current counter and removes wait queue.

Parameters

struct eventfd_ctx * ctx [in] Pointer to eventfd context.

wait_queue_entry_t * wait [in] Wait queue to be removed.

__u64 * cnt [out] Pointer to the 64-bit counter value.

Description

Returns `0` if successful, or the following error codes:

-EAGAIN : The operation would have blocked.

This is used to atomically remove a wait queue entry from the eventfd wait queue head, and read/reset the counter value.

ssize_t **eventfd_ctx_read**(struct eventfd_ctx * *ctx*, int *no_wait*, __u64 * *cnt*)
Reads the eventfd counter or wait if it is zero.

Parameters

struct eventfd_ctx * ctx [in] Pointer to eventfd context.

int no_wait [in] Different from zero if the operation should not block.

__u64 * cnt [out] Pointer to the 64-bit counter value.

Description

Returns `0` if successful, or the following error codes:

- **-EAGAIN** : The operation would have blocked but **no_wait** was non-zero.
- -ERESTARTSYS : A signal interrupted the wait operation.

If **no_wait** is zero, the function might sleep until the eventfd internal counter becomes greater than zero.

struct file * **eventfd_fget**(int *fd*)
Acquire a reference of an eventfd file descriptor.

Parameters

int fd [in] Eventfd file descriptor.

Description

Returns a pointer to the eventfd file structure in case of success, or the following error pointer:

-EBADF : Invalid **fd** file descriptor.

-EINVAL : The **fd** file descriptor is not an eventfd file.

struct eventfd_ctx * **eventfd_ctx_fdget**(int *fd*)
Acquires a reference to the internal eventfd context.

Parameters

int fd [in] Eventfd file descriptor.

Description

Returns a pointer to the internal eventfd context, otherwise the error pointers returned by the following functions:

eventfd_fget

struct eventfd_ctx * **eventfd_ctx_fileget**(struct file * *file*)
Acquires a reference to the internal eventfd context.

Parameters

struct file * file [in] Eventfd file pointer.

Description

Returns a pointer to the internal eventfd context, otherwise the error pointer:

-EINVAL : The **fd** file descriptor is not an eventfd file.

CHAPTER

FOUR

THE FILESYSTEM FOR EXPORTING KERNEL OBJECTS

int **sysfs_create_file_ns**(struct kobject * *kobj*, const struct attribute * *attr*, const void * *ns*)
create an attribute file for an object with custom ns

Parameters

struct kobject * kobj object we're creating for

const struct attribute * attr attribute descriptor

const void * ns namespace the new file should belong to

int **sysfs_add_file_to_group**(struct kobject * *kobj*, const struct attribute * *attr*, const char * *group*)
add an attribute file to a pre-existing group.

Parameters

struct kobject * kobj object we're acting for.

const struct attribute * attr attribute descriptor.

const char * group group name.

int **sysfs_chmod_file**(struct kobject * *kobj*, const struct attribute * *attr*, umode_t *mode*)
update the modified mode value on an object attribute.

Parameters

struct kobject * kobj object we're acting for.

const struct attribute * attr attribute descriptor.

umode_t mode file permissions.

void **sysfs_remove_file_ns**(struct kobject * *kobj*, const struct attribute * *attr*, const void * *ns*)
remove an object attribute with a custom ns tag

Parameters

struct kobject * kobj object we're acting for

const struct attribute * attr attribute descriptor

const void * ns namespace tag of the file to remove

Description

Hash the attribute name and namespace tag and kill the victim.

void **sysfs_remove_file_from_group**(struct kobject * *kobj*, const struct attribute * *attr*, const char * *group*)
remove an attribute file from a group.

Parameters

struct kobject * kobj object we're acting for.

const struct attribute * attr attribute descriptor.

`const char * group` group name.

int **`sysfs_create_bin_file`**(struct kobject * *kobj*, const struct bin_attribute * *attr*)
create binary file for object.

Parameters

`struct kobject * kobj` object.

`const struct bin_attribute * attr` attribute descriptor.

void **`sysfs_remove_bin_file`**(struct kobject * *kobj*, const struct bin_attribute * *attr*)
remove binary file for object.

Parameters

`struct kobject * kobj` object.

`const struct bin_attribute * attr` attribute descriptor.

int **`sysfs_create_link`**(struct kobject * *kobj*, struct kobject * *target*, const char * *name*)
create symlink between two objects.

Parameters

`struct kobject * kobj` object whose directory we're creating the link in.

`struct kobject * target` object we're pointing to.

`const char * name` name of the symlink.

void **`sysfs_remove_link`**(struct kobject * *kobj*, const char * *name*)
remove symlink in object's directory.

Parameters

`struct kobject * kobj` object we're acting for.

`const char * name` name of the symlink to remove.

int **`sysfs_rename_link_ns`**(struct kobject * *kobj*, struct kobject * *targ*, const char * *old*, const char * *new*, const void * *new_ns*)
rename symlink in object's directory.

Parameters

`struct kobject * kobj` object we're acting for.

`struct kobject * targ` object we're pointing to.

`const char * old` previous name of the symlink.

`const char * new` new name of the symlink.

`const void * new_ns` new namespace of the symlink.

Description

A helper function for the common rename symlink idiom.

CHAPTER

FIVE

THE DEBUGFS FILESYSTEM

5.1 debugfs interface

struct dentry * **debugfs_lookup**(const char * *name*, struct dentry * *parent*)
look up an existing debugfs file

Parameters

const char * name a pointer to a string containing the name of the file to look up.

struct dentry * parent a pointer to the parent dentry of the file.

Description

This function will return a pointer to a dentry if it succeeds. If the file doesn't exist or an error occurs, NULL will be returned. The returned dentry must be passed to dput() when it is no longer needed.

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

struct dentry * **debugfs_create_file**(const char * *name*, umode_t *mode*, struct dentry * *parent*, void * *data*, const struct file_operations * *fops*)
create a file in the debugfs filesystem

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have.

struct dentry * 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.

void * 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.

const struct file_operations * 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.

struct dentry * **debugfs_create_file_unsafe**(const char * *name*, umode_t *mode*, struct dentry * *parent*, void * *data*, const struct file_operations * *fops*)
create a file in the debugfs filesystem

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have.

struct dentry * 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.

void * 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.

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

Description

debugfs_create_file_unsafe() is completely analogous to *debugfs_create_file()*, the only difference being that the fops handed it will not get protected against file removals by the debugfs core.

It is your responsibility to protect your struct file_operation methods against file removals by means of *debugfs_use_file_start()* and *debugfs_use_file_finish()*. ->:c:func:*open()* is still protected by debugfs though.

Any struct file_operations defined by means of `DEFINE_DEBUGFS_ATTRIBUTE()` is protected against file removals and thus, may be used here.

struct dentry * **debugfs_create_file_size**(const char * *name*, umode_t *mode*, struct dentry * *parent*, void * *data*, const struct file_operations * *fops*, loff_t *file_size*)
create a file in the debugfs filesystem

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have.

struct dentry * 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.

void * 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.

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

loff_t file_size initial file size

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.

struct dentry * **debugfs_create_dir**(const char * *name*, struct dentry * *parent*)
create a directory in the debugfs filesystem

Parameters

const char * name a pointer to a string containing the name of the directory to create.

struct dentry * 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 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.

struct dentry * **`debugfs_create_automount`**(const char * *name*, struct dentry * *parent*, debugfs_automount_t *f*, void * *data*)
create automount point in the debugfs filesystem

Parameters

`const char * name` a pointer to a string containing the name of the file to create.

`struct dentry * 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.

`debugfs_automount_t f` function to be called when pathname resolution steps on that one.

`void * data` opaque argument to pass to `f()`.

Description

f should return what ->:c:func:*d_automount()* would.

struct dentry * **`debugfs_create_symlink`**(const char * *name*, struct dentry * *parent*, const char * *target*)
create a symbolic link in the debugfs filesystem

Parameters

`const char * name` a pointer to a string containing the name of the symbolic link to create.

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

`const char * 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.

void **`debugfs_remove`**(struct dentry * *dentry*)
removes a file or directory from the debugfs filesystem

Parameters

`struct dentry * dentry` a pointer to a the dentry of the file or directory to be removed. If this parameter is NULL or an error value, nothing will be done.

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.

void **`debugfs_remove_recursive`**(struct dentry * *dentry*)
recursively removes a directory

Parameters

struct dentry * dentry a pointer to a the dentry of the directory to be removed. If this parameter is NULL or an error value, nothing will be done.

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.

struct dentry * **debugfs_rename**(struct dentry * *old_dir*, struct dentry * *old_dentry*, struct dentry * *new_dir*, const char * *new_name*)
rename a file/directory in the debugfs filesystem

Parameters

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

struct dentry * old_dentry dentry of an object to be renamed.

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

const char * 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.

bool **debugfs_initialized**(void)
Tells whether debugfs has been registered

Parameters

void no arguments

int **debugfs_use_file_start**(const struct dentry * *dentry*, int * *srcu_idx*)
mark the beginning of file data access

Parameters

const struct dentry * dentry the dentry object whose data is being accessed.

int * srcu_idx a pointer to some memory to store a SRCU index in.

Description

Up to a matching call to *debugfs_use_file_finish()*, any successive call into the file removing functions *debugfs_remove()* and *debugfs_remove_recursive()* will block. Since associated private file data may only get freed after a successful return of any of the removal functions, you may safely access it after a successful call to *debugfs_use_file_start()* without worrying about lifetime issues.

If -EIO is returned, the file has already been removed and thus, it is not safe to access any of its data. If, on the other hand, it is allowed to access the file data, zero is returned.

Regardless of the return code, any call to *debugfs_use_file_start()* must be followed by a matching call to *debugfs_use_file_finish()*.

void **debugfs_use_file_finish**(int *srcu_idx*)
mark the end of file data access

Parameters

int srcu_idx the SRCU index "created" by a former call to *debugfs_use_file_start()*.

Description

Allow any ongoing concurrent call into *debugfs_remove()* or *debugfs_remove_recursive()* blocked by a former call to *debugfs_use_file_start()* to proceed and return to its caller.

struct dentry * **debugfs_create_u8**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u8 * *value*)
create a debugfs file that is used to read and write an unsigned 8-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u8 * 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.

struct dentry * **debugfs_create_u16**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u16 * *value*)
create a debugfs file that is used to read and write an unsigned 16-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u16 * 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.

struct dentry * **debugfs_create_u32**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u32 * *value*)
create a debugfs file that is used to read and write an unsigned 32-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u32 * 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.

struct dentry * **debugfs_create_u64**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u64 * *value*)
create a debugfs file that is used to read and write an unsigned 64-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u64 * 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.

struct dentry * **debugfs_create_ulong**(const char * *name*, umode_t *mode*, struct dentry * *parent*, unsigned long * *value*)
create a debugfs file that is used to read and write an unsigned long value.

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

unsigned long * 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.

struct dentry * **debugfs_create_x8**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u8 * *value*)
create a debugfs file that is used to read and write an unsigned 8-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u8 * value a pointer to the variable that the file should read to and write from.

struct dentry * **debugfs_create_x16**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u16 * *value*)
create a debugfs file that is used to read and write an unsigned 16-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u16 * value a pointer to the variable that the file should read to and write from.

struct dentry * **debugfs_create_x32**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u32 * *value*)
create a debugfs file that is used to read and write an unsigned 32-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u32 * value a pointer to the variable that the file should read to and write from.

struct dentry * **debugfs_create_x64**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u64 * *value*)
create a debugfs file that is used to read and write an unsigned 64-bit value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

struct dentry * 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.

u64 * value a pointer to the variable that the file should read to and write from.

struct dentry * **debugfs_create_size_t**(const char * *name*, umode_t *mode*, struct dentry * *parent*, size_t * *value*)
create a debugfs file that is used to read and write an size_t value

Parameters

const char * name a pointer to a string containing the name of the file to create.

umode_t mode the permission that the file should have

`struct dentry * 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.

`size_t * value` a pointer to the variable that the file should read to and write from.

struct dentry * **`debugfs_create_atomic_t`**(const char * *name*, umode_t *mode*, struct dentry * *parent*, atomic_t * *value*)
create a debugfs file that is used to read and write an atomic_t value

Parameters

`const char * name` a pointer to a string containing the name of the file to create.

`umode_t mode` the permission that the file should have

`struct dentry * 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.

`atomic_t * value` a pointer to the variable that the file should read to and write from.

struct dentry * **`debugfs_create_bool`**(const char * *name*, umode_t *mode*, struct dentry * *parent*, bool * *value*)
create a debugfs file that is used to read and write a boolean value

Parameters

`const char * name` a pointer to a string containing the name of the file to create.

`umode_t mode` the permission that the file should have

`struct dentry * 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.

`bool * 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.

struct dentry * **`debugfs_create_blob`**(const char * *name*, umode_t *mode*, struct dentry * *parent*, struct debugfs_blob_wrapper * *blob*)
create a debugfs file that is used to read a binary blob

Parameters

`const char * name` a pointer to a string containing the name of the file to create.

`umode_t mode` the permission that the file should have

`struct dentry * 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.

`struct debugfs_blob_wrapper * 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.

struct dentry * **`debugfs_create_u32_array`**(const char * *name*, umode_t *mode*, struct dentry * *parent*, u32 * *array*, u32 *elements*)
    create a debugfs file that is used to read u32 array.

Parameters

`const char * name` a pointer to a string containing the name of the file to create.

`umode_t mode` the permission that the file should have.

`struct dentry * 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.

`u32 * array` u32 array that provides data.

`u32 elements` total number of elements in the array.

Description

This function creates a file in debugfs with the given name that exports **array** as data. If the **mode** variable is so set it can be read from. Writing is not supported. Seek within the file is also not supported. Once array is created its size can not be changed.

The function returns a pointer to dentry on success. If debugfs is not enabled in the kernel, the value `-ENODEV` will be returned.

void **`debugfs_print_regs32`**(struct seq_file * *s*, const struct debugfs_reg32 * *regs*, int *nregs*, void __iomem * *base*, char * *prefix*)
    use seq_print to describe a set of registers

Parameters

`struct seq_file * s` the seq_file structure being used to generate output

`const struct debugfs_reg32 * regs` an array if struct debugfs_reg32 structures

`int nregs` the length of the above array

`void __iomem * base` the base address to be used in reading the registers

`char * prefix` a string to be prefixed to every output line

Description

This function outputs a text block describing the current values of some 32-bit hardware registers. It is meant to be used within debugfs files based on seq_file that need to show registers, intermixed with other information. The prefix argument may be used to specify a leading string, because some peripherals have several blocks of identical registers, for example configuration of dma channels

struct dentry * **`debugfs_create_regset32`**(const char * *name*, umode_t *mode*, struct dentry * *parent*, struct debugfs_regset32 * *regset*)
    create a debugfs file that returns register values

Parameters

`const char * name` a pointer to a string containing the name of the file to create.

`umode_t mode` the permission that the file should have

`struct dentry * 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.

`struct debugfs_regset32 * regset` a pointer to a struct debugfs_regset32, which contains a pointer to an array of register definitions, the array size and the base address where the register bank is to be found.

Description

This function creates a file in debugfs with the given name that reports the names and values of a set of 32-bit registers. 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.

struct dentry * **`debugfs_create_devm_seqfile`**(struct device * *dev*, const char * *name*, struct dentry * *parent*, int (*read_fn) (struct seq_file **s*, void **data*)
create a debugfs file that is bound to device.

Parameters

`struct device * dev` device related to this debugfs file.

`const char * name` name of the debugfs file.

`struct dentry * 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.

`int (*)(struct seq_file *s,void *data) read_fn` function pointer called to print the seq_file content.

CHAPTER

SIX

THE LINUX JOURNALLING API

6.1 Overview

6.1.1 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 *jbd2_journal_init_inode()* call is for journals stored in filesystem inodes, or the *jbd2_journal_init_dev()* call can be used 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 *jbd2_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. The journalling layer expects the space for the journal was already allocated and initialized properly by the userspace tools. When loading the journal you must call *jbd2_journal_load()* to process journal contents. If the client file system detects the journal contents does not need to be processed (or even need not have valid contents), it may call *jbd2_journal_wipe()* to clear the journal contents before calling *jbd2_journal_load()*.

Note that jbd2_journal_wipe(..,0) calls *jbd2_journal_skip_recovery()* for you if it detects any outstanding transactions in the journal and similarly *jbd2_journal_load()* will call *jbd2_journal_recover()* if necessary. I would advise reading `ext4_load_journal()` in fs/ext4/super.c for examples on this stage.

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 *jbd2_journal_start()* which returns a transaction handle.

jbd2_journal_start() and its counterpart *jbd2_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 *jbd2_journal_stop()* the same number of times as *jbd2_journal_start()* before the transaction is completed (or more accurately leaves the update phase). Ext4/VFS makes use of this feature to simplify handling of inode dirtying, quota support, etc.

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 *jbd2_journal_get_create_access()* / *jbd2_journal_get_write_access()* / *jbd2_journal_get_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 *jbd2_journal_dirty_metadata()*. 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 *jbd2_journal_forget()* in much the same way as you might have used `bforget()` in the past.

A *jbd2_journal_flush()* may be called at any time to commit and checkpoint all your transactions.

Then at umount time , in your `put_super()` you can then call *jbd2_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 *jbd2_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 ext4) may be modified in a later syscall.

The second case to bear in mind is that *jbd2_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 *jbd2_journal_stop()*. So to avoid deadlocks you must treat *jbd2_journal_start()* / *jbd2_journal_stop()* as if they were semaphores and include them in your semaphore ordering rules to prevent deadlocks. Note that *jbd2_journal_extend()* has similar blocking behaviour to *jbd2_journal_start()* so you can deadlock here just as easily as on *jbd2_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 ext4_jbd.h to see the basis on which ext4 uses to make these decisions.

Another wriggle to watch out for is your on-disk block allocation strategy. Why? Because, if you do a delete, you need to ensure you haven't reused any of the freed blocks until the transaction freeing these blocks commits. If you reused these blocks and crash happens, there is no way to restore the contents of the reallocated blocks at the end of the last fully committed transaction. One simple way of doing this is to mark blocks as free in internal in-memory block allocation structures only after the transaction freeing them commits. Ext4 uses journal commit callback for this purpose.

With journal commit callbacks you can ask the journalling layer to call a callback function when the transaction is finally committed to disk, so that you can do some of your own management. You ask the journalling layer for calling the callback by simply setting `journal->j_commit_callback` function pointer and that function is called after each transaction commit. You can also use `transaction->t_private_list` for attaching entries to a transaction that need processing when the transaction commits.

JBD2 also provides a way to block all transaction updates via *jbd2_journal_lock_updates()* / *jbd2_journal_unlock_updates()*. Ext4 uses this when it wants a window with a clean and stable fs for a moment. E.g.

```
jbd2_journal_lock_updates() //stop new stuff happening..
jbd2_journal_flush()        // checkpoint everything.
..do stuff on stable fs
jbd2_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.

6.1.2 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.

6.2 Data Types

The journalling layer uses typedefs to 'hide' the concrete definitions of the structures used. As a client of the JBD2 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'.

6.2.1 Structures

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

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**
  The journal_t maintains all of the journaling state information for a single filesystem.

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 **jbd2_inode**

Definition

```
struct jbd2_inode {
};
```

Members

Description

  present in a transaction so that we can sync them during commit.

struct **jbd2_journal_handle**
  The handle_s type is the concrete type associated with handle_t.

Definition

```
struct jbd2_journal_handle {
  union {unnamed_union};
  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;
};
```

Members

{unnamed_union} anonymous

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

struct **journal_s**
The journal_s type is the concrete type associated with journal_t.

Definition

```
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;
  rwlock_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_done_commit;
  wait_queue_head_t j_wait_commit;
  wait_queue_head_t j_wait_updates;
  wait_queue_head_t j_wait_reserved;
  struct mutex j_checkpoint_mutex;
  unsigned long j_head;
  unsigned long j_tail;
  unsigned long j_free;
  unsigned long j_first;
  unsigned long j_last;
  struct block_device * j_dev;
  int j_blocksize;
  unsigned long long j_blk_offset;
  struct block_device * j_fs_dev;
  unsigned int j_maxlen;
  atomic_t j_reserved_credits;
  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;
  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 jbd2_revoke_table_s * j_revoke;
  struct jbd2_revoke_table_s * j_revoke_table;
  struct buffer_head ** j_wbuf;
  int j_wbufsize;
  pid_t j_last_sync_writer;
  spinlock_t j_history_lock;
  struct proc_dir_entry * j_proc_entry;
  struct transaction_stats_s j_stats;
  void * j_private;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
  struct lockdep_map j_trans_commit_map;
#endif
};
```

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_done_commit` Wait queue for waiting for commit to complete

`j_wait_commit` Wait queue to trigger commit

`j_wait_updates` Wait queue to wait for updates to complete

`j_wait_reserved` Wait queue to wait for reserved buffer credits to drop

`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_reserved_credits` Number of buffers reserved from the running transaction

`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` 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` alternate revoke tables for j_revoke

`j_wbuf` array of buffer_heads for jbd2_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_history_lock` Protect the transactions statistics history

`j_proc_entry` procfs entry for the jbd statistics directory

`j_stats` Overall statistics

`j_private` An opaque pointer to fs-private information.

`j_trans_commit_map` Lockdep entity to track transaction commit dependencies

6.3 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

6.3.1 Journal Level

int **`jbd2_journal_force_commit_nested`**(*journal_t * journal*)

Parameters

`journal_t * journal` journal to force Returns true if progress was made.

Description

transaction. This is used for forcing out undo-protected data which contains bitmaps, when the fs is running out of space.

int **`jbd2_journal_force_commit`**(*journal_t * journal*)
force any uncommitted transactions

Parameters

`journal_t * journal` journal to force

Description

Caller want unconditional commit. We can only force the running transaction if we don't have an active handle, otherwise, we will deadlock.

journal_t * **`jbd2_journal_init_dev`**(struct block_device * *bdev*, struct block_device * *fs_dev*, unsigned long long *start*, int *len*, int *blocksize*)
creates and initialises a journal structure

Parameters

`struct block_device * bdev` Block device on which to create the journal

`struct block_device * fs_dev` Device which hold journalled filesystem for this journal.

`unsigned long long start` Block nr Start of journal.

`int len` Length of the journal in blocks.

`int blocksize` blocksize of journalling device

Return

a newly created journal_t *

> jbd2_journal_init_dev creates a journal which maps a fixed contiguous range of blocks on an arbitrary block device.

journal_t * **`jbd2_journal_init_inode`**(struct inode * *inode*)
creates a journal which maps to a inode.

Parameters

`struct inode * inode` An inode to create the journal in

Description

jbd2_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.

void **`jbd2_journal_update_sb_errno`**(*journal_t* * *journal*)
Update error in the journal.

Parameters

`journal_t * journal` The journal to update.

Description

Update a journal's errno. Write updated superblock to disk waiting for IO to complete.

int **`jbd2_journal_load`**(*journal_t* * *journal*)
Read journal from disk.

Parameters

`journal_t * 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.

int **`jbd2_journal_destroy`**(*journal_t* * *journal*)
Release a journal_t structure.

Parameters

`journal_t * 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.

int **`jbd2_journal_check_used_features`**(*journal_t* * *journal*, unsigned long *compat*, unsigned long *ro*, unsigned long *incompat*)
Check if features specified are used.

Parameters

`journal_t * journal` Journal to check.

`unsigned long compat` bitmask of compatible features

`unsigned long ro` bitmask of features that force read-only mount

`unsigned long 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.

int **jbd2_journal_check_available_features**(*journal_t* * *journal*, unsigned long *compat*, unsigned long *ro*, unsigned long *incompat*)
Check feature set in journalling layer

Parameters

journal_t * journal Journal to check.

unsigned long compat bitmask of compatible features

unsigned long ro bitmask of features that force read-only mount

unsigned long 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

int **jbd2_journal_set_features**(*journal_t* * *journal*, unsigned long *compat*, unsigned long *ro*, unsigned long *incompat*)
Mark a given journal feature in the superblock

Parameters

journal_t * journal Journal to act on.

unsigned long compat bitmask of compatible features

unsigned long ro bitmask of features that force read-only mount

unsigned long 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.

int **jbd2_journal_flush**(*journal_t* * *journal*)
Flush journal

Parameters

journal_t * 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.

int **jbd2_journal_wipe**(*journal_t* * *journal*, int *write*)
Wipe journal contents

Parameters

journal_t * journal Journal to act on.

int 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 `jbd2_journal_load()`.

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

void **jbd2_journal_abort**(*journal_t* * *journal*, int *errno*)
Shutdown the journal immediately.

Parameters

`journal_t * journal` the journal to shutdown.

`int 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 jbd2_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 jbd2_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 jbd2_journal_stop, which will receive the -EIO error.

Finally, the jbd2_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).

int **`jbd2_journal_errno`**(*journal_t* * *journal*)
returns the journal's error state.

Parameters

`journal_t * journal` journal to examine.

Description

This is the errno number set with *`jbd2_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.

int **`jbd2_journal_clear_err`**(*journal_t* * *journal*)
clears the journal's error state

Parameters

`journal_t * journal` journal to act on.

Description

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

void **`jbd2_journal_ack_err`**(*journal_t* * *journal*)
Ack journal err.

Parameters

`journal_t * journal` journal to act on.

Description

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

int **jbd2_journal_recover**(*journal_t* * *journal*)
recovers a on-disk journal

Parameters

journal_t * 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.

int **jbd2_journal_skip_recovery**(*journal_t* * *journal*)
Start journal and wipe exiting records

Parameters

journal_t * 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 doesn't appear to be exported..

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.

6.3.2 Transasction Level

handle_t * **jbd2_journal_start**(*journal_t* * *journal*, int *nblocks*)
Obtain a new handle.

Parameters

journal_t * journal Journal to start transaction on.

int 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. Additionally, if rsv_blocks > 0, we also create another handle with rsv_blocks reserved blocks in the journal. This handle is is stored in h_rsv_handle. It is not attached to any particular transaction and thus doesn't block transaction commit. If the caller uses this reserved handle, it has to set h_rsv_handle to NULL as otherwise *jbd2_journal_stop()* on the parent handle will dispose the reserved one. Reserved handle has to be converted to a normal handle using *jbd2_journal_start_reserved()* before it can be used.

Return a pointer to a newly allocated handle, or an ERR_PTR() value on failure.

int **jbd2_journal_start_reserved**(*handle_t* * *handle*, unsigned int *type*, unsigned int *line_no*)
start reserved handle

Parameters

handle_t * handle handle to start

unsigned int type *undescribed*

unsigned int line_no *undescribed*

Description

Start handle that has been previously reserved with jbd2_journal_reserve(). This attaches **handle** to the running transaction (or creates one if there's not transaction running). Unlike *jbd2_journal_start()*

this function cannot block on journal commit, checkpointing, or similar stuff. It can block on memory allocation or frozen journal though.

Return 0 on success, non-zero on error - handle is freed in that case.

int **jbd2_journal_extend**(*handle_t* * *handle*, int *nblocks*)
    extend buffer credits.

Parameters

handle_t * handle handle to 'extend'

int 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 modifications in advance, but can extend its credit if it needs more.

jbd2_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.

int **jbd2__journal_restart**(*handle_t* * *handle*, int *nblocks*, gfp_t *gfp_mask*)
    restart a handle .

Parameters

handle_t * handle handle to restart

int nblocks nr credits requested

gfp_t gfp_mask *undescribed*

Description

Restart a handle for a multi-transaction filesystem operation.

If the *jbd2_journal_extend()* call above fails to grant new buffer credits to a running handle, a call to jbd2_journal_restart will commit the handle's transaction so far and reattach the handle to a new transaction capable of guaranteeing the requested number of credits. We preserve reserved handle if there's any attached to the passed in handle.

void **jbd2_journal_lock_updates**(*journal_t* * *journal*)
    establish a transaction barrier.

Parameters

journal_t * 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.

void **jbd2_journal_unlock_updates**(*journal_t* * *journal*)
    release barrier

Parameters

journal_t * journal Journal to release the barrier on.

Description

Release a transaction barrier obtained with *jbd2_journal_lock_updates()*.

Should be called without the journal lock held.

int **`jbd2_journal_get_write_access`**(*handle_t * handle*, struct buffer_head * *bh*)
    notify intent to modify a buffer for metadata (not data) update.

Parameters

`handle_t * handle` transaction to add buffer modifications to

`struct buffer_head * bh` bh to be used for metadata writes

Return

error code or 0 on success.

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

int **`jbd2_journal_get_create_access`**(*handle_t * handle*, struct buffer_head * *bh*)
    notify intent to use newly created bh

Parameters

`handle_t * handle` transaction to new buffer to

`struct buffer_head * bh` new buffer.

Description

Call this if you create a new bh.

int **`jbd2_journal_get_undo_access`**(*handle_t * handle*, struct buffer_head * *bh*)
    Notify intent to modify metadata with non-rewindable consequences

Parameters

`handle_t * handle` transaction

`struct buffer_head * 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, jbd2_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.

void **`jbd2_journal_set_triggers`**(struct buffer_head * *bh*, struct jbd2_buffer_trigger_type * *type*)
    Add triggers for commit writeout

Parameters

`struct buffer_head * bh` buffer to trigger on

`struct jbd2_buffer_trigger_type * type` struct jbd2_buffer_trigger_type containing the trigger(s).

Description

Set any triggers on this journal_head. This is always safe, because triggers for a committing buffer will be saved off, and triggers for a running transaction will match the buffer in that transaction.

Call with NULL to clear the triggers.

int **jbd2_journal_dirty_metadata**(*handle_t * handle*, struct buffer_head * *bh*)
mark a buffer as containing dirty metadata

Parameters

handle_t * handle transaction to add buffer to.

struct buffer_head * bh buffer to mark

Description

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

The buffer must have previously had *jbd2_journal_get_write_access()* called so that it has a valid journal_head attached to the buffer head.

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.

int **jbd2_journal_forget**(*handle_t * handle*, struct buffer_head * *bh*)
`bforget()` for potentially-journaled buffers.

Parameters

handle_t * handle transaction handle

struct buffer_head * 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.

int **jbd2_journal_stop**(*handle_t * handle*)
complete a transaction

Parameters

handle_t * 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.

jbd2_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 jbd2_journal_abort has been executed since the transaction began.

int **jbd2_journal_try_to_free_buffers**(*journal_t * journal*, struct page * *page*, gfp_t *gfp_mask*)
try to free page buffers.

Parameters

journal_t * journal journal for operation

struct page * page to try and free

gfp_t gfp_mask we use the mask to detect how hard should we try to release buffers. If __GFP_DIRECT_RECLAIM 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 __jbd2_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 jbd2_journal_dirty_data on this buffer. So we need to lock against that. `jbd2_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

int **jbd2_journal_invalidatepage**(*journal_t* * *journal*, struct page * *page*, unsigned int *offset*, unsigned int *length*)

Parameters

journal_t * journal journal to use for flush...

struct page * page page to flush

unsigned int offset start of the range to invalidate

unsigned int length length of the range to invalidate

Description

Reap page buffers containing data after in the specified range in page. Can return -EBUSY if buffers are part of the committing transaction and the page is straddling i_size. Caller then has to wait for current commit and try again.

6.4 See also

Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen Tweedie

Ext3 Journalling FileSystem, OLS 2000, Dr. Stephen Tweedie

CHAPTER

SEVEN

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.

ssize_t **splice_to_pipe**(struct *pipe_inode_info* * *pipe*, struct splice_pipe_desc * *spd*)
fill passed data into a pipe

Parameters

struct pipe_inode_info * pipe pipe to fill

struct splice_pipe_desc * 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.

ssize_t **generic_file_splice_read**(struct file * *in*, loff_t * *ppos*, struct *pipe_inode_info* * *pipe*, size_t *len*, unsigned int *flags*)
splice data from file to a pipe

Parameters

struct file * in file to splice from

loff_t * ppos position in **in**

struct pipe_inode_info * pipe pipe to splice to

size_t len number of bytes to splice

unsigned int flags splice modifier flags

Description

Will read pages from given file and fill them into a pipe. Can be used as long as it has more or less sane ->:c:func:*read_iter()*.

int **splice_from_pipe_feed**(struct *pipe_inode_info* * *pipe*, struct splice_desc * *sd*, splice_actor * *actor*)
feed available data from a pipe to a file

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct splice_desc * sd information to **actor**

splice_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.

int **`splice_from_pipe_next`**(struct *pipe_inode_info* * *pipe*, struct splice_desc * *sd*)
wait for some data to splice from

Parameters

`struct pipe_inode_info * pipe` pipe to splice from

`struct splice_desc * 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.

void **`splice_from_pipe_begin`**(struct splice_desc * *sd*)
start splicing from pipe

Parameters

`struct splice_desc * 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**.

void **`splice_from_pipe_end`**(struct *pipe_inode_info* * *pipe*, struct splice_desc * *sd*)
finish splicing from pipe

Parameters

`struct pipe_inode_info * pipe` pipe to splice from

`struct splice_desc * 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()`*.

ssize_t **`__splice_from_pipe`**(struct *pipe_inode_info* * *pipe*, struct splice_desc * *sd*, splice_actor * *actor*)
splice data from a pipe to given actor

Parameters

`struct pipe_inode_info * pipe` pipe to splice from

`struct splice_desc * sd` information to **actor**

`splice_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.

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*)
splice data from a pipe to a file

Parameters

`struct pipe_inode_info * pipe` pipe to splice from

struct file * out file to splice to

loff_t * ppos position in **out**

size_t len how many bytes to splice

unsigned int flags splice modifier flags

splice_actor * 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()*.

ssize_t **iter_file_splice_write**(struct *pipe_inode_info* * *pipe*, struct file * *out*, loff_t * *ppos*, size_t *len*, unsigned int *flags*)
splice data from a pipe to a file

Parameters

struct pipe_inode_info * pipe pipe info

struct file * out file to write to

loff_t * ppos position in **out**

size_t len number of bytes to splice

unsigned int flags splice modifier flags

Description

Will either move or copy pages (determined by **flags** options) from the given pipe inode to the given file. This one is ->write_iter-based.

ssize_t **generic_splice_sendpage**(struct *pipe_inode_info* * *pipe*, struct file * *out*, loff_t * *ppos*, size_t *len*, unsigned int *flags*)
splice data from a pipe to a socket

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct file * out socket to write to

loff_t * ppos position in **out**

size_t len number of bytes to splice

unsigned int flags splice modifier flags

Description

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

ssize_t **splice_direct_to_actor**(struct file * *in*, struct splice_desc * *sd*, splice_direct_actor * *actor*)
splices data directly between two non-pipes

Parameters

struct file * in file to splice from

struct splice_desc * sd actor information on where to splice to

splice_direct_actor * 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.

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

splices data directly between two files

Parameters

`struct file * in` file to splice from

`loff_t * ppos` input file offset

`struct file * out` file to splice to

`loff_t * opos` output file offset

`size_t len` number of bytes to splice

`unsigned int 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

EIGHT

PIPES API

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

struct **pipe_buffer**
    a linux kernel pipe buffer

Definition

```
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**
    a linux kernel pipe

Definition

```
struct pipe_inode_info {
  struct mutex mutex;
  wait_queue_head_t wait;
  unsigned int nrbufs;
  unsigned int curbuf;
  unsigned int buffers;
  unsigned int readers;
  unsigned int writers;
  unsigned int files;
  unsigned int waiting_writers;
  unsigned int r_counter;
  unsigned int w_counter;
  struct page * tmp_page;
  struct fasync_struct * fasync_readers;
  struct fasync_struct * fasync_writers;
  struct pipe_buffer * bufs;
  struct user_struct * user;
};
```

Members

mutex mutex protecting the whole thing

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

buffers total number of buffers (should be a power of 2)

readers number of current readers of this pipe

writers number of current writers of this pipe

files number of struct file referring this pipe (protected by ->i_lock)

waiting_writers number of writers blocked waiting for room

r_counter reader counter

w_counter writer counter

tmp_page cached released page

fasync_readers reader side fasync

fasync_writers writer side fasync

bufs the circular array of pipe buffers

user the user who created this pipe

void **pipe_buf_get**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
get a reference to a pipe_buffer

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to get a reference to

void **pipe_buf_release**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
put a reference to a pipe_buffer

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to put a reference to

int **pipe_buf_confirm**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
verify contents of the pipe buffer

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to confirm

int **pipe_buf_steal**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
attempt to take ownership of a pipe_buffer

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to attempt to steal

int **generic_pipe_buf_steal**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
attempt to take ownership of a *pipe_buffer*

Parameters

`struct pipe_inode_info * pipe` the pipe that the buffer belongs to

`struct pipe_buffer * 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.

void **`generic_pipe_buf_get`**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
get a reference to a *`struct pipe_buffer`*

Parameters

`struct pipe_inode_info * pipe` the pipe that the buffer belongs to

`struct pipe_buffer * 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.

int **`generic_pipe_buf_confirm`**(struct *pipe_inode_info* * *info*, struct *pipe_buffer* * *buf*)
verify contents of the pipe buffer

Parameters

`struct pipe_inode_info * info` the pipe that the buffer belongs to

`struct pipe_buffer * 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.

void **`generic_pipe_buf_release`**(struct *pipe_inode_info* * *pipe*, struct *pipe_buffer* * *buf*)
put a reference to a *`struct pipe_buffer`*

Parameters

`struct pipe_inode_info * pipe` the pipe that the buffer belongs to

`struct pipe_buffer * buf` the buffer to put a reference to

Description

This function releases a reference to **buf**.