

Linux Filesystems API

Linux Filesystems API

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

Table of Contents

1. The Linux VFS	1
1.1. The Filesystem types.....	1
enum positive_aop_returns	1
sb_end_write	2
sb_end_pagefault	2
sb_end_intwrite.....	3
sb_start_write.....	4
sb_start_pagefault	5
inode_inc_iversion	6
1.2. The Directory Cache	7
__d_drop	7
d_invalidate	8
shrink_dcache_sb	9
have_submounts	10
shrink_dcache_parent	11
check_submounts_and_drop.....	12
d_alloc.....	12
d_alloc_pseudo	13
d_instantiate	14
d_instantiate_no_diralias	15
d_find_any_alias	16
d_obtain_alias	17
d_splice_alias	18
d_add_ci.....	19
d_lookup	20
d_hash_and_lookup	21
d_validate.....	22
d_delete.....	23
d_rehash	24
dentry_update_name_case	25
d_materialise_unique	26
d_path.....	27
d_add.....	28
d_add_unique.....	29
dget_dlock.....	30
d_unhashed	31
1.3. Inode Handling.....	31
inode_init_always	32
drop_nlink.....	32
clear_nlink.....	33
set_nlink.....	34

inc_nlink	35
inode_sb_list_add	36
__insert_inode_hash	36
__remove_inode_hash	37
new_inode	38
unlock_new_inode	39
lock_two_nondirectories	40
unlock_two_nondirectories	41
iget5_locked	41
iget_locked	43
iunique	44
ilookup5_nowait	45
ilookup5	46
ilookup	47
iput	48
bmap	49
touch_atime	50
file_update_time	51
inode_init_owner	52
inode_owner_or_capable	53
inode_dio_wait	53
make_bad_inode	54
is_bad_inode	55
iget_failed	56
1.4. Registration and Superblocks	57
deactivate_locked_super	57
deactivate_super	58
generic_shutdown_super	59
sget	59
iterate_supers_type	60
get_super	61
get_super_thawed	62
freeze_super	63
thaw_super	65
1.5. File Locks	65
posix_lock_file	66
posix_lock_file_wait	67
locks_mandatory_area	67
__break_lease	69
lease_get_mtime	70
generic_setlease	71
vfs_setlease	72
flock_lock_file_wait	73

vfs_test_lock	74
vfs_lock_file.....	75
posix_unblock_lock	76
vfs_cancel_lock.....	77
lock_may_read	78
lock_may_write.....	79
locks_unlink_lock	80
locks_mandatory_locked	81
fcntl_getlease	82
check_conflicting_open	83
fcntl_setlease.....	84
sys_flock	85
1.6. Other Functions.....	86
mpage_readpages	86
mpage_writepages.....	88
generic_permission	89
__inode_permission	90
inode_permission	91
path_get.....	92
path_put.....	92
vfs_path_lookup.....	93
lookup_one_len.....	94
vfs_unlink	95
vfs_link	96
vfs_rename	98
sync_mapping_buffers	99
mark_buffer_dirty	100
__bread.....	101
block_invalidatepage.....	102
ll_rw_block	103
bh_uptodate_or_lock.....	105
bh_submit_read.....	105
bio_reset.....	106
bio_chain.....	107
bio_alloc_bioset	108
bio_put	110
__bio_clone_fast	110
bio_clone_fast	111
bio_clone_bioset	112
bio_get_nr_vecs	113
bio_add_pc_page	114
bio_add_page	115
submit_bio_wait.....	117

bio_advance	117
bio_alloc_pages.....	118
bio_copy_data.....	119
bio_uncopy_user.....	120
bio_copy_user.....	121
bio_map_user.....	122
bio_unmap_user.....	124
bio_map_kern	124
bio_copy_kern.....	125
bio_endio.....	127
bio_endio_nodec	128
bio_split	128
bio_trim.....	130
bioset_create	130
seq_open	131
seq_read	132
seq_lseek.....	133
seq_release	134
seq_escape.....	135
mangle_path.....	136
seq_path	137
seq_write.....	138
seq_pad	139
seq_hlist_start	140
seq_hlist_start_head.....	141
seq_hlist_next	142
seq_hlist_start_rcu	143
seq_hlist_start_head_rcu.....	144
seq_hlist_next_rcu	145
seq_hlist_start_percpu.....	146
seq_hlist_next_percpu.....	147
register_filesystem	148
unregister_filesystem	149
writeback_in_progress	150
__mark_inode_dirty.....	150
writeback_inodes_sb_nr	151
writeback_inodes_sb.....	152
try_to_writeback_inodes_sb_nr.....	153
try_to_writeback_inodes_sb	154
sync_inodes_sb	155
write_inode_now.....	156
sync_inode	157
sync_inode_metadata.....	158

freeze_bdev	159
thaw_bdev	160
bdev_read_page	161
bdev_write_page	162
bdgrab	163
bd_link_disk_holder	164
bd_unlink_disk_holder	166
check_disk_size_change	167
revalidate_disk	167
blkdev_get.....	168
blkdev_get_by_path.....	170
blkdev_get_by_dev	171
lookup_bdev	172
2. The proc filesystem	175
2.1. sysctl interface	175
proc_dostring	175
proc_dointvec.....	176
proc_dointvec_minmax.....	177
proc_doulongvec_minmax.....	178
proc_doulongvec_ms_jiffies_minmax	179
proc_dointvec_jiffies.....	181
proc_dointvec_userhz_jiffies	182
proc_dointvec_ms_jiffies	183
2.2. proc filesystem interface	184
proc_flush_task	185
3. Events based on file descriptors.....	187
eventfd_signal	187
eventfd_ctx_get.....	188
eventfd_ctx_put.....	188
eventfd_ctx_remove_wait_queue.....	189
eventfd_ctx_read.....	190
eventfd_fget	192
eventfd_ctx_fdget.....	192
eventfd_ctx_fileget.....	193
4. The Filesystem for Exporting Kernel Objects.....	195
sysfs_create_file_ns	195
sysfs_add_file_to_group	195
sysfs_chmod_file.....	196
sysfs_remove_file_ns.....	197
sysfs_remove_file_from_group	198
sysfs_create_bin_file.....	199
sysfs_remove_bin_file	200

sysfs_create_link.....	200
sysfs_remove_link	201
sysfs_rename_link_ns	202
5. The debugfs filesystem.....	205
5.1. debugfs interface	205
debugfs_create_file	205
debugfs_create_dir	206
debugfs_create_symlink	207
debugfs_remove	208
debugfs_remove_recursive.....	209
debugfs_rename	210
debugfs_initialized	211
debugfs_create_u8	212
debugfs_create_u16	213
debugfs_create_u32	214
debugfs_create_u64	216
debugfs_create_x8	217
debugfs_create_x16	218
debugfs_create_x32	219
debugfs_create_x64	220
debugfs_create_size_t.....	221
debugfs_create_atomic_t	222
debugfs_create_bool	223
debugfs_create_blob	224
debugfs_create_u32_array	226
debugfs_print_regs32.....	227
debugfs_create_regset32.....	228
6. The Linux Journalling API.....	231
6.1. Overview	231
6.1.1. Details	231
6.1.2. Summary	233
6.2. Data Types	234
6.2.1. Structures	234
typedef handle_t.....	234
typedef journal_t.....	235
struct handle_s	235
struct journal_s.....	237
6.3. Functions.....	242
6.3.1. Journal Level.....	242
journal_init_dev	242
journal_init_inode	244
journal_create.....	244

journal_load	245
journal_destroy	246
journal_check_used_features	247
journal_check_available_features	248
journal_set_features	249
journal_update_format	250
journal_flush	251
journal_wipe	252
journal_abort	253
journal_errno	254
journal_clear_err	255
journal_ack_err	256
journal_recover	256
journal_skip_recovery	257
6.3.2. Transaction Level	258
journal_start	258
journal_extend	259
journal_restart	260
journal_lock_updates	261
journal_unlock_updates	262
journal_get_write_access	263
journal_get_create_access	264
journal_get_undo_access	265
journal_dirty_data	266
journal_dirty_metadata	267
journal_forget	269
journal_stop	270
journal_force_commit	271
journal_try_to_free_buffers	271
journal_invalidatepage	273
6.4. See also	274
7. splice API	275
splice_to_pipe	275
generic_file_splice_read	276
splice_from_pipe_feed	277
splice_from_pipe_next	278
splice_from_pipe_begin	279
splice_from_pipe_end	279
__splice_from_pipe	280
splice_from_pipe	281
iter_file_splice_write	283
generic_splice_sendpage	284
splice_direct_to_actor	285

do_splice_direct	286
8. pipes API.....	289
struct pipe_buffer	289
struct pipe_inode_info	290
generic_pipe_buf_steal	292
generic_pipe_buf_get.....	293
generic_pipe_buf_confirm	294
generic_pipe_buf_release	295

Chapter 1. The Linux VFS

1.1. The Filesystem types

enum positive_aop_returns

LINUX

Kernel Hackers ManualSeptember 2014

Name

enum positive_aop_returns — aop return codes with specific semantics

Synopsis

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

Constants

AOP_WRITEPAGE_ACTIVATE

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

AOP_TRUNCATED_PAGE

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

Description

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

sb_end_write

LINUX

Kernel Hackers ManualSeptember 2014

Name

`sb_end_write` — drop write access to a superblock

Synopsis

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

Arguments

sb

the super we wrote to

Description

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

sb_end_pagefault

LINUX

Kernel Hackers Manual September 2014

Name

`sb_end_pagefault` — drop write access to a superblock from a page fault

Synopsis

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

Arguments

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.

sb_end_intwrite

LINUX

Name

`sb_end_intwrite` — drop write access to a superblock for internal fs purposes

Synopsis

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

Arguments

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.

sb_start_write

LINUX

Name

`sb_start_write` — get write access to a superblock

Synopsis

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

Arguments

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`)

sb_start_pagefault

LINUX

Kernel Hackers Manual September 2014

Name

`sb_start_pagefault` — get write access to a superblock from a page fault

Synopsis

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

Arguments

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`

inode_inc_iversion

LINUX

Name

`inode_inc_iversion` — increments `i_version`

Synopsis

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

Arguments

inode

inode that need to be updated

Description

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

1.2. The Directory Cache

`__d_drop`

LINUX

Name

`__d_drop` — drop a dentry

Synopsis

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

Arguments

dentry

dentry to drop

Description

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

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

`__d_drop` requires `dentry->d_lock`.

d_invalidate

LINUX

Name

`d_invalidate` — invalidate a dentry

Synopsis

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

Arguments

dentry

dentry to invalidate

Description

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

no dcache lock.

shrink_dcache_sb

LINUX

Name

`shrink_dcache_sb` — shrink dcache for a superblock

Synopsis

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

Arguments

sb

superblock

Description

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

have_submounts

LINUX

Kernel Hackers ManualSeptember 2014

Name

`have_submounts` — check for mounts over a dentry

Synopsis

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

Arguments

parent

dentry to check.

Description

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

shrink_dcache_parent

LINUX

Kernel Hackers ManualSeptember 2014

Name

`shrink_dcache_parent` — prune dcache

Synopsis

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

Arguments

parent

parent of entries to prune

Description

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

check_submounts_and_drop

LINUX

Kernel Hackers ManualSeptember 2014

Name

`check_submounts_and_drop` — prune dcache, check for submounts and drop

Synopsis

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

Arguments

dentry

dentry to prune and drop

Description

All done as a single atomic operation relative to `has_unlinked_ancestor`. Returns 0 if successfully unhashed *parent*. If there were submounts then return -EBUSY.

d_alloc

LINUX

Kernel Hackers ManualSeptember 2014

Name

`d_alloc` — allocate a dcache entry

Synopsis

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

Arguments

parent

parent of entry to allocate

name

qstr of the name

Description

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

d_alloc_pseudo

LINUX

Name

`d_alloc_pseudo` — allocate a dentry (for lookup-less filesystems)

Synopsis

```
struct dentry * d_alloc_pseudo (struct super_block * sb, const
struct qstr * name);
```

Arguments

sb

the superblock

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.

`d_instantiate`

LINUX

Name

`d_instantiate` — fill in inode information for a dentry

Synopsis

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

Arguments

entry

dentry to complete

inode

inode to attach to this dentry

Description

Fill in inode information in the entry.

This turns negative dentries into productive full members of society.

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

`d_instantiate_no_diralias`

LINUX

Name

`d_instantiate_no_diralias` — instantiate a non-aliased dentry

Synopsis

```
int d_instantiate_no_diralias (struct dentry * entry, struct
inode * inode);
```

Arguments

entry

dentry to complete

inode

inode to attach to this dentry

Description

Fill in inode information in the entry. 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.

`d_find_any_alias`

LINUX

Name

`d_find_any_alias` — find any alias for a given inode

Synopsis

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

Arguments

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`.

d_obtain_alias

LINUX

Name

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

Synopsis

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

Arguments

inode

inode to allocate the dentry for

Description

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

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

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

d_splice_alias

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

inode

the inode which may have a disconnected dentry

dentry

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

Description

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

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

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

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.

d_add_ci

LINUX

Name

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

Synopsis

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

Arguments

dentry

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

inode

the inode case-insensitive lookup has found

name

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

Description

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

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

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

d_lookup

LINUX

Kernel Hackers ManualSeptember 2014

Name

d_lookup — search for a dentry

Synopsis

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

Arguments

parent

parent dentry

name

qstr of name we wish to find

Returns

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.

d_hash_and_lookup

LINUX

Kernel Hackers Manual September 2014

Name

d_hash_and_lookup — hash the qstr then search for a dentry

Synopsis

```
struct dentry * d_hash_and_lookup (struct dentry * dir, struct  
qstr * name);
```

Arguments

dir

Directory to search in

name

qstr of name we wish to find

Description

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

d_validate

LINUX

Name

`d_validate` — verify dentry provided from insecure source (deprecated)

Synopsis

```
int d_validate (struct dentry * dentry, struct dentry *  
dparent);
```

Arguments

dentry

The dentry alleged to be valid child of *dparent*

dparent

The parent dentry (known to be valid)

Description

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

This function is slow for big directories, and deprecated, do not use it.

d_delete

LINUX

Name

`d_delete` — delete a dentry

Synopsis

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

Arguments

dentry

The dentry to delete

Description

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

d_rehash

LINUX

Name

`d_rehash` — add an entry back to the hash

Synopsis

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

Arguments

entry

dentry to add to the hash

Description

Adds a dentry to the hash according to its name.

dentry_update_name_case

LINUX

Kernel Hackers ManualSeptember 2014

Name

`dentry_update_name_case` — update case insensitive dentry with a new name

Synopsis

```
void dentry_update_name_case (struct dentry * dentry, struct  
qstr * name);
```

Arguments

dentry

dentry to be updated

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).

d_materialise_unique

LINUX

Kernel Hackers ManualSeptember 2014

Name

d_materialise_unique — introduce an inode into the tree

Synopsis

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

Arguments

dentry

candidate dentry

inode

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

Description

Introduces an dentry into the tree, substituting an extant disconnected root directory alias in its place if there is one. Caller must hold the `i_mutex` of the parent directory.

d_path

LINUX

Kernel Hackers Manual September 2014

Name

`d_path` — return the path of a dentry

Synopsis

```
char * d_path (const struct path * path, char * buf, int
buflen);
```

Arguments

path

path to report

buf

buffer to return value in

buflen

buffer length

Description

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

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

“buflen” should be positive.

d_add

LINUX

Kernel Hackers Manual September 2014

Name

d_add — add dentry to hash queues

Synopsis

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

Arguments

entry

dentry to add

inode

The inode to attach to this dentry

Description

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

d_add_unique

LINUX

Kernel Hackers ManualSeptember 2014

Name

`d_add_unique` — add dentry to hash queues without aliasing

Synopsis

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

Arguments

entry

dentry to add

inode

The inode to attach to this dentry

Description

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

dget_dlock

LINUX

Kernel Hackers ManualSeptember 2014

Name

`dget_dlock` — get a reference to a dentry

Synopsis

```
struct dentry * dget_dlock (struct dentry * dentry);
```

Arguments

dentry

dentry to get a reference to

Description

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

d_unhashed

LINUX

Kernel Hackers ManualSeptember 2014

Name

`d_unhashed` — is dentry hashed

Synopsis

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

Arguments

dentry

entry to check

Description

Returns true if the dentry passed is not currently hashed.

1.3. Inode Handling

inode_init_always

LINUX

Kernel Hackers ManualSeptember 2014

Name

`inode_init_always` — perform inode structure initialisation

Synopsis

```
int inode_init_always (struct super_block * sb, struct inode *  
inode);
```

Arguments

sb

superblock inode belongs to

inode

inode to initialise

Description

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

drop_nlink

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

inode

inode

Description

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

clear_nlink

LINUX

Name

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

Synopsis

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

Arguments

inode

inode

Description

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

set_nlink

LINUX

Name

`set_nlink` — directly set an inode's link count

Synopsis

```
void set_nlink (struct inode * inode, unsigned int nlink);
```

Arguments

inode

inode

nlink

new nlink (should be non-zero)

Description

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

inc_nlink

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

inode

inode

Description

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

inode_sb_list_add

LINUX

Kernel Hackers ManualSeptember 2014

Name

`inode_sb_list_add` — add inode to the superblock list of inodes

Synopsis

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

Arguments

inode

inode to add

__insert_inode_hash

LINUX

Kernel Hackers ManualSeptember 2014

Name

`__insert_inode_hash` — hash an inode

Synopsis

```
void __insert_inode_hash (struct inode * inode, unsigned long  
hashval);
```

Arguments

inode

unhashed inode

hashval

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

Description

Add an inode to the inode hash for this superblock.

__remove_inode_hash

LINUX

Name

`__remove_inode_hash` — remove an inode from the hash

Synopsis

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

Arguments

inode

inode to unhash

Description

Remove an inode from the superblock.

new_inode

LINUX

Name

`new_inode` — obtain an inode

Synopsis

```
struct inode * new_inode (struct super_block * sb);
```

Arguments

sb

superblock

Description

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

unlock_new_inode

LINUX

Kernel Hackers ManualSeptember 2014

Name

unlock_new_inode — clear the I_NEW state and wake up any waiters

Synopsis

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

Arguments

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.

lock_two_nondirectories

LINUX

Kernel Hackers ManualSeptember 2014

Name

`lock_two_nondirectories` — take two `i_mutexes` on non-directory objects

Synopsis

```
void lock_two_nondirectories (struct inode * inode1, struct  
inode * inode2);
```

Arguments

inode1

first inode to lock

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.

unlock_two_nondirectories

LINUX

Kernel Hackers ManualSeptember 2014

Name

`unlock_two_nondirectories` — release locks from
`lock_two_nondirectories`

Synopsis

```
void unlock_two_nondirectories (struct inode * inode1, struct  
inode * inode2);
```

Arguments

inode1

first inode to unlock

inode2

second inode to unlock

iget5_locked

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
struct inode * iget5_locked (struct super_block * sb, unsigned
long hashval, int (*test) (struct inode *, void *), int (*set)
(struct inode *, void *), void * data);
```

Arguments

sb

super block of file system

hashval

hash value (usually inode number) to get

test

callback used for comparisons between inodes

set

callback used to initialize a new struct inode

data

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

Description

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.

iget_locked

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
struct inode * iget_locked (struct super_block * sb, unsigned
long ino);
```

Arguments

sb

super block of file system

ino

inode number to get

Description

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`.

iunique

LINUX

Kernel Hackers Manual September 2014

Name

`iunique` — get a unique inode number

Synopsis

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

Arguments

sb

superblock

max_reserved

highest reserved inode number

Description

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

BUGS

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

ilookup5_nowait

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
struct inode * ilookup5_nowait (struct super_block * sb,  
unsigned long hashval, int (*test) (struct inode *, void *),  
void * data);
```

Arguments

sb

super block of file system to search

hashval

hash value (usually inode number) to search for

test

callback used for comparisons between inodes

data

opaque data pointer to pass to *test*

Description

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.

ilookup5

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
struct inode * ilookup5 (struct super_block * sb, unsigned  
long hashval, int (*test) (struct inode *, void *), void *  
data);
```

Arguments

sb

super block of file system to search

hashval

hash value (usually inode number) to search for

test

callback used for comparisons between inodes

data

opaque data pointer to pass to *test*

Description

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.

ilookup

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
struct inode * ilookup (struct super_block * sb, unsigned long  
ino);
```

Arguments

sb

super block of file system to search

ino

inode number to search for

Description

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.

iput

LINUX

Name

`iput` — put an inode

Synopsis

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

Arguments

inode

inode to put

Description

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

Consequently, `iput` can sleep.

bmap

LINUX

Name

`bmap` — find a block number in a file

Synopsis

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

Arguments

inode

inode of file

block

block to find

Description

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

touch_atime

LINUX

Kernel Hackers ManualSeptember 2014

Name

touch_atime — update the access time

Synopsis

```
void touch_atime (const struct path * path);
```

Arguments

path

the struct path to update

Description

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

file_update_time

LINUX

Kernel Hackers ManualSeptember 2014

Name

`file_update_time` — update mtime and ctime time

Synopsis

```
int file_update_time (struct file * file);
```

Arguments

file

file accessed

Description

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

inode_init_owner

LINUX

Kernel Hackers Manual September 2014

Name

`inode_init_owner` — Init uid,gid,mode for new inode according to posix standards

Synopsis

```
void inode_init_owner (struct inode * inode, const struct  
inode * dir, umode_t mode);
```

Arguments

inode

New inode

dir

Directory inode

mode

mode of the new inode

inode_owner_or_capable

LINUX

Kernel Hackers ManualSeptember 2014

Name

`inode_owner_or_capable` — check current task permissions to inode

Synopsis

```
bool inode_owner_or_capable (const struct inode * inode);
```

Arguments

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.

inode_dio_wait

LINUX

Kernel Hackers Manual September 2014

Name

`inode_dio_wait` — wait for outstanding DIO requests to finish

Synopsis

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

Arguments

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`.

make_bad_inode

LINUX

Name

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

Synopsis

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

Arguments

inode

Inode to mark bad

Description

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

is_bad_inode

LINUX

Name

`is_bad_inode` — is an inode errored

Synopsis

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

Arguments

inode

inode to test

Description

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

iget_failed

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

inode

The inode to discard

Description

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

1.4. Registration and Superblocks

deactivate_locked_super

LINUX

Kernel Hackers ManualSeptember 2014

Name

`deactivate_locked_super` — drop an active reference to superblock

Synopsis

```
void deactivate_locked_super (struct super_block * s);
```

Arguments

s

superblock to deactivate

Description

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.

deactivate_super

LINUX

Kernel Hackers ManualSeptember 2014

Name

`deactivate_super` — drop an active reference to superblock

Synopsis

```
void deactivate_super (struct super_block * s);
```

Arguments

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.

generic_shutdown_super

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

sb

superblock to kill

Description

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

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

sget

LINUX

Kernel Hackers ManualSeptember 2014

Name

sget — find or create a superblock

Synopsis

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

Arguments

type

filesystem type superblock should belong to

test

comparison callback

set

setup callback

flags

mount flags

data

argument to each of them

iterate_supers_type

LINUX

Kernel Hackers Manual September 2014

Name

`iterate_supers_type` — call function for superblocks of given type

Synopsis

```
void iterate_supers_type (struct file_system_type * type, void  
(*f) (struct super_block *, void *), void * arg);
```

Arguments

type

fs type

f

function to call

arg

argument to pass to it

Description

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

get_super

LINUX

Kernel Hackers ManualSeptember 2014

Name

`get_super` — get the superblock of a device

Synopsis

```
struct super_block * get_super (struct block_device * bdev);
```

Arguments

bdev

device to get the superblock for

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

get_super_thawed

LINUX

Kernel Hackers ManualSeptember 2014

Name

`get_super_thawed` — get thawed superblock of a device

Synopsis

```
struct super_block * get_super_thawed (struct block_device *
bdev);
```

Arguments

bdev

device to get the superblock for

Description

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

freeze_super

LINUX

Kernel Hackers Manual September 2014

Name

`freeze_super` — lock the filesystem and force it into a consistent state

Synopsis

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

Arguments

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 `->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.

thaw_super

LINUX

Kernel Hackers Manual September 2014

Name

thaw_super — - unlock filesystem

Synopsis

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

Arguments

sb

the super to thaw

Description

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

1.5. File Locks

posix_lock_file

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
int posix_lock_file (struct file * filp, struct file_lock *  
fl, struct file_lock * conflock);
```

Arguments

filp

The file to apply the lock to

fl

The lock to be applied

conflock

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

Description

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

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

posix_lock_file_wait

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
int posix_lock_file_wait (struct file * filp, struct file_lock  
* fl);
```

Arguments

filp

The file to apply the lock to

fl

The lock to be applied

Description

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

locks_mandatory_area

LINUX

Kernel Hackers ManualSeptember 2014

Name

locks_mandatory_area — Check for a conflicting lock

Synopsis

```
int locks_mandatory_area (int read_write, struct inode *  
inode, struct file * filp, loff_t offset, size_t count);
```

Arguments

read_write

FLOCK_VERIFY_WRITE for exclusive access, FLOCK_VERIFY_READ for
shared

inode

the file to check

filp

how the file was opened (if it was)

offset

start of area to check

count

length of area to check

Description

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

__break_lease

LINUX

Kernel Hackers Manual September 2014

Name

`__break_lease` — revoke all outstanding leases on file

Synopsis

```
int __break_lease (struct inode * inode, unsigned int mode,
unsigned int type);
```

Arguments

inode

the inode of the file to return

mode

O_RDONLY: break only write leases; O_WRONLY or O_RDWR: break all leases

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`.

lease_get_mtime

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
void lease_get_mtime (struct inode * inode, struct timespec *  
time);
```

Arguments

inode

the inode

time

pointer to a timespec which will contain the last modified time

Description

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

generic_setlease

LINUX

Kernel Hackers Manual September 2014

Name

`generic_setlease` — sets a lease on an open file

Synopsis

```
int generic_setlease (struct file * filp, long arg, struct  
file_lock ** flp);
```

Arguments

filp

file pointer

arg

type of lease to obtain

flp

input - `file_lock` to use, output - `file_lock` inserted

Description

The (input) `flp->fl_lmops->lm_break` function is required by `break_lease`.
Called with `inode->i_lock` held.

vfs_setlease

LINUX

Kernel Hackers Manual September 2014

Name

`vfs_setlease` — sets a lease on an open file

Synopsis

```
int vfs_setlease (struct file * filp, long arg, struct  
file_lock ** lease);
```

Arguments

filp

file pointer

arg

type of lease to obtain

lease

`file_lock` to use

Description

Call this to establish a lease on the file. The `(*lease)->fl_lmops->lm_break` operation must be set; if not, `break_lease` will oops!

This will call the filesystem's `setlease` file method, if defined. Note that there is no `getlease` method; instead, the filesystem `setlease` method should call back to `setlease` to add a lease to the inode's lease list, where `fcntl_getlease` can find it. Since `fcntl_getlease` only reports whether the current task holds a lease, a cluster filesystem need only do this for leases held by processes on this node.

There is also no `break_lease` method; filesystems that handle their own leases should break leases themselves from the filesystem's `open`, `create`, and (on truncate) `setattr` methods.

Warning

the only current `setlease` methods exist only to disable leases in certain cases. More `vfs` changes may be required to allow a full filesystem lease implementation.

flock_lock_file_wait

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
int flock_lock_file_wait (struct file * filp, struct file_lock
* fl);
```

Arguments

filp

The file to apply the lock to

fl

The lock to be applied

Description

Add a FLOCK style lock to a file.

vfs_test_lock

LINUX

Kernel Hackers ManualSeptember 2014

Name

`vfs_test_lock` — test file byte range lock

Synopsis

```
int vfs_test_lock (struct file * filp, struct file_lock * fl);
```

Arguments

filp

The file to test lock for

fl

The lock to test; also used to hold result

Description

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

vfs_lock_file

LINUX

Kernel Hackers ManualSeptember 2014

Name

`vfs_lock_file` — file byte range lock

Synopsis

```
int vfs_lock_file (struct file * filp, unsigned int cmd,
struct file_lock * fl, struct file_lock * conf);
```

Arguments

filp

The file to apply the lock to

cmd

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

fl

The lock to be applied

conf

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

Description

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

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

To avoid blocking kernel daemons, such as `lockd`, that need to acquire POSIX locks, the `->lock` interface may return asynchronously, before the lock has been granted or denied by the underlying filesystem, if (and only if) `lm_grant` is set. Callers expecting `->lock` to return asynchronously will only use `F_SETLK`, not `F_SETLKW`; they will set `FL_SLEEP` if (and only if) the request is for a blocking lock. When `->lock` does return asynchronously, it must return `FILE_LOCK_DEFERRED`, and call `->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 `->lm_grant` before returning to the caller with a `FILE_LOCK_DEFERRED` return code.

posix_unblock_lock

LINUX

Kernel Hackers Manual September 2014

Name

`posix_unblock_lock` — stop waiting for a file lock

Synopsis

```
int posix_unblock_lock (struct file_lock * waiter);
```

Arguments

waiter

the lock which was waiting

Description

lockd needs to block waiting for locks.

vfs_cancel_lock

LINUX

Kernel Hackers ManualSeptember 2014

Name

`vfs_cancel_lock` — file byte range unblock lock

Synopsis

```
int vfs_cancel_lock (struct file * filp, struct file_lock *  
fl);
```

Arguments

filp

The file to apply the unblock to

fl

The lock to be unblocked

Description

Used by lock managers to cancel blocked requests

lock_may_read

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
int lock_may_read (struct inode * inode, loff_t start,  
unsigned long len);
```

Arguments

inode

the inode that is being read

start

the first byte to read

len

the number of bytes to read

Description

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

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

lock_may_write

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
int lock_may_write (struct inode * inode, loff_t start,
unsigned long len);
```

Arguments

inode

the inode that is being written

start

the first byte to write

len

the number of bytes to write

Description

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

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

locks_unlink_lock

LINUX

Kernel Hackers Manual September 2014

Name

`locks_unlink_lock` — Delete a lock and then free it.

Synopsis

```
void locks_unlink_lock (struct file_lock ** thisfl_p);
```

Arguments

thisfl_p

pointer that points to the `fl_next` field of the previous `inode->i_flock` list entry

Description

Unlink a lock from all lists and free the namespace reference, but don't free it yet. Wake up processes that are blocked waiting for this lock and notify the FS that the lock has been cleared.

Must be called with the `i_lock` held!

locks_mandatory_locked

LINUX

Kernel Hackers ManualSeptember 2014

Name

`locks_mandatory_locked` — Check for an active lock

Synopsis

```
int locks_mandatory_locked (struct file * file);
```

Arguments

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.

fcntl_getlease

LINUX

Kernel Hackers ManualSeptember 2014

Name

`fcntl_getlease` — Enquire what lease is currently active

Synopsis

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

Arguments

filp

the file

Description

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

`F_RDLCK` to indicate a shared lease is held.

`F_WRLCK` to indicate an exclusive lease is held.

`F_UNLCK` to indicate no lease is held.

(if a lease break is pending):

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

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

XXX

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

check_conflicting_open

LINUX

Kernel Hackers Manual September 2014

Name

`check_conflicting_open` — see if the given dentry points to a file that has an existing open that would conflict with the desired lease.

Synopsis

```
int check_conflicting_open (const struct dentry * dentry,  
const long arg);
```

Arguments

dentry

dentry to check

arg

type of lease that we're trying to acquire

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.

fcntl_setlease

LINUX

Kernel Hackers ManualSeptember 2014

Name

`fcntl_setlease` — sets a lease on an open file

Synopsis

```
int fcntl_setlease (unsigned int fd, struct file * filp, long  
arg);
```

Arguments

fd

open file descriptor

filp

file pointer

arg

type of lease to obtain

Description

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

sys_flock

LINUX

Kernel Hackers Manual September 2014

Name

`sys_flock` — flock system call.

Synopsis

```
long sys_flock (unsigned int fd, unsigned int cmd);
```

Arguments

fd

the file descriptor to lock.

cmd

the type of lock to apply.

Description

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

`LOCK_SH` -- a shared lock.

`LOCK_EX` -- an exclusive lock.

`LOCK_UN` -- remove an existing lock.

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

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

1.6. Other Functions

`mpage_readpages`

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

mapping

the `address_space`

pages

The address of a `list_head` which contains the target pages. These pages have their `->index` populated and are otherwise uninitialised. The page at

pages->prev has the lowest file offset, and reads should be issued in *pages->prev* to *pages->next* order.

nr_pages

The number of pages at **pages*

get_block

The filesystem's block mapper function.

Description

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

If anything unusual happens, such as:

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

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

BH_Boundary explanation

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

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

submitted in the following order

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

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

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

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

mpage_writepages

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

mapping

address space structure to write

wbc

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

get_block

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

Description

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

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

generic_permission

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

inode

inode to check access rights for

mask

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

Description

Used to check for read/write/execute permissions on 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.

__inode_permission

LINUX

Kernel Hackers ManualSeptember 2014

Name

`__inode_permission` — Check for access rights to a given inode

Synopsis

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

Arguments

inode

Inode to check permission on

mask

Right to check for (MAY_READ, MAY_WRITE, MAY_EXEC)

Description

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`.

inode_permission

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

inode

Inode to check permission on

mask

Right to check for (MAY_READ, MAY_WRITE, MAY_EXEC)

Description

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*.

path_get

LINUX

Kernel Hackers Manual September 2014

Name

`path_get` — get a reference to a path

Synopsis

```
void path_get (const struct path * path);
```

Arguments

path

path to get the reference to

Description

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

path_put

LINUX

Kernel Hackers ManualSeptember 2014

Name

`path_put` — put a reference to a path

Synopsis

```
void path_put (const struct path * path);
```

Arguments

path

path to put the reference to

Description

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

vfs_path_lookup

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
int vfs_path_lookup (struct dentry * dentry, struct vfsmount *  
mnt, const char * name, unsigned int flags, struct path *  
path);
```

Arguments

dentry

pointer to dentry of the base directory

mnt

pointer to vfs mount of the base directory

name

pointer to file name

flags

lookup flags

path

pointer to struct path to fill

lookup_one_len

LINUX

Kernel Hackers Manual September 2014

Name

`lookup_one_len` — filesystem helper to lookup single pathname component

Synopsis

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

Arguments

name

pathname component to lookup

base

base directory to lookup from

len

maximum length *len* should be interpreted to

Description

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

vfs_unlink

LINUX

Kernel Hackers Manual September 2014

Name

`vfs_unlink` — unlink a filesystem object

Synopsis

```
int vfs_unlink (struct inode * dir, struct dentry * dentry,  
struct inode ** delegated_inode);
```

Arguments

dir

parent directory

dentry

victim

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.

vfs_link

LINUX

Name

`vfs_link` — create a new link

Synopsis

```
int vfs_link (struct dentry * old_dentry, struct inode * dir,  
struct dentry * new_dentry, struct inode ** delegated_inode);
```

Arguments

old_dentry

object to be linked

dir

new parent

new_dentry

where to create the new link

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.

vfs_rename

LINUX

Kernel Hackers Manual September 2014

Name

`vfs_rename` — rename a filesystem object

Synopsis

```
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);
```

Arguments

old_dir

parent of source

old_dentry

source

new_dir

parent of destination

new_dentry

destination

delegated_inode

returns an inode needing a delegation break

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

a) we can get into loop creation. Check is done in `is_subdir`. b) 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. c) 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 `->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. d) 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].

sync_mapping_buffers

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
int sync_mapping_buffers (struct address_space * mapping);
```

Arguments

mapping

the mapping which wants those buffers written

Description

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

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

mark_buffer_dirty

LINUX

Name

`mark_buffer_dirty` — mark a `buffer_head` as needing writeout

Synopsis

```
void mark_buffer_dirty (struct buffer_head * bh);
```

Arguments

bh

the `buffer_head` to mark dirty

Description

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

`mark_buffer_dirty` is atomic. It takes `bh->b_page->mapping->private_lock`, `mapping->tree_lock` and `mapping->host->i_lock`.

__bread

LINUX

Name

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

Synopsis

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

Arguments

bdev

the `block_device` to read from

block

number of block

size

size (in bytes) to read

Description

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

block_invalidatepage

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
void block_invalidatepage (struct page * page, unsigned int  
offset, unsigned int length);
```

Arguments

page

the page which is affected

offset

start of the range to invalidate

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.

ll_rw_block

LINUX

Name

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

Synopsis

```
void ll_rw_block (int rw, int nr, struct buffer_head * bhs[]);
```

Arguments

rw

whether to READ or WRITE or maybe READA (readahead)

nr

number of struct buffer_heads in the array

bhs[]

array of pointers to struct buffer_head

Description

`ll_rw_block` takes an array of pointers to struct buffer_heads, and requests an I/O operation on them, either a READ or a WRITE. The third READA option is described in the documentation for `generic_make_request` which `ll_rw_block` calls.

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

bh_uptodate_or_lock

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bh_uptodate_or_lock` — Test whether the buffer is uptodate

Synopsis

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

Arguments

bh

struct buffer_head

Description

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

bh_submit_read

LINUX

Name

`bh_submit_read` — Submit a locked buffer for reading

Synopsis

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

Arguments

bh

struct buffer_head

Description

Returns zero on success and -EIO on error.

bio_reset

LINUX

Name

`bio_reset` — reinitialize a bio

Synopsis

```
void bio_reset (struct bio * bio);
```

Arguments

bio

bio to reset

Description

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

bio_chain

LINUX

Kernel Hackers Manual September 2014

Name

`bio_chain` — chain bio completions

Synopsis

```
void bio_chain (struct bio * bio, struct bio * parent);
```

Arguments

bio

the target 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*.

bio_alloc_bioset

LINUX

Kernel Hackers Manual September 2014

Name

`bio_alloc_bioset` — allocate a bio for I/O

Synopsis

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

Arguments

gfp_mask

the GFP_ mask given to the slab allocator

nr_iovecs

number of iovecs to pre-allocate

bs

the bio_set to allocate from.

Description

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

RETURNS

Pointer to new bio on success, NULL on failure.

bio_put

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_put` — release a reference to a bio

Synopsis

```
void bio_put (struct bio * bio);
```

Arguments

bio

bio to release reference to

Description

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

__bio_clone_fast

LINUX

Name

`__bio_clone_fast` — clone a bio that shares the original bio's biovec

Synopsis

```
void __bio_clone_fast (struct bio * bio, struct bio *  
bio_src);
```

Arguments

bio

destination bio

bio_src

bio to clone

Description

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

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

`bio_clone_fast`

LINUX

Name

`bio_clone_fast` — clone a bio that shares the original bio's biovec

Synopsis

```
struct bio * bio_clone_fast (struct bio * bio, gfp_t gfp_mask,  
struct bio_set * bs);
```

Arguments

bio

bio to clone

gfp_mask

allocation priority

bs

bio_set to allocate from

Description

Like `__bio_clone_fast`, only also allocates the returned bio

`bio_clone_bioset`

LINUX

Name

`bio_clone_bioset` — clone a bio

Synopsis

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

Arguments

bio_src

bio to clone

gfp_mask

allocation priority

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.

`bio_get_nr_vecs`

LINUX

Name

`bio_get_nr_vecs` — return approx number of vecs

Synopsis

```
int bio_get_nr_vecs (struct block_device * bdev);
```

Arguments

bdev

I/O target

Description

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

bio_add_pc_page

LINUX

Name

`bio_add_pc_page` — attempt to add page to bio

Synopsis

```
int bio_add_pc_page (struct request_queue * q, struct bio *  
bio, struct page * page, unsigned int len, unsigned int  
offset);
```

Arguments

q
the target queue

bio
destination bio

page
page to add

len
vec entry length

offset
vec entry offset

Description

Attempt to add a page to the `bio_vec` maplist. This can fail for a number of reasons, such as the bio being full or target block device limitations. The target block device must allow bio's 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.

bio_add_page

LINUX

Kernel Hackers Manual September 2014

Name

`bio_add_page` — attempt to add page to bio

Synopsis

```
int bio_add_page (struct bio * bio, struct page * page,  
unsigned int len, unsigned int offset);
```

Arguments

bio

destination bio

page

page to add

len

vec entry length

offset

vec entry offset

Description

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

submit_bio_wait

LINUX

Kernel Hackers ManualSeptember 2014

Name

`submit_bio_wait` — submit a bio, and wait until it completes

Synopsis

```
int submit_bio_wait (int rw, struct bio * bio);
```

Arguments

rw

whether to READ or WRITE, or maybe to READA (read ahead)

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.

bio_advance

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_advance` — increment/complete a bio by some number of bytes

Synopsis

```
void bio_advance (struct bio * bio, unsigned bytes);
```

Arguments

bio

bio to advance

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.

bio_alloc_pages

LINUX

Kernel Hackers Manual September 2014

Name

`bio_alloc_pages` — allocates a single page for each bvec in a bio

Synopsis

```
int bio_alloc_pages (struct bio * bio, gfp_t gfp_mask);
```

Arguments

bio

bio to allocate pages for

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.

bio_copy_data

LINUX

Name

`bio_copy_data` — copy contents of data buffers from one chain of bios to another

Synopsis

```
void bio_copy_data (struct bio * dst, struct bio * src);
```

Arguments

dst

destination bio list

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).

`bio_uncopy_user`

LINUX

Name

`bio_uncopy_user` — finish previously mapped bio

Synopsis

```
int bio_uncopy_user (struct bio * bio);
```

Arguments

bio

bio being terminated

Description

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

bio_copy_user

LINUX

Name

`bio_copy_user` — copy user data to bio

Synopsis

```
struct bio * bio_copy_user (struct request_queue * q, struct  
rq_map_data * map_data, unsigned long uaddr, unsigned int len,  
int write_to_vm, gfp_t gfp_mask);
```

Arguments

q

destination block queue

map_data

pointer to the `rq_map_data` holding pages (if necessary)

uaddr

start of user address

len

length in bytes

write_to_vm

bool indicating writing to pages or not

gfp_mask

memory allocation flags

Description

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

bio_map_user

LINUX

Kernel Hackers Manual September 2014

Name

`bio_map_user` — map user address into bio

Synopsis

```
struct bio * bio_map_user (struct request_queue * q, struct  
block_device * bdev, unsigned long uaddr, unsigned int len,  
int write_to_vm, gfp_t gfp_mask);
```

Arguments

q
the struct request_queue for the bio

bdev
destination block device

uaddr
start of user address

len
length in bytes

write_to_vm
bool indicating writing to pages or not

gfp_mask
memory allocation flags

Description

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

bio_unmap_user

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_unmap_user` — unmap a bio

Synopsis

```
void bio_unmap_user (struct bio * bio);
```

Arguments

bio

the bio being unmapped

Description

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

`bio_unmap_user` may sleep.

bio_map_kern

LINUX

Kernel Hackers Manual September 2014

Name

`bio_map_kern` — map kernel address into bio

Synopsis

```
struct bio * bio_map_kern (struct request_queue * q, void *  
data, unsigned int len, gfp_t gfp_mask);
```

Arguments

q

the struct request_queue for the bio

data

pointer to buffer to map

len

length in bytes

gfp_mask

allocation flags for bio allocation

Description

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

bio_copy_kern

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_copy_kern` — copy kernel address into bio

Synopsis

```
struct bio * bio_copy_kern (struct request_queue * q, void *  
data, unsigned int len, gfp_t gfp_mask, int reading);
```

Arguments

q

the struct request_queue for the bio

data

pointer to buffer to copy

len

length in bytes

gfp_mask

allocation flags for bio and page allocation

reading

data direction is READ

Description

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

bio_endio

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_endio` — end I/O on a bio

Synopsis

```
void bio_endio (struct bio * bio, int error);
```

Arguments

bio

bio

error

error, if any

Description

`bio_endio` will end I/O on the whole bio. `bio_endio` is the preferred way to end I/O on a bio, it takes care of clearing `BIO_UPTODATE` on error. *error* is 0 on success, and one of the established -Exxx (-EIO, for instance) error values in

case something went wrong. 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_nodect

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_endio_nodect` — end I/O on a bio, without decrementing `bi_remaining`

Synopsis

```
void bio_endio_nodect (struct bio * bio, int error);
```

Arguments

bio

bio

error

error, if any

Description

For code that has saved and restored `bi_end_io`; thing hard before using this function, probably you should've cloned the entire bio.

bio_split

LINUX

Kernel Hackers Manual September 2014

Name

`bio_split` — split a bio

Synopsis

```
struct bio * bio_split (struct bio * bio, int sectors, gfp_t
gfp, struct bio_set * bs);
```

Arguments

bio

bio to split

sectors

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

gfp

gfp mask

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.

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.

bio_trim

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bio_trim` — trim a bio

Synopsis

```
void bio_trim (struct bio * bio, int offset, int size);
```

Arguments

bio

bio to trim

offset

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

size

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

bio_set_create

LINUX

Name

`bioset_create` — Create a `bio_set`

Synopsis

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

Arguments

pool_size

Number of bio and `bio_vecs` to cache in the mempool

front_pad

Number of bytes to allocate in front of the returned bio

Description

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

seq_open

LINUX

Name

`seq_open` — initialize sequential file

Synopsis

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

Arguments

file

file we initialize

op

method table describing the sequence

Description

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

seq_read

LINUX

Name

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

Synopsis

```
ssize_t seq_read (struct file * file, char __user * buf,
size_t size, loff_t * ppos);
```

Arguments

file

the file to read from

buf

the buffer to read to

size

the maximum number of bytes to read

ppos

the current position in the file

Description

Ready-made `->f_op->read`

seq_lseek

LINUX

Name

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

Synopsis

```
loff_t seq_lseek (struct file * file, loff_t offset, int  
whence);
```

Arguments

file

the file in question

offset

new position

whence

0 for absolute, 1 for relative position

Description

Ready-made `->f_op->llseek`

`seq_release`

LINUX

Name

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

Synopsis

```
int seq_release (struct inode * inode, struct file * file);
```

Arguments

inode

its inode

file

file in question

Description

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

seq_escape

LINUX

Name

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

Synopsis

```
int seq_escape (struct seq_file * m, const char * s, const  
char * esc);
```

Arguments

m

target buffer

s

string

esc

set of characters that need escaping

Description

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

mangle_path

LINUX

Kernel Hackers ManualSeptember 2014

Name

`mangle_path` — mangle and copy path to buffer beginning

Synopsis

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

Arguments

s

buffer start

p

beginning of path in above buffer

esc

set of characters that need escaping

Description

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

seq_path

LINUX

Kernel Hackers Manual September 2014

Name

`seq_path` — `seq_file` interface to print a pathname

Synopsis

```
int seq_path (struct seq_file * m, const struct path * path,  
const char * esc);
```

Arguments

m

the seq_file handle

path

the struct path to print

esc

set of characters to escape in the output

Description

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

seq_write

LINUX

Kernel Hackers ManualSeptember 2014

Name

seq_write — write arbitrary data to buffer

Synopsis

```
int seq_write (struct seq_file * seq, const void * data,
size_t len);
```

Arguments

seq

seq_file identifying the buffer to which data should be written

data

data address

len

number of bytes

Description

Return 0 on success, non-zero otherwise.

seq_pad

LINUX

Kernel Hackers Manual September 2014

Name

seq_pad — write padding spaces to buffer

Synopsis

```
void seq_pad (struct seq_file * m, char c);
```

Arguments

m

seq_file identifying the buffer to which data should be written

c

the byte to append after padding if non-zero

seq_hlist_start

LINUX

Kernel Hackers ManualSeptember 2014

Name

seq_hlist_start — start an iteration of a hlist

Synopsis

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

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

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

seq_hlist_start_head

LINUX

Kernel Hackers ManualSeptember 2014

Name

`seq_hlist_start_head` — start an iteration of a hlist

Synopsis

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

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

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

seq_hlist_next

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

v

the current iterator

head

the head of the hlist

ppos

the current position

Description

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

seq_hlist_start_rcu

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

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

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

`seq_hlist_start_head_rcu`

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

head

the head of the hlist

pos

the start position of the sequence

Description

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

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

seq_hlist_next_rcu

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

v

the current iterator

head

the head of the hlist

ppos

the current position

Description

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

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

seq_hlist_start_percpu

LINUX

Kernel Hackers ManualSeptember 2014

Name

`seq_hlist_start_percpu` — start an iteration of a percpu hlist array

Synopsis

```
struct hlist_node * seq_hlist_start_percpu (struct hlist_head  
__percpu * head, int * cpu, loff_t pos);
```

Arguments

head

pointer to percpu array of struct `hlist_heads`

cpu

pointer to cpu “cursor”

pos

start position of sequence

Description

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

seq_hlist_next_percpu

LINUX

Kernel Hackers ManualSeptember 2014

Name

`seq_hlist_next_percpu` — move to the next position of the percpu hlist array

Synopsis

```
struct hlist_node * seq_hlist_next_percpu (void * v, struct
hlist_head __percpu * head, int * cpu, loff_t * pos);
```

Arguments

*v*pointer to current `hlist_node`*head*pointer to percpu array of struct `hlist_heads`

cpu

pointer to `cpu` “cursor”

pos

start position of sequence

Description

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

register_filesystem

LINUX

Kernel Hackers ManualSeptember 2014

Name

`register_filesystem` — register a new filesystem

Synopsis

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

Arguments

fs

the file system structure

Description

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

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

unregister_filesystem

LINUX

Kernel Hackers Manual September 2014

Name

`unregister_filesystem` — unregister a file system

Synopsis

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

Arguments

fs

filesystem to unregister

Description

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

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

writeback_in_progress

LINUX

Kernel Hackers Manual September 2014

Name

`writeback_in_progress` — determine whether there is writeback in progress

Synopsis

```
int writeback_in_progress (struct backing_dev_info * bdi);
```

Arguments

bdi

the device's `backing_dev_info` structure.

Description

Determine whether there is writeback waiting to be handled against a backing device.

__mark_inode_dirty

LINUX

Name

`__mark_inode_dirty` — internal function

Synopsis

```
void __mark_inode_dirty (struct inode * inode, int flags);
```

Arguments

inode

inode to mark

flags

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

Description

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

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

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

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

writeback_inodes_sb_nr

LINUX

Kernel Hackers ManualSeptember 2014

Name

`writeback_inodes_sb_nr` — writeback dirty inodes from given `super_block`

Synopsis

```
void writeback_inodes_sb_nr (struct super_block * sb, unsigned  
long nr, enum wb_reason reason);
```

Arguments

sb

the superblock

nr

the number of pages to write

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.

writeback_inodes_sb

LINUX

Kernel Hackers Manual September 2014

Name

`writeback_inodes_sb` — writeback dirty inodes from given `super_block`

Synopsis

```
void writeback_inodes_sb (struct super_block * sb, enum  
wb_reason reason);
```

Arguments

sb

the superblock

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.

try_to_writeback_inodes_sb_nr

LINUX

Name

`try_to_writeback_inodes_sb_nr` — try to start writeback if none underway

Synopsis

```
int try_to_writeback_inodes_sb_nr (struct super_block * sb,
unsigned long nr, enum wb_reason reason);
```

Arguments

sb

the superblock

nr

the number of pages to write

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.

try_to_writeback_inodes_sb

LINUX

Name

`try_to_writeback_inodes_sb` — try to start writeback if none underway

Synopsis

```
int try_to_writeback_inodes_sb (struct super_block * sb, enum  
wb_reason reason);
```

Arguments

sb

the superblock

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.

sync_inodes_sb

LINUX

Name

`sync_inodes_sb` — sync sb inode pages

Synopsis

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

Arguments

sb

the superblock

Description

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

write_inode_now

LINUX

Name

`write_inode_now` — write an inode to disk

Synopsis

```
int write_inode_now (struct inode * inode, int sync);
```

Arguments

inode

inode to write to disk

sync

whether the write should be synchronous or not

Description

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

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

sync_inode

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
int sync_inode (struct inode * inode, struct writeback_control  
* wbc);
```

Arguments

inode

the inode to sync

wbc

controls the writeback mode

Description

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

The caller must have a ref on the inode.

sync_inode_metadata

LINUX

Kernel Hackers ManualSeptember 2014

Name

`sync_inode_metadata` — write an inode to disk

Synopsis

```
int sync_inode_metadata (struct inode * inode, int wait);
```

Arguments

inode

the inode to sync

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.

freeze_bdev

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

bdev

blockdevice to lock

Description

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

thaw_bdev

LINUX

Kernel Hackers Manual September 2014

Name

`thaw_bdev` — - unlock filesystem

Synopsis

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

Arguments

bdev

blockdevice to unlock

sb

associated superblock

Description

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

bdev_read_page

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bdev_read_page` — Start reading a page from a block device

Synopsis

```
int bdev_read_page (struct block_device * bdev, sector_t
sector, struct page * page);
```

Arguments

bdev

The device to read the page from

sector

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

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.

bdev_write_page

LINUX

Kernel Hackers Manual September 2014

Name

`bdev_write_page` — Start writing a page to a block device

Synopsis

```
int bdev_write_page (struct block_device * bdev, sector_t
sector, struct page * page, struct writeback_control * wbc);
```

Arguments

bdev

The device to write the page to

sector

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

page

The page to write

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.

bdgrab

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bdgrab` — - Grab a reference to an already referenced block device

Synopsis

```
struct block_device * bdgrab (struct block_device * bdev);
```

Arguments

bdev

Block device to grab a reference to.

bd_link_disk_holder

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bd_link_disk_holder` — create symlinks between holding disk and slave
`bdev`

Synopsis

```
int bd_link_disk_holder (struct block_device * bdev, struct
gendisk * disk);
```

Arguments

bdev

the claimed slave bdev

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.

RETURNS

0 on success, -errno on failure.

bd_unlink_disk_holder

LINUX

Kernel Hackers ManualSeptember 2014

Name

`bd_unlink_disk_holder` — destroy symlinks created by `bd_link_disk_holder`

Synopsis

```
void bd_unlink_disk_holder (struct block_device * bdev, struct  
gendisk * disk);
```

Arguments

bdev

the calimed slave bdev

disk

the holding disk

Description

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

CONTEXT

Might sleep.

check_disk_size_change

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
void check_disk_size_change (struct gendisk * disk, struct  
block_device * bdev);
```

Arguments

disk

struct gendisk to check

bdev

struct bdev to adjust.

Description

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

revalidate_disk

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
int revalidate_disk (struct gendisk * disk);
```

Arguments

disk

struct gendisk to be revalidated

Description

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

blkdev_get

LINUX

Name

`blkdev_get` — open a block device

Synopsis

```
int blkdev_get (struct block_device * bdev, fmode_t mode, void  
* holder);
```

Arguments

bdev

block_device to open

mode

FMODE_* mask

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.

RETURNS

0 on success, -errno on failure.

blkdev_get_by_path

LINUX

Kernel Hackers ManualSeptember 2014

Name

`blkdev_get_by_path` — open a block device by name

Synopsis

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

Arguments

path

path to the block device to open

mode

FMODE_* mask

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.

RETURNS

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

blkdev_get_by_dev

LINUX

Kernel Hackers ManualSeptember 2014

Name

`blkdev_get_by_dev` — open a block device by device number

Synopsis

```
struct block_device * blkdev_get_by_dev (dev_t dev, fmode_t
mode, void * holder);
```

Arguments

dev

device number of block device to open

mode

FMODE_* mask

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.

RETURNS

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

lookup_bdev

LINUX

Name

`lookup_bdev` — lookup a struct `block_device` by name

Synopsis

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

Arguments

pathname

special file representing the block device

Description

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

Chapter 2. The proc filesystem

2.1. sysctl interface

proc_dostring

LINUX

Kernel Hackers Manual September 2014

Name

`proc_dostring` — read a string sysctl

Synopsis

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

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

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

Returns 0 on success.

proc_dointvec

LINUX

Kernel Hackers ManualSeptember 2014

Name

`proc_dointvec` — read a vector of integers

Synopsis

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

Arguments

table

the sysctl table

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

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

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

Returns 0 on success.

proc_dointvec_minmax

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

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

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

Returns 0 on success.

proc_doulongvec_minmax

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

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

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

Returns 0 on success.

proc_doulongvec_ms_jiffies_minmax

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

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

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

Returns 0 on success.

proc_dointvec_jiffies

LINUX

Kernel Hackers Manual September 2014

Name

`proc_dointvec_jiffies` — read a vector of integers as seconds

Synopsis

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

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

file position

Description

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

Returns 0 on success.

proc_dointvec_userhz_jiffies

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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


Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

pointer to the file position

Description

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

Returns 0 on success.

proc_dointvec_ms_jiffies

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

table

the sysctl table

write

TRUE if this is a write to the sysctl file

buffer

the user buffer

lenp

the size of the user buffer

ppos

the current position in the file

Description

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

Returns 0 on success.

2.2. proc filesystem interface

proc_flush_task

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

task

task that should be flushed.

Description

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

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

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

NOTE

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

Chapter 3. Events based on file descriptors

eventfd_signal

LINUX

Kernel Hackers ManualSeptember 2014

Name

`eventfd_signal` — Adds *n* to the eventfd counter.

Synopsis

```
__u64 eventfd_signal (struct eventfd_ctx * ctx, __u64 n);
```

Arguments

ctx

[in] Pointer to the eventfd context.

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.

eventfd_ctx_get

LINUX

Kernel Hackers Manual September 2014

Name

`eventfd_ctx_get` — Acquires a reference to the internal eventfd context.

Synopsis

```
struct eventfd_ctx * eventfd_ctx_get (struct eventfd_ctx *  
ctx);
```

Arguments

ctx

[in] Pointer to the eventfd context.

Returns

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

eventfd_ctx_put

LINUX

Kernel Hackers ManualSeptember 2014

Name

`eventfd_ctx_put` — Releases a reference to the internal eventfd context.

Synopsis

```
void eventfd_ctx_put (struct eventfd_ctx * ctx);
```

Arguments

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`.

eventfd_ctx_remove_wait_queue

LINUX

Name

`eventfd_ctx_remove_wait_queue` — Read the current counter and removes wait queue.

Synopsis

```
int eventfd_ctx_remove_wait_queue (struct eventfd_ctx * ctx,  
wait_queue_t * wait, __u64 * cnt);
```

Arguments

ctx

[in] Pointer to eventfd context.

wait

[in] Wait queue to be removed.

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.

eventfd_ctx_read

LINUX

Kernel Hackers Manual September 2014

Name

`eventfd_ctx_read` — Reads the eventfd counter or wait if it is zero.

Synopsis

```
ssize_t eventfd_ctx_read (struct eventfd_ctx * ctx, int
no_wait, __u64 * cnt);
```

Arguments

ctx

[in] Pointer to eventfd context.

no_wait

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

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.

eventfd_fget

LINUX

Kernel Hackers ManualSeptember 2014

Name

`eventfd_fget` — Acquire a reference of an eventfd file descriptor.

Synopsis

```
struct file * eventfd_fget (int fd);
```

Arguments

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.

eventfd_ctx_fdget

LINUX

Kernel Hackers ManualSeptember 2014

Name

`eventfd_ctx_fdget` — Acquires a reference to the internal eventfd context.

Synopsis

```
struct eventfd_ctx * eventfd_ctx_fdget (int fd);
```

Arguments

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`

eventfd_ctx_fileget

LINUX

Name

`eventfd_ctx_fileget` — Acquires a reference to the internal eventfd context.

Synopsis

```
struct eventfd_ctx * eventfd_ctx_fileget (struct file * file);
```

Arguments

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 4. The Filesystem for Exporting Kernel Objects

sysfs_create_file_ns

LINUX

Kernel Hackers ManualSeptember 2014

Name

`sysfs_create_file_ns` — create an attribute file for an object with custom ns

Synopsis

```
int sysfs_create_file_ns (struct kobject * kobj, const struct attribute * attr, const void * ns);
```

Arguments

kobj

object we're creating for

attr

attribute descriptor

ns

namespace the new file should belong to

sysfs_add_file_to_group

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

group

group name.

sysfs_chmod_file

LINUX

Name

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

Synopsis

```
int sysfs_chmod_file (struct kobject * kobj, const struct
attribute * attr, umode_t mode);
```

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

mode

file permissions.

sysfs_remove_file_ns

LINUX

Name

`sysfs_remove_file_ns` — remove an object attribute with a custom ns tag

Synopsis

```
void sysfs_remove_file_ns (struct kobject * kobj, const struct  
attribute * attr, const void * ns);
```

Arguments

kobj

object we're acting for

attr

attribute descriptor

ns

namespace tag of the file to remove

Description

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

sysfs_remove_file_from_group

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

kobj

object we're acting for.

attr

attribute descriptor.

group

group name.

sysfs_create_bin_file

LINUX

Kernel Hackers ManualSeptember 2014

Name

`sysfs_create_bin_file` — create binary file for object.

Synopsis

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

Arguments

kobj

object.

attr

attribute descriptor.

sysfs_remove_bin_file

LINUX

Kernel Hackers ManualSeptember 2014

Name

`sysfs_remove_bin_file` — remove binary file for object.

Synopsis

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

Arguments

kobj

object.

attr

attribute descriptor.

sysfs_create_link

LINUX

Kernel Hackers ManualSeptember 2014

Name

`sysfs_create_link` — create symlink between two objects.

Synopsis

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

Arguments

kobj

object whose directory we're creating the link in.

target

object we're pointing to.

name

name of the symlink.

sysfs_remove_link

LINUX

Name

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

Synopsis

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

Arguments

kobj

object we’re acting for.

name

name of the symlink to remove.

sysfs_rename_link_ns

LINUX

Name

`sysfs_rename_link_ns` — rename symlink in object’s directory.

Synopsis

```
int sysfs_rename_link_ns (struct kobject * kobj, struct  
kobject * targ, const char * old, const char * new, const  
void * new_ns);
```

Arguments

kobj

object we're acting for.

targ

object we're pointing to.

old

previous name of the symlink.

new

new name of the symlink.

new_ns

new namespace of the symlink.

Description

A helper function for the common rename symlink idiom.

Chapter 5. The debugfs filesystem

5.1. debugfs interface

debugfs_create_file

LINUX

Kernel Hackers ManualSeptember 2014

Name

`debugfs_create_file` — create a file in the debugfs filesystem

Synopsis

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

Arguments

name

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

mode

the permission that the file should have.

parent

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

data

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

fops

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

Description

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

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

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

debugfs_create_dir

LINUX

Kernel Hackers Manual September 2014

Name

`debugfs_create_dir` — create a directory in the debugfs filesystem

Synopsis

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


Arguments

name

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

parent

a pointer to the parent dentry for this file. This should be a directory dentry if set. If this 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.

debugfs_create_symlink

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

parent

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

target

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

Description

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

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

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

debugfs_remove

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

dentry

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

Description

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

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

debugfs_remove_recursive

LINUX

Kernel Hackers ManualSeptember 2014

Name

`debugfs_remove_recursive` — recursively removes a directory

Synopsis

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

Arguments

dentry

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

Description

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

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

debugfs_rename

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

old_dir

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

old_dentry

dentry of an object to be renamed.

new_dir

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

new_name

a pointer to a string containing the target name.

Description

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

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

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

debugfs_initialized

LINUX

Kernel Hackers Manual September 2014

Name

`debugfs_initialized` — Tells whether debugfs has been registered

Synopsis

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

Arguments

void

no arguments

debugfs_create_u8

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

Description

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

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

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

debugfs_create_u16

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

Description

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

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

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

debugfs_create_u32

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

Description

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

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

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

debugfs_create_u64

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

Description

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

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

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

debugfs_create_x8

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

debugfs_create_x16

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

debugfs_create_x32

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

debugfs_create_x64

LINUX

Kernel Hackers Manual September 2014

Name

`debugfs_create_x64` — create a debugfs file that is used to read and write an unsigned 64-bit value

Synopsis

```
struct dentry * debugfs_create_x64 (const char * name, umode_t
mode, struct dentry * parent, u64 * value);
```

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

debugfs_create_size_t

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

debugfs_create_atomic_t

LINUX

Kernel Hackers Manual September 2014

Name

`debugfs_create_atomic_t` — create a debugfs file that is used to read and write an `atomic_t` value

Synopsis

```
struct dentry * debugfs_create_atomic_t (const char * name,  
umode_t mode, struct dentry * parent, atomic_t * value);
```


Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

debugfs_create_bool

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

value

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

Description

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

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

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

debugfs_create_blob

LINUX

Name

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

Synopsis

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

Arguments

name

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

mode

the permission that the file should have

parent

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

blob

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

Description

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

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

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

debugfs_create_u32_array

LINUX

Kernel Hackers Manual September 2014

Name

`debugfs_create_u32_array` — create a `debugfs` file that is used to read `u32` array.

Synopsis

```
struct dentry * debugfs_create_u32_array (const char * name,
umode_t mode, struct dentry * parent, u32 * array, u32
elements);
```

Arguments

name

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

mode

the permission that the file should have.

parent

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

array

u32 array that provides data.

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.

debugfs_print_regs32

LINUX

Kernel Hackers Manual September 2014

Name

`debugfs_print_regs32` — use `seq_print` to describe a set of registers

Synopsis

```
int debugfs_print_regs32 (struct seq_file * s, const struct
debugfs_reg32 * regs, int nregs, void __iomem * base, char *
prefix);
```

Arguments

s

the seq_file structure being used to generate output

regs

an array of struct debugfs_reg32 structures

nregs

the length of the above array

base

the base address to be used in reading the registers

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

debugfs_create_regset32

LINUX

Kernel Hackers Manual September 2014

Name

debugfs_create_regset32 — create a debugfs file that returns register

values

Synopsis

```
struct dentry * debugfs_create_regset32 (const char * name,
umode_t mode, struct dentry * parent, struct debugfs_regset32
* regset);
```

Arguments

name

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

mode

the permission that the file should have

parent

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

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.

Chapter 5. The debugfs filesystem

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

Chapter 6. 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 `journal_init_inode()` call is for journals stored in filesystem inodes, or the `journal_init_dev()` call can be use for journal stored on a raw device (in a continuous range of blocks). A `journal_t` is a typedef for a struct pointer, so when you are finally finished make sure you call `journal_destroy()` on it to free up any used kernel memory.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

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

```

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

```

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

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.

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

```

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

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

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

```

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

6.2. Data Types

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

6.2.1. Structures

typedef handle_t

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

```
typedef handle_t;
```

Description

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

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

This is an opaque datatype.

typedef journal_t

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

```
typedef journal_t;
```

Description

`journal_t` is linked to from the fs superblock structure.

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

This is an opaque datatype.

struct handle_s

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Members

h_transaction

Which compound transaction is this update a part of?

h_buffer_credits

Number of remaining buffers we are allowed to dirty.

h_ref

Reference count on this handle

h_err

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

`h_sync`

flag for sync-on-close

`h_jdata`

flag to force data journaling

`h_aborted`

flag indicating fatal error on handle

`h_lockdep_map`

lockdep info for debugging lock problems

struct journal_s

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

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

Members

`j_flags`

General journaling state flags

`j_errno`

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

j_sb_buffer

First part of superblock buffer

j_superblock

Second part of superblock buffer

j_format_version

Version of the superblock format

j_state_lock

Protect the various scalars in the journal

j_barrier_count

Number of processes waiting to create a barrier lock

j_running_transaction

The current running transaction..

j_committing_transaction

the transaction we are pushing to disk

j_checkpoint_transactions

a linked circular list of all transactions waiting for checkpointing

j_wait_transaction_locked

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

j_wait_logspace

Wait queue for waiting for checkpointing to complete

j_wait_done_commit

Wait queue for waiting for commit to complete

j_wait_checkpoint

Wait queue to trigger checkpointing

j_wait_commit

Wait queue to trigger commit

`j_wait_updates`

Wait queue to wait for updates to complete

`j_checkpoint_mutex`

Mutex for locking against concurrent checkpoints

`j_head`

Journal head - identifies the first unused block in the journal

`j_tail`

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

`j_free`

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

`j_first`

The block number of the first usable block

`j_last`

The block number one beyond the last usable block

`j_dev`

Device where we store the journal

`j_blocksize`

blocksize for the location where we store the journal.

`j_blk_offset`

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

`j_fs_dev`

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

`j_maxlen`

Total maximum capacity of the journal region on disk.

`j_list_lock`

Protects the buffer lists and internal buffer state.

`j_inode`

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

`j_tail_sequence`

Sequence number of the oldest transaction in the log

`j_transaction_sequence`

Sequence number of the next transaction to grant

`j_commit_sequence`

Sequence number of the most recently committed transaction

`j_commit_request`

Sequence number of the most recent transaction wanting commit

`j_commit_waited`

Sequence number of the most recent transaction someone is waiting for to commit.

`j_uuid[16]`

Uuid of client object.

`j_task`

Pointer to the current commit thread for this journal

`j_max_transaction_buffers`

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

`j_commit_interval`

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

`j_commit_timer`

The timer used to wakeup the commit thread

`j_revoke_lock`

Protect the revoke table

`j_revoke`

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

`j_revoke_table[2]`

alternate revoke tables for `j_revoke`

`j_wbuf`

array of `buffer_heads` for `journal_commit_transaction`

`j_wbufsize`

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

`j_last_sync_writer`

most recent pid which did a synchronous write

`j_average_commit_time`

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

`j_private`

An opaque pointer to fs-private information.

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

journal_init_dev

LINUX

Name

`journal_init_dev` — creates and initialises a journal structure

Synopsis

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

Arguments

bdev

Block device on which to create the journal

fs_dev

Device which hold journalled filesystem for this journal.

start

Block nr Start of journal.

len

Length of the journal in blocks.

blocksize

blocksize of journalling device

Returns

a newly created `journal_t *`

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

journal_init_inode

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

inode

An inode to create the journal in

Description

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

journal_create

LINUX

Name

`journal_create` — Initialise the new journal file

Synopsis

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

Arguments

journal

Journal to create. This structure must have been initialised

Description

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

journal_load

LINUX

Name

`journal_load` — Read journal from disk.

Synopsis

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

Arguments

journal

Journal to act on.

Description

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

journal_destroy

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_destroy` — Release a `journal_t` structure.

Synopsis

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


Arguments

journal

Journal to act on.

Description

Release a `journal_t` structure once it is no longer in use by the journaled object.

Return `<0` if we couldn't clean up the journal.

journal_check_used_features

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_check_used_features` — Check if features specified are used.

Synopsis

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

Arguments

journal

Journal to check.

compat

bitmask of compatible features

ro

bitmask of features that force read-only mount

incompat

bitmask of incompatible features

Description

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

journal_check_available_features

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_check_available_features` — Check feature set in journalling layer

Synopsis

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

Arguments

journal

Journal to check.

compat

bitmask of compatible features

ro

bitmask of features that force read-only mount

incompat

bitmask of incompatible features

Description

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

journal_set_features

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

journal

Journal to act on.

compat

bitmask of compatible features

ro

bitmask of features that force read-only mount

incompat

bitmask of incompatible features

Description

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

journal_update_format

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_update_format` — Update on-disk journal structure.

Synopsis

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

Arguments

journal

Journal to act on.

Description

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

journal_flush

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_flush` — Flush journal

Synopsis

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

Arguments

journal

Journal to act on.

Description

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

journal_wipe

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_wipe` — Wipe journal contents

Synopsis

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

Arguments

journal

Journal to act on.

write

flag (see below)

Description

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

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

journal_abort

LINUX

Kernel Hackers Manual September 2014

Name

`journal_abort` — Shutdown the journal immediately.

Synopsis

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

Arguments

journal

the journal to shutdown.

errno

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

Description

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

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

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

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

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

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

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

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

journal_errno

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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


Arguments

journal

journal to examine.

Description

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

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

journal_clear_err

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_clear_err` — clears the journal's error state

Synopsis

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

Arguments

journal

journal to act on.

Description

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

journal_ack_err

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_ack_err` — Ack journal err.

Synopsis

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

Arguments

journal

journal to act on.

Description

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

journal_recover

LINUX

Kernel Hackers Manual September 2014

Name

`journal_recover` — recovers a on-disk journal

Synopsis

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

Arguments

journal

the journal to recover

Description

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

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

journal_skip_recovery

LINUX

Name

`journal_skip_recovery` — Start journal and wipe exiting records

Synopsis

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

Arguments

journal

journal to startup

Description

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

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

6.3.2. Transasction Level

`journal_start`

LINUX

Name

`journal_start` — Obtain a new handle.

Synopsis

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

Arguments

journal

Journal to start transaction on.

nblocks

number of block buffer we might modify

Description

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

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

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

`journal_extend`

LINUX

Name

`journal_extend` — extend buffer credits.

Synopsis

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

Arguments

handle

handle to 'extend'

nblocks

nr blocks to try to extend by.

Description

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

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

Return 0 on success, non-zero on failure.

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

journal_restart

LINUX

Kernel Hackers Manual September 2014

Name

`journal_restart` — restart a handle.

Synopsis

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

Arguments

handle

handle to restart

nblocks

nr credits requested

Description

Restart a handle for a multi-transaction filesystem operation.

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

journal_lock_updates

LINUX

Kernel Hackers ManualSeptember 2014

Name

`journal_lock_updates` — establish a transaction barrier.

Synopsis

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

Arguments

journal

Journal to establish a barrier on.

Description

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

We do not use simple mutex for synchronization as there are syscalls which want to return with filesystem locked and that trips up lockdep. Also hibernate needs to lock filesystem but locked mutex then blocks hibernation. Since locking filesystem is rare operation, we use simple counter and waitqueue for locking.

journal_unlock_updates

LINUX

Kernel Hackers Manual September 2014

Name

`journal_unlock_updates` — release barrier

Synopsis

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

Arguments

journal

Journal to release the barrier on.

Description

Release a transaction barrier obtained with `journal_lock_updates`.

journal_get_write_access

LINUX

Name

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

Synopsis

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

Arguments

handle

transaction to add buffer modifications to

bh

bh to be used for metadata writes

Description

Returns an error code or 0 on success.

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

journal_get_create_access

LINUX

Name

`journal_get_create_access` — notify intent to use newly created bh

Synopsis

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

Arguments

handle

transaction to new buffer to

bh

new buffer.

Description

Call this if you create a new bh.

journal_get_undo_access

LINUX

Name

`journal_get_undo_access` — Notify intent to modify metadata with

non-rewindable consequences

Synopsis

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

Arguments

handle

transaction

bh

buffer to undo

Description

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

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

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

Returns error number or 0 on success.

journal_dirty_data

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

handle

transaction

bh

bufferhead to mark

Description

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

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

Returns error number or 0 on success.

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

journal_dirty_metadata

LINUX

Kernel Hackers Manual September 2014

Name

`journal_dirty_metadata` — mark a buffer as containing dirty metadata

Synopsis

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

Arguments

handle

transaction to add buffer to.

bh

buffer to mark

Description

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

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

Returns error number or 0 on success.

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

buffer

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

journal_forget

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

handle

transaction handle

bh

bh to 'forget'

Description

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

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

Decrements `bh->b_count` by one.

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

journal_stop

LINUX

Kernel Hackers Manual September 2014

Name

`journal_stop` — complete a transaction

Synopsis

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

Arguments

handle

transaction to complete.

Description

All done for a particular handle.

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

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

journal_force_commit

LINUX

Kernel Hackers Manual September 2014

Name

`journal_force_commit` — force any uncommitted transactions

Synopsis

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

Arguments

journal

journal to force

For synchronous operations

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

journal_try_to_free_buffers

LINUX

Name

`journal_try_to_free_buffers` — try to free page buffers.

Synopsis

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

Arguments

journal

journal for operation

page

to try and free

gfp_mask

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

Description

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

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

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

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

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

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

Return 0 on failure, 1 on success

journal_invalidatepage

LINUX

Kernel Hackers Manual September 2014

Name

`journal_invalidatepage` — invalidate a journal page

Synopsis

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

Arguments

journal

journal to use for flush

page

page to flush

offset

offset of the range to invalidate

length

length of the range to invalidate

Description

Reap page buffers containing data in specified range in page.

6.4. See also

[Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen Tweedie
(<http://kernel.org/pub/linux/kernel/people/sct/ext3/journal-design.ps.gz>)]

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

Chapter 7. splice API

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

splice_to_pipe

LINUX

Kernel Hackers ManualSeptember 2014

Name

`splice_to_pipe` — fill passed data into a pipe

Synopsis

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

Arguments

pipe

pipe to fill

spd

data to fill

Description

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

generic_file_splice_read

LINUX

Kernel Hackers ManualSeptember 2014

Name

`generic_file_splice_read` — splice data from file to a pipe

Synopsis

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

Arguments

in

file to splice from

ppos

position in *in*

pipe

pipe to splice to

len

number of bytes to splice

flags

splice modifier flags

Description

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

splice_from_pipe_feed

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

pipe

pipe to splice from

sd

information to *actor*

actor

handler that splices the data

Description

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

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

splice_from_pipe_next

LINUX

Kernel Hackers Manual September 2014

Name

splice_from_pipe_next — wait for some data to splice from

Synopsis

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

Arguments

pipe

pipe to splice from

sd

information about the splice operation

Description

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

splice_from_pipe_begin

LINUX

Kernel Hackers ManualSeptember 2014

Name

`splice_from_pipe_begin` — start splicing from pipe

Synopsis

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

Arguments

sd

information about the splice operation

Description

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

splice_from_pipe_end

LINUX

Kernel Hackers ManualSeptember 2014

Name

`splice_from_pipe_end` — finish splicing from pipe

Synopsis

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

Arguments

pipe

pipe to splice from

sd

information about the splice operation

Description

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

__splice_from_pipe

LINUX

Name

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

Synopsis

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

Arguments

pipe

pipe to splice from

sd

information to *actor*

actor

handler that splices the data

Description

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

splice_from_pipe

LINUX

Name

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

Synopsis

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

Arguments

pipe

pipe to splice from

out

file to splice to

ppos

position in *out*

len

how many bytes to splice

flags

splice modifier flags

actor

handler that splices the data

Description

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

iter_file_splice_write

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

pipe

pipe info

out

file to write to

ppos

position in *out*

len

number of bytes to splice

flags

splice modifier flags

Description

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

generic_splice_sendpage

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

pipe

pipe to splice from

out

socket to write to

ppos

position in *out*

len

number of bytes to splice

flags

splice modifier flags

Description

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

splice_direct_to_actor

LINUX

Kernel Hackers ManualSeptember 2014

Name

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

Synopsis

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

Arguments

in

file to splice from

sd

actor information on where to splice to

actor

handles the data splicing

Description

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

do_splice_direct

LINUX

Kernel Hackers ManualSeptember 2014

Name

`do_splice_direct` — splices data directly between two files

Synopsis

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

Arguments

in

file to splice from

ppos

input file offset

out

file to splice to

opos

output file offset

len

number of bytes to splice

flags

splice modifier flags

Description

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

Chapter 8. pipes API

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

struct pipe_buffer

LINUX

Kernel Hackers ManualSeptember 2014

Name

struct pipe_buffer — a linux kernel pipe buffer

Synopsis

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

Members

page

the page containing the data for the pipe buffer

offset

offset of data inside the *page*

len

length of data inside the *page*

ops

operations associated with this buffer. See *pipe_buf_operations*.

flags

pipe buffer flags. See above.

private

private data owned by the ops.

struct pipe_inode_info

LINUX

Kernel Hackers Manual September 2014

Name

struct pipe_inode_info — a linux kernel pipe

Synopsis

```
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;
};
```

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

generic_pipe_buf_steal

LINUX

Kernel Hackers Manual September 2014

Name

`generic_pipe_buf_steal` — attempt to take ownership of a `pipe_buffer`

Synopsis

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

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer to attempt to steal

Description

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

generic_pipe_buf_get

LINUX

Kernel Hackers ManualSeptember 2014

Name

`generic_pipe_buf_get` — get a reference to a struct `pipe_buffer`

Synopsis

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

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer to get a reference to

Description

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

generic_pipe_buf_confirm

LINUX

Kernel Hackers ManualSeptember 2014

Name

`generic_pipe_buf_confirm` — verify contents of the pipe buffer

Synopsis

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

Arguments

info

the pipe that the buffer belongs to

buf

the buffer to confirm

Description

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

generic_pipe_buf_release

LINUX

Kernel Hackers ManualSeptember 2014

Name

`generic_pipe_buf_release` — put a reference to a struct `pipe_buffer`

Synopsis

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

Arguments

pipe

the pipe that the buffer belongs to

buf

the buffer to put a reference to

Description

This function releases a reference to *buf*.

