
Linux Filesystems Documentation

The kernel development community

Jul 14, 2020

CONTENTS

This under-development manual will, some glorious day, provide comprehensive information on how the Linux virtual filesystem (VFS) layer works, along with the filesystems that sit below it. For now, what we have can be found below.

CORE VFS DOCUMENTATION

See these manuals for documentation about the VFS layer itself and how its algorithms work.

1.1 Overview of the Linux Virtual File System

Original author: Richard Gooch <rgooch@atnf.csiro.au>

- Copyright (C) 1999 Richard Gooch
- Copyright (C) 2005 Pekka Enberg

1.1.1 Introduction

The Virtual File System (also known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to coexist.

VFS system calls `open(2)`, `stat(2)`, `read(2)`, `write(2)`, `chmod(2)` and so on are called from a process context. Filesystem locking is described in the document `Documentation/filesystems/locking.rst`.

Directory Entry Cache (dcache)

The VFS implements the `open(2)`, `stat(2)`, `chmod(2)`, and similar system calls. The pathname argument that is passed to them is used by the VFS to search through the directory entry cache (also known as the dentry cache or dcache). This provides a very fast look-up mechanism to translate a pathname (filename) into a specific dentry. Dentries live in RAM and are never saved to disc: they exist only for performance.

The dentry cache is meant to be a view into your entire filesystem. As most computers cannot fit all dentries in the RAM at the same time, some bits of the cache are missing. In order to resolve your pathname into a dentry, the VFS may have to resort to creating dentries along the way, and then loading the inode. This is done by looking up the inode.

The Inode Object

An individual dentry usually has a pointer to an inode. Inodes are filesystem objects such as regular files, directories, FIFOs and other beasts. They live either on the disc (for block device filesystems) or in the memory (for pseudo filesystems). Inodes that live on the disc are copied into the memory when required and changes to the inode are written back to disc. A single inode can be pointed to by multiple dentries (hard links, for example, do this).

To look up an inode requires that the VFS calls the `lookup()` method of the parent directory inode. This method is installed by the specific filesystem implementation that the inode lives in. Once the VFS has the required dentry (and hence the inode), we can do all those boring things like `open(2)` the file, or `stat(2)` it to peek at the inode data. The `stat(2)` operation is fairly simple: once the VFS has the dentry, it peeks at the inode data and passes some of it back to userspace.

The File Object

Opening a file requires another operation: allocation of a file structure (this is the kernel-side implementation of file descriptors). The freshly allocated file structure is initialized with a pointer to the dentry and a set of file operation member functions. These are taken from the inode data. The `open()` file method is then called so the specific filesystem implementation can do its work. You can see that this is another switch performed by the VFS. The file structure is placed into the file descriptor table for the process.

Reading, writing and closing files (and other assorted VFS operations) is done by using the userspace file descriptor to grab the appropriate file structure, and then calling the required file structure method to do whatever is required. For as long as the file is open, it keeps the dentry in use, which in turn means that the VFS inode is still in use.

1.1.2 Registering and Mounting a Filesystem

To register and unregister a filesystem, use the following API functions:

```
#include <linux/fs.h>

extern int register_filesystem(struct file_system_type *);
extern int unregister_filesystem(struct file_system_type *);
```

The passed `struct file_system_type` describes your filesystem. When a request is made to mount a filesystem onto a directory in your namespace, the VFS will call the appropriate `mount()` method for the specific filesystem. New `vfsmount` referring to the tree returned by `->mount()` will be attached to the mountpoint, so that when pathname resolution reaches the mountpoint it will jump into the root of that `vfsmount`.

You can see all filesystems that are registered to the kernel in the file `/proc/filesystems`.

struct file_system_type

This describes the filesystem. As of kernel 2.6.39, the following members are defined:

```

struct file_system_operations {
    const char *name;
    int fs_flags;
    struct dentry *(*mount) (struct file_system_type *, int,
                            const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
};

```

name the name of the filesystem type, such as “ext2” , “iso9660” , “msdos” and so on

fs_flags various flags (i.e. FS_REQUIRES_DEV, FS_NO_DCACHE, etc.)

mount the method to call when a new instance of this filesystem should be mounted

kill_sb the method to call when an instance of this filesystem should be shut down

owner for internal VFS use: you should initialize this to THIS_MODULE in most cases.

next

for internal VFS use: you should initialize this to NULL

s_lock_key, s_umount_key: lockdep-specific

The mount() method has the following arguments:

struct file_system_type *fs_type describes the filesystem, partly initialized by the specific filesystem code

int flags mount flags

const char *dev_name the device name we are mounting.

void *data arbitrary mount options, usually comes as an ASCII string (see “Mount Options” section)

The mount() method must return the root dentry of the tree requested by caller. An active reference to its superblock must be grabbed and the superblock must be locked. On failure it should return ERR_PTR(error).

The arguments match those of mount(2) and their interpretation depends on filesystem type. E.g. for block filesystems, dev_name is interpreted as block device name, that device is opened and if it contains a suitable filesystem image the method creates and initializes struct super_block accordingly, returning its root dentry to caller.

->mount() may choose to return a subtree of existing filesystem - it doesn't have to create a new one. The main result from the caller's point of view is a reference to dentry at the root of (sub)tree to be attached; creation of new superblock is a common side effect.

The most interesting member of the superblock structure that the mount() method fills in is the "s_op" field. This is a pointer to a "struct super_operations" which describes the next level of the filesystem implementation.

Usually, a filesystem uses one of the generic mount() implementations and provides a fill_super() callback instead. The generic variants are:

mount_bdev mount a filesystem residing on a block device

mount_nodev mount a filesystem that is not backed by a device

mount_single mount a filesystem which shares the instance between all mounts

A fill_super() callback implementation has the following arguments:

struct super_block *sb the superblock structure. The callback must initialize this properly.

void *data arbitrary mount options, usually comes as an ASCII string (see "Mount Options" section)

int silent whether or not to be silent on error

1.1.3 The Superblock Object

A superblock object represents a mounted filesystem.

struct super_operations

This describes how the VFS can manipulate the superblock of your filesystem. As of kernel 2.6.22, the following members are defined:

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*dirty_inode) (struct inode *, int flags);
    int (*write_inode) (struct inode *, int);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);

    int (*show_options)(struct seq_file *, struct dentry *);
};
```

(continues on next page)

(continued from previous page)

```

        ssize_t (*quota_read)(struct super_block *, int, char *, size_t,
↳loff_t);
        ssize_t (*quota_write)(struct super_block *, int, const char *,
↳size_t, loff_t);
        int (*nr_cached_objects)(struct super_block *);
        void (*free_cached_objects)(struct super_block *, int);
};

```

All methods are called without any locks being held, unless otherwise noted. This means that most methods can block safely. All methods are only called from a process context (i.e. not from an interrupt handler or bottom half).

alloc_inode this method is called by `alloc_inode()` to allocate memory for struct inode and initialize it. If this function is not defined, a simple 'struct inode' is allocated. Normally `alloc_inode` will be used to allocate a larger structure which contains a 'struct inode' embedded within it.

destroy_inode this method is called by `destroy_inode()` to release resources allocated for struct inode. It is only required if `->alloc_inode` was defined and simply undoes anything done by `->alloc_inode`.

dirty_inode this method is called by the VFS to mark an inode dirty.

write_inode this method is called when the VFS needs to write an inode to disc. The second parameter indicates whether the write should be synchronous or not, not all filesystems check this flag.

drop_inode called when the last access to the inode is dropped, with the `inode->i_lock` spinlock held.

This method should be either NULL (normal UNIX filesystem semantics) or "generic_delete_inode" (for filesystems that do not want to cache inodes - causing "delete_inode" to always be called regardless of the value of `i_nlink`)

The "generic_delete_inode()" behavior is equivalent to the old practice of using "force_delete" in the `put_inode()` case, but does not have the races that the "force_delete()" approach had.

delete_inode called when the VFS wants to delete an inode

put_super called when the VFS wishes to free the superblock (i.e. unmount). This is called with the superblock lock held

sync_fs called when VFS is writing out all dirty data associated with a superblock. The second parameter indicates whether the method should wait until the write out has been completed. Optional.

freeze_fs called when VFS is locking a filesystem and forcing it into a consistent state. This method is currently used by the Logical Volume Manager (LVM).

unfreeze_fs called when VFS is unlocking a filesystem and making it writable again.

statfs called when the VFS needs to get filesystem statistics.

remount_fs called when the filesystem is remounted. This is called with the kernel lock held

clear_inode called then the VFS clears the inode. Optional

umount_begin called when the VFS is unmounting a filesystem.

show_options called by the VFS to show mount options for /proc/<pid>/mounts. (see “Mount Options” section)

quota_read called by the VFS to read from filesystem quota file.

quota_write called by the VFS to write to filesystem quota file.

nr_cached_objects called by the sb cache shrinking function for the filesystem to return the number of freeable cached objects it contains. Optional.

free_cache_objects called by the sb cache shrinking function for the filesystem to scan the number of objects indicated to try to free them. Optional, but any filesystem implementing this method needs to also implement `>nr_cached_objects` for it to be called correctly.

We can't do anything with any errors that the filesystem might encounter, hence the void return type. This will never be called if the VM is trying to reclaim under GFP_NOFS conditions, hence this method does not need to handle that situation itself.

Implementations must include conditional reschedule calls inside any scanning loop that is done. This allows the VFS to determine appropriate scan batch sizes without having to worry about whether implementations will cause holdoff problems due to large scan batch sizes.

Whoever sets up the inode is responsible for filling in the “i_op” field. This is a pointer to a “struct inode_operations” which describes the methods that can be performed on individual inodes.

struct xattr_handlers

On filesystems that support extended attributes (xattrs), the `s_xattr` superblock field points to a NULL-terminated array of xattr handlers. Extended attributes are name:value pairs.

name Indicates that the handler matches attributes with the specified name (such as “system.posix_acl_access”); the prefix field must be NULL.

prefix Indicates that the handler matches all attributes with the specified name prefix (such as “user.”); the name field must be NULL.

list Determine if attributes matching this xattr handler should be listed for a particular dentry. Used by some listxattr implementations like `generic_listxattr`.

get Called by the VFS to get the value of a particular extended attribute. This method is called by the `getxattr(2)` system call.

set Called by the VFS to set the value of a particular extended attribute. When the new value is NULL, called to remove a particular extended attribute. This method is called by the `setxattr(2)` and `removexattr(2)` system calls.

When none of the xattr handlers of a filesystem match the specified attribute name or when a filesystem doesn't support extended attributes, the various `*xattr(2)` system calls return `-EOPNOTSUPP`.

1.1.4 The Inode Object

An inode object represents an object within the filesystem.

struct inode_operations

This describes how the VFS can manipulate an inode in your filesystem. As of kernel 2.6.22, the following members are defined:

```

struct inode_operations {
    int (*create) (struct inode *, struct dentry *, umode_t, bool);
    struct dentry * (*lookup) (struct inode *, struct dentry *,
↳ unsigned int);
    int (*link) (struct dentry *, struct inode *, struct dentry *);
    int (*unlink) (struct inode *, struct dentry *);
    int (*symlink) (struct inode *, struct dentry *, const char *);
    int (*mkdir) (struct inode *, struct dentry *, umode_t);
    int (*rmdir) (struct inode *, struct dentry *);
    int (*mknod) (struct inode *, struct dentry *, umode_t, dev_t);
    int (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry *, unsigned int);
    int (*readlink) (struct dentry *, char __user *, int);
    const char * (*get_link) (struct dentry *, struct inode *,
                              struct delayed_call *);
    int (*permission) (struct inode *, int);
    int (*get_acl) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (const struct path *, struct kstat *, u32, unsigned_
↳ int);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    void (*update_time) (struct inode *, struct timespec *, int);
    int (*atomic_open) (struct inode *, struct dentry *, struct file *,
                        unsigned open_flag, umode_t create_mode);
    int (*tmpfile) (struct inode *, struct dentry *, umode_t);
};

```

Again, all methods are called without any locks being held, unless otherwise noted.

create called by the `open(2)` and `creat(2)` system calls. Only required if you want to support regular files. The dentry you get should not have an inode (i.e. it should be a negative dentry). Here you will probably call `d_instantiate()` with the dentry and the newly created inode

lookup called when the VFS needs to look up an inode in a parent directory. The name to look for is found in the dentry. This method must call `d_add()` to insert the found inode into the dentry. The “`i_count`” field in the inode structure should be incremented. If the named inode does not exist a NULL inode should be inserted into the dentry (this is called a negative dentry). Returning an error code from this routine must only be done on a real error, otherwise creating inodes with system calls like `create(2)`, `mknod(2)`, `mkdir(2)` and so on will fail. If you wish to overload the dentry methods then you should initialise the “`d_dop`” field in the dentry; this is a pointer to a struct “`dentry_operations`”. This method is called with the directory inode semaphore held

link called by the `link(2)` system call. Only required if you want to support hard

links. You will probably need to call `d_instantiate()` just as you would in the `create()` method

unlink called by the `unlink(2)` system call. Only required if you want to support deleting inodes

symlink called by the `symlink(2)` system call. Only required if you want to support symlinks. You will probably need to call `d_instantiate()` just as you would in the `create()` method

mkdir called by the `mkdir(2)` system call. Only required if you want to support creating subdirectories. You will probably need to call `d_instantiate()` just as you would in the `create()` method

rmdir called by the `rmdir(2)` system call. Only required if you want to support deleting subdirectories

mknod called by the `mknod(2)` system call to create a device (char, block) inode or a named pipe (FIFO) or socket. Only required if you want to support creating these types of inodes. You will probably need to call `d_instantiate()` just as you would in the `create()` method

rename called by the `rename(2)` system call to rename the object to have the parent and name given by the second inode and dentry.

The filesystem must return `-EINVAL` for any unsupported or unknown flags. Currently the following flags are implemented: (1) `RENAME_NOREPLACE`: this flag indicates that if the target of the rename exists the rename should fail with `-EEXIST` instead of replacing the target. The VFS already checks for existence, so for local filesystems the `RENAME_NOREPLACE` implementation is equivalent to plain `rename`. (2) `RENAME_EXCHANGE`: exchange source and target. Both must exist; this is checked by the VFS. Unlike plain `rename`, source and target may be of different type.

get_link called by the VFS to follow a symbolic link to the inode it points to. Only required if you want to support symbolic links. This method returns the symlink body to traverse (and possibly resets the current position with `nd_jump_link()`). If the body won't go away until the inode is gone, nothing else is needed; if it needs to be otherwise pinned, arrange for its release by having `get_link(..., ..., done)` do `set_delayed_call(done, destructor, argument)`. In that case `destructor(argument)` will be called once VFS is done with the body you've returned. May be called in RCU mode; that is indicated by `NULL` dentry argument. If request can't be handled without leaving RCU mode, have it return `ERR_PTR(-ECHILD)`.

If the filesystem stores the symlink target in `->i_link`, the VFS may use it directly without calling `->get_link()`; however, `->get_link()` must still be provided. `->i_link` must not be freed until after an RCU grace period. Writing to `->i_link` post-`iget()` time requires a 'release' memory barrier.

readlink this is now just an override for use by `readlink(2)` for the cases when `->get_link` uses `nd_jump_link()` or object is not in fact a symlink. Normally filesystems should only implement `->get_link` for symlinks and `readlink(2)` will automatically use that.

permission called by the VFS to check for access rights on a POSIX-like filesystem.

May be called in rcu-walk mode (mask & MAY_NOT_BLOCK). If in rcu-walk mode, the filesystem must check the permission without blocking or storing to the inode.

If a situation is encountered that rcu-walk cannot handle, return -ECHILD and it will be called again in ref-walk mode.

setattr called by the VFS to set attributes for a file. This method is called by chmod(2) and related system calls.

getattr called by the VFS to get attributes of a file. This method is called by stat(2) and related system calls.

listxattr called by the VFS to list all extended attributes for a given file. This method is called by the listxattr(2) system call.

update_time called by the VFS to update a specific time or the i_version of an inode. If this is not defined the VFS will update the inode itself and call mark_inode_dirty_sync.

atomic_open called on the last component of an open. Using this optional method the filesystem can look up, possibly create and open the file in one atomic operation. If it wants to leave actual opening to the caller (e.g. if the file turned out to be a symlink, device, or just something filesystem won't do atomic open for), it may signal this by returning finish_no_open(file, dentry). This method is only called if the last component is negative or needs lookup. Cached positive dentries are still handled by f_op->open(). If the file was created, FMODE_CREATED flag should be set in file->f_mode. In case of O_EXCL the method must only succeed if the file didn't exist and hence FMODE_CREATED shall always be set on success.

tmpfile called in the end of O_TMPFILE open(). Optional, equivalent to atomically creating, opening and unlinking a file in given directory.

1.1.5 The Address Space Object

The address space object is used to group and manage pages in the page cache. It can be used to keep track of the pages in a file (or anything else) and also track the mapping of sections of the file into process address spaces.

There are a number of distinct yet related services that an address-space can provide. These include communicating memory pressure, page lookup by address, and keeping track of pages tagged as Dirty or Writeback.

The first can be used independently to the others. The VM can try to either write dirty pages in order to clean them, or release clean pages in order to reuse them. To do this it can call the ->writepage method on dirty pages, and ->releasepage on clean pages with PagePrivate set. Clean pages without PagePrivate and with no external references will be released without notice being given to the address_space.

To achieve this functionality, pages need to be placed on an LRU with lru_cache_add and mark_page_active needs to be called whenever the page is used.

Pages are normally kept in a radix tree index by ->index. This tree maintains information about the PG_Dirty and PG_Writeback status of each page, so that

pages with either of these flags can be found quickly.

The Dirty tag is primarily used by `mpage_writepages` - the default `->writepages` method. It uses the tag to find dirty pages to call `->writepage` on. If `mpage_writepages` is not used (i.e. the address provides its own `->writepages`), the `PAGECACHE_TAG_DIRTY` tag is almost unused. `write_inode_now` and `sync_inode` do use it (through `__sync_single_inode`) to check if `->writepages` has been successful in writing out the whole `address_space`.

The Writeback tag is used by `filemap*wait*` and `sync_page*` functions, via `filemap_fdatawait_range`, to wait for all writeback to complete.

An `address_space` handler may attach extra information to a page, typically using the 'private' field in the 'struct page'. If such information is attached, the `PG_Private` flag should be set. This will cause various VM routines to make extra calls into the `address_space` handler to deal with that data.

An address space acts as an intermediate between storage and application. Data is read into the address space a whole page at a time, and provided to the application either by copying of the page, or by memory-mapping the page. Data is written into the address space by the application, and then written-back to storage typically in whole pages, however the `address_space` has finer control of write sizes.

The read process essentially only requires 'readpage'. The write process is more complicated and uses `write_begin/write_end` or `set_page_dirty` to write data into the `address_space`, and `writepage` and `writepages` to writeback data to storage.

Adding and removing pages to/from an `address_space` is protected by the `inode's i_mutex`.

When data is written to a page, the `PG_Dirty` flag should be set. It typically remains set until `writepage` asks for it to be written. This should clear `PG_Dirty` and set `PG_Writeback`. It can be actually written at any point after `PG_Dirty` is clear. Once it is known to be safe, `PG_Writeback` is cleared.

Writeback makes use of a `writeback_control` structure to direct the operations. This gives the `writepage` and `writepages` operations some information about the nature of and reason for the writeback request, and the constraints under which it is being done. It is also used to return information back to the caller about the result of a `writepage` or `writepages` request.

Handling errors during writeback

Most applications that do buffered I/O will periodically call a file synchronization call (`fsync`, `fdatasync`, `msync` or `sync_file_range`) to ensure that data written has made it to the backing store. When there is an error during writeback, they expect that error to be reported when a file sync request is made. After an error has been reported on one request, subsequent requests on the same file descriptor should return 0, unless further writeback errors have occurred since the previous file synchronization.

Ideally, the kernel would report errors only on file descriptions on which writes were done that subsequently failed to be written back. The generic pagecache infrastructure does not track the file descriptions that have dirtied each individual

page however, so determining which file descriptors should get back an error is not possible.

Instead, the generic writeback error tracking infrastructure in the kernel settles for reporting errors to fsync on all file descriptions that were open at the time that the error occurred. In a situation with multiple writers, all of them will get back an error on a subsequent fsync, even if all of the writes done through that particular file descriptor succeeded (or even if there were no writes on that file descriptor at all).

Filesystems that wish to use this infrastructure should call `mapping_set_error` to record the error in the `address_space` when it occurs. Then, after writing back data from the pagecache in their `file->fsync` operation, they should call `file_check_and_advance_wb_err` to ensure that the struct file's error cursor has advanced to the correct point in the stream of errors emitted by the backing device(s).

struct address_space_operations

This describes how the VFS can manipulate mapping of a file to page cache in your filesystem. The following members are defined:

```

struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*writepages)(struct address_space *, struct writeback_control
↳*);
    int (*set_page_dirty)(struct page *page);
    void (*readahead)(struct readahead_control *);
    int (*readpages)(struct file *filp, struct address_space *mapping,
                    struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping,
                    loff_t pos, unsigned len, unsigned flags,
                    struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
                    loff_t pos, unsigned len, unsigned copied,
                    struct page *page, void *fsdata);
    sector_t (*bmap)(struct address_space *, sector_t);
    void (*invalidatepage) (struct page *, unsigned int, unsigned int);
    int (*releasepage) (struct page *, int);
    void (*freepage)(struct page *);
    ssize_t (*direct_IO)(struct kiocb *, struct iov_iter *iter);
    /* isolate a page for migration */
    bool (*isolate_page) (struct page *, isolate_mode_t);
    /* migrate the contents of a page to the specified target */
    int (*migratepage) (struct page *, struct page *);
    /* put migration-failed page back to right list */
    void (*putback_page) (struct page *);
    int (*launder_page) (struct page *);

    int (*is_partially_uptodate) (struct page *, unsigned long,
                                unsigned long);
    void (*is_dirty_writeback) (struct page *, bool *, bool *);
    int (*error_remove_page) (struct mapping *mapping, struct page
↳*page);

```

(continues on next page)

(continued from previous page)

```
int (*swap_activate)(struct file *);
int (*swap_deactivate)(struct file *);
};
```

writepage called by the VM to write a dirty page to backing store. This may happen for data integrity reasons (i.e. 'sync'), or to free up memory (flush). The difference can be seen in `wbc->sync_mode`. The `PG_Dirty` flag has been cleared and `PageLocked` is true. `writepage` should start writeout, should set `PG_Writeback`, and should make sure the page is unlocked, either synchronously or asynchronously when the write operation completes.

If `wbc->sync_mode` is `WB_SYNC_NONE`, `->writepage` doesn't have to try too hard if there are problems, and may choose to write out other pages from the mapping if that is easier (e.g. due to internal dependencies). If it chooses not to start writeout, it should return `AOP_WRITEPAGE_ACTIVATE` so that the VM will not keep calling `->writepage` on that page.

See the file "Locking" for more details.

readpage called by the VM to read a page from backing store. The page will be Locked when `readpage` is called, and should be unlocked and marked uptodate once the read completes. If `->readpage` discovers that it needs to unlock the page for some reason, it can do so, and then return `AOP_TRUNCATED_PAGE`. In this case, the page will be relocated, relocked and if that all succeeds, `->readpage` will be called again.

writepages called by the VM to write out pages associated with the `address_space` object. If `wbc->sync_mode` is `WBC_SYNC_ALL`, then the `writeback_control` will specify a range of pages that must be written out. If it is `WBC_SYNC_NONE`, then a `nr_to_write` is given and that many pages should be written if possible. If no `->writepages` is given, then `mpage_writepages` is used instead. This will choose pages from the address space that are tagged as `DIRTY` and will pass them to `->writepage`.

set_page_dirty called by the VM to set a page dirty. This is particularly needed if an address space attaches private data to a page, and that data needs to be updated when a page is dirtied. This is called, for example, when a memory mapped page gets modified. If defined, it should set the `PageDirty` flag, and the `PAGECACHE_TAG_DIRTY` tag in the radix tree.

readahead Called by the VM to read pages associated with the `address_space` object. The pages are consecutive in the page cache and are locked. The implementation should decrement the page refcount after starting I/O on each page. Usually the page will be unlocked by the I/O completion handler. If the filesystem decides to stop attempting I/O before reaching the end of the readahead window, it can simply return. The caller will decrement the page refcount and unlock the remaining pages for you. Set `PageUptodate` if the I/O completes successfully. Setting `PageError` on any page will be ignored; simply unlock the page if an I/O error occurs.

readpages called by the VM to read pages associated with the `address_space` object. This is essentially just a vector version of `readpage`. Instead of just one page, several pages are requested. `readpages` is only used for read-ahead, so read errors are ignored. If anything goes wrong, feel free to give up. This

interface is deprecated and will be removed by the end of 2020; implement readahead instead.

write_begin Called by the generic buffered write code to ask the filesystem to prepare to write len bytes at the given offset in the file. The address_space should check that the write will be able to complete, by allocating space if necessary and doing any other internal housekeeping. If the write will update parts of any basic-blocks on storage, then those blocks should be pre-read (if they haven't been read already) so that the updated blocks can be written out properly.

The filesystem must return the locked pagecache page for the specified offset, in *pagep, for the caller to write into.

It must be able to cope with short writes (where the length passed to write_begin is greater than the number of bytes copied into the page).

flags is a field for AOP_FLAG_XXX flags, described in include/linux/fs.h.

A void * may be returned in fsdata, which then gets passed into write_end.

Returns 0 on success; < 0 on failure (which is the error code), in which case write_end is not called.

write_end After a successful write_begin, and data copy, write_end must be called. len is the original len passed to write_begin, and copied is the amount that was able to be copied.

The filesystem must take care of unlocking the page and releasing its refcount, and updating i_size.

Returns < 0 on failure, otherwise the number of bytes (<= 'copied') that were able to be copied into pagecache.

bmap called by the VFS to map a logical block offset within object to physical block number. This method is used by the FIBMAP ioctl and for working with swap-files. To be able to swap to a file, the file must have a stable mapping to a block device. The swap system does not go through the filesystem but instead uses bmap to find out where the blocks in the file are and uses those addresses directly.

invalidatepage If a page has PagePrivate set, then invalidatepage will be called when part or all of the page is to be removed from the address space. This generally corresponds to either a truncation, punch hole or a complete invalidation of the address space (in the latter case 'offset' will always be 0 and 'length' will be PAGE_SIZE). Any private data associated with the page should be updated to reflect this truncation. If offset is 0 and length is PAGE_SIZE, then the private data should be released, because the page must be able to be completely discarded. This may be done by calling the ->releasepage function, but in this case the release MUST succeed.

releasepage releasepage is called on PagePrivate pages to indicate that the page should be freed if possible. ->releasepage should remove any private data from the page and clear the PagePrivate flag. If releasepage() fails for some reason, it must indicate failure with a 0 return value. releasepage() is used in two distinct though related cases. The first is when the VM finds a clean

page with no active users and wants to make it a free page. If `->releasepage` succeeds, the page will be removed from the `address_space` and become free.

The second case is when a request has been made to invalidate some or all pages in an `address_space`. This can happen through the `fdadvise(POSIX_FADV_DONTNEED)` system call or by the filesystem explicitly requesting it as `nfs` and `9fs` do (when they believe the cache may be out of date with storage) by calling `invalidate_inode_pages2()`. If the filesystem makes such a call, and needs to be certain that all pages are invalidated, then its `releasepage` will need to ensure this. Possibly it can clear the `PageUptodate` bit if it cannot free private data yet.

freepage `freepage` is called once the page is no longer visible in the page cache in order to allow the cleanup of any private data. Since it may be called by the memory reclaimer, it should not assume that the original `address_space` mapping still exists, and it should not block.

direct_IO called by the generic read/write routines to perform `direct_IO` - that is IO requests which bypass the page cache and transfer data directly between the storage and the application's address space.

isolate_page Called by the VM when isolating a movable non-lru page. If page is successfully isolated, VM marks the page as `PG_isolated` via `__SetPageIsolated`.

migrate_page This is used to compact the physical memory usage. If the VM wants to relocate a page (maybe off a memory card that is signalling imminent failure) it will pass a new page and an old page to this function. `migrate_page` should transfer any private data across and update any references that it has to the page.

putback_page Called by the VM when isolated page's migration fails.

launder_page Called before freeing a page - it writes back the dirty page. To prevent redirtying the page, it is kept locked during the whole operation.

is_partially_uptodate Called by the VM when reading a file through the page-cache when the underlying blocksize \neq pagesize. If the required block is up to date then the read can complete without needing the IO to bring the whole page up to date.

is_dirty_writeback Called by the VM when attempting to reclaim a page. The VM uses dirty and writeback information to determine if it needs to stall to allow flushers a chance to complete some IO. Ordinarily it can use `PageDirty` and `PageWriteback` but some filesystems have more complex state (unstable pages in NFS prevent reclaim) or do not set those flags due to locking problems. This callback allows a filesystem to indicate to the VM if a page should be treated as dirty or writeback for the purposes of stalling.

error_remove_page normally set to `generic_error_remove_page` if truncation is ok for this address space. Used for memory failure handling. Setting this implies you deal with pages going away under you, unless you have them locked or reference counts increased.

swap_activate Called when `swapon` is used on a file to allocate space if necessary and pin the block lookup information in memory. A return value of zero indicates success, in which case this file can be used to back swapspace.

swap_deactivate Called during swapon on files where swap_activate was successful.

1.1.6 The File Object

A file object represents a file opened by a process. This is also known as an “open file description” in POSIX parlance.

struct file_operations

This describes how the VFS can manipulate an open file. As of kernel 4.18, the following members are defined:

```

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t
↳ t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll) (struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
↳ long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
↳ loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long,
↳ unsigned long, unsigned long, unsigned long);
    int (*check_flags) (int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write) (struct pipe_inode_info *, struct file *,
↳ loff_t *, size_t, unsigned int);
    ssize_t (*splice_read) (struct file *, loff_t *, struct pipe_inode_
↳ info *, size_t, unsigned int);
    int (*setlease) (struct file *, long, struct file_lock **, void **);
    long (*fallocate) (struct file *file, int mode, loff_t offset,
        loff_t len);
    void (*show_fdinfo) (struct seq_file *m, struct file *f);
#ifdef CONFIG_MMU
    unsigned (*mmap_capabilities) (struct file *);
#endif
    ssize_t (*copy_file_range) (struct file *, loff_t, struct file *,
↳ loff_t, size_t, unsigned int);

```

(continues on next page)

(continued from previous page)

```
loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                          struct file *file_out, loff_t pos_out,
                          loff_t len, unsigned int remap_flags);
int (*fadvise)(struct file *, loff_t, loff_t, int);
};
```

Again, all methods are called without any locks being held, unless otherwise noted.

llseek called when the VFS needs to move the file position index

read called by read(2) and related system calls

read_iter possibly asynchronous read with iov_iter as destination

write called by write(2) and related system calls

write_iter possibly asynchronous write with iov_iter as source

iopoll called when aio wants to poll for completions on HIPRI iocbs

iterate called when the VFS needs to read the directory contents

iterate_shared called when the VFS needs to read the directory contents when filesystem supports concurrent dir iterators

poll called by the VFS when a process wants to check if there is activity on this file and (optionally) go to sleep until there is activity. Called by the select(2) and poll(2) system calls

unlocked_ioctl called by the ioctl(2) system call.

compat_ioctl

called by the **ioctl(2)** system call when **32 bit system calls** are used on 64 bit kernels.

mmap called by the mmap(2) system call

open called by the VFS when an inode should be opened. When the VFS opens a file, it creates a new “struct file”. It then calls the open method for the newly allocated file structure. You might think that the open method really belongs in “struct inode_operations”, and you may be right. I think it’s done the way it is because it makes filesystems simpler to implement. The open() method is a good place to initialize the “private_data” member in the file structure if you want to point to a device structure

flush called by the close(2) system call to flush a file

release called when the last reference to an open file is closed

fsync called by the fsync(2) system call. Also see the section above entitled “Handling errors during writeback” .

fasync called by the fcntl(2) system call when asynchronous (non-blocking) mode is enabled for a file

lock called by the fcntl(2) system call for F_GETLK, F_SETLK, and F_SETLKW commands

get_unmapped_area called by the mmap(2) system call

check_flags called by the `fcntl(2)` system call for `F_SETFL` command

flock called by the `flock(2)` system call

splice_write called by the VFS to splice data from a pipe to a file. This method is used by the `splice(2)` system call

splice_read called by the VFS to splice data from file to a pipe. This method is used by the `splice(2)` system call

setlease called by the VFS to set or release a file lock lease. `setlease` implementations should call `generic_setlease` to record or remove the lease in the inode after setting it.

fallocate called by the VFS to preallocate blocks or punch a hole.

copy_file_range called by the `copy_file_range(2)` system call.

remap_file_range called by the `ioctl(2)` system call for `FICLONERANGE` and `FICLONE` and `FIDEDUPERANGE` commands to remap file ranges. An implementation should remap `len` bytes at `pos_in` of the source file into the dest file at `pos_out`. Implementations must handle callers passing in `len == 0`; this means “remap to the end of the source file”. The return value should be the number of bytes remapped, or the usual negative error code if errors occurred before any bytes were remapped. The `remap_flags` parameter accepts `REMAP_FILE_*` flags. If `REMAP_FILE_DEDUP` is set then the implementation must only remap if the requested file ranges have identical contents. If `REMAP_CAN_SHORTEN` is set, the caller is ok with the implementation shortening the request length to satisfy alignment or EOF requirements (or any other reason).

advise possibly called by the `advise64()` system call.

Note that the file operations are implemented by the specific filesystem in which the inode resides. When opening a device node (character or block special) most filesystems will call special support routines in the VFS which will locate the required device driver information. These support routines replace the filesystem file operations with those for the device driver, and then proceed to call the new `open()` method for the file. This is how opening a device file in the filesystem eventually ends up calling the device driver `open()` method.

1.1.7 Directory Entry Cache (dcache)

struct dentry_operations

This describes how a filesystem can overload the standard dentry operations. Dentries and the dcache are the domain of the VFS and the individual filesystem implementations. Device drivers have no business here. These methods may be set to `NULL`, as they are either optional or the VFS uses a default. As of kernel 2.6.22, the following members are defined:

```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, unsigned int);
    int (*d_weak_revalidate)(struct dentry *, unsigned int);
    int (*d_hash)(const struct dentry *, struct qstr *);
```

(continues on next page)

(continued from previous page)

```
int (*d_compare)(const struct dentry *,
                unsigned int, const char *, const struct qstr *);
int (*d_delete)(const struct dentry *);
int (*d_init)(struct dentry *);
void (*d_release)(struct dentry *);
void (*d_iput)(struct dentry *, struct inode *);
char *(*d_dname)(struct dentry *, char *, int);
struct vfsmount *(*d_automount)(struct path *);
int (*d_manage)(const struct path *, bool);
struct dentry *(*d_real)(struct dentry *, const struct inode *);
};
```

d_revalidate called when the VFS needs to revalidate a dentry. This is called whenever a name look-up finds a dentry in the dcache. Most local filesystems leave this as NULL, because all their dentries in the dcache are valid. Network filesystems are different since things can change on the server without the client necessarily being aware of it.

This function should return a positive value if the dentry is still valid, and zero or a negative error code if it isn't.

d_revalidate may be called in rcu-walk mode (flags & LOOKUP_RCU). If in rcu-walk mode, the filesystem must revalidate the dentry without blocking or storing to the dentry, **d_parent** and **d_inode** should not be used without care (because they can change and, in **d_inode** case, even become NULL under us).

If a situation is encountered that rcu-walk cannot handle, return -ECHILD and it will be called again in ref-walk mode.

_weak_revalidate called when the VFS needs to revalidate a “jumped” dentry. This is called when a path-walk ends at dentry that was not acquired by doing a lookup in the parent directory. This includes “/” , “.” and “..” , as well as procfs-style symlinks and mountpoint traversal.

In this case, we are less concerned with whether the dentry is still fully correct, but rather that the inode is still valid. As with **d_revalidate**, most local filesystems will set this to NULL since their dcache entries are always valid.

This function has the same return code semantics as **d_revalidate**.

d_weak_revalidate is only called after leaving rcu-walk mode.

d_hash called when the VFS adds a dentry to the hash table. The first dentry passed to **d_hash** is the parent directory that the name is to be hashed into.

Same locking and synchronisation rules as **d_compare** regarding what is safe to dereference etc.

d_compare called to compare a dentry name with a given name. The first dentry is the parent of the dentry to be compared, the second is the child dentry. **len** and **name** string are properties of the dentry to be compared. **qstr** is the name to compare it with.

Must be constant and idempotent, and should not take locks if possible, and should not or store into the dentry. Should not dereference pointers outside

the dentry without lots of care (eg. `d_parent`, `d_inode`, `d_name` should not be used).

However, our `vfsmount` is pinned, and RCU held, so the dentries and inodes won't disappear, neither will our `sb` or filesystem module. `->d_sb` may be used.

It is a tricky calling convention because it needs to be called under "rcu-walk", ie. without any locks or references on things.

d_delete called when the last reference to a dentry is dropped and the dcache is deciding whether or not to cache it. Return 1 to delete immediately, or 0 to cache the dentry. Default is NULL which means to always cache a reachable dentry. `d_delete` must be constant and idempotent.

d_init called when a dentry is allocated

d_release called when a dentry is really deallocated

d_iput called when a dentry loses its inode (just prior to its being deallocated). The default when this is NULL is that the VFS calls `iput()`. If you define this method, you must call `iput()` yourself

d_dname called when the pathname of a dentry should be generated. Useful for some pseudo filesystems (`sockfs`, `pipefs`, ...) to delay pathname generation. (Instead of doing it when dentry is created, it's done only when the path is needed.). Real filesystems probably don't want to use it, because their dentries are present in global dcache hash, so their hash should be an invariant. As no lock is held, `d_dname()` should not try to modify the dentry itself, unless appropriate SMP safety is used. CAUTION : `d_path()` logic is quite tricky. The correct way to return for example "Hello" is to put it at the end of the buffer, and returns a pointer to the first char. `dynamic_dname()` helper function is provided to take care of this.

Example :

```
static char *pipefs_dname(struct dentry *dent, char *buffer, int buflen)
{
    return dynamic_dname(dentry, buffer, buflen, "pipe:[%lu]",
                        dentry->d_inode->i_ino);
}
```

d_automount called when an automount dentry is to be traversed (optional). This should create a new VFS mount record and return the record to the caller. The caller is supplied with a path parameter giving the automount directory to describe the automount target and the parent VFS mount record to provide inheritable mount parameters. NULL should be returned if someone else managed to make the automount first. If the `vfsmount` creation failed, then an error code should be returned. If `-EISDIR` is returned, then the directory will be treated as an ordinary directory and returned to `pathwalk` to continue walking.

If a `vfsmount` is returned, the caller will attempt to mount it on the mountpoint and will remove the `vfsmount` from its expiration list in the case of failure. The `vfsmount` should be returned with 2 refs on it to prevent automatic expiration - the caller will clean up the additional ref.

This function is only used if `DCACHE_NEED_AUTOMOUNT` is set on the dentry. This is set by `__d_instantiate()` if `S_AUTOMOUNT` is set on the inode being added.

d_manage called to allow the filesystem to manage the transition from a dentry (optional). This allows autofs, for example, to hold up clients waiting to explore behind a ‘mountpoint’ while letting the daemon go past and construct the subtree there. 0 should be returned to let the calling process continue. `-EISDIR` can be returned to tell pathwalk to use this directory as an ordinary directory and to ignore anything mounted on it and not to check the automount flag. Any other error code will abort pathwalk completely.

If the ‘`rcu_walk`’ parameter is true, then the caller is doing a pathwalk in RCU-walk mode. Sleeping is not permitted in this mode, and the caller can be asked to leave it and call again by returning `-ECHILD`. `-EISDIR` may also be returned to tell pathwalk to ignore `d_automount` or any mounts.

This function is only used if `DCACHE_MANAGE_TRANSIT` is set on the dentry being transited from.

d_real overlay/union type filesystems implement this method to return one of the underlying dentries hidden by the overlay. It is used in two different modes:

Called from `file_dentry()` it returns the real dentry matching the inode argument. The real dentry may be from a lower layer already copied up, but still referenced from the file. This mode is selected with a non-NULL inode argument.

With NULL inode the topmost real underlying dentry is returned.

Each dentry has a pointer to its parent dentry, as well as a hash list of child dentries. Child dentries are basically like files in a directory.

Directory Entry Cache API

There are a number of functions defined which permit a filesystem to manipulate dentries:

dget open a new handle for an existing dentry (this just increments the usage count)

dput close a handle for a dentry (decrements the usage count). If the usage count drops to 0, and the dentry is still in its parent’s hash, the “`d_delete`” method is called to check whether it should be cached. If it should not be cached, or if the dentry is not hashed, it is deleted. Otherwise cached dentries are put into an LRU list to be reclaimed on memory shortage.

d_drop this unhashes a dentry from its parents hash list. A subsequent call to `dput()` will deallocate the dentry if its usage count drops to 0

d_delete delete a dentry. If there are no other open references to the dentry then the dentry is turned into a negative dentry (the `d_iput()` method is called). If there are other references, then `d_drop()` is called instead

d_add add a dentry to its parents hash list and then calls `d_instantiate()`

d_instantiate add a dentry to the alias hash list for the inode and updates the “d_inode” member. The “i_count” member in the inode structure should be set/incremented. If the inode pointer is NULL, the dentry is called a “negative dentry” . This function is commonly called when an inode is created for an existing negative dentry

d_lookup look up a dentry given its parent and path name component It looks up the child of that given name from the dcache hash table. If it is found, the reference count is incremented and the dentry is returned. The caller must use dput() to free the dentry when it finishes using it.

1.1.8 Mount Options

Parsing options

On mount and remount the filesystem is passed a string containing a comma separated list of mount options. The options can have either of these forms:

option option=value

The <linux/parser.h> header defines an API that helps parse these options. There are plenty of examples on how to use it in existing filesystems.

Showing options

If a filesystem accepts mount options, it must define show_options() to show all the currently active options. The rules are:

- options **MUST** be shown which are not default or their values differ from the default
- options **MAY** be shown which are enabled by default or have their default value

Options used only internally between a mount helper and the kernel (such as file descriptors), or which only have an effect during the mounting (such as ones controlling the creation of a journal) are exempt from the above rules.

The underlying reason for the above rules is to make sure, that a mount can be accurately replicated (e.g. umounting and mounting again) based on the information found in /proc/mounts.

1.1.9 Resources

(Note some of these resources are not up-to-date with the latest kernel version.)

Creating Linux virtual filesystems. 2002 <<http://lwn.net/Articles/13325/>>

The Linux Virtual File-system Layer by Neil Brown. 1999 <<http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>>

A tour of the Linux VFS by Michael K. Johnson. 1996 <<http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html>>

A small trail through the Linux kernel by Andries Brouwer. 2001

[<http://www.win.tue.nl/~aeb/linux/vfs/trail.html>](http://www.win.tue.nl/~aeb/linux/vfs/trail.html)

1.2 Pathname lookup

This write-up is based on three articles published at lwn.net:

- [<https://lwn.net/Articles/649115/>](https://lwn.net/Articles/649115/) Pathname lookup in Linux
- [<https://lwn.net/Articles/649729/>](https://lwn.net/Articles/649729/) RCU-walk: faster pathname lookup in Linux
- [<https://lwn.net/Articles/650786/>](https://lwn.net/Articles/650786/) A walk among the symlinks

Written by Neil Brown with help from Al Viro and Jon Corbet. It has subsequently been updated to reflect changes in the kernel including:

- per-directory parallel name lookup.
- `openat2()` resolution restriction flags.

1.2.1 Introduction to pathname lookup

The most obvious aspect of pathname lookup, which very little exploration is needed to discover, is that it is complex. There are many rules, special cases, and implementation alternatives that all combine to confuse the unwary reader. Computer science has long been acquainted with such complexity and has tools to help manage it. One tool that we will make extensive use of is “divide and conquer”. For the early parts of the analysis we will divide off symlinks - leaving them until the final part. Well before we get to symlinks we have another major division based on the VFS’ s approach to locking which will allow us to review “REF-walk” and “RCU-walk” separately. But we are getting ahead of ourselves. There are some important low level distinctions we need to clarify first.

There are two sorts of ...

Pathnames (sometimes “file names”), used to identify objects in the filesystem, will be familiar to most readers. They contain two sorts of elements: “slashes” that are sequences of one or more “/” characters, and “components” that are sequences of one or more non- “/” characters. These form two kinds of paths. Those that start with slashes are “absolute” and start from the filesystem root. The others are “relative” and start from the current directory, or from some other location specified by a file descriptor given to a “XXXat” system call such as `openat()`.

It is tempting to describe the second kind as starting with a component, but that isn’ t always accurate: a pathname can lack both slashes and components, it can be empty, in other words. This is generally forbidden in POSIX, but some of those “xxx`at`” system calls in Linux permit it when the `AT_EMPTY_PATH` flag is given. For example, if you have an open file descriptor on an executable file you can execute it by calling `execveat()` passing the file descriptor, an empty path, and the `AT_EMPTY_PATH` flag.

These paths can be divided into two sections: the final component and everything else. The “everything else” is the easy bit. In all cases it must identify a directory that already exists, otherwise an error such as `ENOENT` or `ENOTDIR` will be reported.

The final component is not so simple. Not only do different system calls interpret it quite differently (e.g. some create it, some do not), but it might not even exist: neither the empty pathname nor the pathname that is just slashes have a final component. If it does exist, it could be `“.”` or `“..”` which are handled quite differently from other components.

If a pathname ends with a slash, such as `“/tmp/foo/”` it might be tempting to consider that to have an empty final component. In many ways that would lead to correct results, but not always. In particular, `mkdir()` and `rmdir()` each create or remove a directory named by the final component, and they are required to work with pathnames ending in `“/”`. According to [POSIX](#)

A pathname that contains at least one non- `<slash>` character and that ends with one or more trailing `<slash>` characters shall not be resolved successfully unless the last pathname component before the trailing `<slash>` characters names an existing directory or a directory entry that is to be created for a directory immediately after the pathname is resolved.

The Linux pathname walking code (mostly in `fs/namei.c`) deals with all of these issues: breaking the path into components, handling the “everything else” quite separately from the final component, and checking that the trailing slash is not used where it isn’t permitted. It also addresses the important issue of concurrent access.

While one process is looking up a pathname, another might be making changes that affect that lookup. One fairly extreme case is that if `“a/b”` were renamed to `“a/c/b”` while another process were looking up `“a/b/..”`, that process might successfully resolve on `“a/c”`. Most races are much more subtle, and a big part of the task of pathname lookup is to prevent them from having damaging effects. Many of the possible races are seen most clearly in the context of the `“dcache”` and an understanding of that is central to understanding pathname lookup.

More than just a cache

The `“dcache”` caches information about names in each filesystem to make them quickly available for lookup. Each entry (known as a `“dentry”`) contains three significant fields: a component name, a pointer to a parent dentry, and a pointer to the `“inode”` which contains further information about the object in that parent with the given name. The inode pointer can be `NULL` indicating that the name doesn’t exist in the parent. While there can be linkage in the dentry of a directory to the dentries of the children, that linkage is not used for pathname lookup, and so will not be considered here.

The dcache has a number of uses apart from accelerating lookup. One that will be particularly relevant is that it is closely integrated with the mount table that records which filesystem is mounted where. What the mount table actually stores is which dentry is mounted on top of which other dentry.

When considering the dcache, we have another of our “two types” distinctions: there are two types of filesystems.

Some filesystems ensure that the information in the dcache is always completely accurate (though not necessarily complete). This can allow the VFS to determine if a particular file does or doesn't exist without checking with the filesystem, and means that the VFS can protect the filesystem against certain races and other problems. These are typically “local” filesystems such as ext3, XFS, and Btrfs.

Other filesystems don't provide that guarantee because they cannot. These are typically filesystems that are shared across a network, whether remote filesystems like NFS and 9P, or cluster filesystems like ocfs2 or cephfs. These filesystems allow the VFS to revalidate cached information, and must provide their own protection against awkward races. The VFS can detect these filesystems by the `DCACHE_OP_REVALIDATE` flag being set in the dentry.

REF-walk: simple concurrency management with refcounts and spinlocks

With all of those divisions carefully classified, we can now start looking at the actual process of walking along a path. In particular we will start with the handling of the “everything else” part of a pathname, and focus on the “REF-walk” approach to concurrency management. This code is found in the `link_path_walk()` function, if you ignore all the places that only run when “LOOKUP_RCU” (indicating the use of RCU-walk) is set.

REF-walk is fairly heavy-handed with locks and reference counts. Not as heavy-handed as in the old “big kernel lock” days, but certainly not afraid of taking a lock when one is needed. It uses a variety of different concurrency controls. A background understanding of the various primitives is assumed, or can be gleaned from elsewhere such as in [Meet the Lockers](#).

The locking mechanisms used by REF-walk include:

dentry->d_lockref

This uses the lockref primitive to provide both a spinlock and a reference count. The special-sauce of this primitive is that the conceptual sequence “lock; inc_ref; unlock;” can often be performed with a single atomic memory operation.

Holding a reference on a dentry ensures that the dentry won't suddenly be freed and used for something else, so the values in various fields will behave as expected. It also protects the `->d_inode` reference to the inode to some extent.

The association between a dentry and its inode is fairly permanent. For example, when a file is renamed, the dentry and inode move together to the new location. When a file is created the dentry will initially be negative (i.e. `d_inode` is NULL), and will be assigned to the new inode as part of the act of creation.

When a file is deleted, this can be reflected in the cache either by setting `d_inode` to NULL, or by removing it from the hash table (described shortly) used to look up the name in the parent directory. If the dentry is still in use the second option is used as it is perfectly legal to keep using an open file after it has been deleted and having the dentry around helps. If the dentry is not otherwise in use (i.e. if the

refcount in `d_lockref` is one), only then will `d_inode` be set to `NULL`. Doing it this way is more efficient for a very common case.

So as long as a counted reference is held to a dentry, a non-`NULL` `->d_inode` value will never be changed.

dentry->d_lock

`d_lock` is a synonym for the spinlock that is part of `d_lockref` above. For our purposes, holding this lock protects against the dentry being renamed or unlinked. In particular, its parent (`d_parent`), and its name (`d_name`) cannot be changed, and it cannot be removed from the dentry hash table.

When looking for a name in a directory, REF-walk takes `d_lock` on each candidate dentry that it finds in the hash table and then checks that the parent and name are correct. So it doesn't lock the parent while searching in the cache; it only locks children.

When looking for the parent for a given name (to handle `“..”`), REF-walk can take `d_lock` to get a stable reference to `d_parent`, but it first tries a more lightweight approach. As seen in `dget_parent()`, if a reference can be claimed on the parent, and if subsequently `d_parent` can be seen to have not changed, then there is no need to actually take the lock on the child.

rename_lock

Looking up a given name in a given directory involves computing a hash from the two values (the name and the dentry of the directory), accessing that slot in a hash table, and searching the linked list that is found there.

When a dentry is renamed, the name and the parent dentry can both change so the hash will almost certainly change too. This would move the dentry to a different chain in the hash table. If a filename search happened to be looking at a dentry that was moved in this way, it might end up continuing the search down the wrong chain, and so miss out on part of the correct chain.

The name-lookup process (`d_lookup()`) does not try to prevent this from happening, but only to detect when it happens. `rename_lock` is a seqlock that is updated whenever any dentry is renamed. If `d_lookup` finds that a rename happened while it unsuccessfully scanned a chain in the hash table, it simply tries again.

`rename_lock` is also used to detect and defend against potential attacks against `LOOKUP_BENEATH` and `LOOKUP_IN_ROOT` when resolving `“..”` (where the parent directory is moved outside the root, bypassing the `path_equal()` check). If `rename_lock` is updated during the lookup and the path encounters a `“..”`, a potential attack occurred and `handle_dots()` will bail out with `-EAGAIN`.

inode->i_rwsem

`i_rwsem` is a read/write semaphore that serializes all changes to a particular directory. This ensures that, for example, an `unlink()` and a `rename()` cannot both happen at the same time. It also keeps the directory stable while the filesystem is asked to look up a name that is not currently in the dcache or, optionally, when the list of entries in a directory is being retrieved with `readdir()`.

This has a complementary role to that of `d_lock`: `i_rwsem` on a directory protects all of the names in that directory, while `d_lock` on a name protects just one name in a directory. Most changes to the dcache hold `i_rwsem` on the relevant directory inode and briefly take `d_lock` on one or more the dentries while the change happens. One exception is when idle dentries are removed from the dcache due to memory pressure. This uses `d_lock`, but `i_rwsem` plays no role.

The semaphore affects pathname lookup in two distinct ways. Firstly it prevents changes during lookup of a name in a directory. `walk_component()` uses `lookup_fast()` first which, in turn, checks to see if the name is in the cache, using only `d_lock` locking. If the name isn't found, then `walk_component()` falls back to `lookup_slow()` which takes a shared lock on `i_rwsem`, checks again that the name isn't in the cache, and then calls in to the filesystem to get a definitive answer. A new dentry will be added to the cache regardless of the result.

Secondly, when pathname lookup reaches the final component, it will sometimes need to take an exclusive lock on `i_rwsem` before performing the last lookup so that the required exclusion can be achieved. How path lookup chooses to take, or not take, `i_rwsem` is one of the issues addressed in a subsequent section.

If two threads attempt to look up the same name at the same time - a name that is not yet in the dcache - the shared lock on `i_rwsem` will not prevent them both adding new dentries with the same name. As this would result in confusion an extra level of interlocking is used, based around a secondary hash table (`in_lookup_hashtable`) and a per-dentry flag bit (`DCACHE_PAR_LOOKUP`).

To add a new dentry to the cache while only holding a shared lock on `i_rwsem`, a thread must call `d_alloc_parallel()`. This allocates a dentry, stores the required name and parent in it, checks if there is already a matching dentry in the primary or secondary hash tables, and if not, stores the newly allocated dentry in the secondary hash table, with `DCACHE_PAR_LOOKUP` set.

If a matching dentry was found in the primary hash table then that is returned and the caller can know that it lost a race with some other thread adding the entry. If no matching dentry is found in either cache, the newly allocated dentry is returned and the caller can detect this from the presence of `DCACHE_PAR_LOOKUP`. In this case it knows that it has won any race and now is responsible for asking the filesystem to perform the lookup and find the matching inode. When the lookup is complete, it must call `d_lookup_done()` which clears the flag and does some other house keeping, including removing the dentry from the secondary hash table - it will normally have been added to the primary hash table already. Note that a `struct waitqueue_head` is passed to `d_alloc_parallel()`, and `d_lookup_done()` must be called while this `waitqueue_head` is still in scope.

If a matching dentry is found in the secondary hash table, `d_alloc_parallel()` has a little more work to do. It first waits for `DCACHE_PAR_LOOKUP` to be cleared, us-

ing a `wait_queue` that was passed to the instance of `d_alloc_parallel()` that won the race and that will be woken by the call to `d_lookup_done()`. It then checks to see if the dentry has now been added to the primary hash table. If it has, the dentry is returned and the caller just sees that it lost any race. If it hasn't been added to the primary hash table, the most likely explanation is that some other dentry was added instead using `d_splice_alias()`. In any case, `d_alloc_parallel()` repeats all the look ups from the start and will normally return something from the primary hash table.

mnt->mnt_count

`mnt_count` is a per-CPU reference counter on “mount” structures. Per-CPU here means that incrementing the count is cheap as it only uses CPU-local memory, but checking if the count is zero is expensive as it needs to check with every CPU. Taking a `mnt_count` reference prevents the mount structure from disappearing as the result of regular unmount operations, but does not prevent a “lazy” unmount. So holding `mnt_count` doesn't ensure that the mount remains in the namespace and, in particular, doesn't stabilize the link to the mounted-on dentry. It does, however, ensure that the mount data structure remains coherent, and it provides a reference to the root dentry of the mounted filesystem. So a reference through `->mnt_count` provides a stable reference to the mounted dentry, but not the mounted-on dentry.

mount_lock

`mount_lock` is a global seqlock, a bit like `rename_lock`. It can be used to check if any change has been made to any mount points.

While walking down the tree (away from the root) this lock is used when crossing a mount point to check that the crossing was safe. That is, the value in the seqlock is read, then the code finds the mount that is mounted on the current directory, if there is one, and increments the `mnt_count`. Finally the value in `mount_lock` is checked against the old value. If there is no change, then the crossing was safe. If there was a change, the `mnt_count` is decremented and the whole process is retried.

When walking up the tree (towards the root) by following a “..” link, a little more care is needed. In this case the seqlock (which contains both a counter and a spinlock) is fully locked to prevent any changes to any mount points while stepping up. This locking is needed to stabilize the link to the mounted-on dentry, which the `refcount` on the mount itself doesn't ensure.

`mount_lock` is also used to detect and defend against potential attacks against `LOOKUP_BENEATH` and `LOOKUP_IN_ROOT` when resolving “..” (where the parent directory is moved outside the root, bypassing the `path_equal()` check). If `mount_lock` is updated during the lookup and the path encounters a “..”, a potential attack occurred and `handle_dots()` will bail out with `-EAGAIN`.

RCU

Finally the global (but extremely lightweight) RCU read lock is held from time to time to ensure certain data structures don't get freed unexpectedly.

In particular it is held while scanning chains in the dcache hash table, and the mount point hash table.

Bringing it together with `struct nameidata`

Throughout the process of walking a path, the current status is stored in a `struct nameidata`, “namei” being the traditional name - dating all the way back to [First Edition Unix](#) - of the function that converts a “name” to an “inode”. `struct nameidata` contains (among other fields):

`struct path path`

A `path` contains a `struct vfsmount` (which is embedded in a `struct mount`) and a `struct dentry`. Together these record the current status of the walk. They start out referring to the starting point (the current working directory, the root directory, or some other directory identified by a file descriptor), and are updated on each step. A reference through `d_lockref` and `mnt_count` is always held.

`struct qstr last`

This is a string together with a length (i.e. `_not_nul` terminated) that is the “next” component in the pathname.

`int last_type`

This is one of `LAST_NORM`, `LAST_ROOT`, `LAST_DOT` or `LAST_DOTDOT`. The `last` field is only valid if the type is `LAST_NORM`.

`struct path root`

This is used to hold a reference to the effective root of the filesystem. Often that reference won't be needed, so this field is only assigned the first time it is used, or when a non-standard root is requested. Keeping a reference in the `nameidata` ensures that only one root is in effect for the entire path walk, even if it races with a `chroot()` system call.

It should be noted that in the case of `LOOKUP_IN_ROOT` or `LOOKUP_BENEATH`, the effective root becomes the directory file descriptor passed to `openat2()` (which exposes these `LOOKUP_` flags).

The root is needed when either of two conditions holds: (1) either the pathname or a symbolic link starts with a “/”, or (2) a “.” component is being handled, since “.” from the root must always stay at the root. The value used is usually

the current root directory of the calling process. An alternate root can be provided as when `sysctl()` calls `file_open_root()`, and when NFSv4 or Btrfs call `mount_subtree()`. In each case a pathname is being looked up in a very specific part of the filesystem, and the lookup must not be allowed to escape that subtree. It works a bit like a local `chroot()`.

Ignoring the handling of symbolic links, we can now describe the “`link_path_walk()`” function, which handles the lookup of everything except the final component as:

Given a path (name) and a nameidata structure (nd), check that the current directory has execute permission and then advance name over one component while updating `last_type` and `last`. If that was the final component, then return, otherwise call `walk_component()` and repeat from the top.

`walk_component()` is even easier. If the component is `LAST_DOTS`, it calls `handle_dots()` which does the necessary locking as already described. If it finds a `LAST_NORM` component it first calls “`lookup_fast()`” which only looks in the dcache, but will ask the filesystem to revalidate the result if it is that sort of filesystem. If that doesn't get a good result, it calls “`lookup_slow()`” which takes `i_rwsem`, rechecks the cache, and then asks the filesystem to find a definitive answer. Each of these will call `follow_managed()` (as described below) to handle any mount points.

In the absence of symbolic links, `walk_component()` creates a new struct `path` containing a counted reference to the new dentry and a reference to the new `vfsmount` which is only counted if it is different from the previous `vfsmount`. It then calls `path_to_nameidata()` to install the new struct `path` in the struct `nameidata` and drop the unneeded references.

This “hand-over-hand” sequencing of getting a reference to the new dentry before dropping the reference to the previous dentry may seem obvious, but is worth pointing out so that we will recognize its analogue in the “RCU-walk” version.

Handling the final component

`link_path_walk()` only walks as far as setting `nd->last` and `nd->last_type` to refer to the final component of the path. It does not call `walk_component()` that last time. Handling that final component remains for the caller to sort out. Those callers are `path_lookupat()`, `path_parentat()`, `path_mountpoint()` and `path_openat()` each of which handles the differing requirements of different system calls.

`path_parentat()` is clearly the simplest - it just wraps a little bit of house-keeping around `link_path_walk()` and returns the parent directory and final component to the caller. The caller will be either aiming to create a name (via `filename_create()`) or remove or rename a name (in which case `user_path_parent()` is used). They will use `i_rwsem` to exclude other changes while they validate and then perform their operation.

`path_lookupat()` is nearly as simple - it is used when an existing object is wanted such as by `stat()` or `chmod()`. It essentially just calls `walk_component()` on the

final component through a call to `lookup_last()`. `path_lookupat()` returns just the final dentry.

`path_mountpoint()` handles the special case of unmounting which must not try to revalidate the mounted filesystem. It effectively contains, through a call to `mountpoint_last()`, an alternate implementation of `lookup_slow()` which skips that step. This is important when unmounting a filesystem that is inaccessible, such as one provided by a dead NFS server.

Finally `path_openat()` is used for the `open()` system call; it contains, in support functions starting with “`do_last()`”, all the complexity needed to handle the different subtleties of `O_CREAT` (with or without `O_EXCL`), final “/” characters, and trailing symbolic links. We will revisit this in the final part of this series, which focuses on those symbolic links. “`do_last()`” will sometimes, but not always, take `i_rwsem`, depending on what it finds.

Each of these, or the functions which call them, need to be alert to the possibility that the final component is not `LAST_NORM`. If the goal of the lookup is to create something, then any value for `last_type` other than `LAST_NORM` will result in an error. For example if `path_parentat()` reports `LAST_DOTDOT`, then the caller won't try to create that name. They also check for trailing slashes by testing `last.name[last.len]`. If there is any character beyond the final component, it must be a trailing slash.

Revalidation and automounts

Apart from symbolic links, there are only two parts of the “REF-walk” process not yet covered. One is the handling of stale cache entries and the other is automounts.

On filesystems that require it, the lookup routines will call the `->d_revalidate()` dentry method to ensure that the cached information is current. This will often confirm validity or update a few details from a server. In some cases it may find that there has been change further up the path and that something that was thought to be valid previously isn't really. When this happens the lookup of the whole path is aborted and retried with the “`LOOKUP_REVAL`” flag set. This forces revalidation to be more thorough. We will see more details of this retry process in the next article.

Automount points are locations in the filesystem where an attempt to lookup a name can trigger changes to how that lookup should be handled, in particular by mounting a filesystem there. These are covered in greater detail in `autofs.txt` in the Linux documentation tree, but a few notes specifically related to path lookup are in order here.

The Linux VFS has a concept of “managed” dentries which is reflected in function names such as “`follow_managed()`”. There are three potentially interesting things about these dentries corresponding to three different flags that might be set in `dentry->d_flags`:

DCACHE_MANAGE_TRANSIT

If this flag has been set, then the filesystem has requested that the `d_manage()` dentry operation be called before handling any possible mount point. This can perform two particular services:

It can block to avoid races. If an automount point is being unmounted, the `d_manage()` function will usually wait for that process to complete before letting the new lookup proceed and possibly trigger a new automount.

It can selectively allow only some processes to transit through a mount point. When a server process is managing automounts, it may need to access a directory without triggering normal automount processing. That server process can identify itself to the `autofs` filesystem, which will then give it a special pass through `d_manage()` by returning `-EISDIR`.

DCACHE_MOUNTED

This flag is set on every dentry that is mounted on. As Linux supports multiple filesystem namespaces, it is possible that the dentry may not be mounted on in this namespace, just in some other. So this flag is seen as a hint, not a promise.

If this flag is set, and `d_manage()` didn't return `-EISDIR`, `lookup_mnt()` is called to examine the mount hash table (honoring the `mount_lock` described earlier) and possibly return a new `vfsmount` and a new dentry (both with counted references).

DCACHE_NEED_AUTOMOUNT

If `d_manage()` allowed us to get this far, and `lookup_mnt()` didn't find a mount point, then this flag causes the `d_automount()` dentry operation to be called.

The `d_automount()` operation can be arbitrarily complex and may communicate with server processes etc. but it should ultimately either report that there was an error, that there was nothing to mount, or should provide an updated `struct path` with new dentry and `vfsmount`.

In the latter case, `finish_automount()` will be called to safely install the new mount point into the mount table.

There is no new locking of import here and it is important that no locks (only counted references) are held over this processing due to the very real possibility of extended delays. This will become more important next time when we examine RCU-walk which is particularly sensitive to delays.

1.2.2 RCU-walk - faster pathname lookup in Linux

RCU-walk is another algorithm for performing pathname lookup in Linux. It is in many ways similar to REF-walk and the two share quite a bit of code. The significant difference in RCU-walk is how it allows for the possibility of concurrent access.

We noted that REF-walk is complex because there are numerous details and special cases. RCU-walk reduces this complexity by simply refusing to handle a number of cases - it instead falls back to REF-walk. The difficulty with RCU-walk comes from a different direction: unfamiliarity. The locking rules when depending on RCU are quite different from traditional locking, so we will spend a little extra time when we come to those.

Clear demarcation of roles

The easiest way to manage concurrency is to forcibly stop any other thread from changing the data structures that a given thread is looking at. In cases where no other thread would even think of changing the data and lots of different threads want to read at the same time, this can be very costly. Even when using locks that permit multiple concurrent readers, the simple act of updating the count of the number of current readers can impose an unwanted cost. So the goal when reading a shared data structure that no other process is changing is to avoid writing anything to memory at all. Take no locks, increment no counts, leave no footprints.

The REF-walk mechanism already described certainly doesn't follow this principle, but then it is really designed to work when there may well be other threads modifying the data. RCU-walk, in contrast, is designed for the common situation where there are lots of frequent readers and only occasional writers. This may not be common in all parts of the filesystem tree, but in many parts it will be. For the other parts it is important that RCU-walk can quickly fall back to using REF-walk.

Pathname lookup always starts in RCU-walk mode but only remains there as long as what it is looking for is in the cache and is stable. It dances lightly down the cached filesystem image, leaving no footprints and carefully watching where it is, to be sure it doesn't trip. If it notices that something has changed or is changing, or if something isn't in the cache, then it tries to stop gracefully and switch to REF-walk.

This stopping requires getting a counted reference on the current `vfsmount` and `dentry`, and ensuring that these are still valid - that a path walk with REF-walk would have found the same entries. This is an invariant that RCU-walk must guarantee. It can only make decisions, such as selecting the next step, that are decisions which REF-walk could also have made if it were walking down the tree at the same time. If the graceful stop succeeds, the rest of the path is processed with the reliable, if slightly sluggish, REF-walk. If RCU-walk finds it cannot stop gracefully, it simply gives up and restarts from the top with REF-walk.

This pattern of "try RCU-walk, if that fails try REF-walk" can be clearly seen in functions like `filename_lookup()`, `filename_parentat()`, `filename_mountpoint()`, `do_filp_open()`, and `do_file_open_root()`. These five correspond roughly to the four `path_`*`` functions we met earlier, each of which calls `link_path_walk()`. The `path_*` functions are called using different mode flags

until a mode is found which works. They are first called with LOOKUP_RCU set to request “RCU-walk” . If that fails with the error ECHILD they are called again with no special flag to request “REF-walk” . If either of those report the error ESTALE a final attempt is made with LOOKUP_REVAL set (and no LOOKUP_RCU) to ensure that entries found in the cache are forcibly revalidated - normally entries are only revalidated if the filesystem determines that they are too old to trust.

The LOOKUP_RCU attempt may drop that flag internally and switch to REF-walk, but will never then try to switch back to RCU-walk. Places that trip up RCU-walk are much more likely to be near the leaves and so it is very unlikely that there will be much, if any, benefit from switching back.

RCU and seqlocks: fast and light

RCU is, unsurprisingly, critical to RCU-walk mode. The `rcu_read_lock()` is held for the entire time that RCU-walk is walking down a path. The particular guarantee it provides is that the key data structures - dentries, inodes, super_blocks, and mounts - will not be freed while the lock is held. They might be unlinked or invalidated in one way or another, but the memory will not be repurposed so values in various fields will still be meaningful. This is the only guarantee that RCU provides; everything else is done using seqlocks.

As we saw above, REF-walk holds a counted reference to the current dentry and the current vfsmount, and does not release those references before taking references to the “next” dentry or vfsmount. It also sometimes takes the `d_lock` spinlock. These references and locks are taken to prevent certain changes from happening. RCU-walk must not take those references or locks and so cannot prevent such changes. Instead, it checks to see if a change has been made, and aborts or retries if it has.

To preserve the invariant mentioned above (that RCU-walk may only make decisions that REF-walk could have made), it must make the checks at or near the same places that REF-walk holds the references. So, when REF-walk increments a reference count or takes a spinlock, RCU-walk samples the status of a seqlock using `read_seqcount_begin()` or a similar function. When REF-walk decrements the count or drops the lock, RCU-walk checks if the sampled status is still valid using `read_seqcount_retry()` or similar.

However, there is a little bit more to seqlocks than that. If RCU-walk accesses two different fields in a seqlock-protected structure, or accesses the same field twice, there is no a priori guarantee of any consistency between those accesses. When consistency is needed - which it usually is - RCU-walk must take a copy and then use `read_seqcount_retry()` to validate that copy.

`read_seqcount_retry()` not only checks the sequence number, but also imposes a memory barrier so that no memory-read instruction from before the call can be delayed until after the call, either by the CPU or by the compiler. A simple example of this can be seen in `slow_dentry_cmp()` which, for filesystems which do not use simple byte-wise name equality, calls into the filesystem to compare a name against a dentry. The length and name pointer are copied into local variables, then `read_seqcount_retry()` is called to confirm the two are consistent, and only then is `->d_compare()` called. When standard filename comparison is used, `dentry_cmp()` is called instead. Notably it does `_not_` use

`read_seqcount_retry()`, but instead has a large comment explaining why the consistency guarantee isn't necessary. A subsequent `read_seqcount_retry()` will be sufficient to catch any problem that could occur at this point.

With that little refresher on seqlocks out of the way we can look at the bigger picture of how RCU-walk uses seqlocks.

`mount_lock` and `nd->m_seq`

We already met the `mount_lock` seqlock when REF-walk used it to ensure that crossing a mount point is performed safely. RCU-walk uses it for that too, but for quite a bit more.

Instead of taking a counted reference to each `vfsmount` as it descends the tree, RCU-walk samples the state of `mount_lock` at the start of the walk and stores this initial sequence number in the `struct nameidata` in the `m_seq` field. This one lock and one sequence number are used to validate all accesses to all `vfsmounts`, and all mount point crossings. As changes to the mount table are relatively rare, it is reasonable to fall back on REF-walk any time that any “mount” or “unmount” happens.

`m_seq` is checked (using `read_seqretry()`) at the end of an RCU-walk sequence, whether switching to REF-walk for the rest of the path or when the end of the path is reached. It is also checked when stepping down over a mount point (in `__follow_mount_rcu()`) or up (in `follow_dotdot_rcu()`). If it is ever found to have changed, the whole RCU-walk sequence is aborted and the path is processed again by REF-walk.

If RCU-walk finds that `mount_lock` hasn't changed then it can be sure that, had REF-walk taken counted references on each `vfsmount`, the results would have been the same. This ensures the invariant holds, at least for `vfsmount` structures.

`dentry->d_seq` and `nd->seq`

In place of taking a count or lock on `d_reflock`, RCU-walk samples the per-dentry `d_seq` seqlock, and stores the sequence number in the `seq` field of the `nameidata` structure, so `nd->seq` should always be the current sequence number of `nd->dentry`. This number needs to be revalidated after copying, and before using, the name, parent, or inode of the dentry.

The handling of the name we have already looked at, and the parent is only accessed in `follow_dotdot_rcu()` which fairly trivially follows the required pattern, though it does so for three different cases.

When not at a mount point, `d_parent` is followed and its `d_seq` is collected. When we are at a mount point, we instead follow the `mnt->mnt_mountpoint` link to get a new dentry and collect its `d_seq`. Then, after finally finding a `d_parent` to follow, we must check if we have landed on a mount point and, if so, must find that mount point and follow the `mnt->mnt_root` link. This would imply a somewhat unusual, but certainly possible, circumstance where the starting point of the path lookup was in part of the filesystem that was mounted on, and so not visible from the root.

The inode pointer, stored in `->d_inode`, is a little more interesting. The inode will always need to be accessed at least twice, once to determine if it is NULL and once to verify access permissions. Symlink handling requires a validated inode pointer too. Rather than revalidating on each access, a copy is made on the first access and it is stored in the `inode` field of `nameidata` from where it can be safely accessed without further validation.

`lookup_fast()` is the only lookup routine that is used in RCU-mode, `lookup_slow()` being too slow and requiring locks. It is in `lookup_fast()` that we find the important “hand over hand” tracking of the current dentry.

The current dentry and current seq number are passed to `__d_lookup_rcu()` which, on success, returns a new dentry and a new seq number. `lookup_fast()` then copies the inode pointer and revalidates the new seq number. It then validates the old dentry with the old seq number one last time and only then continues. This process of getting the seq number of the new dentry and then checking the seq number of the old exactly mirrors the process of getting a counted reference to the new dentry before dropping that for the old dentry which we saw in REF-walk.

No inode-`>i_rwsem` or even `rename_lock`

A semaphore is a fairly heavyweight lock that can only be taken when it is permissible to sleep. As `rcu_read_lock()` forbids sleeping, `inode->i_rwsem` plays no role in RCU-walk. If some other thread does take `i_rwsem` and modifies the directory in a way that RCU-walk needs to notice, the result will be either that RCU-walk fails to find the dentry that it is looking for, or it will find a dentry which `read_seqretry()` won't validate. In either case it will drop down to REF-walk mode which can take whatever locks are needed.

Though `rename_lock` could be used by RCU-walk as it doesn't require any sleeping, RCU-walk doesn't bother. REF-walk uses `rename_lock` to protect against the possibility of hash chains in the dcache changing while they are being searched. This can result in failing to find something that actually is there. When RCU-walk fails to find something in the dentry cache, whether it is really there or not, it already drops down to REF-walk and tries again with appropriate locking. This neatly handles all cases, so adding extra checks on `rename_lock` would bring no significant value.

unlazy walk() and complete_walk()

That “dropping down to REF-walk” typically involves a call to `unlazy_walk()`, so named because “RCU-walk” is also sometimes referred to as “lazy walk”. `unlazy_walk()` is called when following the path down to the current `vfs-mount/dentry` pair seems to have proceeded successfully, but the next step is problematic. This can happen if the next name cannot be found in the dcache, if permission checking or name revalidation couldn't be achieved while the `rcu_read_lock()` is held (which forbids sleeping), if an automount point is found, or in a couple of cases involving symlinks. It is also called from `complete_walk()` when the lookup has reached the final component, or the very end of the path, depending on which particular flavor of lookup is used.

Other reasons for dropping out of RCU-walk that do not trigger a call to `unlazy_walk()` are when some inconsistency is found that cannot be handled immediately, such as `mount_lock` or one of the `d_seq` seqlocks reporting a change. In these cases the relevant function will return `-ECHILD` which will percolate up until it triggers a new attempt from the top using REF-walk.

For those cases where `unlazy_walk()` is an option, it essentially takes a reference on each of the pointers that it holds (`vfsmount`, `dentry`, and possibly some symbolic links) and then verifies that the relevant seqlocks have not been changed. If there have been changes, it, too, aborts with `-ECHILD`, otherwise the transition to REF-walk has been a success and the lookup process continues.

Taking a reference on those pointers is not quite as simple as just incrementing a counter. That works to take a second reference if you already have one (often indirectly through another object), but it isn't sufficient if you don't actually have a counted reference at all. For `dentry->d_lockref`, it is safe to increment the reference counter to get a reference unless it has been explicitly marked as "dead" which involves setting the counter to `-128`. `lockref_get_not_dead()` achieves this.

For `mnt->mnt_count` it is safe to take a reference as long as `mount_lock` is then used to validate the reference. If that validation fails, it may not be safe to just drop that reference in the standard way of calling `mnt_put()` - an unmount may have progressed too far. So the code in `legitimize_mnt()`, when it finds that the reference it got might not be safe, checks the `MNT_SYNC_UMOUNT` flag to determine if a simple `mnt_put()` is correct, or if it should just decrement the count and pretend none of this ever happened.

Taking care in filesystems

RCU-walk depends almost entirely on cached information and often will not call into the filesystem at all. However there are two places, besides the already-mentioned component-name comparison, where the file system might be included in RCU-walk, and it must know to be careful.

If the filesystem has non-standard permission-checking requirements - such as a networked filesystem which may need to check with the server - the `i_op->permission` interface might be called during RCU-walk. In this case an extra "MAY_NOT_BLOCK" flag is passed so that it knows not to sleep, but to return `-ECHILD` if it cannot complete promptly. `i_op->permission` is given the inode pointer, not the `dentry`, so it doesn't need to worry about further consistency checks. However if it accesses any other filesystem data structures, it must ensure they are safe to be accessed with only the `rcu_read_lock()` held. This typically means they must be freed using `kfree_rcu()` or similar.

If the filesystem may need to revalidate `dcache` entries, then `d_op->d_revalidate` may be called in RCU-walk too. This interface is passed the `dentry` but does not have access to the `inode` or the `seq` number from the `nameidata`, so it needs to be extra careful when accessing fields in the `dentry`. This "extra care" typically involves using `READ_ONCE()` to access fields, and verifying the result is not `NULL` before using it. This pattern can be seen in `nfs_lookup_revalidate()`.

A pair of patterns

In various places in the details of REF-walk and RCU-walk, and also in the big picture, there are a couple of related patterns that are worth being aware of.

The first is “try quickly and check, if that fails try slowly” . We can see that in the high-level approach of first trying RCU-walk and then trying REF-walk, and in places where `unlazy_walk()` is used to switch to REF-walk for the rest of the path. We also saw it earlier in `dget_parent()` when following a “.” link. It tries a quick way to get a reference, then falls back to taking locks if needed.

The second pattern is “try quickly and check, if that fails try again - repeatedly” . This is seen with the use of `rename_lock` and `mount_lock` in REF-walk. RCU-walk doesn't make use of this pattern - if anything goes wrong it is much safer to just abort and try a more sedate approach.

The emphasis here is “try quickly and check” . It should probably be “try quickly and carefully, then check” . The fact that checking is needed is a reminder that the system is dynamic and only a limited number of things are safe at all. The most likely cause of errors in this whole process is assuming something is safe when in reality it isn't. Careful consideration of what exactly guarantees the safety of each access is sometimes necessary.

1.2.3 A walk among the symlinks

There are several basic issues that we will examine to understand the handling of symbolic links: the symlink stack, together with cache lifetimes, will help us understand the overall recursive handling of symlinks and lead to the special care needed for the final component. Then a consideration of access-time updates and summary of the various flags controlling lookup will finish the story.

The symlink stack

There are only two sorts of filesystem objects that can usefully appear in a path prior to the final component: directories and symlinks. Handling directories is quite straightforward: the new directory simply becomes the starting point at which to interpret the next component on the path. Handling symbolic links requires a bit more work.

Conceptually, symbolic links could be handled by editing the path. If a component name refers to a symbolic link, then that component is replaced by the body of the link and, if that body starts with a `'/'` , then all preceding parts of the path are discarded. This is what the `readlink -f` command does, though it also edits out “.” and “..” components.

Directly editing the path string is not really necessary when looking up a path, and discarding early components is pointless as they aren't looked at anyway. Keeping track of all remaining components is important, but they can of course be kept separately; there is no need to concatenate them. As one symlink may easily refer to another, which in turn can refer to a third, we may need to keep the remaining components of several paths, each to be processed when the preceding ones are completed. These path remnants are kept on a stack of limited size.

There are two reasons for placing limits on how many symlinks can occur in a single path lookup. The most obvious is to avoid loops. If a symlink referred to itself either directly or through intermediaries, then following the symlink can never complete successfully - the error `EL0OP` must be returned. Loops can be detected without imposing limits, but limits are the simplest solution and, given the second reason for restriction, quite sufficient.

The second reason was [outlined recently](#) by Linus:

Because it's a latency and DoS issue too. We need to react well to true loops, but also to "very deep" non-loops. It's not about memory use, it's about users triggering unreasonable CPU resources.

Linux imposes a limit on the length of any pathname: `PATH_MAX`, which is 4096. There are a number of reasons for this limit; not letting the kernel spend too much time on just one path is one of them. With symbolic links you can effectively generate much longer paths so some sort of limit is needed for the same reason. Linux imposes a limit of at most 40 symlinks in any one path lookup. It previously imposed a further limit of eight on the maximum depth of recursion, but that was raised to 40 when a separate stack was implemented, so there is now just the one limit.

The `nameidata` structure that we met in an earlier article contains a small stack that can be used to store the remaining part of up to two symlinks. In many cases this will be sufficient. If it isn't, a separate stack is allocated with room for 40 symlinks. Pathname lookup will never exceed that stack as, once the 40th symlink is detected, an error is returned.

It might seem that the name remnants are all that needs to be stored on this stack, but we need a bit more. To see that, we need to move on to cache lifetimes.

Storage and lifetime of cached symlinks

Like other filesystem resources, such as inodes and directory entries, symlinks are cached by Linux to avoid repeated costly access to external storage. It is particularly important for RCU-walk to be able to find and temporarily hold onto these cached entries, so that it doesn't need to drop down into REF-walk.

While each filesystem is free to make its own choice, symlinks are typically stored in one of two places. Short symlinks are often stored directly in the inode. When a filesystem allocates a `struct inode` it typically allocates extra space to store private data (a common [object-oriented design pattern](#) in the kernel). This will sometimes include space for a symlink. The other common location is in the page cache, which normally stores the content of files. The pathname in a symlink can be seen as the content of that symlink and can easily be stored in the page cache just like file content.

When neither of these is suitable, the next most likely scenario is that the filesystem will allocate some temporary memory and copy or construct the symlink content into that memory whenever it is needed.

When the symlink is stored in the inode, it has the same lifetime as the inode which, itself, is protected by RCU or by a counted reference on the dentry. This means that the mechanisms that pathname lookup uses to access the dcache and icache (inode cache) safely are quite sufficient for accessing some cached symlinks

safely. In these cases, the `i_link` pointer in the inode is set to point to wherever the symlink is stored and it can be accessed directly whenever needed.

When the symlink is stored in the page cache or elsewhere, the situation is not so straightforward. A reference on a dentry or even on an inode does not imply any reference on cached pages of that inode, and even an `rcu_read_lock()` is not sufficient to ensure that a page will not disappear. So for these symlinks the pathname lookup code needs to ask the filesystem to provide a stable reference and, significantly, needs to release that reference when it is finished with it.

Taking a reference to a cache page is often possible even in RCU-walk mode. It does require making changes to memory, which is best avoided, but that isn't necessarily a big cost and it is better than dropping out of RCU-walk mode completely. Even filesystems that allocate space to copy the symlink into can use `GFP_ATOMIC` to often successfully allocate memory without the need to drop out of RCU-walk. If a filesystem cannot successfully get a reference in RCU-walk mode, it must return `-ECHILD` and `unlazy_walk()` will be called to return to REF-walk mode in which the filesystem is allowed to sleep.

The place for all this to happen is the `i_op->follow_link()` inode method. In the present mainline code this is never actually called in RCU-walk mode as the rewrite is not quite complete. It is likely that in a future release this method will be passed an inode pointer when called in RCU-walk mode so it both (1) knows to be careful, and (2) has the validated pointer. Much like the `i_op->permission()` method we looked at previously, `->follow_link()` would need to be careful that all the data structures it references are safe to be accessed while holding no counted reference, only the RCU lock. Though getting a reference with `->follow_link()` is not yet done in RCU-walk mode, the code is ready to release the reference when that does happen.

This need to drop the reference to a symlink adds significant complexity. It requires a reference to the inode so that the `i_op->put_link()` inode operation can be called. In REF-walk, that reference is kept implicitly through a reference to the dentry, so keeping the `struct path` of the symlink is easiest. For RCU-walk, the pointer to the inode is kept separately. To allow switching from RCU-walk back to REF-walk in the middle of processing nested symlinks we also need the seq number for the dentry so we can confirm that switching back was safe.

Finally, when providing a reference to a symlink, the filesystem also provides an opaque "cookie" that must be passed to `->put_link()` so that it knows what to free. This might be the allocated memory area, or a pointer to the `struct page` in the page cache, or something else completely. Only the filesystem knows what it is.

In order for the reference to each symlink to be dropped when the walk completes, whether in RCU-walk or REF-walk, the symlink stack needs to contain, along with the path remnants:

- the `struct path` to provide a reference to the inode in REF-walk
- the `struct inode *` to provide a reference to the inode in RCU-walk
- the seq to allow the path to be safely switched from RCU-walk to REF-walk
- the cookie that tells `->put_path()` what to put.

This means that each entry in the symlink stack needs to hold five pointers and an integer instead of just one pointer (the path remnant). On a 64-bit system, this is about 40 bytes per entry; with 40 entries it adds up to 1600 bytes total, which is less than half a page. So it might seem like a lot, but is by no means excessive.

Note that, in a given stack frame, the path remnant (name) is not part of the symlink that the other fields refer to. It is the remnant to be followed once that symlink has been fully parsed.

Following the symlink

The main loop in `link_path_walk()` iterates seamlessly over all components in the path and all of the non-final symlinks. As symlinks are processed, the name pointer is adjusted to point to a new symlink, or is restored from the stack, so that much of the loop doesn't need to notice. Getting this name variable on and off the stack is very straightforward; pushing and popping the references is a little more complex.

When a symlink is found, `walk_component()` returns the value 1 (0 is returned for any other sort of success, and a negative number is, as usual, an error indicator). This causes `get_link()` to be called; it then gets the link from the filesystem. Providing that operation is successful, the old path name is placed on the stack, and the new value is used as the name for a while. When the end of the path is found (i.e. `*name` is `'\0'`) the old name is restored off the stack and path walking continues.

Pushing and popping the reference pointers (inode, cookie, etc.) is more complex in part because of the desire to handle tail recursion. When the last component of a symlink itself points to a symlink, we want to pop the symlink-just-completed off the stack before pushing the symlink-just-found to avoid leaving empty path remnants that would just get in the way.

It is most convenient to push the new symlink references onto the stack in `walk_component()` immediately when the symlink is found; `walk_component()` is also the last piece of code that needs to look at the old symlink as it walks that last component. So it is quite convenient for `walk_component()` to release the old symlink and pop the references just before pushing the reference information for the new symlink. It is guided in this by two flags; `WALK_GET`, which gives it permission to follow a symlink if it finds one, and `WALK_PUT`, which tells it to release the current symlink after it has been followed. `WALK_PUT` is tested first, leading to a call to `put_link()`. `WALK_GET` is tested subsequently (by `should_follow_link()`) leading to a call to `pick_link()` which sets up the stack frame.

Symlinks with no final component

A pair of special-case symlinks deserve a little further explanation. Both result in a new `struct path` (with `mount` and `dentry`) being set up in the `nameidata`, and result in `get_link()` returning `NULL`.

The more obvious case is a symlink to `“/”`. All symlinks starting with `“/”` are detected in `get_link()` which resets the `nameidata` to point to the effective filesystem root. If the symlink only contains `“/”` then there is nothing more to do, no components at all, so `NULL` is returned to indicate that the symlink can be released and the stack frame discarded.

The other case involves things in `/proc` that look like symlinks but aren't really (and are therefore commonly referred to as “magic-links”):

```
$ ls -l /proc/self/fd/1
lrwx----- 1 neilb neilb 64 Jun 13 10:19 /proc/self/fd/1 -> /dev/pts/4
```

Every open file descriptor in any process is represented in `/proc` by something that looks like a symlink. It is really a reference to the target file, not just the name of it. When you `readlink` these objects you get a name that might refer to the same file - unless it has been unlinked or mounted over. When `walk_component()` follows one of these, the `->follow_link()` method in “`procfs`” doesn't return a string name, but instead calls `nd_jump_link()` which updates the `nameidata` in place to point to that target. `->follow_link()` then returns `NULL`. Again there is no final component and `get_link()` reports this by leaving the `last_type` field of `nameidata` as `LAST_BIND`.

Following the symlink in the final component

All this leads to `link_path_walk()` walking down every component, and following all symbolic links it finds, until it reaches the final component. This is just returned in the `last` field of `nameidata`. For some callers, this is all they need; they want to create that `last` name if it doesn't exist or give an error if it does. Other callers will want to follow a symlink if one is found, and possibly apply special handling to the last component of that symlink, rather than just the last component of the original file name. These callers potentially need to call `link_path_walk()` again and again on successive symlinks until one is found that doesn't point to another symlink.

This case is handled by the relevant caller of `link_path_walk()`, such as `path_lookupat()` using a loop that calls `link_path_walk()`, and then handles the final component. If the final component is a symlink that needs to be followed, then `trailing_symlink()` is called to set things up properly and the loop repeats, calling `link_path_walk()` again. This could loop as many as 40 times if the last component of each symlink is another symlink.

The various functions that examine the final component and possibly report that it is a symlink are `lookup_last()`, `mountpoint_last()` and `do_last()`, each of which use the same convention as `walk_component()` of returning 1 if a symlink was found that needs to be followed.

Of these, `do_last()` is the most interesting as it is used for opening a file. Part

of `do_last()` runs with `i_rwsem` held and this part is in a separate function: `lookup_open()`.

Explaining `do_last()` completely is beyond the scope of this article, but a few highlights should help those interested in exploring the code.

1. Rather than just finding the target file, `do_last()` needs to open it. If the file was found in the dcache, then `vfs_open()` is used for this. If not, then `lookup_open()` will either call `atomic_open()` (if the filesystem provides it) to combine the final lookup with the open, or will perform the separate `lookup_real()` and `vfs_create()` steps directly. In the later case the actual “open” of this newly found or created file will be performed by `vfs_open()`, just as if the name were found in the dcache.
2. `vfs_open()` can fail with `-EOPENSTALE` if the cached information wasn't quite current enough. Rather than restarting the lookup from the top with `LOOKUP_REVAL` set, `lookup_open()` is called instead, giving the filesystem a chance to resolve small inconsistencies. If that doesn't work, only then is the lookup restarted from the top.
3. An open with `O_CREAT` **does** follow a symlink in the final component, unlike other creation system calls (like `mkdir`). So the sequence:

```
ln -s bar /tmp/foo
echo hello > /tmp/foo
```

will create a file called `/tmp/bar`. This is not permitted if `O_EXCL` is set but otherwise is handled for an `O_CREAT` open much like for a non-creating open: `should_follow_link()` returns 1, and so does `do_last()` so that `trailing_symlink()` gets called and the open process continues on the symlink that was found.

Updating the access time

We previously said of RCU-walk that it would “take no locks, increment no counts, leave no footprints.” We have since seen that some “footprints” can be needed when handling symlinks as a counted reference (or even a memory allocation) may be needed. But these footprints are best kept to a minimum.

One other place where walking down a symlink can involve leaving footprints in a way that doesn't affect directories is in updating access times. In Unix (and Linux) every filesystem object has a “last accessed time”, or “`atime`”. Passing through a directory to access a file within is not considered to be an access for the purposes of `atime`; only listing the contents of a directory can update its `atime`. Symlinks are different it seems. Both reading a symlink (with `readlink()`) and looking up a symlink on the way to some other destination can update the `atime` on that symlink.

It is not clear why this is the case; POSIX has little to say on the subject. The [clearest statement](#) is that, if a particular implementation updates a timestamp in a place not specified by POSIX, this must be documented “except that any changes caused by pathname resolution need not be documented”. This seems to imply that POSIX doesn't really care about access-time updates during pathname lookup.

An examination of history shows that prior to [Linux 1.3.87](#), the ext2 filesystem, at least, didn't update atime when following a link. Unfortunately we have no record of why that behavior was changed.

In any case, access time must now be updated and that operation can be quite complex. Trying to stay in RCU-walk while doing it is best avoided. Fortunately it is often permitted to skip the atime update. Because atime updates cause performance problems in various areas, Linux supports the `relatime` mount option, which generally limits the updates of atime to once per day on files that aren't being changed (and symlinks never change once created). Even without `relatime`, many filesystems record atime with a one-second granularity, so only one update per second is required.

It is easy to test if an atime update is needed while in RCU-walk mode and, if it isn't, the update can be skipped and RCU-walk mode continues. Only when an atime update is actually required does the path walk drop down to REF-walk. All of this is handled in the `get_link()` function.

A few flags

A suitable way to wrap up this tour of pathname walking is to list the various flags that can be stored in the `nameidata` to guide the lookup process. Many of these are only meaningful on the final component, others reflect the current state of the pathname lookup, and some apply restrictions to all path components encountered in the path lookup.

And then there is `LOOKUP_EMPTY`, which doesn't fit conceptually with the others. If this is not set, an empty pathname causes an error very early on. If it is set, empty pathnames are not considered to be an error.

Global state flags

We have already met two global state flags: `LOOKUP_RCU` and `LOOKUP_REVAL`. These select between one of three overall approaches to lookup: RCU-walk, REF-walk, and REF-walk with forced revalidation.

`LOOKUP_PARENT` indicates that the final component hasn't been reached yet. This is primarily used to tell the audit subsystem the full context of a particular access being audited.

`LOOKUP_ROOT` indicates that the `root` field in the `nameidata` was provided by the caller, so it shouldn't be released when it is no longer needed.

`LOOKUP_JUMPED` means that the current `dentry` was chosen not because it had the right name but for some other reason. This happens when following `..`, following a symlink to `/`, crossing a mount point or accessing a `/proc/$PID/fd/$FD` symlink (also known as a "magic link"). In this case the filesystem has not been asked to revalidate the name (with `d_revalidate()`). In such cases the `inode` may still need to be revalidated, so `d_op->d_weak_revalidate()` is called if `LOOKUP_JUMPED` is set when the look completes - which may be at the final component or, when creating, unlinking, or renaming, at the penultimate component.

Resolution-restriction flags

In order to allow userspace to protect itself against certain race conditions and attack scenarios involving changing path components, a series of flags are available which apply restrictions to all path components encountered during path lookup. These flags are exposed through `openat2()`'s `resolve` field.

`LOOKUP_NO_SYMLINKS` blocks all symlink traversals (including magic-links). This is distinctly different from `LOOKUP_FOLLOW`, because the latter only relates to restricting the following of trailing symlinks.

`LOOKUP_NO_MAGICLINKS` blocks all magic-link traversals. Filesystems must ensure that they return errors from `nd_jump_link()`, because that is how `LOOKUP_NO_MAGICLINKS` and other magic-link restrictions are implemented.

`LOOKUP_NO_XDEV` blocks all `vfsmount` traversals (this includes both `bind`-mounts and ordinary mounts). Note that the `vfsmount` which contains the lookup is determined by the first mountpoint the path lookup reaches - absolute paths start with the `vfsmount` of `/`, and relative paths start with the `dfd`'s `vfsmount`. Magic-links are only permitted if the `vfsmount` of the path is unchanged.

`LOOKUP_BENEATH` blocks any path components which resolve outside the starting point of the resolution. This is done by blocking `nd_jump_root()` as well as blocking `“..”` if it would jump outside the starting point. `rename_lock` and `mount_lock` are used to detect attacks against the resolution of `“..”`. Magic-links are also blocked.

`LOOKUP_IN_ROOT` resolves all path components as though the starting point were the filesystem root. `nd_jump_root()` brings the resolution back to to the starting point, and `“..”` at the starting point will act as a no-op. As with `LOOKUP_BENEATH`, `rename_lock` and `mount_lock` are used to detect attacks against `“..”` resolution. Magic-links are also blocked.

Final-component flags

Some of these flags are only set when the final component is being considered. Others are only checked for when considering that final component.

`LOOKUP_AUTOMOUNT` ensures that, if the final component is an automount point, then the mount is triggered. Some operations would trigger it anyway, but operations like `stat()` deliberately don't. `statfs()` needs to trigger the mount but otherwise behaves a lot like `stat()`, so it sets `LOOKUP_AUTOMOUNT`, as does `quotactl()` and the handling of `“mount --bind”`.

`LOOKUP_FOLLOW` has a similar function to `LOOKUP_AUTOMOUNT` but for symlinks. Some system calls set or clear it implicitly, while others have API flags such as `AT_SYMLINK_FOLLOW` and `UMOUNT_NOFOLLOW` to control it. Its effect is similar to `WALK_GET` that we already met, but it is used in a different way.

`LOOKUP_DIRECTORY` insists that the final component is a directory. Various callers set this and it is also set when the final component is found to be followed by a slash.

Finally `LOOKUP_OPEN`, `LOOKUP_CREATE`, `LOOKUP_EXCL`, and `LOOKUP_RENAME_TARGET` are not used directly by the VFS but are made available to the filesystem and

particularly the `->d_revalidate()` method. A filesystem can choose not to bother revalidating too hard if it knows that it will be asked to open or create the file soon. These flags were previously useful for `->lookup()` too but with the introduction of `->atomic_open()` they are less relevant there.

End of the road

Despite its complexity, all this pathname lookup code appears to be in good shape - various parts are certainly easier to understand now than even a couple of releases ago. But that doesn't mean it is "finished". As already mentioned, RCU-walk currently only follows symlinks that are stored in the inode so, while it handles many ext4 symlinks, it doesn't help with NFS, XFS, or Btrfs. That support is not likely to be long delayed.

1.3 Linux Filesystems API summary

This section contains API-level documentation, mostly taken from the source code itself.

1.3.1 The Linux VFS

The Filesystem types

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

Constants

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

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

Description

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

struct **address_space**
Contents of a cacheable, mappable object.

Definition

```
struct address_space {
    struct inode          *host;
    struct xarray         i_pages;
    gfp_t gfp_mask;
    atomic_t i_mmap_writable;
#ifdef CONFIG_READ_ONLY_THP_FOR_FS;
    atomic_t nr_thps;
#endif;
    struct rb_root_cached i_mmap;
    struct rw_semaphore  i_mmap_rwsem;
    unsigned long        nrpages;
    unsigned long        nrexceptional;
    pgoff_t writeback_index;
    const struct address_space_operations *a_ops;
    unsigned long        flags;
    errseq_t wb_err;
    spinlock_t private_lock;
    struct list_head     private_list;
    void *private_data;
};
```

Members

host Owner, either the inode or the block_device.

i_pages Cached pages.

gfp_mask Memory allocation flags to use for allocating pages.

i_mmap_writable Number of VM_SHARED mappings.

nr_thps Number of THPs in the pagecache (non-shmem only).

i_mmap Tree of private and shared mappings.

i_mmap_rwsem Protects **i_mmap** and **i_mmap_writable**.

nrpages Number of page entries, protected by the **i_pages** lock.

nrexceptional Shadow or DAX entries, protected by the **i_pages** lock.

writeback_index Writeback starts here.

a_ops Methods.

flags Error bits and flags (AS_*).

wb_err The most recent error which has occurred.

private_lock For use by the owner of the address_space.

private_list For use by the owner of the address_space.

private_data For use by the owner of the address_space.

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

Parameters

struct super_block * sb the super we wrote to

Description

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

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

Parameters

struct super_block * sb the super we wrote to

Description

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

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

Parameters

struct super_block * sb the super we wrote to

Description

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

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

Parameters

struct super_block * sb the super we write to

Description

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

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

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

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

Parameters

struct super_block * sb the super we write to

Description

When a process starts handling write page fault, it should embed the operation into `sb_start_pagefault()` - `sb_end_pagefault()` pair to get exclusion against file system freezing. This is needed since the page fault is going to dirty a page.

This function increments number of running page faults preventing freezing. If the file system is already frozen, the function waits until the file system is thawed.

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

mmap_lock -> `sb_start_pagefault`

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

Parameters

struct address_space * mapping mapping in which to set writeback error

int err error to be set in mapping

Description

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

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

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

Parameters

struct address_space * mapping mapping to check for writeback errors

errseq_t since previously-sampled `errseq_t`

Description

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

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

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

Parameters

struct address_space * mapping mapping to be sampled

Description

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

`errseq_t` **file_sample_sb_err**(struct file * file)
sample the current `errseq_t` to test for later errors

Parameters

struct file * file undescribed

Description

Grab the most current superblock-level `errseq_t` value for the given struct file.

The Directory Cache

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

Parameters

struct inode * inode inode to find an alias for

Description

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

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

Parameters

struct super_block * sb superblock

Description

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

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

Parameters

const struct path * parent path to check.

Description

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

void shrink_dcache_parent(struct dentry * parent)
prune dcache

Parameters

struct dentry * parent parent of entries to prune

Description

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

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

Parameters

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

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

Parameters

`struct dentry * parent` parent of entry to allocate

`const struct qstr * name` qstr of the name

Description

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

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

Parameters

`struct dentry * entry` dentry to complete

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

Description

Fill in inode information in the entry.

This turns negative dentries into productive full members of society.

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

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

Parameters

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

Description

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

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

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

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

Parameters

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

Description

Obtain an IS_ROOT dentry for the root of a filesystem.

We must ensure that directory inodes only ever have one dentry. If a dentry is found, that is returned instead of allocating a new one.

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

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

Parameters

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

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

struct qstr * name the case-exact name to be associated with the returned dentry

Description

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

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

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

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

Parameters

const struct dentry * parent parent dentry

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

Return

dentry, or NULL

Description

d_lookup searches the children of the parent dentry for the name in question. If the dentry is found its reference count is incremented and the dentry is returned. The caller must use dput to free the entry when it has finished using it. NULL is returned if the dentry does not exist.

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

Parameters

struct dentry * dir Directory to search in

struct qstr * name qstr of name we wish to find

Description

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

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

Parameters

struct dentry * dentry The dentry to delete

Description

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

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

Parameters

struct dentry * entry dentry to add to the hash

Description

Adds a dentry to the hash according to its name.

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

Parameters

struct dentry * entry dentry to add

struct inode * inode The inode to attach to this dentry

Description

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

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

Parameters

struct dentry * entry dentry to add

struct inode * inode The inode to go with this dentry

Description

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

Parent directory should be locked.

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

Parameters

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

Description

If inode is a directory and has an IS_ROOT alias, then d_move that in place of the given dentry and return it, else simply d_add the inode to the dentry and return NULL.

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

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

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

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

bool **is_subdir**(struct dentry * new_dentry, struct dentry * old_dentry)
is new dentry a subdirectory of old_dentry

Parameters

struct dentry * new_dentry new dentry
struct dentry * old_dentry old dentry

Description

Returns true if new_dentry is a subdirectory of the parent (at any depth). Returns false otherwise. Caller must ensure that "new_dentry" is pinned before calling is_subdir()

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

Parameters

struct dentry * dentry dentry to get a reference to

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

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

Parameters

const struct dentry * dentry entry to check

Returns true if the dentry passed is not currently hashed.

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

Parameters

const struct dentry * dentry The dentry in question

Description

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

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

bool **d_really_is_positive**(const struct dentry * dentry)

Determine if a dentry is really positive (ignoring fallthroughs)

Parameters

const struct dentry * dentry The dentry in question

Description

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

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

struct inode * **d_inode**(const struct dentry * dentry)

Get the actual inode of this dentry

Parameters

const struct dentry * dentry The dentry to query

Description

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

struct inode * **d_inode_rcu**(const struct dentry * dentry)

Get the actual inode of this dentry with `READ_ONCE()`

Parameters

const struct dentry * dentry The dentry to query

Description

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

struct inode * **d_backing_inode**(const struct dentry * upper)

Get upper or lower inode we should be using

Parameters

const struct dentry * upper The upper layer

Description

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

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

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

Parameters

struct dentry * upper The upper layer

Description

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

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

```
struct dentry * d_real(struct dentry * dentry, const struct inode * inode)
    Return the real dentry
```

Parameters

struct dentry * dentry the dentry to query

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

Description

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

See also: [Documentation/filesystems/vfs.rst](#)

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

Parameters

const struct dentry * dentry The dentry to query

Description

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

Inode Handling

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

Parameters

struct super_block * sb superblock inode belongs to

struct inode * inode inode to initialise

Description

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

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

Parameters

struct inode * inode inode

Description

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

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

Parameters

struct inode * inode inode

Description

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

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

Parameters

struct inode * inode inode

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

Description

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

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

Parameters

struct inode * inode inode

Description

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

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

Parameters

struct inode * inode inode to add

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

Parameters

struct inode * inode unhashed inode

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

Add an inode to the inode hash for this superblock.

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

Parameters

struct inode * inode inode to unhash

Remove an inode from the superblock.

```
void evict_inodes(struct super_block * sb)
    evict all evictable inodes for a superblock
```

Parameters

struct super_block * sb superblock to operate on

Description

Make sure that no inodes with zero refcount are retained. This is called by superblock shutdown after having `SB_ACTIVE` flag removed, so any inode reaching zero refcount during or after that call will be immediately evicted.

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

Parameters

struct super_block * sb superblock

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

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

Parameters

struct inode * inode new inode to unlock

Description

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

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

Parameters

struct inode * inode1 first inode to lock

struct inode * inode2 second inode to lock

Description

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

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

Parameters

struct inode * inode1 first inode to unlock

struct inode * inode2 second inode to unlock

struct inode * **inode_insert5**(struct inode *inode, unsigned long hashval,
int (*test)(struct inode *, void *), int (*set)
(struct inode *, void *), void * data)
obtain an inode from a mounted file system

Parameters

struct inode * inode pre-allocated inode to use for insert to cache

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

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

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

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

Description

Search for the inode specified by **hashval** and **data** in the inode cache, and if present it is return it with an increased reference count. This is a variant of `iget5_locked()` for callers that don't want to fail on memory allocation of inode.

If the inode is not in cache, insert the pre-allocated inode to cache and return it locked, hashed, and with the `I_NEW` flag set. The file system gets to fill it in before unlocking it via `unlock_new_inode()`.

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

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

Parameters

struct super_block * sb super block of file system

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

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

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

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

Description

Search for the inode specified by **hashval** and **data** in the inode cache, and if present it is return it with an increased reference count. This is a generalized version of `iget_locked()` for file systems where the inode number is not sufficient for unique identification of an inode.

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

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

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

Parameters

struct super_block * sb super block of file system

unsigned long ino inode number to get

Description

Search for the inode specified by **ino** in the inode cache and if present return it with an increased reference count. This is for file systems where the inode number is sufficient for unique identification of an inode.

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

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

Parameters

struct super_block * sb superblock

ino_t max_reserved highest reserved inode number

Obtain an inode number that is unique on the system for a given superblock. This is used by file systems that have no natural permanent inode numbering

system. An inode number is returned that is higher than the reserved limit but unique.

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

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

Parameters

struct super_block * sb super block of file system to search

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

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

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

Description

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

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

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.

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

Parameters

struct super_block * sb super block of file system to search

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

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

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

Description

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

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

Note

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

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

Parameters

struct super_block * sb super block of file system to search

unsigned long ino inode number to search for

Description

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

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

Parameters

struct super_block * sb super block of file system to search

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

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

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

Description

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

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

struct inode * **find_inode_rcu**(struct super_block * sb, unsigned
long hashval, int (*test)(struct inode *,
void *), void * data)
find an inode in the inode cache

Parameters

struct super_block * sb Super block of file system to search

unsigned long hashval Key to hash

int (*)(struct inode *, void *) test Function to test match on an inode

void * data Data for test function

Description

Search for the inode specified by **hashval** and **data** in the inode cache, where the helper function **test** will return 0 if the inode does not match and 1 if it does. The **test** function must be responsible for taking the `i_lock` spin_lock and checking `i_state` for an inode being freed or being initialized.

If successful, this will return the inode for which the **test** function returned 1 and NULL otherwise.

The **test** function is not permitted to take a ref on any inode presented. It is also not permitted to sleep.

The caller must hold the RCU read lock.

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

Find an inode in the inode cache

Parameters

struct super_block * sb Super block of file system to search

unsigned long ino The inode number to match

Description

Search for the inode specified by **hashval** and **data** in the inode cache, where the helper function **test** will return 0 if the inode does not match and 1 if it does. The **test** function must be responsible for taking the `i_lock` spin_lock and checking `i_state` for an inode being freed or being initialized.

If successful, this will return the inode for which the **test** function returned 1 and NULL otherwise.

The **test** function is not permitted to take a ref on any inode presented. It is also not permitted to sleep.

The caller must hold the RCU read lock.

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

Parameters

struct inode * inode inode to put

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.

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

Parameters

struct inode * inode inode owning the block number being requested

sector_t * block pointer containing the block to find

Replaces the value in `*block` with the block number on the device holding corresponding to the requested block number in the file. That is, asked for

block 4 of inode 1 the function will replace the 4 in *block, with disk block relative to the disk start that holds that block of the file.

Returns -EINVAL in case of error, 0 otherwise. If mapping falls into a hole, returns 0 and *block is also set to 0.

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

Parameters

struct file * file file accessed

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

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

Parameters

struct inode * inode New inode

const struct inode * dir Directory inode

umode_t mode mode of the new inode

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

Parameters

const struct inode * inode inode being checked

Description

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

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

Parameters

struct inode * inode inode to wait for

Description

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

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

struct timespec64 **timestamp_truncate**(struct timespec64 t, struct inode
* inode)
Truncate timespec to a granularity

Parameters

struct timespec64 t Timespec

struct inode * inode inode being updated

Description

Truncate a timespec to the granularity supported by the fs containing the inode. Always rounds down. gran must not be 0 nor greater than a second (NSEC_PER_SEC, or 10^9 ns).

struct timespec64 current_time(struct inode * inode)
Return FS time

Parameters

struct inode * inode inode.

Description

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

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

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

Parameters

struct inode * inode Inode to mark bad

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

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

Parameters

struct inode * inode inode to test

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

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

Parameters

struct inode * inode The inode to discard

Description

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

Registration and Superblocks

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

Parameters

struct super_block * s superblock to deactivate

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

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

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

Parameters

struct super_block * s superblock to deactivate

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

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

Parameters

struct super_block * sb superblock to kill

`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.

struct super_block * **sget_fc**(struct fs_context * fc, int (*test)(struct super_block *, struct fs_context *), int (*set)(struct super_block *, struct fs_context *))

Find or create a superblock

Parameters

struct fs_context * fc Filesystem context.

int (*)(struct super_block *, struct fs_context *) test Comparison callback

int (*)(struct super_block *, struct fs_context *) set Setup callback

Description

Find or create a superblock using the parameters stored in the filesystem context and the two callback functions.

If an extant superblock is matched, then that will be returned with an elevated reference count that the caller must transfer or discard.

If no match is made, a new superblock will be allocated and basic initialisation will be performed (`s_type`, `s_fs_info` and `s_id` will be set and the `set()` callback will be invoked), the superblock will be published and it will be returned in a partially constructed state with `SB_BORN` and `SB_ACTIVE` as yet unset.

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

Parameters

struct file_system_type * type filesystem type superblock should belong to

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

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

int flags mount flags

void * data argument to each of them

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

Parameters

struct file_system_type * type fs type

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

void * arg argument to pass to it

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

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

Parameters

struct block_device * bdev device to get the superblock for

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

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

Parameters

struct block_device * bdev device to get the superblock for

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

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

Parameters

struct block_device * bdev device to get the superblock for

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

```
int get_anon_bdev(dev_t * p)
    Allocate a block device for filesystems which don't have one.
```

Parameters

dev_t * p Pointer to a dev_t.

Description

Filesystems which don't use real block devices can call this function to allocate a virtual block device.

Context

Any context. Frequently called while holding sb_lock.

Return

0 on success, -EMFILE if there are no anonymous bdevs left or -ENOMEM if memory allocation failed.

```
int vfs_get_super(struct fs_context * fc, enum vfs_get_super_keying keying,
                  int (*fill_super)(struct super_block *sb, struct fs_context
                  *fc))
    Get a superblock with a search key set in s_fs_info.
```

Parameters

struct fs_context * fc The filesystem context holding the parameters

enum vfs_get_super_keying keying How to distinguish superblocks

int (*)(struct super_block *sb, struct fs_context *fc) fill_super
Helper to initialise a new superblock

Description

Search for a superblock and create a new one if not found. The search criterion is controlled by **keying**. If the search fails, a new superblock is created and **fill_super()** is called to initialise it.

keying can take one of a number of values:

- (1) **vfs_get_single_super** - Only one superblock of this type may exist on the system. This is typically used for special system filesystems.
- (2) **vfs_get_keyed_super** - Multiple superblocks may exist, but they must have distinct keys (where the key is in s_fs_info). Searching for the same key again will turn up the superblock for that key.

- (3) `vfs_get_independent_super` - Multiple superblocks may exist and are unkeyed. Each call will get a new superblock.

A permissions check is made by `sget_fc()` unless we're getting a superblock for a kernel-internal mount or a submount.

```
int get_tree_bdev(struct fs_context * fc, int (*fill_super)(struct super_block  
*, struct fs_context *))
```

Get a superblock based on a single block device

Parameters

struct fs_context * fc The filesystem context holding the parameters

int (*)(struct super_block *, struct fs_context *) fill_super Helper to initialise a new superblock

```
int vfs_get_tree(struct fs_context * fc)
```

Get the mountable root

Parameters

struct fs_context * fc The superblock configuration context.

Description

The filesystem is invoked to get or create a superblock which can then later be used for mounting. The filesystem places a pointer to the root to be used for mounting in **fc->root**.

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

lock the filesystem and force it into a consistent state

Parameters

struct super_block * sb the super to lock

Description

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

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

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

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

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

`SB_FREEZE_FS`: The file system is frozen. Now all internal sources of fs modification are blocked (e.g. XFS preallocation truncation on inode reclaim). This is usually implemented by blocking new transactions for filesystems that have them and need this additional guard. After all internal writers are finished we call `->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.

File Locks

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

Parameters

struct file_lock * waiter the lock which was waiting

lockd/nfsd need to disconnect the lock while working on it.

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

Parameters

struct file * filp The file to apply the lock to

struct file_lock * fl The lock to be applied

struct file_lock * conflock Place to return a copy of the conflicting lock, if found.

Description

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

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

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

Parameters

struct inode * inode the file to check

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

loff_t start first byte in the file to check

loff_t end lastbyte in the file to check

unsigned char type F_WRLCK for a write lock, else F_RDLCK

Description

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

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

Parameters

struct inode * inode the inode of the file to return

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

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

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

void **lease_get_mtime**(struct inode * inode, struct timespec64 * time)
update modified time of an inode with exclusive lease

Parameters

struct inode * inode the inode

struct timespec64 * time pointer to a timespec which contains the last modified time

Description

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

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

Parameters

struct file * filp file pointer

long arg type of lease to obtain

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

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

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

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

Parameters

struct file * filp file pointer

long arg type of lease to obtain

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

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

Description

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

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

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

Parameters

struct inode * inode inode of the file to apply to

struct file_lock * fl The lock to be applied

Description

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

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

Parameters

struct file * filp The file to test lock for

struct file_lock * fl The lock to test; also used to hold result

Description

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

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

Parameters

struct file * filp The file to apply the lock to

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

struct file_lock * fl The lock to be applied

struct file_lock * conf Place to return a copy of the conflicting lock, if found.

Description

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

If the filesystem defines a private ->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.

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

Parameters

struct file * filp The file to apply the unblock to

struct file_lock * fl The lock to be unblocked

Description

Used by lock managers to cancel blocked requests

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

Parameters

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

struct file_lock * fl The lock to be applied

Description

Apply a POSIX style lock request to an inode.

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

Parameters

struct file * file the file to check

Description

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

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

Parameters

struct file * filp the file

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

`F_RDLCK` to indicate a shared lease is held.

`F_WRLCK` to indicate an exclusive lease is held.

F_UNLCK to indicate no lease is held.

(if a lease break is pending):

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

F_UNLCK to indicate the lease needs to be removed.

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

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

Parameters

struct file * filp file to check

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

int flags current lock flags

Description

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

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

Parameters

unsigned int fd open file descriptor

struct file * filp file pointer

long arg type of lease to obtain

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

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

Parameters

struct inode * inode inode of the file to apply to

struct file_lock * fl The lock to be applied

Description

Apply a FLOCK style lock request to an inode.

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

Parameters

unsigned int fd the file descriptor to lock.

unsigned int cmd the type of lock to apply.

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.

pid_t locks_translate_pid(struct file_lock * fl, struct pid_namespace * ns)
translate a file_lock's fl_pid number into a namespace

Parameters

struct file_lock * fl The file_lock who's fl_pid should be translated

struct pid_namespace * ns The namespace into which the pid should be translated

Description

Used to translate a fl_pid into a namespace virtual pid number

Other Functions

void mpage_readahead(struct readahead_control * rac,
get_block_t get_block)
start reads against pages

Parameters

struct readahead_control * rac Describes which pages to read.

get_block_t get_block The filesystem's block mapper function.

Description

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

If anything unusual happens, such as:

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

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

BH_Boundary explanation:

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

So an mpage read of the first 16 blocks of an ext2 file will cause I/O to be submitted in the following order:

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

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

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

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

```
int mpage_writepages(struct address_space * mapping, struct write-
                    back_control * wbc, get_block_t get_block)
    walk the list of dirty pages of the given address space & writepage() all of
    them
```

Parameters

struct address_space * mapping address space structure to write

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

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

Description

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

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

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

Parameters

struct inode * inode inode to check access rights for

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

Description

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

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

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

Parameters

struct inode * inode Inode to check permission on

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

Description

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

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

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

Parameters

const struct path * path path to get the reference to

Description

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

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

Parameters

const struct path * path path to put the reference to

Description

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

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

Parameters

struct dentry * dentry pointer to dentry of the base directory

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

const char * name pointer to file name

unsigned int flags lookup flags

struct path * path pointer to struct path to fill

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

Parameters

const char * name pathname component to lookup
struct dentry * base base directory to lookup from
int len maximum length **len** should be interpreted to

Description

Look up a dentry by name in the dcache, returning NULL if it does not currently exist. The function does not try to create a dentry.

Note that this routine is purely a helper for filesystem usage and should not be called by generic code.

The caller must hold base->i_mutex.

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

Parameters

const char * name pathname component to lookup
struct dentry * base base directory to lookup from
int len maximum length **len** should be interpreted to

Description

Note that this routine is purely a helper for filesystem usage and should not be called by generic code.

The caller must hold base->i_mutex.

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

Parameters

const char * name pathname component to lookup
struct dentry * base base directory to lookup from
int len maximum length **len** should be interpreted to

Description

Note that this routine is purely a helper for filesystem usage and should not be called by generic code.

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

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

Parameters

struct inode * dir parent directory
struct dentry * dentry victim

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

Description

The caller must hold `dir->i_mutex`.

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

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

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

Parameters

struct dentry * old_dentry object to be linked

struct inode * dir new parent

struct dentry * new_dentry where to create the new link

struct inode ** delegated_inode returns inode needing a delegation break

Description

The caller must hold `dir->i_mutex`

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

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

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

Parameters

struct inode * old_dir parent of source

struct dentry * old_dentry source

struct inode * new_dir parent of destination

struct dentry * new_dentry destination

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

unsigned int flags rename flags

Description

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

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

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

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

- a) we can get into loop creation.
- 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].

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

Parameters

struct dentry * dentry dentry on which to get symbolic link

char __user * buffer user memory pointer

int buflen size of buffer

Description

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

Does not call security hook.

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

Parameters

struct dentry * dentry dentry on which to get symbolic link

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

Description

Calls security hook and `i_op->get_link()` on the supplied inode.

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

Does not work on “special” symlinks like `/proc/$$/fd/N`

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

Parameters

struct address_space * mapping the mapping which wants those buffers written

Description

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

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

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

Parameters

struct buffer_head * bh the buffer_head to mark dirty

Description

`mark_buffer_dirty()` will set the dirty bit against the buffer, then set its backing page dirty, then tag the page as dirty in the page cache 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`, `i_pages` lock and `mapping->host->i_lock`.

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

Parameters

struct block_device * bdev the block_device to read from

sector_t block number of block

unsigned size size (in bytes) to read

gfp_t gfp page allocation flag

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

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

Parameters

struct page * page the page which is affected

unsigned int offset start of the range to invalidate

unsigned int length length of the range to invalidate

Description

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

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

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

Parameters

struct block_device * bdev Block device to clean buffers in

sector_t block Start of a range of blocks to clean

sector_t len Number of blocks to clean

Description

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

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

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

Parameters

int op whether to READ or WRITE

int op_flags req_flag_bits

int nr number of struct `buffer_head`s in the array

struct buffer_head * bhs array of pointers to struct `buffer_head`

Description

`ll_rw_block()` takes an array of pointers to `struct buffer_heads`, and requests an I/O operation on them, either a `REQ_OP_READ` or a `REQ_OP_WRITE`. **op_flags** contains flags modifying the detailed I/O behavior, most notably `REQ_RAHEAD`.

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

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

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

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

Parameters

struct buffer_head * bh struct buffer_head

Description

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

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

Parameters

struct buffer_head * bh struct buffer_head

Description

Returns zero on success and -EIO on error.

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

Parameters

struct bio * bio bio to reset

Description

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

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

Parameters

struct bio * bio the target bio

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

Description

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

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

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

Parameters

gfp_t gfp_mask the GFP_* mask given to the slab allocator

unsigned int nr_iovecs number of iovecs to pre-allocate

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

Description

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

When **bs** is not NULL, if `__GFP_DIRECT_RECLAIM` is set then `bio_alloc` will always be able to allocate a bio. This is due to the mempool guarantees. To make this work, callers must never allocate more than 1 bio at a time from this pool. Callers that need to allocate more than 1 bio must always submit the previously allocated bio for IO before attempting to allocate a new one. Failure to do so can cause deadlocks under memory pressure.

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

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

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

Return

Pointer to new bio on success, NULL on failure.

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

Parameters

struct bio * bio bio to release reference to

Description

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

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

Parameters

struct bio * bio destination bio

struct bio * bio_src bio to clone

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

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

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

Parameters

struct bio * bio bio to clone

gfp_t gfp_mask allocation priority

struct bio_set * bs bio_set to allocate from

Like `__bio_clone_fast`, only also allocates the returned bio

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

Parameters

struct request_queue * q the target queue

struct bio * bio destination bio

struct page * page page to add

unsigned int len vec entry length

unsigned int offset vec entry offset

Description

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

This should only be used by passthrough bios.

```
bool __bio_try_merge_page(struct bio * bio, struct page * page, unsigned
    int len, unsigned int off, bool * same_page)
    try appending data to an existing bvec.
```

Parameters

struct bio * bio destination bio

struct page * page start page to add

unsigned int len length of the data to add

unsigned int off offset of the data relative to **page**

bool * same_page return if the segment has been merged inside the same page

Description

Try to add the data at **page + off** to the last bvec of **bio**. This is a useful optimisation for file systems with a block size smaller than the page size.

Warn if (**len, off**) crosses pages in case that **same_page** is true.

Return true on success or false on failure.

```
void __bio_add_page(struct bio * bio, struct page * page, unsigned int len,
                  unsigned int off)
    add page(s) to a bio in a new segment
```

Parameters

struct bio * bio destination bio

struct page * page start page to add

unsigned int len length of the data to add, may cross pages

unsigned int off offset of the data relative to **page**, may cross pages

Description

Add the data at **page + off** to **bio** as a new bvec. The caller must ensure that **bio** has space for another bvec.

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

Parameters

struct bio * bio destination bio

struct page * page start page to add

unsigned int len vec entry length, may cross pages

unsigned int offset vec entry offset relative to **page**, may cross pages

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

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

Parameters

struct bio * bio bio to add pages to

struct iov_iter * iter iov iterator describing the region to be added

Description

This takes either an iterator pointing to user memory, or one pointing to kernel pages (BVEC iterator). If we're adding user pages, we pin them and map them into the kernel. On IO completion, the caller should put those pages. If we're

adding kernel pages, and the caller told us it's safe to do so, we just have to add the pages to the bio directly. We don't grab an extra reference to those pages (the user should already have that), and we don't put the page on IO completion. The caller needs to check if the bio is flagged `BIO_NO_PAGE_REF` on IO completion. If it isn't, then pages should be released.

The function tries, but does not guarantee, to pin as many pages as fit into the bio, or are requested in `*iter`, whatever is smaller. If MM encounters an error pinning the requested pages, it stops. Error is returned only if 0 pages could be pinned.

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

Parameters

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

Description

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

WARNING: Unlike to how `submit_bio()` is usually used, this function does not result in bio reference to be consumed. The caller must drop the reference on his own.

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

Parameters

struct bio * bio bio to advance

unsigned bytes number of bytes to complete

Description

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

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

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

Parameters

struct bio * dst destination bio

struct bio * src source bio

Description

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).

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

Parameters

struct bio * dst destination bio list

struct bio * src source bio list

Description

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

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

Parameters

struct bio * bio bio

Description

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

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

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

Parameters

struct bio * bio bio to split

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

gfp_t gfp gfp mask

struct bio_set * bs bio set to allocate from

Description

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

Unless this is a discard request the newly allocated bio will point to **bio**'s `bi_io_vec`. It is the caller' s responsibility to ensure that neither **bio** nor **bs** are freed before the split bio.

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

Parameters

struct bio * bio bio to trim

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

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

int **bio_set_init**(struct bio_set * bs, unsigned int pool_size, unsigned int front_pad, int flags)
Initialize a bio_set

Parameters

struct bio_set * bs pool to initialize

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

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

int flags Flags to modify behavior, currently BIOSET_NEED_BVECS and BIOSET_NEED_RESCUER

Description

Set up a **bio_set** to be used with **bio_alloc_bioset**. Allows the caller to ask for a number of bytes to be allocated in front of the bio. Front pad allocation is useful for embedding the bio inside another structure, to avoid allocating extra data to go with the bio. Note that the bio must be embedded at the END of that structure always, or things will break badly. If BIOSET_NEED_BVECS is set in **flags**, a separate pool will be allocated for allocating iovecs. This pool is not needed e.g. for **bio_clone_fast()**. If BIOSET_NEED_RESCUER is set, a workqueue is created which can be used to dispatch queued requests when the mempool runs out of space.

void **bio_disassociate_blkcg**(struct bio * bio)
puts back the blkcg reference if associated

Parameters

struct bio * bio target bio

Description

Helper to disassociate the blkcg from **bio** if a blkcg is associated.

void **bio_associate_blkcg_from_css**(struct bio * bio, struct cgroup_subsys_state * css)
associate a bio with a specified css

Parameters

struct bio * bio target bio

struct cgroup_subsys_state * css target css

Description

Associate **bio** with the blkcg found by combining the css' s blkcg and the request_queue of the **bio**. This falls back to the queue' s root_blkcg if the association fails with the css.

void **bio_associate_blkcg**(struct bio * bio)
associate a bio with a blkcg

Parameters

struct bio * bio target bio

Description

Associate **bio** with the blkcg found from the bio' s css and request_queue. If one is not found, **bio_lookup_blkcg()** creates the blkcg. If a blkcg is already associated, the css is reused and association redone as the request_queue may have changed.

void **bio_clone_blkcg_association**(struct bio * dst, struct bio * src)
 clone blkcg association from src to dst bio

Parameters

struct bio * dst destination bio

struct bio * src source bio

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

Parameters

struct file * file file we initialize

const struct seq_operations * op method table describing the sequence

`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” .

Note

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

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

Parameters

struct file * file the file to read from

char __user * buf the buffer to read to

size_t size the maximum number of bytes to read

loff_t * ppos the current position in the file

Ready-made `->f_op->read()`

loff_t **seq_llseek**(struct file * file, loff_t offset, int whence)
`->llseek()` method for sequential files.

Parameters

struct file * file the file in question

loff_t offset new position

int whence 0 for absolute, 1 for relative position

Ready-made `->f_op->llseek()`

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

Parameters

struct inode * inode its inode

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

struct file * file file in question

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

Parameters

struct seq_file * m target buffer

const char * s string

const char * esc set of characters that need escaping

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

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

Parameters

char * s buffer start

const char * p beginning of path in above buffer

const char * esc set of characters that need escaping

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

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

Parameters

struct seq_file * m the seq_file handle

const struct path * path the struct path to print

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

Description

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

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

Parameters

struct seq_file * m the seq_file handle

struct file * file the struct file to print

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

Description

return the absolute path to the file.

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

Parameters

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

const void * data data address

size_t len number of bytes

Description

Return 0 on success, non-zero otherwise.

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

Parameters

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

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

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

Parameters

struct hlist_head * head the head of the hlist

loff_t pos the start position of the sequence

Description

Called at seq_file->op->start().

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

Parameters

struct hlist_head * head the head of the hlist

loff_t pos the start position of the sequence

Description

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

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

Parameters

void * v the current iterator

struct hlist_head * head the head of the hlist

loff_t * ppos the current position

Description

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

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

Parameters

struct hlist_head * head the head of the hlist

loff_t pos the start position of the sequence

Description

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

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

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

Parameters

struct hlist_head * head the head of the hlist

loff_t pos the start position of the sequence

Description

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

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

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

Parameters

void * v the current iterator

struct hlist_head * head the head of the hlist

loff_t * ppos the current position

Description

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

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

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

Parameters

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

int * cpu pointer to cpu “cursor”

loff_t pos start position of sequence

Description

Called at seq_file->op->start().

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

Parameters

void * v pointer to current hlist_node

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

int * cpu pointer to cpu “cursor”

loff_t * pos start position of sequence

Description

Called at seq_file->op->next().

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

Parameters

struct file_system_type * fs the file system structure

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

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

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

Parameters

struct file_system_type * fs filesystem to unregister

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

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

```
void wbc_attach_and_unlock_inode(struct writeback_control * wbc, struct
                                inode * inode)
    associate wbc with target inode and unlock it
```

Parameters

struct writeback_control * wbc writeback_control of interest

struct inode * inode target inode

Description

inode is locked and about to be written back under the control of **wbc**. Record **inode**'s writeback context into **wbc** and unlock the `i_lock`. On writeback completion, `wbc_detach_inode()` should be called. This is used to track the cgroup writeback context.

```
void wbc_detach_inode(struct writeback_control * wbc)
    disassociate wbc from inode and perform foreign detection
```

Parameters

struct writeback_control * wbc writeback_control of the just finished writeback

Description

To be called after a writeback attempt of an inode finishes and undoes `wbc_attach_and_unlock_inode()`. Can be called under any context.

As concurrent write sharing of an inode is expected to be very rare and memcg only tracks page ownership on first-use basis severely confining the usefulness of such sharing, cgroup writeback tracks ownership per-inode. While the support for concurrent write sharing of an inode is deemed unnecessary, an inode being written to by different cgroups at different points in time is a lot more common, and, more importantly, charging only by first-use can too readily lead to grossly incorrect behaviors (single foreign page can lead to gigabytes of writeback to be incorrectly attributed).

To resolve this issue, cgroup writeback detects the majority dirtier of an inode and transfers the ownership to it. To avoid unnecessary oscillation, the detection mechanism keeps track of history and gives out the switch verdict only if the foreign usage pattern is stable over a certain amount of time and/or writeback attempts.

On each writeback attempt, **wbc** tries to detect the majority writer using Boyer-Moore majority vote algorithm. In addition to the byte count from the majority voting, it also counts the bytes written for the current `wb` and the last round's winner `wb` (max of last round's current `wb`, the winner from two rounds ago, and the last round's majority candidate). Keeping track of the historical winner helps the algorithm to semi-reliably detect the most active writer even when it's not the absolute majority.

Once the winner of the round is determined, whether the winner is foreign or not and how much IO time the round consumed is recorded in `inode->i_wb_frn_history`.

If the amount of recorded foreign IO time is over a certain threshold, the switch verdict is given.

```
void wbc_account_cgroup_owner(struct writeback_control *wbc, struct
                               page *page, size_t bytes)
    account writeback to update inode cgroup ownership
```

Parameters

struct writeback_control * wbc writeback_control of the writeback in progress

struct page * page page being written out

size_t bytes number of bytes being written out

Description

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

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

Parameters

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

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

Description

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

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

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

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

Parameters

struct inode * inode inode to mark

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

Description

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

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

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

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

Note that for blockdevs, `inode->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.

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

Parameters

struct super_block * sb the superblock

unsigned long nr the number of pages to write

enum wb_reason reason reason why some writeback work initiated

Description

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

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

Parameters

struct super_block * sb the superblock

enum wb_reason reason reason why some writeback work was initiated

Description

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

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

Parameters

struct super_block * sb the superblock

enum wb_reason reason reason why some writeback work was initiated

Description

Invoke `__writeback_inodes_sb_nr` if no writeback is currently underway.

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

Parameters

struct super_block * sb the superblock

Description

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

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

Parameters

struct inode * inode inode to write to disk

int sync whether the write should be synchronous or not

Description

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

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

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

Parameters

struct inode * inode the inode to sync

struct writeback_control * wbc controls the writeback mode

Description

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

The caller must have a ref on the inode.

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

Parameters

struct inode * inode the inode to sync

int wait wait for I/O to complete.

Description

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

Note

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

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

- lock a filesystem and force it into a consistent state

Parameters

struct block_device * bdev blockdevice to lock

Description

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

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

- unlock filesystem

Parameters

struct block_device * bdev blockdevice to unlock

struct super_block * sb associated superblock

Description

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

struct block_device * bdgrab(struct block_device * bdev)

- Grab a reference to an already referenced block device

Parameters

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

struct block_device * bd_start_claiming(struct block_device * bdev, void * holder)
start claiming a block device

Parameters

struct block_device * bdev block device of interest

void * holder holder trying to claim **bdev**

Description

bdev is about to be opened exclusively. Check **bdev** can be opened exclusively and mark that an exclusive open is in progress. Each successful call to this function must be matched with a call to either `bd_finish_claiming()` or `bd_abort_claiming()` (which do not fail).

This function is used to gain exclusive access to the block device without actually causing other exclusive open attempts to fail. It should be used when the open sequence itself requires exclusive access but may subsequently fail.

Context

Might sleep.

Return

Pointer to the block device containing **bdev** on success, `ERR_PTR()` value on failure.

void bd_finish_claiming(struct block_device * bdev, struct block_device * whole, void * holder)
finish claiming of a block device

Parameters

struct block_device * bdev block device of interest

struct block_device * whole whole block device (returned from `bd_start_claiming()`)

void * holder holder that has claimed **bdev**

Description

Finish exclusive open of a block device. Mark the device as exclusively open by the holder and wake up all waiters for exclusive open to finish.

```
void bd_abort_claiming(struct block_device * bdev, struct block_device
                       * whole, void * holder)
    abort claiming of a block device
```

Parameters

struct block_device * bdev block device of interest

struct block_device * whole whole block device (returned from `bd_start_claiming()`)

void * holder holder that has claimed **bdev**

Description

Abort claiming of a block device when the exclusive open failed. This can be also used when exclusive open is not actually desired and we just needed to block other exclusive openers for a while.

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

Parameters

struct block_device * bdev the claimed slave bdev

struct gendisk * disk the holding disk

Description

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

This functions creates the following sysfs symlinks.

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

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

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

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

Context

Might sleep.

Return

0 on success, `-errno` on failure.

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

Parameters

struct block_device * bdev the calimed slave bdev

struct gendisk * disk the holding disk

Description

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

Context

Might sleep.

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

Parameters

struct gendisk * disk struct gendisk to be revalidated

Description

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

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

Parameters

struct block_device * bdev block_device to open

fmode_t mode FMODE_* mask

void * holder exclusive holder identifier

Description

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

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

Context

Might sleep.

Return

0 on success, -errno on failure.

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

Parameters

const char * path path to the block device to open

fmode_t mode FMODE_* mask

void * holder exclusive holder identifier

Description

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

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

Context

Might sleep.

Return

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

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

Parameters

dev_t dev device number of block device to open

fmode_t mode `FMODE_*` mask

void * holder exclusive holder identifier

Description

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

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

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

Context

Might sleep.

Return

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

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

Parameters

const char * pathname special file representing the block device

Description

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

```
struct file * anon_inode_getfile(const char * name,  const struct
                                file_operations * fops,  void * priv,
                                int flags)
    creates a new file instance by hooking it up to an anonymous inode, and a
    dentry that describe the "class" of the file
```

Parameters

const char * name [in] name of the “class” of the new file
const struct file_operations * fops [in] file operations for the new file
void * priv [in] private data for the new file (will be file’ s private_data)
int flags [in] flags

Description

Creates a new file by hooking it on a single inode. This is useful for files that do not need to have a full-fledged inode in order to operate correctly. All the files created with `anon_inode_getfile()` will share a single inode, hence saving memory and avoiding code duplication for the file/inode/dentry setup. Returns the newly created file* or an error pointer.

int **anon_inode_getfd**(const char * name, const struct file_operations * fops, void * priv, int flags)
creates a new file instance by hooking it up to an anonymous inode, and a dentry that describe the “class” of the file

Parameters

const char * name [in] name of the “class” of the new file
const struct file_operations * fops [in] file operations for the new file
void * priv [in] private data for the new file (will be file’ s private_data)
int flags [in] flags

Description

Creates a new file by hooking it on a single inode. This is useful for files that do not need to have a full-fledged inode in order to operate correctly. All the files created with `anon_inode_getfd()` will share a single inode, hence saving memory and avoiding code duplication for the file/inode/dentry setup. Returns new descriptor or an error code.

int **setattr_prepare**(struct dentry * dentry, struct iattr * attr)
check if attribute changes to a dentry are allowed

Parameters

struct dentry * dentry dentry to check
struct iattr * attr attributes to change

Description

Check if we are allowed to change the attributes contained in **attr** in the given dentry. This includes the normal unix access permission checks, as well as checks for rlimits and others. The function also clears SGID bit from mode if user is not allowed to set it. Also file capabilities and IMA extended attributes are cleared if `ATTR_KILL_PRIV` is set.

Should be called as the first thing in `->setattr` implementations, possibly after taking additional locks.

int **inode_newsize_ok**(const struct inode * inode, loff_t offset)
may this inode be truncated to a given size

Parameters

const struct inode * inode the inode to be truncated

loff_t offset the new size to assign to the inode

Description

inode_newsize_ok must be called with i_mutex held.

inode_newsize_ok will check filesystem limits and ulimits to check that the new inode size is within limits. inode_newsize_ok will also send SIGXFSZ when necessary. Caller must not proceed with inode size change if failure is returned. **inode** must be a file (not directory), with appropriate permissions to allow truncate (inode_newsize_ok does NOT check these conditions).

Return

0 on success, -ve errno on failure

void **setattr_copy**(struct inode * inode, const struct iattr * attr)
copy simple metadata updates into the generic inode

Parameters

struct inode * inode the inode to be updated

const struct iattr * attr the new attributes

Description

setattr_copy must be called with i_mutex held.

setattr_copy updates the inode's metadata with that specified in attr. Noticeably missing is inode size update, which is more complex as it requires pagecache updates.

The inode is not marked as dirty after this operation. The rationale is that for "simple" filesystems, the struct inode is the inode storage. The caller is free to mark the inode dirty afterwards if needed.

int **notify_change**(struct dentry * dentry, struct iattr * attr, struct inode
** delegated_inode)
modify attributes of a filesystem object

Parameters

struct dentry * dentry object affected

struct iattr * attr new attributes

struct inode ** delegated_inode returns inode, if the inode is delegated

Description

The caller must hold the i_mutex on the affected object.

If notify_change discovers a delegation 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. Also, passing NULL is fine for callers holding the file open for write, as there can be no conflicting delegation in that case.

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

Parameters

`const struct path * path` path to report

`char * buf` buffer to return value in

`int buflen` buffer length

Description

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

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

"buflen" should be positive.

`struct page * dax_layout_busy_page(struct address_space * mapping)`
find first pinned page in **mapping**

Parameters

`struct address_space * mapping` address space to scan for a page with ref count > 1

Description

DAX requires ZONE_DEVICE mapped pages. These pages are never 'onlined' to the page allocator so they are considered idle when `page->count == 1`. A filesystem uses this interface to determine if any page in the mapping is busy, i.e. for DMA, or other `get_user_pages()` usages.

It is expected that the filesystem is holding locks to block the establishment of new mappings in this `address_space`. I.e. it expects to be able to run `unmap_mapping_range()` and subsequently not race `mapping_mapped()` becoming true.

`ssize_t dax_iomap_rw(struct kiocb * iocb, struct iov_iter * iter, const struct iomap_ops * ops)`
Perform I/O to a DAX file

Parameters

`struct kiocb * iocb` The control block for this I/O

`struct iov_iter * iter` The addresses to do I/O from or to

`const struct iomap_ops * ops` iomap ops passed from the file system

Description

This function performs read and write operations to directly mapped persistent memory. The callers needs to take care of read/write exclusion and evicting any page cache pages in the region under I/O.

```
vm_fault_t dax_iomap_fault(struct vm_fault *vmf, enum
                           page_entry_size pe_size, pfn_t *pfnp, int
                           *iomap_errp, const struct iomap_ops *ops)
    handle a page fault on a DAX file
```

Parameters

struct vm_fault * vmf The description of the fault

enum page_entry_size pe_size Size of the page to fault in

pfn_t * pfnp PFN to insert for synchronous faults if fsync is required

int * iomap_errp Storage for detailed error code in case of error

const struct iomap_ops * ops Iomap ops passed from the file system

Description

When a page fault occurs, filesystems may call this helper in their fault handler for DAX files. `dax_iomap_fault()` assumes the caller has done all the necessary locking for page fault to proceed successfully.

```
vm_fault_t dax_finish_sync_fault(struct vm_fault *vmf, enum
                                  page_entry_size pe_size, pfn_t pfn)
    finish synchronous page fault
```

Parameters

struct vm_fault * vmf The description of the fault

enum page_entry_size pe_size Size of entry to be inserted

pfn_t pfn PFN to insert

Description

This function ensures that the file range touched by the page fault is stored persistently on the media and handles inserting of appropriate page table entry.

```
void dio_end_io(struct bio * bio)
    handle the end io action for the given bio
```

Parameters

struct bio * bio The direct io bio thats being completed

Description

This is meant to be called by any filesystem that uses their own `dio_submit_t` so that the DIO specific endio actions are dealt with after the filesystem has done it's completion work.

```
int simple_setattr(struct dentry * dentry, struct iattr * iattr)
    setattr for simple filesystem
```

Parameters

struct dentry * dentry dentry

struct iattr * iattr iattr structure

Description

Returns 0 on success, -error on failure.

simple_setattr is a simple ->setattr implementation without a proper implementation of size changes.

It can either be used for in-memory filesystems or special files on simple regular filesystems. Anything that needs to change on-disk or wire state on size changes needs its own setattr method.

int **simple_write_end**(struct file * file, struct address_space * mapping,
 loff_t pos, unsigned len, unsigned copied, struct page
 * page, void * fsdata)
 .write_end helper for non-block-device FSes

Parameters

struct file * file See .write_end of address_space_operations

struct address_space * mapping “

loff_t pos “

unsigned len “

unsigned copied “

struct page * page “

void * fsdata “

Description

simple_write_end does the minimum needed for updating a page after writing is done. It has the same API signature as the .write_end of address_space_operations vector. So it can just be set onto .write_end for FSes that don't need any other processing. i_mutex is assumed to be held. Block based filesystems should use generic_write_end(). Use ONLY with simple_readpage()

NOTE

Even though i_size might get updated by this function, mark_inode_dirty is not called, so a filesystem that actually does store data in .write_inode should extend on what's done here with a call to mark_inode_dirty() in the case that i_size has changed.

ssize_t **simple_read_from_buffer**(void __user * to, size_t count,
 loff_t * ppos, const void * from,
 size_t available)
 copy data from the buffer to user space

Parameters

void __user * to the user space buffer to read to

size_t count the maximum number of bytes to read

loff_t * ppos the current position in the buffer

const void * from the buffer to read from

size_t available the size of the buffer

Description

The `simple_read_from_buffer()` function reads up to **count** bytes from the buffer **from** at offset **ppos** into the user space address starting at **to**.

On success, the number of bytes read is returned and the offset **ppos** is advanced by this number, or negative value is returned on error.

```
ssize_t simple_write_to_buffer(void * to, size_t available, loff_t * ppos,
                              const void __user * from, size_t count)
    copy data from user space to the buffer
```

Parameters

void * to the buffer to write to

size_t available the size of the buffer

loff_t * ppos the current position in the buffer

const void __user * from the user space buffer to read from

size_t count the maximum number of bytes to read

Description

The `simple_write_to_buffer()` function reads up to **count** bytes from the user space address starting at **from** into the buffer **to** at offset **ppos**.

On success, the number of bytes written is returned and the offset **ppos** is advanced by this number, or negative value is returned on error.

```
ssize_t memory_read_from_buffer(void * to, size_t count, loff_t * ppos, const
                               void * from, size_t available)
    copy data from the buffer
```

Parameters

void * to the kernel space buffer to read to

size_t count the maximum number of bytes to read

loff_t * ppos the current position in the buffer

const void * from the buffer to read from

size_t available the size of the buffer

Description

The `memory_read_from_buffer()` function reads up to **count** bytes from the buffer **from** at offset **ppos** into the kernel space address starting at **to**.

On success, the number of bytes read is returned and the offset **ppos** is advanced by this number, or negative value is returned on error.

```
struct dentry * generic_fh_to_dentry(struct super_block * sb, struct
                                     fid * fid, int fh_len, int fh_type,
                                     struct inode *(*get_inode)(struct
                                     super_block *sb, u64 ino, u32 gen))
    generic helper for the fh_to_dentry export operation
```

Parameters

struct super_block * sb filesystem to do the file handle conversion on

struct fid * fid file handle to convert

int fh_len length of the file handle in bytes

int fh_type type of file handle

struct inode *(*)(struct super_block *sb, u64 ino, u32 gen) get_inode
filesystem callback to retrieve inode

Description

This function decodes **fid** as long as it has one of the well-known Linux filehandle types and calls **get_inode** on it to retrieve the inode for the object specified in the file handle.

```
struct dentry * generic_fh_to_parent(struct super_block * sb, struct
                                     fid * fid, int fh_len, int fh_type,
                                     struct inode *(*get_inode)(struct
                                     super_block *sb, u64 ino, u32 gen))
    generic helper for the fh_to_parent export operation
```

Parameters

struct super_block * sb filesystem to do the file handle conversion on

struct fid * fid file handle to convert

int fh_len length of the file handle in bytes

int fh_type type of file handle

struct inode *(*)(struct super_block *sb, u64 ino, u32 gen) get_inode
filesystem callback to retrieve inode

Description

This function decodes **fid** as long as it has one of the well-known Linux filehandle types and calls **get_inode** on it to retrieve the inode for the `_parent_object` specified in the file handle if it is specified in the file handle, or NULL otherwise.

```
int __generic_file_fsync(struct file * file, loff_t start, loff_t end,
                          int datasync)
    generic fsync implementation for simple filesystems
```

Parameters

struct file * file file to synchronize

loff_t start start offset in bytes

loff_t end end offset in bytes (inclusive)

int datasync only synchronize essential metadata if true

Description

This is a generic implementation of the fsync method for simple filesystems which track all non-inode metadata in the buffers list hanging off the `address_space` structure.

int **generic_file_fsync**(struct file * file, loff_t start, loff_t end,
int datasync)
generic fsync implementation for simple filesystems with flush

Parameters

struct file * file file to synchronize

loff_t start start offset in bytes

loff_t end end offset in bytes (inclusive)

int datasync only synchronize essential metadata if true

int **generic_check_addressable**(unsigned blocksize_bits, u64 num_blocks)
Check addressability of file system

Parameters

unsigned blocksize_bits log of file system block size

u64 num_blocks number of blocks in file system

Description

Determine whether a file system with **num_blocks** blocks (and a block size of $2^{blocksize_bits}$) is addressable by the sector_t and page cache of the system. Return 0 if so and -EFBIG otherwise.

int **simple_nosetlease**(struct file * filp, long arg, struct file_lock ** flp, void
** priv)
generic helper for prohibiting leases

Parameters

struct file * filp file pointer

long arg type of lease to obtain

struct file_lock ** flp new lease supplied for insertion

void ** priv private data for lm_setup operation

Description

Generic helper for filesystems that do not wish to allow leases to be set. All arguments are ignored and it just returns -EINVAL.

const char * **simple_get_link**(struct dentry * dentry, struct inode * inode,
struct delayed_call * done)
generic helper to get the target of “fast” symlinks

Parameters

struct dentry * dentry not used here

struct inode * inode the symlink inode

struct delayed_call * done not used here

Description

Generic helper for filesystems to use for symlink inodes where a pointer to the symlink target is stored in ->i_link. NOTE: this isn't normally called, since as an optimization the path lookup code uses any non-NULL ->i_link directly, without calling

->get_link(). But ->get_link() still must be set, to mark the inode_operations as being for a symlink.

Return

the symlink target

```
int posix_acl_update_mode(struct inode * inode, umode_t * mode_p, struct
                           posix_acl ** acl)
    update mode in set_acl
```

Parameters

struct inode * inode target inode

umode_t * mode_p mode (pointer) for update

struct posix_acl ** acl acl pointer

Description

Update the file mode when setting an ACL: compute the new file permission bits based on the ACL. In addition, if the ACL is equivalent to the new file mode, set ***acl** to NULL to indicate that no ACL should be set.

As with chmod, clear the setgid bit if the caller is not in the owning group or capable of CAP_FSETID (see inode_change_ok).

Called from set_acl inode operations.

```
void generic_fillattr(struct inode * inode, struct kstat * stat)
    Fill in the basic attributes from the inode struct
```

Parameters

struct inode * inode Inode to use as the source

struct kstat * stat Where to fill in the attributes

Description

Fill in the basic attributes in the kstat structure from data that's to be found on the VFS inode structure. This is the default if no getattr inode operation is supplied.

```
int vfs_getattr_nosec(const struct path * path, struct kstat * stat,
                       u32 request_mask, unsigned int query_flags)
    getattr without security checks
```

Parameters

const struct path * path file to get attributes from

struct kstat * stat structure to return attributes in

u32 request_mask STATX_xxx flags indicating what the caller wants

unsigned int query_flags Query mode (KSTAT_QUERY_FLAGS)

Description

Get attributes without calling security_inode_getattr.

Currently the only caller other than `vfs_getattr` is internal to the filehandle lookup code, which uses only the inode number and returns no attributes to any user. Any other code probably wants `vfs_getattr`.

`int vfs_statx_fd`(unsigned int fd, struct kstat * stat, u32 request_mask, unsigned int query_flags)
Get the enhanced basic attributes by file descriptor

Parameters

unsigned int fd The file descriptor referring to the file of interest

struct kstat * stat The result structure to fill in.

u32 request_mask STATX_XXX flags indicating what the caller wants

unsigned int query_flags Query mode (KSTAT_QUERY_FLAGS)

Description

This function is a wrapper around `vfs_getattr`(). The main difference is that it uses a file descriptor to determine the file location.

0 will be returned on success, and a -ve error code if unsuccessful.

`int vfs_statx`(int dfd, const char __user * filename, int flags, struct kstat * stat, u32 request_mask)
Get basic and extra attributes by filename

Parameters

int dfd A file descriptor representing the base dir for a relative filename

const char __user * filename The name of the file of interest

int flags Flags to control the query

struct kstat * stat The result structure to fill in.

u32 request_mask STATX_XXX flags indicating what the caller wants

Description

This function is a wrapper around `vfs_getattr`(). The main difference is that it uses a filename and base directory to determine the file location. Additionally, the use of `AT_SYMLINK_NOFOLLOW` in flags will prevent a symlink at the given name from being referenced.

0 will be returned on success, and a -ve error code if unsuccessful.

`int vfs_fsync_range`(struct file * file, loff_t start, loff_t end, int datasync)
helper to sync a range of data & metadata to disk

Parameters

struct file * file file to sync

loff_t start offset in bytes of the beginning of data range to sync

loff_t end offset in bytes of the end of data range (inclusive)

int datasync perform only datasync

Description

Write back data in range **start..**end**** and metadata for **file** to disk. If **datasync** is set only metadata needed to access modified file data is written.

```
int vfs_fsync(struct file * file, int datasync)
    perform a fsync or fdatasync on a file
```

Parameters

struct file * file file to sync

int datasync only perform a fdatasync operation

Description

Write back data and metadata for **file** to disk. If **datasync** is set only metadata needed to access modified file data is written.

```
const char * xattr_full_name(const struct xattr_handler * handler, const
                               char * name)
    Compute full attribute name from suffix
```

Parameters

const struct xattr_handler * handler handler of the xattr_handler operation

const char * name name passed to the xattr_handler operation

Description

The get and set xattr handler operations are called with the remainder of the attribute name after skipping the handler's prefix: for example, "foo" is passed to the get operation of a handler with prefix "user." to get attribute "user.foo". The full name is still "there" in the name though.

Note

the list xattr handler operation when called from the vfs is passed a NULL name; some file systems use this operation internally, with varying semantics.

1.3.2 The proc filesystem

sysctl interface

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

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

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos pointer to the file position

Description

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

Returns 0 on success.

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

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos the current position in the file

Description

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

Returns 0 on success.

int proc_do_large_bitmap(`struct ctl_table * table`, `int write`, `void * buffer`, `size_t * lenp`, `loff_t * ppos`)
read/write from/to a large bitmap

Parameters

struct ctl_table * table the sysctl table

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

void * buffer the user buffer

size_t * lenp the size of the user buffer

loff_t * ppos file position

Description

The bitmap is stored at `table->data` and the bitmap length (in bits) in `table->maxlen`.

We use a range comma separated format (e.g. `1,3-4,10-10`) so that large bitmaps may be represented in a compact manner. Writing into the file will clear the bitmap then update it with the given input.

Returns 0 on success.

proc filesystem interface

void **proc_flush_pid**(struct pid * pid)
Remove dcache entries for **pid** from the /proc dcache.

Parameters

struct pid * pid pid that should be flushed.

Description

This function walks a list of inodes (that belong to any proc filesystem) that are attached to the pid and flushes them from the dentry cache.

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 provided to flush those useless dcache entries when a process is reaped.

NOTE

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

1.3.3 Events based on file descriptors

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

Parameters

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

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

Description

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

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

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

Parameters

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

Description

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

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

Read the current counter and removes wait queue.

Parameters

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

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

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

Description

Returns 0 if successful, or the following error codes:

-EAGAIN : The operation would have blocked.

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

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

Acquire a reference of an eventfd file descriptor.

Parameters

int fd [in] Eventfd file descriptor.

Description

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

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

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

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

Acquires a reference to the internal eventfd context.

Parameters

int fd [in] Eventfd file descriptor.

Description

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

eventfd_fget

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

Acquires a reference to the internal eventfd context.

Parameters

struct file * file [in] Eventfd file pointer.

Description

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

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

Description

Undo the effects of `sysfs_break_active_protection()`. Since this function calls `kernfs_put()` on the `kernfs` node that corresponds to the 'attr' argument passed to `sysfs_break_active_protection()` that attribute may have been removed between the `sysfs_break_active_protection()` and `sysfs_unbreak_active_protection()` calls, it is not safe to access **kn** after this function has returned.

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

Parameters

struct kobject * kobj object we' re acting for
const struct attribute * attr attribute descriptor
const void * ns namespace tag of the file to remove

Description

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

```
bool sysfs_remove_file_self(struct kobject * kobj, const struct attribute
                             * attr)
    remove an object attribute from its own method
```

Parameters

struct kobject * kobj object we' re acting for
const struct attribute * attr attribute descriptor

Description

See `kernfs_remove_self()` for details.

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

Parameters

struct kobject * kobj object we' re acting for.
const struct attribute * attr attribute descriptor.
const char * group group name.

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

Parameters

struct kobject * kobj object.
const struct bin_attribute * attr attribute descriptor.

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


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

Parameters

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

struct kobject * target object we' re pointing to.

const char * name name of the symlink.

This function does the same as `sysfs_create_link()`, but it doesn' t warn if the link already exists.

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

Parameters

struct kobject * kobj object we' re acting for.

const char * name name of the symlink to remove.

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

Parameters

struct kobject * kobj object we' re acting for.

struct kobject * targ object we' re pointing to.

const char * old previous name of the symlink.

const char * new new name of the symlink.

const void * new_ns new namespace of the symlink.

A helper function for the common rename symlink idiom.

1.3.5 The debugfs filesystem

debugfs interface

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

Parameters

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

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

Description

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

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

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

Parameters

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

umode_t mode the permission that the file should have.

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

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

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

Description

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

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

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

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

Parameters

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

umode_t mode the permission that the file should have.

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

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

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

Description

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

It is your responsibility to protect your struct `file_operations` methods against file removals by means of `debugfs_file_get()` and `debugfs_file_put()`. `->open()` is still protected by debugfs though.

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

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

Parameters

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

umode_t mode the permission that the file should have.

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

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

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

loff_t file_size initial file size

Description

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

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

Parameters

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

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

Description

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

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

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

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

Parameters

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

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

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

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

Description

`f` should return what `->d_automount()` would.

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

Parameters

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

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

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

Description

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

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

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

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

Parameters

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

Description

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

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

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

Parameters

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

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

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

const char * new_name a pointer to a string containing the target name.

Description

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

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

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

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

Parameters

void no arguments

```
int debugfs_file_get(struct dentry * dentry)
    mark the beginning of file data access
```

Parameters

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

Description

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

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

void **debugfs_file_put**(struct dentry * dentry)
mark the end of file data access

Parameters

struct dentry * dentry the dentry object formerly passed to `debugfs_file_get()`.

Description

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

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

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

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

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

Parameters

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

umode_t mode the permission that the file should have.

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

u32 * array u32 array that provides data.

u32 elements total number of elements in the array.

Description

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

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

Parameters

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

const struct debugfs_reg32 * regs an array of struct debugfs_reg32 structures

int nregs the length of the above array

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

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

Description

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

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

Parameters

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

umode_t mode the permission that the file should have

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

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

Description

This function creates a file in debugfs with the given name that reports the names and values of a set of 32-bit registers. If the **mode** variable is so set it can be read from. Writing is not supported.

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

Parameters

struct device * dev device related to this debugfs file.

const char * name name of the debugfs file.

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

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

1.4 splice and pipes

1.4.1 splice API

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

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

Parameters

struct pipe_inode_info * pipe pipe to fill

struct splice_pipe_desc * spd data to fill

Description

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

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

Parameters

struct file * in file to splice from

loff_t * ppos position in **in**

struct pipe_inode_info * pipe pipe to splice to

size_t len number of bytes to splice

unsigned int flags splice modifier flags

Description

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

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

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct splice_desc * sd information to **actor**

splice_actor * actor handler that splices the data

Description

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

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

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

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct splice_desc * sd information about the splice operation

Description

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

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

Parameters

struct splice_desc * sd information about the splice operation

Description

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

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

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct splice_desc * sd information about the splice operation

Description

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

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

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct splice_desc * sd information to **actor**

splice_actor * actor handler that splices the data

Description

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

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

Parameters

struct pipe_inode_info * pipe pipe to splice from

struct file * out file to splice to

loff_t * ppos position in **out**

size_t len how many bytes to splice

unsigned int flags splice modifier flags

splice_actor * actor handler that splices the data

Description

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

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

Parameters

struct pipe_inode_info * pipe pipe info

struct file * out file to write to

loff_t * ppos position in **out**
size_t len number of bytes to splice
unsigned int flags splice modifier flags

Description

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

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

Parameters

struct pipe_inode_info * pipe pipe to splice from
struct file * out socket to write to
loff_t * ppos position in **out**
size_t len number of bytes to splice
unsigned int flags splice modifier flags

Description

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

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

Parameters

struct file * in file to splice from
struct splice_desc * sd actor information on where to splice to
splice_direct_actor * actor handles the data splicing

Description

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

long do_splice_direct(struct file * in, loff_t * ppos, struct file * out, loff_t * opos, size_t len, unsigned int flags)
splices data directly between two files

Parameters

struct file * in file to splice from
loff_t * ppos input file offset
struct file * out file to splice to
loff_t * opos output file offset

size_t len number of bytes to splice

unsigned int flags splice modifier flags

Description

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

1.4.2 pipes API

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

struct **pipe_buffer**
a linux kernel pipe buffer

Definition

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

Members

page the page containing the data for the pipe buffer

offset offset of data inside the **page**

len length of data inside the **page**

ops operations associated with this buffer. See **pipe_buf_operations**.

flags pipe buffer flags. See above.

private private data owned by the ops.

struct **pipe_inode_info**
a linux kernel pipe

Definition

```
struct pipe_inode_info {
    struct mutex mutex;
    wait_queue_head_t rd_wait, wr_wait;
    unsigned int head;
    unsigned int tail;
    unsigned int max_usage;
    unsigned int ring_size;
#ifdef CONFIG_WATCH_QUEUE;
    bool note_loss;
#endif;
    unsigned int nr_accounted;
```

(continues on next page)

(continued from previous page)

```
unsigned int readers;
unsigned int writers;
unsigned int files;
unsigned int r_counter;
unsigned int w_counter;
struct page *tmp_page;
struct fasync_struct *fasync_readers;
struct fasync_struct *fasync_writers;
struct pipe_buffer *bufs;
struct user_struct *user;
#ifdef CONFIG_WATCH_QUEUE;
struct watch_queue *watch_queue;
#endif;
};
```

Members

mutex mutex protecting the whole thing

rd_wait reader wait point in case of empty pipe

wr_wait writer wait point in case of full pipe

head The point of buffer production

tail The point of buffer consumption

max_usage The maximum number of slots that may be used in the ring

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

note_loss The next read() should insert a data-lost message

nr_accounted The amount this pipe accounts for in user->pipe_bufs

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)

r_counter reader counter

w_counter writer counter

tmp_page cached released page

fasync_readers reader side fasync

fasync_writers writer side fasync

bufs the circular array of pipe buffers

user the user who created this pipe

watch_queue If this pipe is a watch_queue, this is the stuff for that

bool **pipe_empty**(unsigned int head, unsigned int tail)
Return true if the pipe is empty

Parameters

unsigned int head The pipe ring head pointer

unsigned int tail The pipe ring tail pointer

unsigned int **pipe_occupancy**(unsigned int head, unsigned int tail)
Return number of slots used in the pipe

Parameters

unsigned int head The pipe ring head pointer

unsigned int tail The pipe ring tail pointer

bool **pipe_full**(unsigned int head, unsigned int tail, unsigned int limit)
Return true if the pipe is full

Parameters

unsigned int head The pipe ring head pointer

unsigned int tail The pipe ring tail pointer

unsigned int limit The maximum amount of slots available.

unsigned int **pipe_space_for_user**(unsigned int head, unsigned int tail,
struct pipe_inode_info * pipe)
Return number of slots available to userspace

Parameters

unsigned int head The pipe ring head pointer

unsigned int tail The pipe ring tail pointer

struct pipe_inode_info * pipe The pipe info structure

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

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to get a reference to

Return

true if the reference was successfully obtained.

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

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to put a reference to

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

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to confirm

bool **pipe_buf_try_steal**(struct pipe_inode_info * pipe, struct pipe_buffer * buf)
attempt to take ownership of a pipe_buffer

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to attempt to steal

bool **generic_pipe_buf_try_steal**(struct pipe_inode_info * pipe, struct pipe_buffer * buf)
attempt to take ownership of a pipe_buffer

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to attempt to steal

Description

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

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

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to get a reference to

Description

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

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

Parameters

struct pipe_inode_info * pipe the pipe that the buffer belongs to

struct pipe_buffer * buf the buffer to put a reference to

Description

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

1.5 Locking

The text below describes the locking rules for VFS-related methods. It is (believed to be) up-to-date. Please, if you change anything in prototypes or locking protocols - update this file. And update the relevant instances in the tree, don't leave that to maintainers of filesystems/devices/ etc. At the very least, put the list of dubious cases in the end of this file. Don't turn it into log - maintainers of out-of-the-tree code are supposed to be able to use diff(1).

Thing currently missing here: socket operations. Alexey?

1.5.1 dentry_operations

prototypes:

```
int (*d_revalidate)(struct dentry *, unsigned int);
int (*d_weak_revalidate)(struct dentry *, unsigned int);
int (*d_hash)(const struct dentry *, struct qstr *);
int (*d_compare)(const struct dentry *,
                 unsigned int, const char *, const struct qstr *);
int (*d_delete)(struct dentry *);
int (*d_init)(struct dentry *);
void (*d_release)(struct dentry *);
void (*d_iput)(struct dentry *, struct inode *);
char *(*d_dname)((struct dentry *dentry, char *buffer, int buflen);
struct vfsmount *(*d_automount)(struct path *path);
int (*d_manage)(const struct path *, bool);
struct dentry *(*d_real)(struct dentry *, const struct inode *);
```

locking rules:

ops	rename_lock	->d_lock	may block	rcu-walk
d_revalidate:	no	no	yes (ref-walk)	maybe
d_weak_revalidate:	no	no	yes	no
d_hash	no	no	no	maybe
d_compare:	yes	no	no	maybe
d_delete:	no	yes	no	no
d_init:	no	no	yes	no
d_release:	no	no	yes	no
d_prune:	no	yes	no	no
d_iput:	no	no	yes	no
d_dname:	no	no	no	no
d_automount:	no	no	yes	no
d_manage:	no	no	yes (ref-walk)	maybe
d_real	no	no	yes	no

1.5.2 inode_operations

prototypes:

```
int (*create) (struct inode *,struct dentry *,umode_t, bool);
struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
int (*link) (struct dentry *,struct inode *,struct dentry *);
int (*unlink) (struct inode *,struct dentry *);
int (*symlink) (struct inode *,struct dentry *,const char *);
int (*mkdir) (struct inode *,struct dentry *,umode_t);
int (*rmdir) (struct inode *,struct dentry *);
int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
int (*rename) (struct inode *, struct dentry *,
               struct inode *, struct dentry *, unsigned int);
int (*readlink) (struct dentry *, char __user *,int);
const char *(*get_link) (struct dentry *, struct inode *, struct delayed_
↳call *);
void (*truncate) (struct inode *);
int (*permission) (struct inode *, int, unsigned int);
int (*get_acl)(struct inode *, int);
int (*setattr) (struct dentry *, struct iattr *);
int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
ssize_t (*listxattr) (struct dentry *, char *, size_t);
int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start, u64_
↳len);
void (*update_time)(struct inode *, struct timespec *, int);
int (*atomic_open)(struct inode *, struct dentry *,
                  struct file *, unsigned open_flag,
                  umode_t create_mode);
int (*tmpfile) (struct inode *, struct dentry *, umode_t);
```

locking rules: all may block

ops	i_rwsem(inode)
lookup:	shared
create:	exclusive
link:	exclusive (both)
mknod:	exclusive
symlink:	exclusive
mkdir:	exclusive
unlink:	exclusive (both)
rmdir:	exclusive (both)(see below)
rename:	exclusive (all) (see below)
readlink:	no
get_link:	no
setattr:	exclusive
permission:	no (may not block if called in rcu-walk mode)
get_acl:	no
getattr:	no
listxattr:	no
fiemap:	no
update_time:	no
atomic_open:	shared (exclusive if O_CREAT is set in open flags)
tmpfile:	no

Additionally, `->rmdir()`, `->unlink()` and `->rename()` have `->i_rwsem` exclusive on victim. `cross-directory ->rename()` has (per-superblock) `->s_vfs_rename_sem`.

See `Documentation/filesystems/directory-locking.rst` for more detailed discussion of the locking scheme for directory operations.

1.5.3 xattr_handler operations

prototypes:

```
bool (*list)(struct dentry *dentry);
int (*get)(const struct xattr_handler *handler, struct dentry *dentry,
           struct inode *inode, const char *name, void *buffer,
           size_t size);
int (*set)(const struct xattr_handler *handler, struct dentry *dentry,
           struct inode *inode, const char *name, const void *buffer,
           size_t size, int flags);
```

locking rules: all may block

ops	i_rwsem(inode)
list:	no
get:	no
set:	exclusive

1.5.4 super_operations

prototypes:

```
struct inode *(*alloc_inode)(struct super_block *sb);
void (*free_inode)(struct inode *);
void (*destroy_inode)(struct inode *);
void (*dirty_inode) (struct inode *, int flags);
int (*write_inode) (struct inode *, struct writeback_control *wbc);
int (*drop_inode) (struct inode *);
void (*evict_inode) (struct inode *);
void (*put_super) (struct super_block *);
int (*sync_fs)(struct super_block *sb, int wait);
int (*freeze_fs) (struct super_block *);
int (*unfreeze_fs) (struct super_block *);
int (*statfs) (struct dentry *, struct kstatfs *);
int (*remount_fs) (struct super_block *, int *, char *);
void (*umount_begin) (struct super_block *);
int (*show_options)(struct seq_file *, struct dentry *);
ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
ssize_t (*quota_write)(struct super_block *, int, const char *, size_t,
↳loff_t);
int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
```

locking rules: All may block [not true, see below]

ops	s_umount	note
alloc_inode:		
free_inode:		called from RCU callback
destroy_inode:		
dirty_inode:		
write_inode:		
drop_inode:		!!!inode->i_lock!!!
evict_inode:		
put_super:	write	
sync_fs:	read	
freeze_fs:	write	
unfreeze_fs:	write	
statfs:	maybe(read)	(see below)
remount_fs:	write	
umount_begin:	no	
show_options:	no	(namespace_sem)
quota_read:	no	(see below)
quota_write:	no	(see below)
bdev_try_to_free_page:	no	(see below)

->statfs() has s_umount (shared) when called by ustat(2) (native or compat), but that's an accident of bad API; s_umount is used to pin the superblock down when we only have dev_t given us by userland to identify the superblock. Everything else (statfs(), fstatfs(), etc.) doesn't hold it when calling ->statfs() - superblock is pinned down by resolving the pathname passed to syscall.

->quota_read() and ->quota_write() functions are both guaranteed to be the only ones operating on the quota file by the quota code (via dqio_sem) (unless an admin really wants to screw up something and writes to quota files with quotas on). For other details about locking see also dquot_operations section.

->bdev_try_to_free_page is called from the ->releasepage handler of the block device inode. See there for more details.

1.5.5 file_system_type

prototypes:

```
struct dentry *(*mount) (struct file_system_type *, int,
                        const char *, void *);
void (*kill_sb) (struct super_block *);
```

locking rules:

ops	may block
mount	yes
kill_sb	yes

->mount() returns ERR_PTR or the root dentry; its superblock should be locked on return.

->kill_sb() takes a write-locked superblock, does all shutdown work on it, unlocks and drops the reference.

1.5.6 address_space_operations

prototypes:

```
int (*writepage)(struct page *page, struct writeback_control *wbc);
int (*readpage)(struct file *, struct page *);
int (*writepages)(struct address_space *, struct writeback_control *);
int (*set_page_dirty)(struct page *page);
void (*readahead)(struct readahead_control *);
int (*readpages)(struct file *filp, struct address_space *mapping,
                 struct list_head *pages, unsigned nr_pages);
int (*write_begin)(struct file *, struct address_space *mapping,
                  loff_t pos, unsigned len, unsigned flags,
                  struct page **pagep, void **fsdata);
int (*write_end)(struct file *, struct address_space *mapping,
                 loff_t pos, unsigned len, unsigned copied,
                 struct page *page, void *fsdata);
sector_t (*bmap)(struct address_space *, sector_t);
void (*invalidatepage)(struct page *, unsigned int, unsigned int);
int (*releasepage)(struct page *, int);
void (*freepage)(struct page *);
int (*direct_IO)(struct kiocb *, struct iov_iter *iter);
bool (*isolate_page)(struct page *, isolate_mode_t);
int (*migratepage)(struct address_space *, struct page *, struct page *);
void (*putback_page)(struct page *);
int (*launder_page)(struct page *);
int (*is_partially_uptodate)(struct page *, unsigned long, unsigned long);
int (*error_remove_page)(struct address_space *, struct page *);
int (*swap_activate)(struct file *);
int (*swap_deactivate)(struct file *);
```

locking rules: All except set_page_dirty and freepage may block

ops	PageLocked(page)	i_rwsem
writepage:	yes, unlocks (see below)	
readpage:	yes, unlocks	
writepages:		
set_page_dirty	no	
readahead:	yes, unlocks	
readpages:	no	
write_begin:	locks the page	exclusive
write_end:	yes, unlocks	exclusive
bmap:		
invalidatepage:	yes	
releasepage:	yes	
freepage:	yes	
direct_IO:		
isolate_page:	yes	
migratepage:	yes (both)	
putback_page:	yes	
launder_page:	yes	
is_partially_uptodate:	yes	
error_remove_page:	yes	
swap_activate:	no	
swap_deactivate:	no	

->write_begin(), ->write_end() and ->readpage() may be called from the request handler (/dev/loop).

->readpage() unlocks the page, either synchronously or via I/O completion.

->readahead() unlocks the pages that I/O is attempted on like ->readpage().

->readpages() populates the pagecache with the passed pages and starts I/O against them. They come unlocked upon I/O completion.

->writepage() is used for two purposes: for “memory cleansing” and for “sync” . These are quite different operations and the behaviour may differ depending upon the mode.

If writepage is called for sync (wbc->sync_mode != WBC_SYNC_NONE) then it must start I/O against the page, even if that would involve blocking on in-progress I/O.

If writepage is called for memory cleansing (sync_mode == WBC_SYNC_NONE) then its role is to get as much writeout underway as possible. So writepage should try to avoid blocking against currently-in-progress I/O.

If the filesystem is not called for “sync” and it determines that it would need to block against in-progress I/O to be able to start new I/O against the page the filesystem should redirty the page with redirty_page_for_writepage(), then unlock the page and return zero. This may also be done to avoid internal deadlocks, but rarely.

If the filesystem is called for sync then it must wait on any in-progress I/O and then start new I/O.

The filesystem should unlock the page synchronously, before returning to

the caller, unless `->writepage()` returns special `WRITEPAGE_ACTIVATE` value. `WRITEPAGE_ACTIVATE` means that page cannot really be written out currently, and VM should stop calling `->writepage()` on this page for some time. VM does this by moving page to the head of the active list, hence the name.

Unless the filesystem is going to `redirty_page_for_writepage()`, unlock the page and return zero, `writepage` must run `set_page_writeback()` against the page, followed by unlocking it. Once `set_page_writeback()` has been run against the page, write I/O can be submitted and the write I/O completion handler must run `end_page_writeback()` once the I/O is complete. If no I/O is submitted, the filesystem must run `end_page_writeback()` against the page before returning from `writepage`.

That is: after 2.5.12, pages which are under writeout are not locked. Note, if the filesystem needs the page to be locked during writeout, that is ok, too, the page is allowed to be unlocked at any point in time between the calls to `set_page_writeback()` and `end_page_writeback()`.

Note, failure to run either `redirty_page_for_writepage()` or the combination of `set_page_writeback()/end_page_writeback()` on a page submitted to `writepage` will leave the page itself marked clean but it will be tagged as dirty in the radix tree. This incoherency can lead to all sorts of hard-to-debug problems in the filesystem like having dirty inodes at umount and losing written data.

`->writepages()` is used for periodic writeback and for syscall-initiated sync operations. The `address_space` should start I/O against at least `*nr_to_write` pages. `*nr_to_write` must be decremented for each page which is written. The `address_space` implementation may write more (or less) pages than `*nr_to_write` asks for, but it should try to be reasonably close. If `nr_to_write` is `NULL`, all dirty pages must be written.

`writepages` should only write pages which are present on `mapping->io_pages`.

`->set_page_dirty()` is called from various places in the kernel when the target page is marked as needing writeback. It may be called under spinlock (it cannot block) and is sometimes called with the page not locked.

`->bmap()` is currently used by legacy `ioctl()` (`FIBMAP`) provided by some filesystems and by the swapper. The latter will eventually go away. Please, keep it that way and don't breed new callers.

`->invalidatepage()` is called when the filesystem must attempt to drop some or all of the buffers from the page when it is being truncated. It returns zero on success. If `->invalidatepage` is zero, the kernel uses `block_invalidatepage()` instead.

`->releasepage()` is called when the kernel is about to try to drop the buffers from the page in preparation for freeing it. It returns zero to indicate that the buffers are (or may be) freeable. If `->releasepage` is zero, the kernel assumes that the fs has no private interest in the buffers.

`->freepage()` is called when the kernel is done dropping the page from the page cache.

`->launder_page()` may be called prior to releasing a page if it is still found to be dirty. It returns zero if the page was successfully cleaned, or an error value if not. Note that in order to prevent the page getting mapped back in and redirtied, it needs to be kept locked across the entire operation.

->swap_activate will be called with a non-zero argument on files backing (non block device backed) swapfiles. A return value of zero indicates success, in which case this file can be used for backing swapspace. The swapspace operations will be proxied to the address space operations.

->swap_deactivate() will be called in the sys_swapoff() path after ->swap_activate() returned success.

1.5.7 file_lock_operations

prototypes:

```
void (*fl_copy_lock)(struct file_lock *, struct file_lock *);
void (*fl_release_private)(struct file_lock *);
```

locking rules:

ops	inode->i_lock	may block
fl_copy_lock:	yes	no
fl_release_private:	maybe	maybe[1]

1.5.8 lock_manager_operations

prototypes:

```
void (*lm_notify)(struct file_lock *); /* unblock callback */
int (*lm_grant)(struct file_lock *, struct file_lock *, int);
void (*lm_break)(struct file_lock *); /* break_lease callback */
int (*lm_change)(struct file_lock **, int);
bool (*lm_breaker_owns_lease)(struct file_lock *);
```

locking rules:

1.5.9 buffer_head

prototypes:

```
void (*b_end_io)(struct buffer_head *bh, int uptodate);
```

locking rules:

called from interrupts. In other words, extreme care is needed here. bh is locked, but that's all warranties we have here. Currently only RAID1, highmem, fs/buffer.c, and fs/ntfs/aops.c are providing these. Block devices call this method upon the IO completion.

1.5.10 block_device_operations

prototypes:

```
int (*open) (struct block_device *, fmode_t);
int (*release) (struct gendisk *, fmode_t);
int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned,
↳long);
int (*direct_access) (struct block_device *, sector_t, void **,
                    unsigned long *);
int (*media_changed) (struct gendisk *);
void (*unlock_native_capacity) (struct gendisk *);
int (*revalidate_disk) (struct gendisk *);
int (*getgeo)(struct block_device *, struct hd_geometry *);
void (*swap_slot_free_notify) (struct block_device *, unsigned long);
```

locking rules:

ops	bd_mutex
open:	yes
release:	yes
ioctl:	no
compat_ioctl:	no
direct_access:	no
media_changed:	no
unlock_native_capacity:	no
revalidate_disk:	no
getgeo:	no
swap_slot_free_notify:	no (see below)

media_changed, unlock_native_capacity and revalidate_disk are called only from check_disk_change().

swap_slot_free_notify is called with swap_lock and sometimes the page lock held.

1.5.11 file_operations

prototypes:

```
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
int (*iterate) (struct file *, struct dir_context *);
int (*iterate_shared) (struct file *, struct dir_context *);
__poll_t (*poll) (struct file *, struct poll_table_struct *);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
```

(continues on next page)

(continued from previous page)

```
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, loff_t start, loff_t end, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
                 loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
                 loff_t *);
ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                   void __user *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                   loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                  unsigned long, unsigned long);
int (*check_flags)(int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
                      size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
                      size_t, unsigned int);
int (*setlease)(struct file *, long, struct file_lock **, void **);
long (*fallocate)(struct file *, int, loff_t, loff_t);
```

locking rules: All may block.

->llseek() locking has moved from llseek to the individual llseek implementations. If your fs is not using generic_file_llseek, you need to acquire and release the appropriate locks in your ->llseek(). For many filesystems, it is probably safe to acquire the inode mutex or just to use i_size_read() instead. Note: this does not protect the file->f_pos against concurrent modifications since this is something the userspace has to take care about.

->iterate() is called with i_rwsem exclusive.

->iterate_shared() is called with i_rwsem at least shared.

->fasync() is responsible for maintaining the FASYNC bit in filp->f_flags. Most instances call fasync_helper(), which does that maintenance, so it's not normally something one needs to worry about. Return values > 0 will be mapped to zero in the VFS layer.

->readdir() and ->iocctl() on directories must be changed. Ideally we would move ->readdir() to inode_operations and use a separate method for directory ->iocctl() or kill the latter completely. One of the problems is that for anything that resembles union-mount we won't have a struct file for all components. And there are other reasons why the current interface is a mess...

->read on directories probably must go away - we should just enforce -EISDIR in sys_read() and friends.

->setlease operations should call generic_setlease() before or after setting the lease within the individual filesystem to record the result of the operation

1.5.12 dquot_operations

prototypes:

```
int (*write_dquot) (struct dquot *);
int (*acquire_dquot) (struct dquot *);
int (*release_dquot) (struct dquot *);
int (*mark_dirty) (struct dquot *);
int (*write_info) (struct super_block *, int);
```

These operations are intended to be more or less wrapping functions that ensure a proper locking wrt the filesystem and call the generic quota operations.

What filesystem should expect from the generic quota functions:

ops	FS recursion	Held locks when called
write_dquot:	yes	dqonoff_sem or dqptr_sem
acquire_dquot:	yes	dqonoff_sem or dqptr_sem
release_dquot:	yes	dqonoff_sem or dqptr_sem
mark_dirty:	no	•
write_info:	yes	dqonoff_sem

FS recursion means calling `->quota_read()` and `->quota_write()` from superblock operations.

More details about quota locking can be found in `fs/dquot.c`.

1.5.13 vm_operations_struct

prototypes:

```
void (*open)(struct vm_area_struct*);
void (*close)(struct vm_area_struct*);
vm_fault_t (*fault)(struct vm_area_struct*, struct vm_fault *);
vm_fault_t (*page_mkwrite)(struct vm_area_struct *, struct vm_fault *);
vm_fault_t (*pfn_mkwrite)(struct vm_area_struct *, struct vm_fault *);
int (*access)(struct vm_area_struct *, unsigned long, void*, int, int);
```

locking rules:

`->fault()` is called when a previously not present pte is about to be faulted in. The filesystem must find and return the page associated with the passed in “pgoff” in the `vm_fault` structure. If it is possible that the page may be truncated and/or invalidated, then the filesystem must lock the page, then ensure it is not already truncated (the page lock will block subsequent truncate), and then return with `VM_FAULT_LOCKED`, and the page locked. The VM will unlock the page.

`->map_pages()` is called when VM asks to map easy accessible pages. Filesystem should find and map pages associated with offsets from “start_pgoff” till “end_pgoff”

. ->map_pages() is called with page table locked and must not block. If it's not possible to reach a page without blocking, filesystem should skip it. Filesystem should use do_set_pte() to setup page table entry. Pointer to entry associated with the page is passed in "pte" field in vm_fault structure. Pointers to entries for other offsets should be calculated relative to "pte" .

->page_mkwrite() is called when a previously read-only pte is about to become writeable. The filesystem again must ensure that there are no truncate/invalidate races, and then return with the page locked. If the page has been truncated, the filesystem should not look up a new page like the ->fault() handler, but simply return with VM_FAULT_NOPAGE, which will cause the VM to retry the fault.

->pfn_mkwrite() is the same as page_mkwrite but when the pte is VM_PFNMAP or VM_MIXEDMAP with a page-less entry. Expected return is VM_FAULT_NOPAGE. Or one of the VM_FAULT_ERROR types. The default behavior after this call is to make the pte read-write, unless pfn_mkwrite returns an error.

->access() is called when get_user_pages() fails in access_process_vm(), typically used to debug a process through /proc/pid/mem or ptrace. This function is needed only for VM_IO | VM_PFNMAP VMAs.

Dubious stuff

(if you break something or notice that it is broken and do not fix it yourself - at least put it here)

1.6 Directory Locking

Locking scheme used for directory operations is based on two kinds of locks - per-inode (->i_rwsem) and per-filesystem (->s_vfs_rename_mutex).

When taking the i_rwsem on multiple non-directory objects, we always acquire the locks in order by increasing address. We'll call that "inode pointer" order in the following.

For our purposes all operations fall in 5 classes:

1) read access. Locking rules: caller locks directory we are accessing. The lock is taken shared.

2) object creation. Locking rules: same as above, but the lock is taken exclusive.

3) object removal. Locking rules: caller locks parent, finds victim, locks victim and calls the method. Locks are exclusive.

4) rename() that is `_not_cross-directory`. Locking rules: caller locks the parent and finds source and target. In case of exchange (with `RENAME_EXCHANGE` in flags argument) lock both. In any case, if the target already exists, lock it. If the source is a non-directory, lock it. If we need to lock both, lock them in inode pointer order. Then call the method. All locks are exclusive. NB: we might get away with locking the the source (and target in exchange case) shared.

5) link creation. Locking rules:

- lock parent

- check that source is not a directory
- lock source
- call the method.

All locks are exclusive.

6) cross-directory rename. The trickiest in the whole bunch. Locking rules:

- lock the filesystem
- lock parents in “ancestors first” order.
- find source and target.
- if old parent is equal to or is a descendent of target fail with -ENOTEMPTY
- if new parent is equal to or is a descendent of source fail with -ELOOP
- If it’ s an exchange, lock both the source and the target.
- If the target exists, lock it. If the source is a non-directory, lock it. If we need to lock both, do so in inode pointer order.
- call the method.

All `->i_rwsem` are taken exclusive. Again, we might get away with locking the the source (and target in exchange case) shared.

The rules above obviously guarantee that all directories that are going to be read, modified or removed by method will be locked by caller.

If no directory is its own ancestor, the scheme above is deadlock-free.

Proof:

First of all, at any moment we have a partial ordering of the objects - $A < B$ iff A is an ancestor of B.

That ordering can change. However, the following is true:

- (1) if object removal or non-cross-directory rename holds lock on A and attempts to acquire lock on B, A will remain the parent of B until we acquire the lock on B. (Proof: only cross-directory rename can change the parent of object and it would have to lock the parent).
- (2) if cross-directory rename holds the lock on filesystem, order will not change until rename acquires all locks. (Proof: other cross-directory renames will be blocked on filesystem lock and we don’ t start changing the order until we had acquired all locks).
- (3) locks on non-directory objects are acquired only after locks on directory objects, and are acquired in inode pointer order. (Proof: all operations but renames take lock on at most one non-directory object, except renames, which take locks on source and target in inode pointer order in the case they are not directories.)

Now consider the minimal deadlock. Each process is blocked on attempt to acquire some lock and already holds at least one lock. Let’ s consider the set of contended locks. First of all, filesystem lock is not contended, since any process blocked on it is not holding any locks. Thus all processes are blocked on `->i_rwsem`.

By (3), any process holding a non-directory lock can only be waiting on another non-directory lock with a larger address. Therefore the process holding the “largest” such lock can always make progress, and non-directory objects are not included in the set of contended locks.

Thus link creation can't be a part of deadlock - it can't be blocked on source and it means that it doesn't hold any locks.

Any contended object is either held by cross-directory rename or has a child that is also contended. Indeed, suppose that it is held by operation other than cross-directory rename. Then the lock this operation is blocked on belongs to child of that object due to (1).

It means that one of the operations is cross-directory rename. Otherwise the set of contended objects would be infinite - each of them would have a contended child and we had assumed that no object is its own descendent. Moreover, there is exactly one cross-directory rename (see above).

Consider the object blocking the cross-directory rename. One of its descendents is locked by cross-directory rename (otherwise we would again have an infinite set of contended objects). But that means that cross-directory rename is taking locks out of order. Due to (2) the order hadn't changed since we had acquired filesystem lock. But locking rules for cross-directory rename guarantee that we do not try to acquire lock on descendent before the lock on ancestor. Contradiction. I.e. deadlock is impossible. Q.E.D.

These operations are guaranteed to avoid loop creation. Indeed, the only operation that could introduce loops is cross-directory rename. Since the only new (parent, child) pair added by rename() is (new parent, source), such loop would have to contain these objects and the rest of it would have to exist before rename(). I.e. at the moment of loop creation rename() responsible for that would be holding filesystem lock and new parent would have to be equal to or a descendent of source. But that means that new parent had been equal to or a descendent of source since the moment when we had acquired filesystem lock and rename() would fail with -ELOOP in that case.

While this locking scheme works for arbitrary DAGs, it relies on ability to check that directory is a descendent of another object. Current implementation assumes that directory graph is a tree. This assumption is also preserved by all operations (cross-directory rename on a tree that would not introduce a cycle will leave it a tree and link() fails for directories).

Notice that “directory” in the above == “anything that might have children” , so if we are going to introduce hybrid objects we will need either to make sure that link(2) doesn't work for them or to make changes in `is_subdir()` that would make it work even in presence of such beasts.

1.7 The Devpts Filesystem

Each mount of the devpts filesystem is now distinct such that ptys and their indices allocated in one mount are independent from ptys and their indices in all other mounts.

All mounts of the devpts filesystem now create a `/dev/pts/ptmx` node with permissions `0000`.

To retain backwards compatibility the a ptmx device node (aka any node created with `mknod name c 5 2`) when opened will look for an instance of devpts under the name `pts` in the same directory as the ptmx device node.

As an option instead of placing a `/dev/ptmx` device node at `/dev/ptmx` it is possible to place a symlink to `/dev/pts/ptmx` at `/dev/ptmx` or to bind mount `/dev/pts/ptmx` to `/dev/ptmx`. If you opt for using the devpts filesystem in this manner devpts should be mounted with the `ptmxmode=0666`, or `chmod 0666 /dev/pts/ptmx` should be called.

Total count of pty pairs in all instances is limited by `sysctls`:

<code>kernel.pty.max = 4096</code>	- global limit
<code>kernel.pty.reserve = 1024</code>	- reserved for filesystems mounted from the ↵ ↪ initial mount namespace
<code>kernel.pty.nr</code>	- current count of ptys

Per-instance limit could be set by adding mount option `max=<count>`.

This feature was added in kernel 3.4 together with `sysctl kernel.pty.reserve`.

In kernels older than 3.4 `sysctl kernel.pty.max` works as per-instance limit.

1.8 Linux Directory Notification

Stephen Rothwell <sfr@canb.auug.org.au>

The intention of directory notification is to allow user applications to be notified when a directory, or any of the files in it, are changed. The basic mechanism involves the application registering for notification on a directory using a `fcntl(2)` call and the notifications themselves being delivered using signals.

The application decides which “events” it wants to be notified about. The currently defined events are:

<code>DN_ACCESS</code>	A file in the directory was accessed (read)
<code>DN_MODIFY</code>	A file in the directory was modified (write,truncate)
<code>DN_CREATE</code>	A file was created in the directory
<code>DN_DELETE</code>	A file was unlinked from directory
<code>DN_RENAME</code>	A file in the directory was renamed
<code>DN_ATTRIB</code>	A file in the directory had its attributes changed (chmod,chown)

Usually, the application must reregister after each notification, but if `DN_MULTISHOT` is or'ed with the event mask, then the registration will remain until explicitly removed (by registering for no events).

By default, `SIGIO` will be delivered to the process and no other useful information. However, if the `F_SETSIG` `fcntl(2)` call is used to let the kernel know which signal to deliver, a `siginfo` structure will be passed to the signal handler and the `si_fd` member of that structure will contain the file descriptor associated with the directory in which the event occurred.

Preferably the application will choose one of the real time signals (`SIGRTMIN + <n>`) so that the notifications may be queued. This is especially important if `DN_MULTISHOT` is specified. Note that `SIGRTMIN` is often blocked, so it is better to use (at least) `SIGRTMIN + 1`.

1.8.1 Implementation expectations (features and bugs :-))

The notification should work for any local access to files even if the actual file system is on a remote server. This implies that remote access to files served by local user mode servers should be notified. Also, remote accesses to files served by a local kernel NFS server should be notified.

In order to make the impact on the file system code as small as possible, the problem of hard links to files has been ignored. So if a file (x) exists in two directories (a and b) then a change to the file using the name "a/x" should be notified to a program expecting notifications on directory "a", but will not be notified to one expecting notifications on directory "b".

Also, files that are unlinked, will still cause notifications in the last directory that they were linked to.

1.8.2 Configuration

`Dnotify` is controlled via the `CONFIG_DNOTIFY` configuration option. When disabled, `fcntl(fd, F_NOTIFY, ...)` will return `-EINVAL`.

1.8.3 Example

See `tools/testing/selftests/filesystems/dnotify_test.c` for an example.

1.8.4 NOTE

Beginning with Linux 2.6.13, `dnotify` has been replaced by `inotify`. See [Documentation/filesystems/inotify.rst](#) for more information on it.

1.9 Fiemap ioctl

The fiemap ioctl is an efficient method for userspace to get file extent mappings. Instead of block-by-block mapping (such as bmap), fiemap returns a list of extents.

1.9.1 Request Basics

A fiemap request is encoded within struct fiemap:

```
struct fiemap {
    __u64  fm_start;          /* logical offset (inclusive) at
                             * which to start mapping (in) */
    __u64  fm_length;        /* logical length of mapping which
                             * userspace cares about (in) */
    __u32  fm_flags;         /* FIEMAP_FLAG_* flags for request (in/out)
    ↪*/
    __u32  fm_mapped_extents; /* number of extents that were
                             * mapped (out) */
    __u32  fm_extent_count; /* size of fm_extents array (in) */
    __u32  fm_reserved;
    struct fiemap_extent fm_extents[0]; /* array of mapped extents (out)
    ↪*/
};
```

fm_start, and fm_length specify the logical range within the file which the process would like mappings for. Extents returned mirror those on disk - that is, the logical offset of the 1st returned extent may start before fm_start, and the range covered by the last returned extent may end after fm_length. All offsets and lengths are in bytes.

Certain flags to modify the way in which mappings are looked up can be set in fm_flags. If the kernel doesn't understand some particular flags, it will return EBADR and the contents of fm_flags will contain the set of flags which caused the error. If the kernel is compatible with all flags passed, the contents of fm_flags will be unmodified. It is up to userspace to determine whether rejection of a particular flag is fatal to its operation. This scheme is intended to allow the fiemap interface to grow in the future but without losing compatibility with old software.

fm_extent_count specifies the number of elements in the fm_extents[] array that can be used to return extents. If fm_extent_count is zero, then the fm_extents[] array is ignored (no extents will be returned), and the fm_mapped_extents count will hold the number of extents needed in fm_extents[] to hold the file's current mapping. Note that there is nothing to prevent the file from changing between calls to FIEMAP.

The following flags can be set in fm_flags:

FIEMAP_FLAG_SYNC If this flag is set, the kernel will sync the file before mapping extents.

FIEMAP_FLAG_XATTR If this flag is set, the extents returned will describe the inodes extended attribute lookup tree, instead of its data tree.

1.9.2 Extent Mapping

Extent information is returned within the embedded `fm_extents` array which userspace must allocate along with the `fiemap` structure. The number of elements in the `fiemap_extents[]` array should be passed via `fm_extent_count`. The number of extents mapped by kernel will be returned via `fm_mapped_extents`. If the number of `fiemap_extents` allocated is less than would be required to map the requested range, the maximum number of extents that can be mapped in the `fm_extent[]` array will be returned and `fm_mapped_extents` will be equal to `fm_extent_count`. In that case, the last extent in the array will not complete the requested range and will not have the `FIEMAP_EXTENT_LAST` flag set (see the next section on extent flags).

Each extent is described by a single `fiemap_extent` structure as returned in `fm_extents`:

```
struct fiemap_extent {
    __u64      fe_logical; /* logical offset in bytes for the start_
↳of
                * the extent */
    __u64      fe_physical; /* physical offset in bytes for the start
                * of the extent */
    __u64      fe_length; /* length in bytes for the extent */
    __u64      fe_reserved64[2];
    __u32      fe_flags; /* FIEMAP_EXTENT_* flags for this extent_
↳*/
    __u32      fe_reserved[3];
};
```

All offsets and lengths are in bytes and mirror those on disk. It is valid for an extents logical offset to start before the request or its logical length to extend past the request. Unless `FIEMAP_EXTENT_NOT_ALIGNED` is returned, `fe_logical`, `fe_physical`, and `fe_length` will be aligned to the block size of the file system. With the exception of extents flagged as `FIEMAP_EXTENT_MERGED`, adjacent extents will not be merged.

The `fe_flags` field contains flags which describe the extent returned. A special flag, `FIEMAP_EXTENT_LAST` is always set on the last extent in the file so that the process making `fiemap` calls can determine when no more extents are available, without having to call the `ioctl` again.

Some flags are intentionally vague and will always be set in the presence of other more specific flags. This way a program looking for a general property does not have to know all existing and future flags which imply that property.

For example, if `FIEMAP_EXTENT_DATA_INLINE` or `FIEMAP_EXTENT_DATA_TAIL` are set, `FIEMAP_EXTENT_NOT_ALIGNED` will also be set. A program looking for inline or tail-packed data can key on the specific flag. Software which simply cares not to try operating on non-aligned extents however, can just key on `FIEMAP_EXTENT_NOT_ALIGNED`, and not have to worry about all present and future flags which might imply unaligned data. Note that the opposite is not true - it would be valid for `FIEMAP_EXTENT_NOT_ALIGNED` to appear alone.

FIEMAP_EXTENT_LAST This is generally the last extent in the file. A mapping attempt past this extent may return nothing. Some implementations set this

flag to indicate this extent is the last one in the range queried by the user (via `fiemap->fm_length`).

FIEMAP_EXTENT_UNKNOWN The location of this extent is currently unknown. This may indicate the data is stored on an inaccessible volume or that no storage has been allocated for the file yet.

FIEMAP_EXTENT_DEALLOC This will also set `FIEMAP_EXTENT_UNKNOWN`.
Delayed allocation - while there is data for this extent, its physical location has not been allocated yet.

FIEMAP_EXTENT_ENCODED This extent does not consist of plain filesystem blocks but is encoded (e.g. encrypted or compressed). Reading the data in this extent via I/O to the block device will have undefined results.

Note that it is always undefined to try to update the data in-place by writing to the indicated location without the assistance of the filesystem, or to access the data using the information returned by the `FIEMAP` interface while the filesystem is mounted. In other words, user applications may only read the extent data via I/O to the block device while the filesystem is unmounted, and then only if the `FIEMAP_EXTENT_ENCODED` flag is clear; user applications must not try reading or writing to the filesystem via the block device under any other circumstances.

FIEMAP_EXTENT_DATA_ENCRYPTED This will also set `FIEMAP_EXTENT_ENCODED` The data in this extent has been encrypted by the file system.

FIEMAP_EXTENT_NOT_ALIGNED Extent offsets and length are not guaranteed to be block aligned.

FIEMAP_EXTENT_DATA_INLINE This will also set `FIEMAP_EXTENT_NOT_ALIGNED` Data is located within a meta data block.

FIEMAP_EXTENT_DATA_TAIL This will also set `FIEMAP_EXTENT_NOT_ALIGNED` Data is packed into a block with data from other files.

FIEMAP_EXTENT_UNWRITTEN Unwritten extent - the extent is allocated but its data has not been initialized. This indicates the extent's data will be all zero if read through the filesystem but the contents are undefined if read directly from the device.

FIEMAP_EXTENT_MERGED This will be set when a file does not support extents, i.e., it uses a block based addressing scheme. Since returning an extent for each block back to userspace would be highly inefficient, the kernel will try to merge most adjacent blocks into 'extents'.

1.9.3 VFS -> File System Implementation

File systems wishing to support fiemap must implement a `->fiemap` callback on their `inode_operations` structure. The fs `->fiemap` call is responsible for defining its set of supported fiemap flags, and calling a helper function on each discovered extent:

```
struct inode_operations {
    ...

    int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                  u64 len);
```

`->fiemap` is passed `struct fiemap_extent_info` which describes the fiemap request:

```
struct fiemap_extent_info {
    unsigned int fi_flags;           /* Flags as passed from user */
    unsigned int fi_extents_mapped; /* Number of mapped extents */
    unsigned int fi_extents_max;    /* Size of fiemap_extent array */
    struct fiemap_extent *fi_extents_start; /* Start of fiemap_extent_
↪array */
};
```

It is intended that the file system should not need to access any of this structure directly. Filesystem handlers should be tolerant to signals and return `EINTR` once fatal signal received.

Flag checking should be done at the beginning of the `->fiemap` callback via the `fiemap_prep()` helper:

```
int fiemap_prep(struct inode *inode, struct fiemap_extent_info *fieinfo,
                u64 start, u64 *len, u32 supported_flags);
```

The `struct fieinfo` should be passed in as received from `ioctl_fiemap()`. The set of fiemap flags which the fs understands should be passed via `fs_flags`. If `fiemap_prep` finds invalid user flags, it will place the bad values in `fieinfo->fi_flags` and return `-EBADR`. If the file system gets `-EBADR`, from `fiemap_prep()`, it should immediately exit, returning that error back to `ioctl_fiemap()`. Additionally the range is validate against the supported maximum file size.

For each extent in the request range, the file system should call the helper function, `fiemap_fill_next_extent()`:

```
int fiemap_fill_next_extent(struct fiemap_extent_info *info, u64 logical,
                           u64 phys, u64 len, u32 flags, u32 dev);
```

`fiemap_fill_next_extent()` will use the passed values to populate the next free extent in the `fm_extents` array. 'General' extent flags will automatically be set from specific flags on behalf of the calling file system so that the userspace API is not broken.

`fiemap_fill_next_extent()` returns 0 on success, and 1 when the user-supplied `fm_extents` array is full. If an error is encountered while copying the extent to user memory, `-EFAULT` will be returned.

1.10 File management in the Linux kernel

This document describes how locking for files (`struct file`) and file descriptor table (`struct files`) works.

Up until 2.6.12, the file descriptor table has been protected with a lock (`files->file_lock`) and reference count (`files->count`). `->file_lock` protected accesses to all the file related fields of the table. `->count` was used for sharing the file descriptor table between tasks cloned with `CLONE_FILES` flag. Typically this would be the case for posix threads. As with the common refcounting model in the kernel, the last task doing a `put_files_struct()` frees the file descriptor (`fd`) table. The files (`struct file`) themselves are protected using reference count (`->f_count`).

In the new lock-free model of file descriptor management, the reference counting is similar, but the locking is based on RCU. The file descriptor table contains multiple elements - the fd sets (`open_fds` and `close_on_exec`, the array of file pointers, the sizes of the sets and the array etc.). In order for the updates to appear atomic to a lock-free reader, all the elements of the file descriptor table are in a separate structure - `struct fdtable`. `files_struct` contains a pointer to `struct fdtable` through which the actual fd table is accessed. Initially the `fdtable` is embedded in `files_struct` itself. On a subsequent expansion of `fdtable`, a new `fdtable` structure is allocated and `files->fdtab` points to the new structure. The `fdtable` structure is freed with RCU and lock-free readers either see the old `fdtable` or the new `fdtable` making the update appear atomic. Here are the locking rules for the `fdtable` structure -

1. All references to the `fdtable` must be done through the `files_fdtable()` macro:

```
struct fdtable *fdt;

rcu_read_lock();

fdt = files_fdtable(files);
....
if (n <= fdt->max_fds)
    ....
...
rcu_read_unlock();
```

`files_fdtable()` uses `rcu_dereference()` macro which takes care of the memory barrier requirements for lock-free dereference. The `fdtable` pointer must be read within the read-side critical section.

2. Reading of the `fdtable` as described above must be protected by `rcu_read_lock()/rcu_read_unlock()`.
3. For any update to the fd table, `files->file_lock` must be held.
4. To look up the file structure given an fd, a reader must use either `fcheck()` or `fcheck_files()` APIs. These take care of barrier requirements due to lock-free lookup.

An example:

```
struct file *file;

rcu_read_lock();
file = fcheck(fd);
if (file) {
    ...
}
....
rcu_read_unlock();
```

5. Handling of the file structures is special. Since the look-up of the fd (`fget()/fget_light()`) are lock-free, it is possible that look-up may race with the last `put()` operation on the file structure. This is avoided using `atomic_long_inc_not_zero()` on `->f_count`:

```
rcu_read_lock();
file = fcheck_files(files, fd);
if (file) {
    if (atomic_long_inc_not_zero(&file->f_count))
        *fput_needed = 1;
    else
        /* Didn't get the reference, someone's freed */
        file = NULL;
}
rcu_read_unlock();
....
return file;
```

`atomic_long_inc_not_zero()` detects if refcounts is already zero or goes to zero during increment. If it does, we fail `fget()/fget_light()`.

6. Since both `fdtable` and `file` structures can be looked up lock-free, they must be installed using `rcu_assign_pointer()` API. If they are looked up lock-free, `rcu_dereference()` must be used. However it is advisable to use `files_fdtable()` and `fcheck()/fcheck_files()` which take care of these issues.
7. While updating, the `fdtable` pointer must be looked up while holding `files->file_lock`. If `->file_lock` is dropped, then another thread expand the `files` thereby creating a new `fdtable` and making the earlier `fdtable` pointer stale.

For example:

```
spin_lock(&files->file_lock);
fd = locate_fd(files, file, start);
if (fd >= 0) {
    /* locate_fd() may have expanded fdtable, load the ptr */
    fdt = files_fdtable(files);
    __set_open_fd(fd, fdt);
    __clear_close_on_exec(fd, fdt);
    spin_unlock(&files->file_lock);
    ....
}
```

Since `locate_fd()` can drop `->file_lock` (and reacquire `->file_lock`), the `fdtable` pointer (`fdt`) must be loaded after `locate_fd()`.

1.11 File Locking Release Notes

Andy Walker <andy@lysaker.kvaerner.no>

12 May 1997

1.11.1 1. What' s New?

1.1 Broken Flock Emulation

The old flock(2) emulation in the kernel was swapped for proper BSD compatible flock(2) support in the 1.3.x series of kernels. With the release of the 2.1.x kernel series, support for the old emulation has been totally removed, so that we don' t need to carry this baggage forever.

This should not cause problems for anybody, since everybody using a 2.1.x kernel should have updated their C library to a suitable version anyway (see the file "Documentation/process/changes.rst" .)

1.2 Allow Mixed Locks Again

1.2.1 Typical Problems - Sendmail

Because sendmail was unable to use the old flock() emulation, many sendmail installations use fcntl() instead of flock(). This is true of Slackware 3.0 for example. This gave rise to some other subtle problems if sendmail was configured to rebuild the alias file. Sendmail tried to lock the aliases.dir file with fcntl() at the same time as the GDBM routines tried to lock this file with flock(). With pre 1.3.96 kernels this could result in deadlocks that, over time, or under a very heavy mail load, would eventually cause the kernel to lock solid with deadlocked processes.

1.2.2 The Solution

The solution I have chosen, after much experimentation and discussion, is to make flock() and fcntl() locks oblivious to each other. Both can exists, and neither will have any effect on the other.

I wanted the two lock styles to be cooperative, but there were so many race and deadlock conditions that the current solution was the only practical one. It puts us in the same position as, for example, SunOS 4.1.x and several other commercial Unices. The only OS' s that support cooperative flock()/fcntl() are those that emulate flock() using fcntl(), with all the problems that implies.

1.3 Mandatory Locking As A Mount Option

Mandatory locking, as described in ‘Documentation/filesystems/mandatory-locking.rst’ was prior to this release a general configuration option that was valid for all mounted filesystems. This had a number of inherent dangers, not the least of which was the ability to freeze an NFS server by asking it to read a file for which a mandatory lock existed.

From this release of the kernel, mandatory locking can be turned on and off on a per-filesystem basis, using the mount options ‘mand’ and ‘nomand’ . The default is to disallow mandatory locking. The intention is that mandatory locking only be enabled on a local filesystem as the specific need arises.

1.12 Mandatory File Locking For The Linux Operating System

Andy Walker <andy@lysaker.kvaerner.no>

15 April 1996

(Updated September 2007)

1.12.1 0. Why you should avoid mandatory locking

The Linux implementation is prey to a number of difficult-to-fix race conditions which in practice make it not dependable:

- The write system call checks for a mandatory lock only once at its start. It is therefore possible for a lock request to be granted after this check but before the data is modified. A process may then see file data change even while a mandatory lock was held.
- Similarly, an exclusive lock may be granted on a file after the kernel has decided to proceed with a read, but before the read has actually completed, and the reading process may see the file data in a state which should not have been visible to it.
- Similar races make the claimed mutual exclusion between lock and mmap similarly unreliable.

1.12.2 1. What is mandatory locking?

Mandatory locking is kernel enforced file locking, as opposed to the more usual cooperative file locking used to guarantee sequential access to files among processes. File locks are applied using the flock() and fcntl() system calls (and the lockf() library routine which is a wrapper around fcntl().) It is normally a process’ responsibility to check for locks on a file it wishes to update, before applying its own lock, updating the file and unlocking it again. The most commonly used example of this (and in the case of sendmail, the most troublesome) is access to a user’ s mailbox. The mail user agent and the mail transfer agent must guard against

updating the mailbox at the same time, and prevent reading the mailbox while it is being updated.

In a perfect world all processes would use and honour a cooperative, or “advisory” locking scheme. However, the world isn’ t perfect, and there’ s a lot of poorly written code out there.

In trying to address this problem, the designers of System V UNIX came up with a “mandatory” locking scheme, whereby the operating system kernel would block attempts by a process to write to a file that another process holds a “read” -or- “shared” lock on, and block attempts to both read and write to a file that a process holds a “write ” -or- “exclusive” lock on.

The System V mandatory locking scheme was intended to have as little impact as possible on existing user code. The scheme is based on marking individual files as candidates for mandatory locking, and using the existing `fcntl()/lockf()` interface for applying locks just as if they were normal, advisory locks.

Note:

1. In saying “file” in the paragraphs above I am actually not telling the whole truth. System V locking is based on `fcntl()`. The granularity of `fcntl()` is such that it allows the locking of byte ranges in files, in addition to entire files, so the mandatory locking rules also have byte level granularity.
 2. POSIX.1 does not specify any scheme for mandatory locking, despite borrowing the `fcntl()` locking scheme from System V. The mandatory locking scheme is defined by the System V Interface Definition (SVID) Version 3.
-

1.12.3 2. Marking a file for mandatory locking

A file is marked as a candidate for mandatory locking by setting the group-id bit in its file mode but removing the group-execute bit. This is an otherwise meaningless combination, and was chosen by the System V implementors so as not to break existing user programs.

Note that the group-id bit is usually automatically cleared by the kernel when a `setgid` file is written to. This is a security measure. The kernel has been modified to recognize the special case of a mandatory lock candidate and to refrain from clearing this bit. Similarly the kernel has been modified not to run mandatory lock candidates with `setgid` privileges.

1.12.4 3. Available implementations

I have considered the implementations of mandatory locking available with SunOS 4.1.x, Solaris 2.x and HP-UX 9.x.

Generally I have tried to make the most sense out of the behaviour exhibited by these three reference systems. There are many anomalies.

All the reference systems reject all calls to `open()` for a file on which another process has outstanding mandatory locks. This is in direct contravention of SVID 3,

which states that only calls to `open()` with the `O_TRUNC` flag set should be rejected. The Linux implementation follows the SVID definition, which is the “Right Thing”, since only calls with `O_TRUNC` can modify the contents of the file.

HP-UX even disallows `open()` with `O_TRUNC` for a file with advisory locks, not just mandatory locks. That would appear to contravene POSIX.1.

`mmap()` is another interesting case. All the operating systems mentioned prevent mandatory locks from being applied to an `mmap()`'ed file, but HP-UX also disallows advisory locks for such a file. SVID actually specifies the paranoid HP-UX behaviour.

In my opinion only `MAP_SHARED` mappings should be immune from locking, and then only from mandatory locks - that is what is currently implemented.

SunOS is so hopeless that it doesn't even honour the `O_NONBLOCK` flag for mandatory locks, so reads and writes to locked files always block when they should return `EAGAIN`.

I'm afraid that this is such an esoteric area that the semantics described below are just as valid as any others, so long as the main points seem to agree.

1.12.5 4. Semantics

1. Mandatory locks can only be applied via the `fcntl()/lockf()` locking interface - in other words the System V/POSIX interface. BSD style locks using `flock()` never result in a mandatory lock.
2. If a process has locked a region of a file with a mandatory read lock, then other processes are permitted to read from that region. If any of these processes attempts to write to the region it will block until the lock is released, unless the process has opened the file with the `O_NONBLOCK` flag in which case the system call will return immediately with the error status `EAGAIN`.
3. If a process has locked a region of a file with a mandatory write lock, all attempts to read or write to that region block until the lock is released, unless a process has opened the file with the `O_NONBLOCK` flag in which case the system call will return immediately with the error status `EAGAIN`.
4. Calls to `open()` with `O_TRUNC`, or to `creat()`, on an existing file that has any mandatory locks owned by other processes will be rejected with the error status `EAGAIN`.
5. Attempts to apply a mandatory lock to a file that is memory mapped and shared (via `mmap()` with `MAP_SHARED`) will be rejected with the error status `EAGAIN`.
6. Attempts to create a shared memory map of a file (via `mmap()` with `MAP_SHARED`) that has any mandatory locks in effect will be rejected with the error status `EAGAIN`.

1.12.6 5. Which system calls are affected?

Those which modify a file's contents, not just the inode. That gives `read()`, `write()`, `readv()`, `writev()`, `open()`, `creat()`, `mmap()`, `truncate()` and `ftruncate()`. `truncate()` and `ftruncate()` are considered to be “write” actions for the purposes of mandatory locking.

The affected region is usually defined as stretching from the current position for the total number of bytes read or written. For the `truncate` calls it is defined as the bytes of a file removed or added (we must also consider bytes added, as a lock can specify just “the whole file” , rather than a specific range of bytes.)

Note 3: I may have overlooked some system calls that need mandatory lock checking in my eagerness to get this code out the door. Please let me know, or better still fix the system calls yourself and submit a patch to me or Linus.

1.12.7 6. Warning!

Not even root can override a mandatory lock, so runaway processes can wreak havoc if they lock crucial files. The way around it is to change the file permissions (remove the `setgid` bit) before trying to read or write to it. Of course, that might be a bit tricky if the system is hung :-)

1.12.8 7. The “mand” mount option

Mandatory locking is disabled on all filesystems by default, and must be administratively enabled by mounting with “`-o mand`”. That mount option is only allowed if the mounting task has the `CAP_SYS_ADMIN` capability.

Since kernel v4.5, it is possible to disable mandatory locking altogether by setting `CONFIG_MANDATORY_FILE_LOCKING` to “`n`”. A kernel with this disabled will reject attempts to mount filesystems with the “`mand`” mount option with the error status `EPERM`.

1.13 FILESYSTEM Mount API

1.13.1 Overview

The creation of new mounts is now to be done in a multistep process:

- (1) Create a filesystem context.
- (2) Parse the parameters and attach them to the context. Parameters are expected to be passed individually from userspace, though legacy binary parameters can also be handled.
- (3) Validate and pre-process the context.
- (4) Get or create a superblock and mountable root.
- (5) Perform the mount.
- (6) Return an error message attached to the context.

(7) Destroy the context.

To support this, the `file_system_type` struct gains two new fields:

```
int (*init_fs_context)(struct fs_context *fc);
const struct fs_parameter_description *parameters;
```

The first is invoked to set up the filesystem-specific parts of a filesystem context, including the additional space, and the second points to the parameter description for validation at registration time and querying by a future system call.

Note that security initialisation is done after the filesystem is called so that the namespaces may be adjusted first.

1.13.2 The Filesystem context

The creation and reconfiguration of a superblock is governed by a filesystem context. This is represented by the `fs_context` structure:

```
struct fs_context {
    const struct fs_context_operations *ops;
    struct file_system_type *fs_type;
    void *fs_private;
    struct dentry *root;
    struct user_namespace *user_ns;
    struct net *net_ns;
    const struct cred *cred;
    char *source;
    char *subtype;
    void *security;
    void *s_fs_info;
    unsigned int sb_flags;
    unsigned int sb_flags_mask;
    unsigned int s_iflags;
    unsigned int lsm_flags;
    enum fs_context_purpose purpose:8;
    ...
};
```

The `fs_context` fields are as follows:

- `const struct fs_context_operations *ops`

These are operations that can be done on a filesystem context (see below). This must be set by the `->init_fs_context()` `file_system_type` operation.

- `struct file_system_type *fs_type`

A pointer to the `file_system_type` of the filesystem that is being constructed or reconfigured. This retains a reference on the type owner.

- `void *fs_private`

A pointer to the file system's private data. This is where the filesystem will need to store any options it parses.

- `struct dentry *root`

A pointer to the root of the mountable tree (and indirectly, the superblock thereof). This is filled in by the `->get_tree()` op. If this is set, an active reference on `root->d_sb` must also be held.

- `struct user_namespace *user_ns`
`struct net *net_ns`

There are a subset of the namespaces in use by the invoking process. They retain references on each namespace. The subscribed namespaces may be replaced by the filesystem to reflect other sources, such as the parent mount superblock on an automount.

- `const struct cred *cred`

The mounter's credentials. This retains a reference on the credentials.

- `char *source`

This specifies the source. It may be a block device (e.g. `/dev/sda1`) or something more exotic, such as the “`host:/path`” that NFS desires.

- `char *subtype`

This is a string to be added to the type displayed in `/proc/mounts` to qualify it (used by FUSE). This is available for the filesystem to set if desired.

- `void *security`

A place for the LSMs to hang their security data for the superblock. The relevant security operations are described below.

- `void *s_fs_info`

The proposed `s_fs_info` for a new superblock, set in the superblock by `sget_fc()`. This can be used to distinguish superblocks.

- `unsigned int sb_flags`
`unsigned int sb_flags_mask`

Which bits `SB_*` flags are to be set/cleared in `super_block::s_flags`.

- `unsigned int s_iflags`

These will be bitwise-OR'd with `s->s_iflags` when a superblock is created.

- `enum fs_context_purpose`

This indicates the purpose for which the context is intended. The available values are:

FS_CONTEXT_FOR_MOUNT,	New superblock for explicit mount
FS_CONTEXT_FOR_SUBMOUNT,	New automatic submount of extant mount
FS_CONTEXT_FOR_RECONFIGURE,	Change an existing mount

The mount context is created by calling `vfs_new_fs_context()` or `vfs_dup_fs_context()` and is destroyed with `put_fs_context()`. Note that the structure is not refcounted.

VFS, security and filesystem mount options are set individually with `vfs_parse_mount_option()`. Options provided by the old `mount(2)` system call as a page of data can be parsed with `generic_parse_monolithic()`.

When mounting, the filesystem is allowed to take data from any of the pointers and attach it to the superblock (or whatever), provided it clears the pointer in the mount context.

The filesystem is also allowed to allocate resources and pin them with the mount context. For instance, NFS might pin the appropriate protocol version module.

1.13.3 The Filesystem Context Operations

The filesystem context points to a table of operations:

```
struct fs_context_operations {
    void (*free)(struct fs_context *fc);
    int (*dup)(struct fs_context *fc, struct fs_context *src_fc);
    int (*parse_param)(struct fs_context *fc,
                      struct fs_parameter *param);
    int (*parse_monolithic)(struct fs_context *fc, void *data);
    int (*get_tree)(struct fs_context *fc);
    int (*reconfigure)(struct fs_context *fc);
};
```

These operations are invoked by the various stages of the mount procedure to manage the filesystem context. They are as follows:

- `void (*free)(struct fs_context *fc);`

Called to clean up the filesystem-specific part of the filesystem context when the context is destroyed. It should be aware that parts of the context may have been removed and NULL'd out by `->get_tree()`.

- `int (*dup)(struct fs_context *fc, struct fs_context *src_fc);`

Called when a filesystem context has been duplicated to duplicate the filesystem-private data. An error may be returned to indicate failure to do this.

Warning: Note that even if this fails, `put_fs_context()` will be called immediately thereafter, so `->dup()` must make the filesystem-private data safe for `->free()`.

- ```
int (*parse_param)(struct fs_context *fc,
 struct fs_parameter *param);
```

Called when a parameter is being added to the filesystem context. `param` points to the key name and maybe a value object. VFS-specific options will have been weeded out and `fc->sb_flags` updated in the context. Security options will also have been weeded out and `fc->security` updated.

The parameter can be parsed with `fs_parse()` and `fs_lookup_param()`. Note that the source(s) are presented as parameters named “source” .

If successful, 0 should be returned or a negative error code otherwise.

- ```
int (*parse_monolithic)(struct fs_context *fc, void *data);
```

Called when the `mount(2)` system call is invoked to pass the entire data page in one go. If this is expected to be just a list of “key[=val]” items separated by commas, then this may be set to `NULL`.

The return value is as for `->parse_param()`.

If the filesystem (e.g. NFS) needs to examine the data first and then finds it's the standard key-val list then it may pass it off to `generic_parse_monolithic()`.

- ```
int (*get_tree)(struct fs_context *fc);
```

Called to get or create the mountable root and superblock, using the information stored in the filesystem context (reconfiguration goes via a different vector). It may detach any resources it desires from the filesystem context and transfer them to the superblock it creates.

On success it should set `fc->root` to the mountable root and return 0. In the case of an error, it should return a negative error code.

The phase on a userspace-driven context will be set to only allow this to be called once on any particular context.

- ```
int (*reconfigure)(struct fs_context *fc);
```

Called to effect reconfiguration of a superblock using information stored in the filesystem context. It may detach any resources it desires from the filesystem context and transfer them to the superblock. The superblock can be found from `fc->root->d_sb`.

On success it should return 0. In the case of an error, it should return a negative error code.

Note: `reconfigure` is intended as a replacement for `remount_fs`.

1.13.4 Filesystem context Security

The filesystem context contains a security pointer that the LSMs can use for building up a security context for the superblock to be mounted. There are a number of operations used by the new mount code for this purpose:

- ```
int security_fs_context_alloc(struct fs_context *fc,
 struct dentry *reference);
```

Called to initialise `fc->security` (which is preset to `NULL`) and allocate any resources needed. It should return 0 on success or a negative error code on failure.

`reference` will be non-`NULL` if the context is being created for superblock reconfiguration (`FS_CONTEXT_FOR_RECONFIGURE`) in which case it indicates the root dentry of the superblock to be reconfigured. It will also be non-`NULL` in the case of a submount (`FS_CONTEXT_FOR_SUBMOUNT`) in which case it indicates the automount point.

- ```
int security_fs_context_dup(struct fs_context *fc,
                           struct fs_context *src_fc);
```

Called to initialise `fc->security` (which is preset to `NULL`) and allocate any resources needed. The original filesystem context is pointed to by `src_fc` and may be used for reference. It should return 0 on success or a negative error code on failure.

- ```
void security_fs_context_free(struct fs_context *fc);
```

Called to clean up anything attached to `fc->security`. Note that the contents may have been transferred to a superblock and the pointer cleared during `get_tree`.

- ```
int security_fs_context_parse_param(struct fs_context *fc,
                                   struct fs_parameter *param);
```

Called for each mount parameter, including the source. The arguments are as for the `->parse_param()` method. It should return 0 to indicate that the parameter should be passed on to the filesystem, 1 to indicate that the parameter should be discarded or an error to indicate that the parameter should be rejected.

The value pointed to by `param` may be modified (if a string) or stolen (provided the value pointer is `NULL`'d out). If it is stolen, 1 must be returned to prevent it being passed to the filesystem.

- ```
int security_fs_context_validate(struct fs_context *fc);
```

Called after all the options have been parsed to validate the collection as a whole and to do any necessary allocation so that `security_sb_get_tree()` and `security_sb_reconfigure()` are less likely to fail. It should return 0 or a negative error code.

In the case of reconfiguration, the target superblock will be accessible via `fc->root`.

```
• int security_sb_get_tree(struct fs_context *fc);
```

Called during the mount procedure to verify that the specified superblock is allowed to be mounted and to transfer the security data there. It should return 0 or a negative error code.

```
• void security_sb_reconfigure(struct fs_context *fc);
```

Called to apply any reconfiguration to an LSM's context. It must not fail. Error checking and resource allocation must be done in advance by the parameter parsing and validation hooks.

```
• int security_sb_mountpoint(struct fs_context *fc,
 struct path *mountpoint,
 unsigned int mnt_flags);
```

Called during the mount procedure to verify that the root dentry attached to the context is permitted to be attached to the specified mountpoint. It should return 0 on success or a negative error code on failure.

### 1.13.5 VFS Filesystem context API

There are four operations for creating a filesystem context and one for destroying a context:

```
• struct fs_context *fs_context_for_mount(struct file_system_type *fs_
 ↪type,
 unsigned int sb_flags);
```

Allocate a filesystem context for the purpose of setting up a new mount, whether that be with a new superblock or sharing an existing one. This sets the superblock flags, initialises the security and calls `fs_type->init_fs_context()` to initialise the filesystem private data.

`fs_type` specifies the filesystem type that will manage the context and `sb_flags` presets the superblock flags stored therein.

```
• struct fs_context *fs_context_for_reconfigure(
 struct dentry *dentry,
 unsigned int sb_flags,
 unsigned int sb_flags_mask);
```

Allocate a filesystem context for the purpose of reconfiguring an existing superblock. `dentry` provides a reference to the superblock to be configured. `sb_flags` and `sb_flags_mask` indicate which superblock flags need changing and to what.

```
• struct fs_context *fs_context_for_submount(
 struct file_system_type *fs_type,
 struct dentry *reference);
```

Allocate a filesystem context for the purpose of creating a new mount for an automount point or other derived superblock. `fs_type` specifies the filesystem type that will manage the context and the reference `dentry` supplies the

parameters. Namespaces are propagated from the reference dentry's superblock also.

Note that it's not a requirement that the reference dentry be of the same filesystem type as `fs_type`.

- ```
struct fs_context *vfs_dup_fs_context(struct fs_context *src_fc);
```

Duplicate a filesystem context, copying any options noted and duplicating or additionally referencing any resources held therein. This is available for use where a filesystem has to get a mount within a mount, such as NFS4 does by internally mounting the root of the target server and then doing a private pathwalk to the target directory.

The purpose in the new context is inherited from the old one.

- ```
void put_fs_context(struct fs_context *fc);
```

Destroy a filesystem context, releasing any resources it holds. This calls the `->free()` operation. This is intended to be called by anyone who created a filesystem context.

**Warning:** filesystem contexts are not refcounted, so this causes unconditional destruction.

In all the above operations, apart from the `put` op, the return is a mount context pointer or a negative error code.

For the remaining operations, if an error occurs, a negative error code will be returned.

- ```
int vfs_parse_fs_param(struct fs_context *fc,  
                      struct fs_parameter *param);
```

Supply a single mount parameter to the filesystem context. This include the specification of the source/device which is specified as the "source" parameter (which may be specified multiple times if the filesystem supports that).

`param` specifies the parameter key name and the value. The parameter is first checked to see if it corresponds to a standard mount flag (in which case it is used to set an `SB_xxx` flag and consumed) or a security option (in which case the LSM consumes it) before it is passed on to the filesystem.

The parameter value is typed and can be one of:

<code>fs_value_is_flag</code>	Parameter not given a value
<code>fs_value_is_string</code>	Value is a string
<code>fs_value_is_blob</code>	Value is a binary blob
<code>fs_value_is_filename</code>	Value is a filename* + <code>dirfd</code>
<code>fs_value_is_file</code>	Value is an open file (file*)

If there is a value, that value is stored in a union in the struct in one of `param->{string,blob,name,file}`. Note that the function may steal and clear

the pointer, but then becomes responsible for disposing of the object.

- ```
int vfs_parse_fs_string(struct fs_context *fc, const char *key,
 const char *value, size_t v_size);
```

A wrapper around `vfs_parse_fs_param()` that copies the value string it is passed.

- ```
int generic_parse_monolithic(struct fs_context *fc, void *data);
```

Parse a `sys_mount()` data page, assuming the form to be a text list consisting of `key[=val]` options separated by commas. Each item in the list is passed to `vfs_mount_option()`. This is the default when the `->parse_monolithic()` method is NULL.

- ```
int vfs_get_tree(struct fs_context *fc);
```

Get or create the mountable root and superblock, using the parameters in the filesystem context to select/configure the superblock. This invokes the `->get_tree()` method.

- ```
struct vfsmount *vfs_create_mount(struct fs_context *fc);
```

Create a mount given the parameters in the specified filesystem context. Note that this does not attach the mount to anything.

1.13.6 Superblock Creation Helpers

A number of VFS helpers are available for use by filesystems for the creation or looking up of superblocks.

- ```
struct super_block *
sget_fc(struct fs_context *fc,
 int (*test)(struct super_block *sb, struct fs_context *fc),
 int (*set)(struct super_block *sb, struct fs_context *fc));
```

This is the core routine. If `test` is non-NULL, it searches for an existing superblock matching the criteria held in the `fs_context`, using the `test` function to match them. If no match is found, a new superblock is created and the `set` function is called to set it up.

Prior to the `set` function being called, `fc->s_fs_info` will be transferred to `sb->s_fs_info` - and `fc->s_fs_info` will be cleared if `set` returns success (ie. 0).

The following helpers all wrap `sget_fc()`:

- ```
int vfs_get_super(struct fs_context *fc,
                 enum vfs_get_super_keying keying,
                 int (*fill_super)(struct super_block *sb,
                                   struct fs_context *fc))
```

This creates/looks up a deviceless superblock. The `keying` indicates how many superblocks of this type may exist and in what manner they may be shared:

- (1) `vfs_get_single_super`

Only one such superblock may exist in the system. Any further attempt to get a new superblock gets this one (and any parameter differences are ignored).

(2) `vfs_get_keyed_super`

Multiple superblocks of this type may exist and they're keyed on their `s_fs_info` pointer (for example this may refer to a namespace).

(3) `vfs_get_independent_super`

Multiple independent superblocks of this type may exist. This function never matches an existing one and always creates a new one.

1.14 PARAMETER DESCRIPTION

Parameters are described using structures defined in `linux/fs_parser.h`. There's a core description struct that links everything together:

```
struct fs_parameter_description {
    const struct fs_parameter_spec *specs;
    const struct fs_parameter_enum *enums;
};
```

For example:

```
enum {
    Opt_autocell,
    Opt_bar,
    Opt_dyn,
    Opt_foo,
    Opt_source,
};

static const struct fs_parameter_description afs_fs_parameters = {
    .specs      = afs_param_specs,
    .enums      = afs_param_enums,
};
```

The members are as follows:

(1) `const struct fs_parameter_specification *specs;`

Table of parameter specifications, terminated with a null entry, where the entries are of type:

```
struct fs_parameter_spec {
    const char      *name;
    u8              opt;
    enum fs_parameter_type type:8;
    unsigned short  flags;
};
```

The 'name' field is a string to match exactly to the parameter key (no wildcards, patterns and no case-independence) and 'opt' is the value that will be returned by the `fs_parser()` function in the case of a successful match.

The ‘type’ field indicates the desired value type and must be one of:

TYPE NAME	EXPECTED VALUE	RESULT IN
fs_param_is_flag	No value	n/a
fs_param_is_bool	Boolean value	result->boolean
fs_param_is_u32	32-bit unsigned int	result->uint_32
fs_param_is_u32_octal	32-bit octal int	result->uint_32
fs_param_is_u32_hex	32-bit hex int	result->uint_32
fs_param_is_s32	32-bit signed int	result->int_32
fs_param_is_u64	64-bit unsigned int	result->uint_64
fs_param_is_enum	Enum value name	result->uint_32
fs_param_is_string	Arbitrary string	param->string
fs_param_is_blob	Binary blob	param->blob
fs_param_is_blockdev	Blockdev path	<ul style="list-style-type: none"> • Needs lookup
fs_param_is_path	Path	<ul style="list-style-type: none"> • Needs lookup
fs_param_is_fd	File descriptor	result->int_32

Note that if the value is of fs_param_is_bool type, fs_parse() will try to match any string value against “0” , “1” , “no” , “yes” , “false” , “true” .

Each parameter can also be qualified with ‘flags’ :

fs_param_v_optional	The value is optional
fs_param_neg_with_no	result->negated set if key is prefixed with “no”
fs_param_neg_with_empty	result->negated set if value is “”
fs_param_deprecated	The parameter is deprecated.

These are wrapped with a number of convenience wrappers:

MACRO	SPECIFIES
fsparam_flag()	fs_param_is_flag
fsparam_flag_no()	fs_param_is_flag, fs_param_neg_with_no
fsparam_bool()	fs_param_is_bool
fsparam_u32()	fs_param_is_u32
fsparam_u32oct()	fs_param_is_u32_octal
fsparam_u32hex()	fs_param_is_u32_hex
fsparam_s32()	fs_param_is_s32
fsparam_u64()	fs_param_is_u64
fsparam_enum()	fs_param_is_enum
fsparam_string()	fs_param_is_string
fsparam_blob()	fs_param_is_blob
fsparam_bdev()	fs_param_is_blockdev
fsparam_path()	fs_param_is_path
fsparam_fd()	fs_param_is_fd

all of which take two arguments, name string and option number - for example:

```
static const struct fs_parameter_spec afs_param_specs[] = {
    fsparam_flag    ("autocell",    Opt_autocell),
    fsparam_flag    ("dyn",         Opt_dyn),
    fsparam_string  ("source",      Opt_source),
    fsparam_flag_no ("foo",         Opt_foo),
    {}
};
```

An addition macro, `__fsparam()` is provided that takes an additional pair of arguments to specify the type and the flags for anything that doesn't match one of the above macros.

(2) `const struct fs_parameter_enum *enums;`

Table of enum value names to integer mappings, terminated with a null entry. This is of type:

```
struct fs_parameter_enum {
    u8      opt;
    char    name[14];
    u8      value;
};
```

Where the array is an unsorted list of { parameter ID, name }-keyed elements that indicate the value to map to, e.g.:

```
static const struct fs_parameter_enum afs_param_enums[] = {
    { Opt_bar,  "x",    1},
    { Opt_bar,  "y",    23},
    { Opt_bar,  "z",    42},
};
```

If a parameter of type `fs_param_is_enum` is encountered, `fs_parse()` will try to look the value up in the enum table and the result will be stored in the parse result.

The parser should be pointed to by the parser pointer in the `file_system_type` struct as this will provide validation on registration (if `CONFIG_VALIDATE_FS_PARSER=y`) and will allow the description to be queried from userspace using the `fsinfo()` syscall.

1.14.1 Parameter Helper Functions

A number of helper functions are provided to help a filesystem or an LSM process the parameters it is given.

- `int lookup_constant(const struct constant_table tbl[], const char *name, int not_found);`

Look up a constant by name in a table of name -> integer mappings. The table is an array of elements of the following type:

```
struct constant_table {
    const char    *name;
    int           value;
};
```

If a match is found, the corresponding value is returned. If a match isn't found, the `not_found` value is returned instead.

- ```
bool validate_constant_table(const struct constant_table *tbl,
 size_t tbl_size,
 int low, int high, int special);
```

Validate a constant table. Checks that all the elements are appropriately ordered, that there are no duplicates and that the values are between low and high inclusive, though provision is made for one allowable special value outside of that range. If no special value is required, `special` should just be set to lie inside the low-to-high range.

If all is good, `true` is returned. If the table is invalid, errors are logged to `dmesg` and `false` is returned.

- ```
bool fs_validate_description(const struct fs_parameter_description
↪ *desc);
```

This performs some validation checks on a parameter description. It returns `true` if the description is good and `false` if it is not. It will log errors to `dmesg` if validation fails.

- ```
int fs_parse(struct fs_context *fc,
 const struct fs_parameter_description *desc,
 struct fs_parameter *param,
 struct fs_parse_result *result);
```

This is the main interpreter of parameters. It uses the parameter description to look up a parameter by key name and to convert that to an option number (which it returns).

If successful, and if the parameter type indicates the result is a boolean, integer or enum type, the value is converted by this function and the result stored in `result->{boolean,int_32,uint_32,uint_64}`.

If a match isn't initially made, the key is prefixed with "no" and no value is present then an attempt will be made to look up the key with the prefix removed. If this matches a parameter for which the type has flag `fs_param_neg_with_no` set, then a match will be made and `result->negated` will be set to `true`.

If the parameter isn't matched, `-ENOPARAM` will be returned; if the parameter is matched, but the value is erroneous, `-EINVAL` will be returned; otherwise the parameter's option number will be returned.

- ```
int fs_lookup_param(struct fs_context *fc,
                   struct fs_parameter *value,
                   bool want_bdev,
                   struct path *_path);
```

This takes a parameter that carries a string or filename type and attempts to do a path lookup on it. If the parameter expects a blockdev, a check is made that the inode actually represents one.

Returns 0 if successful and *_path will be set; returns a negative error code if not.

1.15 Quota subsystem

Quota subsystem allows system administrator to set limits on used space and number of used inodes (inode is a filesystem structure which is associated with each file or directory) for users and/or groups. For both used space and number of used inodes there are actually two limits. The first one is called softlimit and the second one hardlimit. A user can never exceed a hardlimit for any resource (unless he has CAP_SYS_RESOURCE capability). User is allowed to exceed softlimit but only for limited period of time. This period is called “grace period” or “grace time” . When grace time is over, user is not able to allocate more space/inodes until he frees enough of them to get below softlimit.

Quota limits (and amount of grace time) are set independently for each filesystem.

For more details about quota design, see the documentation in quota-tools package (<http://sourceforge.net/projects/linuxquota>).

1.15.1 Quota netlink interface

When user exceeds a softlimit, runs out of grace time or reaches hardlimit, quota subsystem traditionally printed a message to the controlling terminal of the process which caused the excess. This method has the disadvantage that when user is using a graphical desktop he usually cannot see the message. Thus quota netlink interface has been designed to pass information about the above events to userspace. There they can be captured by an application and processed accordingly.

The interface uses generic netlink framework (see <http://lwn.net/Articles/208755/> and <http://people.suug.ch/~tgr/libnl/> for more details about this layer). The name of the quota generic netlink interface is “VFS_DQUOT” . Definitions of constants below are in <linux/quota.h>. Since the quota netlink protocol is not namespace aware, quota netlink messages are sent only in initial network namespace.

Currently, the interface supports only one message type QUOTA_NL_C_WARNING. This command is used to send a notification about any of the above mentioned events. Each message has six attributes. These are (type of the argument is in parentheses):

QUOTA_NL_A_QTYPE (u32)

- type of quota being exceeded (one of USRQUOTA, GRPQUOTA)

QUOTA_NL_A_EXCESS_ID (u64)

- UID/GID (depends on quota type) of user / group whose limit is being exceeded.

QUOTA_NL_A_CAUSED_ID (u64)

- UID of a user who caused the event

QUOTA_NL_A_WARNING (u32)

- what kind of limit is exceeded:

QUOTA_NL_IHARDWARN inode hardlimit

QUOTA_NL_ISOFTLONGWARN inode softlimit is exceeded longer than given grace period

QUOTA_NL_ISOFTWARN inode softlimit

QUOTA_NL_BHARDWARN space (block) hardlimit

QUOTA_NL_BSOFTLONGWARN space (block) softlimit is exceeded longer than given grace period.

QUOTA_NL_BSOFTWARN space (block) softlimit

- four warnings are also defined for the event when user stops exceeding some limit:

QUOTA_NL_IHARDBELOW inode hardlimit

QUOTA_NL_ISOFTBELOW inode softlimit

QUOTA_NL_BHARDBELOW space (block) hardlimit

QUOTA_NL_BSOFTBELOW space (block) softlimit

QUOTA_NL_A_DEV_MAJOR (u32)

- major number of a device with the affected filesystem

QUOTA_NL_A_DEV_MINOR (u32)

- minor number of a device with the affected filesystem

1.16 The seq_file Interface

Copyright 2003 Jonathan Corbet <corbet@lwn.net>

This file is originally from the LWN.net Driver Porting series at <https://lwn.net/Articles/driver-porting/>

There are numerous ways for a device driver (or other kernel component) to provide information to the user or system administrator. One useful technique is the creation of virtual files, in debugfs, /proc or elsewhere. Virtual files can provide human-readable output that is easy to get at without any special utility programs; they can also make life easier for script writers. It is not surprising that the use of virtual files has grown over the years.

Creating those files correctly has always been a bit of a challenge, however. It is not that hard to make a virtual file which returns a string. But life gets trickier if the output is long - anything greater than an application is likely to read in a single operation. Handling multiple reads (and seeks) requires careful attention to the reader's position within the virtual file - that position is, likely as not, in the middle

of a line of output. The kernel has traditionally had a number of implementations that got this wrong.

The 2.6 kernel contains a set of functions (implemented by Alexander Viro) which are designed to make it easy for virtual file creators to get it right.

The `seq_file` interface is available via `<linux/seq_file.h>`. There are three aspects to `seq_file`:

- An iterator interface which lets a virtual file implementation step through the objects it is presenting.
- Some utility functions for formatting objects for output without needing to worry about things like output buffers.
- A set of canned `file_operations` which implement most operations on the virtual file.

We'll look at the `seq_file` interface via an extremely simple example: a loadable module which creates a file called `/proc/sequence`. The file, when read, simply produces a set of increasing integer values, one per line. The sequence will continue until the user loses patience and finds something better to do. The file is seekable, in that one can do something like the following:

```
dd if=/proc/sequence of=out1 count=1
dd if=/proc/sequence skip=1 of=out2 count=1
```

Then concatenate the output files `out1` and `out2` and get the right result. Yes, it is a thoroughly useless module, but the point is to show how the mechanism works without getting lost in other details. (Those wanting to see the full source for this module can find it at <https://lwn.net/Articles/22359/>).

1.16.1 Deprecated `create_proc_entry`

Note that the above article uses `create_proc_entry` which was removed in kernel 3.10. Current versions require the following update:

```
- entry = create_proc_entry("sequence", 0, NULL);
- if (entry)
-     entry->proc_fops = &ct_file_ops;
+ entry = proc_create("sequence", 0, NULL, &ct_file_ops);
```

1.16.2 The iterator interface

Modules implementing a virtual file with `seq_file` must implement an iterator object that allows stepping through the data of interest during a “session” (roughly one `read()` system call). If the iterator is able to move to a specific position - like the file they implement, though with freedom to map the position number to a sequence location in whatever way is convenient - the iterator need only exist transiently during a session. If the iterator cannot easily find a numerical position but works well with a `first/next` interface, the iterator can be stored in the private data area and continue from one session to the next.

A `seq_file` implementation that is formatting firewall rules from a table, for example, could provide a simple iterator that interprets position `N` as the `N`th rule in the chain. A `seq_file` implementation that presents the content of a, potentially volatile, linked list might record a pointer into that list, providing that can be done without risk of the current location being removed.

Positioning can thus be done in whatever way makes the most sense for the generator of the data, which need not be aware of how a position translates to an offset in the virtual file. The one obvious exception is that a position of zero should indicate the beginning of the file.

The `/proc/sequence` iterator just uses the count of the next number it will output as its position.

Four functions must be implemented to make the iterator work. The first, called `start()`, starts a session and takes a position as an argument, returning an iterator which will start reading at that position. The `pos` passed to `start()` will always be either zero, or the most recent `pos` used in the previous session.

For our simple sequence example, the `start()` function looks like:

```
static void *ct_seq_start(struct seq_file *s, loff_t *pos)
{
    loff_t *spos = kmalloc(sizeof(loff_t), GFP_KERNEL);
    if (!spos)
        return NULL;
    *spos = *pos;
    return spos;
}
```

The entire data structure for this iterator is a single `loff_t` value holding the current position. There is no upper bound for the sequence iterator, but that will not be the case for most other `seq_file` implementations; in most cases the `start()` function should check for a “past end of file” condition and return `NULL` if need be.

For more complicated applications, the private field of the `seq_file` structure can be used to hold state from session to session. There is also a special value which can be returned by the `start()` function called `SEQ_START_TOKEN`; it can be used if you wish to instruct your `show()` function (described below) to print a header at the top of the output. `SEQ_START_TOKEN` should only be used if the offset is zero, however.

The next function to implement is called, amazingly, `next()`; its job is to move the iterator forward to the next position in the sequence. The example module can simply increment the position by one; more useful modules will do what is needed to step through some data structure. The `next()` function returns a new iterator, or `NULL` if the sequence is complete. Here’s the example version:

```
static void *ct_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    loff_t *spos = v;
    *pos = ++*spos;
    return spos;
}
```

The `stop()` function closes a session; its job, of course, is to clean up. If dynamic

memory is allocated for the iterator, `stop()` is the place to free it; if a lock was taken by `start()`, `stop()` must release that lock. The value that `*pos` was set to by the last `next()` call before `stop()` is remembered, and used for the first `start()` call of the next session unless `lseek()` has been called on the file; in that case next `start()` will be asked to start at position zero:

```
static void ct_seq_stop(struct seq_file *s, void *v)
{
    kfree(v);
}
```

Finally, the `show()` function should format the object currently pointed to by the iterator for output. The example module's `show()` function is:

```
static int ct_seq_show(struct seq_file *s, void *v)
{
    loff_t *spos = v;
    seq_printf(s, "%lld\n", (long long)*spos);
    return 0;
}
```

If all is well, the `show()` function should return zero. A negative error code in the usual manner indicates that something went wrong; it will be passed back to user space. This function can also return `SEQ_SKIP`, which causes the current item to be skipped; if the `show()` function has already generated output before returning `SEQ_SKIP`, that output will be dropped.

We will look at `seq_printf()` in a moment. But first, the definition of the `seq_file` iterator is finished by creating a `seq_operations` structure with the four functions we have just defined:

```
static const struct seq_operations ct_seq_ops = {
    .start = ct_seq_start,
    .next  = ct_seq_next,
    .stop  = ct_seq_stop,
    .show  = ct_seq_show
};
```

This structure will be needed to tie our iterator to the `/proc` file in a little bit.

It's worth noting that the iterator value returned by `start()` and manipulated by the other functions is considered to be completely opaque by the `seq_file` code. It can thus be anything that is useful in stepping through the data to be output. Counters can be useful, but it could also be a direct pointer into an array or linked list. Anything goes, as long as the programmer is aware that things can happen between calls to the iterator function. However, the `seq_file` code (by design) will not sleep between the calls to `start()` and `stop()`, so holding a lock during that time is a reasonable thing to do. The `seq_file` code will also avoid taking any other locks while the iterator is active.

1.16.3 Formatted output

The `seq_file` code manages positioning within the output created by the iterator and getting it into the user's buffer. But, for that to work, that output must be passed to the `seq_file` code. Some utility functions have been defined which make this task easy.

Most code will simply use `seq_printf()`, which works pretty much like `printf()`, but which requires the `seq_file` pointer as an argument.

For straight character output, the following functions may be used:

```
seq_putc(struct seq_file *m, char c);
seq_puts(struct seq_file *m, const char *s);
seq_escape(struct seq_file *m, const char *s, const char *esc);
```

The first two output a single character and a string, just like one would expect. `seq_escape()` is like `seq_puts()`, except that any character in `s` which is in the string `esc` will be represented in octal form in the output.

There are also a pair of functions for printing filenames:

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

Here, `path` indicates the file of interest, and `esc` is a set of characters which should be escaped in the output. A call to `seq_path()` will output the path relative to the current process's filesystem root. If a different root is desired, it can be used with `seq_path_root()`. If it turns out that `path` cannot be reached from `root`, `seq_path_root()` returns `SEQ_SKIP`.

A function producing complicated output may want to check:

```
bool seq_has_overflowed(struct seq_file *m);
```

and avoid further `seq_<output>` calls if `true` is returned.

A `true` return from `seq_has_overflowed` means that the `seq_file` buffer will be discarded and the `seq_show` function will attempt to allocate a larger buffer and retry printing.

1.16.4 Making it all work

So far, we have a nice set of functions which can produce output within the `seq_file` system, but we have not yet turned them into a file that a user can see. Creating a file within the kernel requires, of course, the creation of a set of `file_operations` which implement the operations on that file. The `seq_file` interface provides a set of canned operations which do most of the work. The virtual file author still must implement the `open()` method, however, to hook everything up. The `open` function is often a single line, as in the example module:

```
static int ct_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &ct_seq_ops);
}
```

Here, the call to `seq_open()` takes the `seq_operations` structure we created before, and gets set up to iterate through the virtual file.

On a successful open, `seq_open()` stores the `struct seq_file` pointer in `file->private_data`. If you have an application where the same iterator can be used for more than one file, you can store an arbitrary pointer in the private field of the `seq_file` structure; that value can then be retrieved by the iterator functions.

There is also a wrapper function to `seq_open()` called `seq_open_private()`. It `kmallocs` a zero filled block of memory and stores a pointer to it in the private field of the `seq_file` structure, returning 0 on success. The block size is specified in a third parameter to the function, e.g.:

```
static int ct_open(struct inode *inode, struct file *file)
{
    return seq_open_private(file, &ct_seq_ops,
                           sizeof(struct mystruct));
}
```

There is also a variant function, `__seq_open_private()`, which is functionally identical except that, if successful, it returns the pointer to the allocated memory block, allowing further initialisation e.g.:

```
static int ct_open(struct inode *inode, struct file *file)
{
    struct mystruct *p =
        __seq_open_private(file, &ct_seq_ops, sizeof(*p));

    if (!p)
        return -ENOMEM;

    p->foo = bar; /* initialize my stuff */
    ...
    p->baz = true;

    return 0;
}
```

A corresponding close function, `seq_release_private()` is available which frees the memory allocated in the corresponding open.

The other operations of interest - `read()`, `llseek()`, and `release()` - are all implemented by the `seq_file` code itself. So a virtual file's `file_operations` structure will look like:

```
static const struct file_operations ct_file_ops = {
    .owner    = THIS_MODULE,
    .open     = ct_open,
    .read     = seq_read,
    .llseek   = seq_llseek,
```

(continues on next page)

(continued from previous page)

```
.release = seq_release
};
```

There is also a `seq_release_private()` which passes the contents of the `seq_file` private field to `kfree()` before releasing the structure.

The final step is the creation of the `/proc` file itself. In the example code, that is done in the initialization code in the usual way:

```
static int ct_init(void)
{
    struct proc_dir_entry *entry;

    proc_create("sequence", 0, NULL, &ct_file_ops);
    return 0;
}

module_init(ct_init);
```

And that is pretty much it.

1.16.5 seq_list

If your file will be iterating through a linked list, you may find these routines useful:

```
struct list_head *seq_list_start(struct list_head *head,
                                loff_t pos);
struct list_head *seq_list_start_head(struct list_head *head,
                                       loff_t pos);
struct list_head *seq_list_next(void *v, struct list_head *head,
                                loff_t *ppos);
```

These helpers will interpret `pos` as a position within the list and iterate accordingly. Your `start()` and `next()` functions need only invoke the `seq_list_*` helpers with a pointer to the appropriate `list_head` structure.

1.16.6 The extra-simple version

For extremely simple virtual files, there is an even easier interface. A module can define only the `show()` function, which should create all the output that the virtual file will contain. The file's `open()` method then calls:

```
int single_open(struct file *file,
               int (*show)(struct seq_file *m, void *p),
               void *data);
```

When output time comes, the `show()` function will be called once. The data value given to `single_open()` can be found in the private field of the `seq_file` structure. When using `single_open()`, the programmer should use `single_release()` instead of `seq_release()` in the `file_operations` structure to avoid a memory leak.

1.17 Shared Subtrees

1.17.1 1) Overview

Consider the following situation:

A process wants to clone its own namespace, but still wants to access the CD that got mounted recently. Shared subtree semantics provide the necessary mechanism to accomplish the above.

It provides the necessary building blocks for features like per-user-namespace and versioned filesystem.

1.17.2 2) Features

Shared subtree provides four different flavors of mounts; struct `vfsmount` to be precise

- a. shared mount
- b. slave mount
- c. private mount
- d. unbindable mount

2a) A shared mount can be replicated to as many mountpoints and all the replicas continue to be exactly same.

Here is an example:

Let' s say `/mnt` has a mount that is shared:

```
mount --make-shared /mnt
```

Note: `mount(8)` command now supports the `-make-shared` flag, so the sample 'smount' program is no longer needed and has been removed.

```
# mount --bind /mnt /tmp
```

The above command replicates the mount at `/mnt` to the mountpoint `/tmp` and the contents of both the mounts remain identical.

```
#ls /mnt
a b c

#ls /tmp
a b c
```

Now let' s say we mount a device at `/tmp/a`:

```
# mount /dev/sd0 /tmp/a

#ls /tmp/a
t1 t2 t3
```

(continues on next page)

(continued from previous page)

```
#ls /mnt/a
t1 t2 t3
```

Note that the mount has propagated to the mount at /mnt as well.

And the same is true even when /dev/sd0 is mounted on /mnt/a. The contents will be visible under /tmp/a too.

2b) A slave mount is like a shared mount except that mount and umount events only propagate towards it.

All slave mounts have a master mount which is a shared.

Here is an example:

Let' s say /mnt has a mount which is shared. # mount -make-shared /mnt

Let' s bind mount /mnt to /tmp # mount -bind /mnt /tmp

the new mount at /tmp becomes a shared mount and it is a replica of the mount at /mnt.

Now let' s make the mount at /tmp; a slave of /mnt # mount -make-slave /tmp

let' s mount /dev/sd0 on /mnt/a # mount /dev/sd0 /mnt/a

```
#ls /mnt/a t1 t2 t3
```

```
#ls /tmp/a t1 t2 t3
```

Note the mount event has propagated to the mount at /tmp

However let' s see what happens if we mount something on the mount at /tmp

```
# mount /dev/sd1 /tmp/b
```

```
#ls /tmp/b s1 s2 s3
```

```
#ls /mnt/b
```

Note how the mount event has not propagated to the mount at /mnt

2c) A private mount does not forward or receive propagation.

This is the mount we are familiar with. Its the default type.

2d) A unbindable mount is a unbindable private mount

let' s say we have a mount at /mnt and we make it unbindable:

```
# mount --make-unbindable /mnt
```

Let's try to bind mount this mount somewhere else::

```
# mount --bind /mnt /tmp
mount: wrong fs type, bad option, bad superblock on /mnt,
      or too many mounted file systems
```

Binding a unbindable mount is a invalid operation.

3) Setting mount states

The mount command (util-linux package) can be used to set mount states:

```
mount --make-shared mountpoint
mount --make-slave mountpoint
mount --make-private mountpoint
mount --make-unbindable mountpoint
```

1.17.3 4) Use cases

- A) A process wants to clone its own namespace, but still wants to access the CD that got mounted recently.

Solution:

The system administrator can make the mount at /cdrom shared:

```
mount --bind /cdrom /cdrom
mount --make-shared /cdrom
```

Now any process that clones off a new namespace will have a mount at /cdrom which is a replica of the same mount in the parent namespace.

So when a CD is inserted and mounted at /cdrom that mount gets propagated to the other mount at /cdrom in all the other clone namespaces.

- B) A process wants its mounts invisible to any other process, but still be able to see the other system mounts.

Solution:

To begin with, the administrator can mark the entire mount tree as shareable:

```
mount --make-rshared /
```

A new process can clone off a new namespace. And mark some part of its namespace as slave:

```
mount --make-rslave /myprivatetree
```

Hence forth any mounts within the /myprivatetree done by the process will not show up in any other namespace. However mounts done in the parent namespace under /myprivatetree still shows up in the process' s namespace.

Apart from the above semantics this feature provides the building blocks to solve the following problems:

- C) Per-user namespace

The above semantics allows a way to share mounts across namespaces. But namespaces are associated with processes. If namespaces are made first class objects with user API to associate/disassociate a namespace with userid, then each user could have his/her own namespace and tailor it to his/her requirements. This needs to be supported in PAM.

D) Versioned files

If the entire mount tree is visible at multiple locations, then an underlying versioning file system can return different versions of the file depending on the path used to access that file.

An example is:

```
mount --make-shared /
mount --rbind / /view/v1
mount --rbind / /view/v2
mount --rbind / /view/v3
mount --rbind / /view/v4
```

and if /usr has a versioning filesystem mounted, then that mount appears at /view/v1/usr, /view/v2/usr, /view/v3/usr and /view/v4/usr too

A user can request v3 version of the file /usr/fs/namespace.c by accessing /view/v3/usr/fs/namespace.c . The underlying versioning filesystem can then decipher that v3 version of the filesystem is being requested and return the corresponding inode.

1.17.4 5) Detailed semantics

The section below explains the detailed semantics of bind, rbind, move, mount, umount and clone-namespace operations.

Note: the word 'vfsmount' and the noun 'mount' have been used to mean the same thing, throughout this document.

5a) Mount states

A given mount can be in one of the following states

- 1) shared
- 2) slave
- 3) shared and slave
- 4) private
- 5) unbindable

A 'propagation event' is defined as event generated on a vfsmount that leads to mount or unmount actions in other vfsmounts.

A ‘peer group’ is defined as a group of vfstmounts that propagate events to each other.

(1) Shared mounts

A ‘shared mount’ is defined as a vfstmount that belongs to a ‘peer group’ .

For example:

```
mount --make-shared /mnt
mount --bind /mnt /tmp
```

The mount at /mnt and that at /tmp are both shared and belong to the same peer group. Anything mounted or unmounted under /mnt or /tmp reflect in all the other mounts of its peer group.

(2) Slave mounts

A ‘slave mount’ is defined as a vfstmount that receives propagation events and does not forward propagation events.

A slave mount as the name implies has a master mount from which mount/unmount events are received. Events do not propagate from the slave mount to the master. Only a shared mount can be made a slave by executing the following command:

```
mount --make-slave mount
```

A shared mount that is made as a slave is no more shared unless modified to become shared.

(3) Shared and Slave

A vfstmount can be both shared as well as slave. This state indicates that the mount is a slave of some vfstmount, and has its own peer group too. This vfstmount receives propagation events from its master vfstmount, and also forwards propagation events to its ‘peer group’ and to its slave vfstmounts.

Strictly speaking, the vfstmount is shared having its own peer group, and this peer-group is a slave of some other peer group.

Only a slave vfstmount can be made as ‘shared and slave’ by either executing the following command:

```
mount --make-shared mount
```

or by moving the slave vfstmount under a shared vfstmount.

(4) Private mount

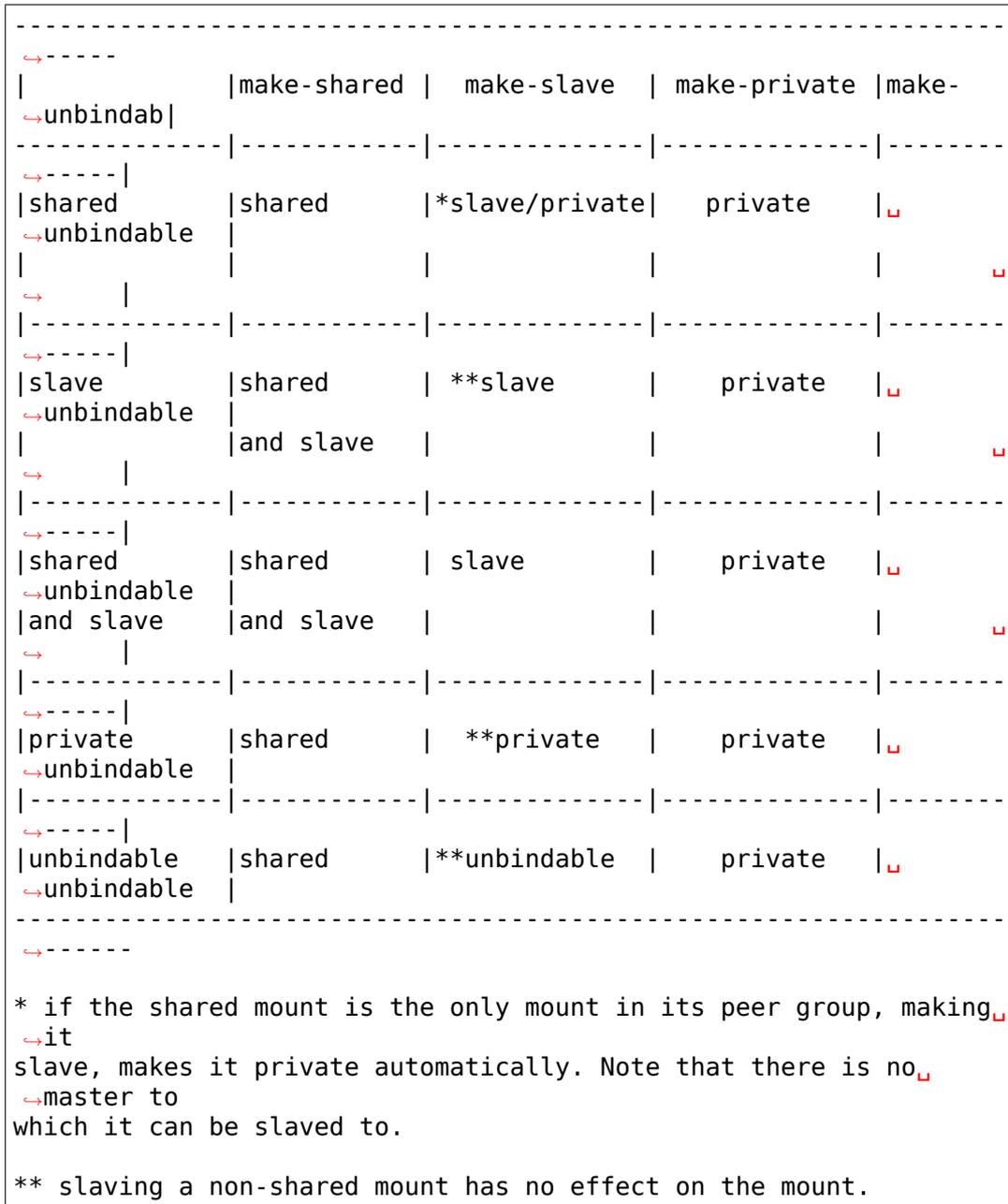
A ‘private mount’ is defined as vfstmount that does not receive or forward any propagation events.

(5) Unbindable mount

A ‘unbindable mount’ is defined as vfstype that does not receive or forward any propagation events and cannot be bind mounted.

State diagram:

The state diagram below explains the state transition of a mount, in response to various commands:



Apart from the commands listed below, the ‘move’ operation also changes the state of a mount depending on type of the destination mount. Its explained in section 5d.

5b) Bind semantics

Consider the following command:

```
mount --bind A/a B/b
```

where 'A' is the source mount, 'a' is the dentry in the mount 'A', 'B' is the destination mount and 'b' is the dentry in the destination mount.

The outcome depends on the type of mount of 'A' and 'B'. The table below contains quick reference:

BIND MOUNT OPERATION			
source(A) ->	shared	private	slave
unbindable			
dest(B)			
v			
shared	shared	shared	shared &
slave	invalid		
non-shared	shared	private	slave
invalid			

Details:

1. **'A' is a shared mount and 'B' is a shared mount. A new mount 'C'** which is clone of 'A', is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. Also new mount 'C1', 'C2', 'C3' ...are created and mounted at the dentry 'b' on all mounts where 'B' propagates to. A new propagation tree containing 'C1', ..., 'Cn' is created. This propagation tree is identical to the propagation tree of 'B'. And finally the peer-group of 'C' is merged with the peer group of 'A'.
2. **'A' is a private mount and 'B' is a shared mount. A new mount 'C'** which is clone of 'A', is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. Also new mount 'C1', 'C2', 'C3' ...are created and mounted at the dentry 'b' on all mounts where 'B' propagates to. A new propagation tree is set containing all new mounts 'C', 'C1', ..., 'Cn' with exactly the same configuration as the propagation tree for 'B'.
3. **'A' is a slave mount of mount 'Z' and 'B' is a shared mount. A new mount 'C'** which is clone of 'A', is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. Also new mounts 'C1', 'C2', 'C3' ...are created and mounted at the dentry 'b' on all mounts where 'B' propagates to. A new propagation tree containing the new mounts 'C', 'C1', ..., 'Cn'

is created. This propagation tree is identical to the propagation tree for 'B'. And finally the mount 'C' and its peer group is made the slave of mount 'Z'. In other words, mount 'C' is in the state 'slave and shared'.

4. **'A' is a unbindable mount and 'B' is a shared mount. This is a invalid operation.**
5. **'A' is a private mount and 'B' is a non-shared(private or slave or unbindable) mount.** A new mount 'C' which is clone of 'A', is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'.
6. **'A' is a shared mount and 'B' is a non-shared mount. A new mount 'C'** which is a clone of 'A' is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. 'C' is made a member of the peer-group of 'A'.
7. **'A' is a slave mount of mount 'Z' and 'B' is a non-shared mount. A** new mount 'C' which is a clone of 'A' is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. Also 'C' is set as a slave mount of 'Z'. In other words 'A' and 'C' are both slave mounts of 'Z'. All mount/unmount events on 'Z' propagates to 'A' and 'C'. But mount/unmount on 'A' do not propagate anywhere else. Similarly mount/unmount on 'C' do not propagate anywhere else.
8. **'A' is a unbindable mount and 'B' is a non-shared mount. This is a invalid operation.** A unbindable mount cannot be bind mounted.

5c) Rbind semantics

rbind is same as bind. Bind replicates the specified mount. Rbind replicates all the mounts in the tree belonging to the specified mount. Rbind mount is bind mount applied to all the mounts in the tree.

If the source tree that is rbind has some unbindable mounts, then the subtree under the unbindable mount is pruned in the new location.

eg:

let's say we have the following mount tree:



Let's say all the mount except the mount C in the tree are of a type other than unbindable.

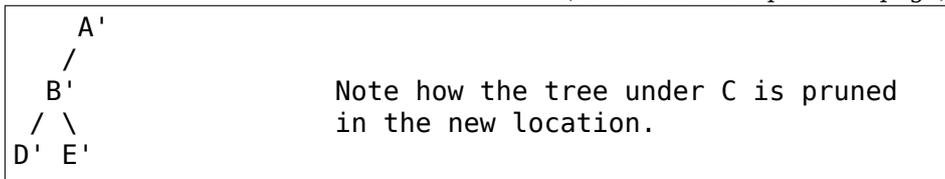
If this tree is rbound to say Z

We will have the following tree at the new location:



(continues on next page)

(continued from previous page)



5d) Move semantics

Consider the following command

```
mount -move A B/b
```

where 'A' is the source mount, 'B' is the destination mount and 'b' is the dentry in the destination mount.

The outcome depends on the type of the mount of 'A' and 'B'. The table below is a quick reference:

MOVE MOUNT OPERATION			

→			
→			
	*****	*****	*****
	source(A) ->	shared	private
→	slave	unbindable	
	dest(B)		
→			
→			
	v		
→			
	*****	*****	*****
	shared	shared	shared
→	shared and slave	invalid	
→			
	non-shared	shared	private
→	slave	unbindable	
	*****	*****	*****

Note: moving a mount residing under a shared mount is invalid.

Details follow:

1. 'A' is a shared mount and 'B' is a shared mount. The mount 'A' is mounted on mount 'B' at dentry 'b'. Also new mounts 'A1', 'A2' ... 'An' are created and mounted at dentry 'b' on all mounts that receive propagation from mount 'B'. A new propagation tree is created in the exact same configuration as that of 'B'. This new propagation tree contains all the new mounts 'A1', 'A2' ... 'An'. And this new propagation tree is appended to the already existing propagation tree of 'A'.
2. 'A' is a private mount and 'B' is a shared mount. The mount 'A' is

mounted on mount 'B' at dentry 'b' . Also new mount 'A1' , 'A2' ... 'An' are created and mounted at dentry 'b' on all mounts that receive propagation from mount 'B' . The mount 'A' becomes a shared mount and a propagation tree is created which is identical to that of 'B' . This new propagation tree contains all the new mounts 'A1' , 'A2' ... 'An' .

3. **'A' is a slave mount of mount 'Z' and 'B' is a shared mount. The** mount 'A' is mounted on mount 'B' at dentry 'b' . Also new mounts 'A1' , 'A2' ... 'An' are created and mounted at dentry 'b' on all mounts that receive propagation from mount 'B' . A new propagation tree is created in the exact same configuration as that of 'B' . This new propagation tree contains all the new mounts 'A1' , 'A2' ... 'An' . And this new propagation tree is appended to the already existing propagation tree of 'A' . Mount 'A' continues to be the slave mount of 'Z' but it also becomes 'shared' .
4. **'A' is a unbindable mount and 'B' is a shared mount. The operation** is invalid. Because mounting anything on the shared mount 'B' can create new mounts that get mounted on the mounts that receive propagation from 'B' . And since the mount 'A' is unbindable, cloning it to mount at other mountpoints is not possible.
5. **'A' is a private mount and 'B' is a non-shared(private or slave or unbindable) mount.** The mount 'A' is mounted on mount 'B' at dentry 'b' .
6. **'A' is a shared mount and 'B' is a non-shared mount. The mount 'A'** is mounted on mount 'B' at dentry 'b' . Mount 'A' continues to be a shared mount.
7. **'A' is a slave mount of mount 'Z' and 'B' is a non-shared mount.** The mount 'A' is mounted on mount 'B' at dentry 'b' . Mount 'A' continues to be a slave mount of mount 'Z' .
8. **'A' is a unbindable mount and 'B' is a non-shared mount. The mount** 'A' is mounted on mount 'B' at dentry 'b' . Mount 'A' continues to be a unbindable mount.

5e) Mount semantics

Consider the following command:

```
mount device B/b
```

'B' is the destination mount and 'b' is the dentry in the destination mount.

The above operation is the same as bind operation with the exception that the source mount is always a private mount.

5f) Unmount semantics

Consider the following command:

```
umount A
```

where 'A' is a mount mounted on mount 'B' at dentry 'b' .

If mount 'B' is shared, then all most-recently-mounted mounts at dentry 'b' on mounts that receive propagation from mount 'B' and does not have sub-mounts within them are unmounted.

Example: Let's say 'B1' , 'B2' , 'B3' are shared mounts that propagate to each other.

let's say 'A1' , 'A2' , 'A3' are first mounted at dentry 'b' on mount 'B1' , 'B2' and 'B3' respectively.

let's say 'C1' , 'C2' , 'C3' are next mounted at the same dentry 'b' on mount 'B1' , 'B2' and 'B3' respectively.

if 'C1' is unmounted, all the mounts that are most-recently-mounted on 'B1' and on the mounts that 'B1' propagates-to are unmounted.

'B1' propagates to 'B2' and 'B3' . And the most recently mounted mount on 'B2' at dentry 'b' is 'C2' , and that of mount 'B3' is 'C3' .

So all 'C1' , 'C2' and 'C3' should be unmounted.

If any of 'C2' or 'C3' has some child mounts, then that mount is not unmounted, but all other mounts are unmounted. However if 'C1' is told to be unmounted and 'C1' has some sub-mounts, the unmount operation is failed entirely.

5g) Clone Namespace

A cloned namespace contains all the mounts as that of the parent namespace.

Let's say 'A' and 'B' are the corresponding mounts in the parent and the child namespace.

If 'A' is shared, then 'B' is also shared and 'A' and 'B' propagate to each other.

If 'A' is a slave mount of 'Z' , then 'B' is also the slave mount of 'Z' .

If 'A' is a private mount, then 'B' is a private mount too.

If 'A' is unbindable mount, then 'B' is a unbindable mount too.

6) Quiz

A. What is the result of the following command sequence?

```
mount --bind /mnt /mnt
mount --make-shared /mnt
mount --bind /mnt /tmp
mount --move /tmp /mnt/1
```

what should be the contents of /mnt /mnt/1 /mnt/1/1 should be? Should they all be identical? or should /mnt and /mnt/1 be identical only?

B. What is the result of the following command sequence?

```
mount --make-rshared /
mkdir -p /v/1
mount --rbind / /v/1
```

what should be the content of /v/1/v/1 be?

C. What is the result of the following command sequence?

```
mount --bind /mnt /mnt
mount --make-shared /mnt
mkdir -p /mnt/1/2/3 /mnt/1/test
mount --bind /mnt/1 /tmp
mount --make-slave /mnt
mount --make-shared /mnt
mount --bind /mnt/1/2 /tmp1
mount --make-slave /mnt
```

At this point we have the first mount at /tmp and its root dentry is 1. Let's call this mount 'A' And then we have a second mount at /tmp1 with root dentry 2. Let's call this mount 'B' Next we have a third mount at /mnt with root dentry mnt. Let's call this mount 'C'

'B' is the slave of 'A' and 'C' is a slave of 'B' A -> B -> C

at this point if we execute the following command

```
mount --bind /bin /tmp/test
```

The mount is attempted on 'A'

will the mount propagate to 'B' and 'C' ?

what would be the contents of /mnt/1/test be?

7) FAQ

Q1. Why is bind mount needed? How is it different from symbolic links?

symbolic links can get stale if the destination mount gets unmounted or moved. Bind mounts continue to exist even if the other mount is unmounted or moved.

Q2. Why can't the shared subtree be implemented using exportfs?

exportfs is a heavyweight way of accomplishing part of what shared subtree can do. I cannot imagine a way to implement the semantics of slave mount using exportfs?

Q3 Why is unbindable mount needed?

Let's say we want to replicate the mount tree at multiple locations within the same subtree.

if one rbind mounts a tree within the same subtree 'n' times the number of mounts created is an exponential function of 'n'. Having unbindable mount can help prune the unneeded bind mounts. Here is an example.

step 1: let' s say the root tree has just two directories with one vfstmount:

```
      root
     /   \
    tmp   usr
```

And we want to replicate the tree at multiple mountpoints under /root/tmp

step 2:

```
mount --make-shared /root
mkdir -p /tmp/m1
mount --rbind /root /tmp/m1
```

the new tree now looks like this:

```
      root
     /   \
    tmp   usr
   /
  m1
 /  \
tmp  usr
 /
m1
```

it has two vfstmounts

step 3:

```
mkdir -p /tmp/m2
mount --rbind /root /tmp/m2
```

the new tree now looks like this::

```
      root
     /   \
    tmp   usr
   /     \
  m1      m2
 /  \    /  \
tmp  usr tmp  usr
 /  \  /
m1  m2 m1
 /  \ /  \
tmp  usr tmp  usr
 /  \ /  \
m1  m2 m1  m2
 /  \
tmp  usr
 /  \
m1  m2
```

(continues on next page)

(continued from previous page)

```
it has 6 vfmounts
```

step 4:

```
:: mkdir -p /tmp/m3 mount -rbind /root /tmp/m3
```

I won't draw the tree..but it has 24 vfmounts

at step i the number of vfmounts is $V[i] = i \cdot V[i-1]$. This is an exponential function. And this tree has way more mounts than what we really needed in the first place.

One could use a series of `umount` at each step to prune out the unneeded mounts. But there is a better solution. Unclonable mounts come in handy here.

step 1: let's say the root tree has just two directories with one vfmount:

```

      root
     /  \
    tmp  usr

```

How do we set up the same tree at multiple locations under `/root/tmp`

step 2:

```
mount --bind /root/tmp /root/tmp
mount --make-rshared /root
mount --make-unbindable /root/tmp
mkdir -p /tmp/m1
mount --rbind /root /tmp/m1
```

the new tree now looks like this:

```

      root
     /  \
    tmp  usr
   /
  m1
 /  \
tmp  usr

```

step 3:

```
mkdir -p /tmp/m2
mount --rbind /root /tmp/m2
```

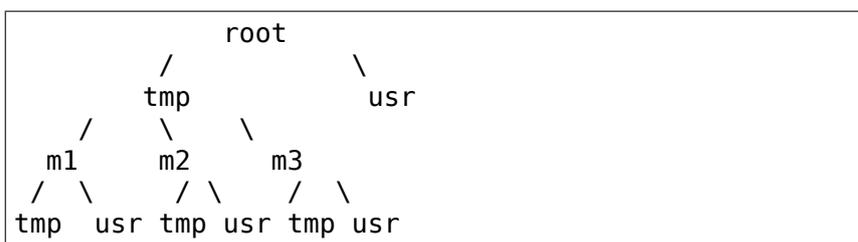
the new tree now looks like this:



step 4:

```
mkdir -p /tmp/m3
mount --rbind /root /tmp/m3
```

the new tree now looks like this:



8) Implementation

8A) Datastructure

4 new fields are introduced to struct vfstmount:

- ->mnt_share
- ->mnt_slave_list
- ->mnt_slave
- ->mnt_master

->**mnt_share** links together all the mount to/from which this vfstmount send/receives propagation events.

->**mnt_slave_list** links all the mounts to which this vfstmount propagates to.

->**mnt_slave** links together all the slaves that its master vfstmount propagates to.

->**mnt_master** points to the master vfstmount from which this vfstmount receives propagation.

->**mnt_flags** takes two more flags to indicate the propagation status of the vfstmount. MNT_SHARE indicates that the vfstmount is a shared vfstmount. MNT_UNCLONABLE indicates that the vfstmount cannot be replicated.

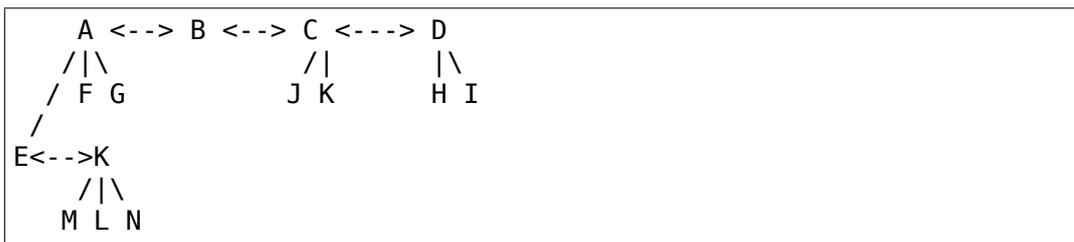
All the shared vfstmounts in a peer group form a cyclic list through ->mnt_share.

All vfstmounts with the same ->mnt_master form on a cyclic list anchored in ->mnt_master->mnt_slave_list and going through ->mnt_slave.

->mnt_master can point to arbitrary (and possibly different) members of master peer group. To find all immediate slaves of a peer group you need to go through _all_ ->mnt_slave_list of its members. Conceptually it's just a single set - distribution among the individual lists does not affect propagation or the way propagation tree is modified by operations.

All vfstmounts in a peer group have the same ->mnt_master. If it is non-NULL, they form a contiguous (ordered) segment of slave list.

An example propagation tree looks as shown in the figure below. [NOTE: Though it looks like a forest, if we consider all the shared mounts as a conceptual entity called 'pnode', it becomes a tree]:



In the above figure A,B,C and D all are shared and propagate to each other. 'A' has got 3 slave mounts 'E' 'F' and 'G' 'C' has got 2 slave mounts 'J' and 'K' and 'D' has got two slave mounts 'H' and 'I'. 'E' is also shared with 'K' and they propagate to each other. And 'K' has 3 slaves 'M', 'L' and 'N'

A's ->mnt_share links with the ->mnt_share of 'B' 'C' and 'D'

A's ->mnt_slave_list links with ->mnt_slave of 'E', 'K', 'F' and 'G'

E's ->mnt_share links with ->mnt_share of K

'E', 'K', 'F', 'G' have their ->mnt_master point to struct vfstmount of 'A'

'M', 'L', 'N' have their ->mnt_master point to struct vfstmount of 'K'

K's ->mnt_slave_list links with ->mnt_slave of 'M', 'L' and 'N'

C's ->mnt_slave_list links with ->mnt_slave of 'J' and 'K'

J and K's ->mnt_master points to struct vfstmount of C

and finally D's ->mnt_slave_list links with ->mnt_slave of 'H' and 'I'

'H' and 'I' have their ->mnt_master pointing to struct vfstmount of 'D'

.

NOTE: The propagation tree is orthogonal to the mount tree.

8B Locking:

->mnt_share, ->mnt_slave, ->mnt_slave_list, ->mnt_master are protected by namespace_sem (exclusive for modifications, shared for reading).

Normally we have ->mnt_flags modifications serialized by vfstmount_lock. There are two exceptions: do_add_mount() and

clone_mnt(). The former modifies a vfstmount that has not been visible in any shared data structures yet. The latter holds namespace_sem and the only references to vfstmount are in lists that can't be traversed without namespace_sem.

8C Algorithm:

The crux of the implementation resides in rbind/move operation.

The overall algorithm breaks the operation into 3 phases: (look at attach_recursive_mnt() and propagate_mnt())

1. prepare phase.
2. commit phases.
3. abort phases.

Prepare phase:

for each mount in the source tree:

- a) **Create the necessary number of mount trees to** be attached to each of the mounts that receive propagation from the destination mount.
- b) Do not attach any of the trees to its destination. However note down its ->mnt_parent and ->mnt_mountpoint
- c) Link all the new mounts to form a propagation tree that is identical to the propagation tree of the destination mount.

If this phase is successful, there should be 'n' new propagation trees; where 'n' is the number of mounts in the source tree. Go to the commit phase

Also there should be 'm' new mount trees, where 'm' is the number of mounts to which the destination mount propagates to.

if any memory allocations fail, go to the abort phase.

Commit phase attach each of the mount trees to their corresponding destination mounts.

Abort phase delete all the newly created trees.

Note: all the propagation related functionality resides in the file pn-ode.c

version 0.1 (created the initial document, Ram Pai linuxram@us.ibm.com)

version 0.2 (Incorporated comments from Al Viro)

1.18 Accessing PCI device resources through sysfs

sysfs, usually mounted at /sys, provides access to PCI resources on platforms that support it. For example, a given bus might look like this:

```

/sys/devices/pci0000:17
|-- 0000:17:00.0
|   |-- class
|   |-- config
|   |-- device
|   |-- enable
|   |-- irq
|   |-- local_cpus
|   |-- remove
|   |-- resource
|   |-- resource0
|   |-- resource1
|   |-- resource2
|   |-- revision
|   |-- rom
|   |-- subsystem_device
|   |-- subsystem_vendor
|   |-- vendor
|-- ...

```

The topmost element describes the PCI domain and bus number. In this case, the domain number is 0000 and the bus number is 17 (both values are in hex). This bus contains a single function device in slot 0. The domain and bus numbers are reproduced for convenience. Under the device directory are several files, each with their own function.

file	function
class	PCI class (ascii, ro)
config	PCI config space (binary, rw)
device	PCI device (ascii, ro)
enable	Whether the device is enabled (ascii, rw)
irq	IRQ number (ascii, ro)
local_cpus	nearby CPU mask (cpumask, ro)
remove	remove device from kernel's list (ascii, wo)
resource	PCI resource host addresses (ascii, ro)
resource0..N	PCI resource N, if present (binary, mmap, rw ¹)
resource0_wc..N_wc	PCI WC map resource N, if prefetchable (binary, mmap)
revision	PCI revision (ascii, ro)
rom	PCI ROM resource, if present (binary, ro)
subsystem_device	PCI subsystem device (ascii, ro)
subsystem_vendor	PCI subsystem vendor (ascii, ro)
vendor	PCI vendor (ascii, ro)

¹ rw for RESOURCE_IO (I/O port) regions only

```
ro - read only file
rw - file is readable and writable
wo - write only file
mmap - file is mmapable
ascii - file contains ascii text
binary - file contains binary data
cpumask - file contains a cpumask type
```

The read only files are informational, writes to them will be ignored, with the exception of the 'rom' file. Writable files can be used to perform actions on the device (e.g. changing config space, detaching a device). mmapable files are available via an mmap of the file at offset 0 and can be used to do actual device programming from userspace. Note that some platforms don't support mmapming of certain resources, so be sure to check the return value from any attempted mmap. The most notable of these are I/O port resources, which also provide read/write access.

The 'enable' file provides a counter that indicates how many times the device has been enabled. If the 'enable' file currently returns '4', and a '1' is echoed into it, it will then return '5'. Echoing a '0' into it will decrease the count. Even when it returns to 0, though, some of the initialisation may not be reversed.

The 'rom' file is special in that it provides read-only access to the device's ROM file, if available. It's disabled by default, however, so applications should write the string "1" to the file to enable it before attempting a read call, and disable it following the access by writing "0" to the file. Note that the device must be enabled for a rom read to return data successfully. In the event a driver is not bound to the device, it can be enabled using the 'enable' file, documented above.

The 'remove' file is used to remove the PCI device, by writing a non-zero integer to the file. This does not involve any kind of hot-plug functionality, e.g. powering off the device. The device is removed from the kernel's list of PCI devices, the sysfs directory for it is removed, and the device will be removed from any drivers attached to it. Removal of PCI root buses is disallowed.

1.18.1 Accessing legacy resources through sysfs

Legacy I/O port and ISA memory resources are also provided in sysfs if the underlying platform supports them. They're located in the PCI class hierarchy, e.g.:

```
/sys/class/pci_bus/0000:17/
|-- bridge -> ../../../../devices/pci0000:17
|-- cpuaffinity
|-- legacy_io
`-- legacy_mem
```

The legacy_io file is a read/write file that can be used by applications to do legacy port I/O. The application should open the file, seek to the desired port (e.g. 0x3e8) and do a read or a write of 1, 2 or 4 bytes. The legacy_mem file should be mmapmed with an offset corresponding to the memory offset desired, e.g. 0xa0000 for the VGA frame buffer. The application can then simply dereference the returned pointer (after checking for errors of course) to access legacy memory space.

1.18.2 Supporting PCI access on new platforms

In order to support PCI resource mapping as described above, Linux platform code should ideally define `ARCH_GENERIC_PCI_MMAP_RESOURCE` and use the generic implementation of that functionality. To support the historical interface of `mmap()` through files in `/proc/bus/pci`, platforms may also set `HAVE_PCI_MMAP`.

Alternatively, platforms which set `HAVE_PCI_MMAP` may provide their own implementation of `pci_mmap_page_range()` instead of defining `ARCH_GENERIC_PCI_MMAP_RESOURCE`.

Platforms which support write-combining maps of PCI resources must define `arch_can_pci_mmap_wc()` which shall evaluate to non-zero at runtime when write-combining is permitted. Platforms which support maps of I/O resources define `arch_can_pci_mmap_io()` similarly.

Legacy resources are protected by the `HAVE_PCI_LEGACY` define. Platforms wishing to support legacy functionality should define it and provide `pci_legacy_read`, `pci_legacy_write` and `pci_mmap_legacy_page_range` functions.

1.19 Sysfs tagging

(Taken almost verbatim from Eric Biederman's netns tagging patch commit msg)

The problem. Network devices show up in `sysfs` and with the network namespace active multiple devices with the same name can show up in the same directory, ouch!

To avoid that problem and allow existing applications in network namespaces to see the same interface that is currently presented in `sysfs`, `sysfs` now has tagging directory support.

By using the network namespace pointers as tags to separate out the the `sysfs` directory entries we ensure that we don't have conflicts in the directories and applications only see a limited set of the network devices.

Each `sysfs` directory entry may be tagged with a namespace via the `void *ns` member of its `kernfs_node`. If a directory entry is tagged, then `kernfs_node->flags` will have a flag between `KOBJ_NS_TYPE_NONE` and `KOBJ_NS_TYPES`, and `ns` will point to the namespace to which it belongs.

Each `sysfs` superblock's `kernfs_super_info` contains an array `void *ns[KOBJ_NS_TYPES]`. When a task in a tagging namespace `kobj_nstype` first mounts `sysfs`, a new superblock is created. It will be differentiated from other `sysfs` mounts by having its `s_fs_info->ns[kobj_nstype]` set to the new namespace. Note that through `bind` mounting and mounts propagation, a task can easily view the contents of other namespaces' `sysfs` mounts. Therefore, when a namespace exits, it will call `kobj_ns_exit()` to invalidate any `kernfs_node->ns` pointers pointing to it.

Users of this interface:

- define a type in the `kobj_ns_type` enumeration.
- call `kobj_ns_type_register()` with its `kobj_ns_type_operations` which has

- `current_ns()` which returns current' s namespace
- `netlink_ns()` which returns a socket' s namespace
- `initial_ns()` which returns the initial namesapce
- call `kobj_ns_exit()` when an individual tag is no longer valid

1.20 Automount Support

Support is available for filesystems that wish to do automounting support (such as kAFS which can be found in `fs/afs/` and NFS in `fs/nfs/`). This facility includes allowing in-kernel mounts to be performed and mountpoint degradation to be requested. The latter can also be requested by userspace.

1.20.1 In-Kernel Automounting

See section “Mount Traps” of `Documentation/filesystems/autofs.rst`

Then from userspace, you can just do something like:

```
[root@andromeda root]# mount -t afs \#root.afs. /afs
[root@andromeda root]# ls /afs
asd cambridge cambridge.redhat.com grand.central.org
[root@andromeda root]# ls /afs/cambridge
afsdoc
[root@andromeda root]# ls /afs/cambridge/afsdoc/
ChangeLog html LICENSE pdf RELNOTES-1.2.2
```

And then if you look in the mountpoint catalogue, you' ll see something like:

```
[root@andromeda root]# cat /proc/mounts
...
#root.afs. /afs afs rw 0 0
#root.cell. /afs/cambridge.redhat.com afs rw 0 0
#afsdoc. /afs/cambridge.redhat.com/afsdoc afs rw 0 0
```

1.20.2 Automatic Mountpoint Expiry

Automatic expiration of mountpoints is easy, provided you' ve mounted the mountpoint to be expired in the automounting procedure outlined separately.

To do expiration, you need to follow these steps:

- (1) Create at least one list off which the `vfsmounts` to be expired can be hung.
- (2) When a new mountpoint is created in the `->d_automount` method, add the `mnt` to the list using `mnt_set_expiry()`:

```
mnt_set_expiry(newmnt, &afs_vfsmounts);
```

- (3) When you want mountpoints to be expired, call `mark_mounts_for_expiry()` with a pointer to this list. This will process the list, marking every `vfsmount` thereon for potential expiry on the next call.

If a vfmount was already flagged for expiry, and if its usage count is 1 (it's only referenced by its parent vfmount), then it will be deleted from the namespace and thrown away (effectively unmounted).

It may prove simplest to simply call this at regular intervals, using some sort of timed event to drive it.

The expiration flag is cleared by calls to `mntput`. This means that expiration will only happen on the second expiration request after the last time the mountpoint was accessed.

If a mountpoint is moved, it gets removed from the expiration list. If a bind mount is made on an expirable mount, the new vfmount will not be on the expiration list and will not expire.

If a namespace is copied, all mountpoints contained therein will be copied, and the copies of those that are on an expiration list will be added to the same expiration list.

1.20.3 Userspace Driven Expiry

As an alternative, it is possible for userspace to request expiry of any mountpoint (though some will be rejected - the current process's idea of the rootfs for example). It does this by passing the `MNT_EXPIRE` flag to `umount()`. This flag is considered incompatible with `MNT_FORCE` and `MNT_DETACH`.

If the mountpoint in question is referenced by something other than `umount()` or its parent mountpoint, an `EBUSY` error will be returned and the mountpoint will not be marked for expiration or unmounted.

If the mountpoint was not already marked for expiry at that time, an `EAGAIN` error will be given and it won't be unmounted.

Otherwise if it was already marked and it wasn't referenced, unmounting will take place as usual.

Again, the expiration flag is cleared every time anything other than `umount()` looks at a mountpoint.

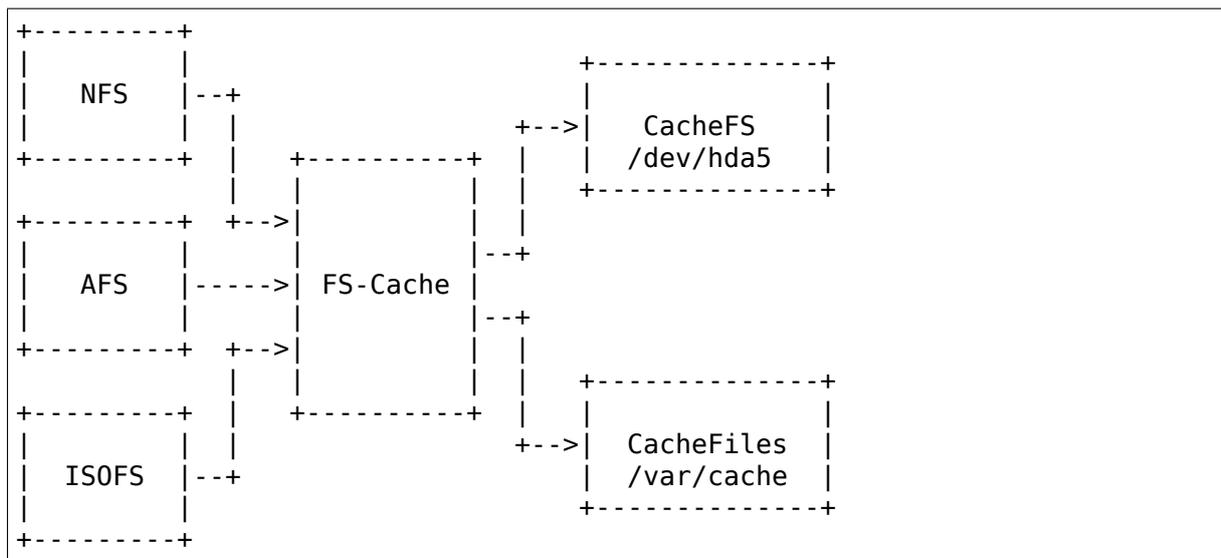
1.21 Filesystem Caching

1.21.1 General Filesystem Caching

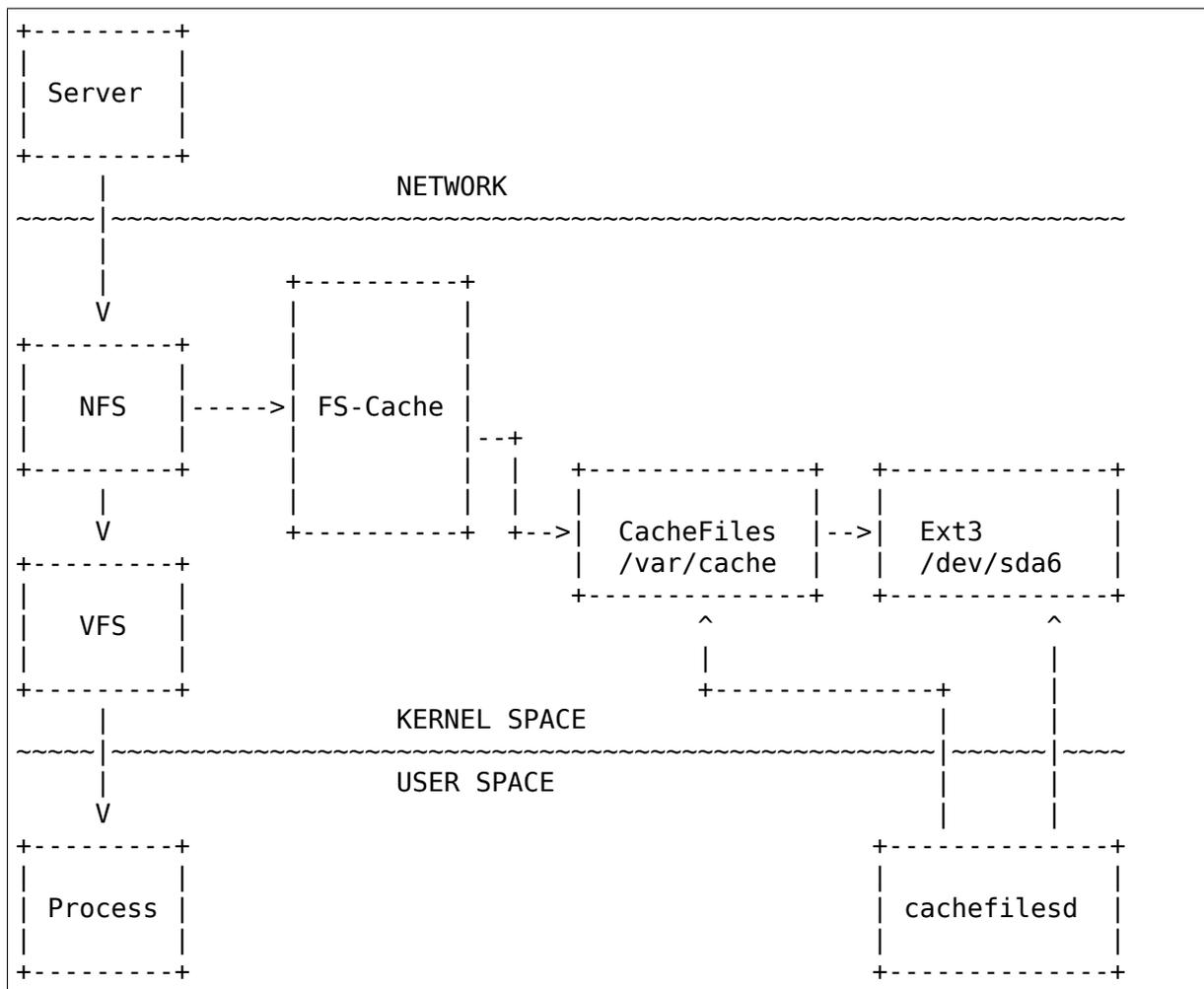
Overview

This facility is a general purpose cache for network filesystems, though it could be used for caching other things such as ISO9660 filesystems too.

FS-Cache mediates between cache backends (such as CacheFS) and network filesystems:



Or to look at it another way, FS-Cache is a module that provides a caching facility to a network filesystem such that the cache is transparent to the user:



FS-Cache does not follow the idea of completely loading every netfs file opened in its entirety into a cache before permitting it to be accessed and then serving the pages out of that cache rather than the netfs inode because:

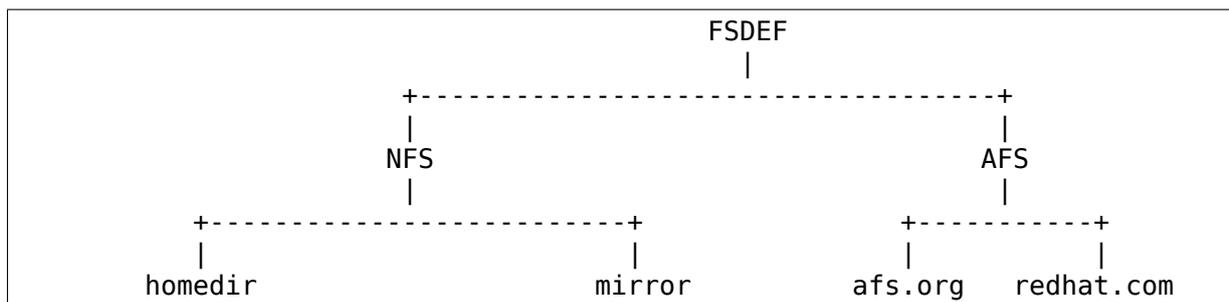
- (1) It must be practical to operate without a cache.
- (2) The size of any accessible file must not be limited to the size of the cache.
- (3) The combined size of all opened files (this includes mapped libraries) must not be limited to the size of the cache.
- (4) The user should not be forced to download an entire file just to do a one-off access of a small portion of it (such as might be done with the “file” program).

It instead serves the cache out in PAGE_SIZE chunks as and when requested by the netfs('s) using it.

FS-Cache provides the following facilities:

- (1) More than one cache can be used at once. Caches can be selected explicitly by use of tags.
- (2) Caches can be added / removed at any time.
- (3) The netfs is provided with an interface that allows either party to withdraw caching facilities from a file (required for (2)).
- (4) The interface to the netfs returns as few errors as possible, preferring rather to let the netfs remain oblivious.
- (5) Cookies are used to represent indices, files and other objects to the netfs. The simplest cookie is just a NULL pointer - indicating nothing cached there.
- (6) The netfs is allowed to propose - dynamically - any index hierarchy it desires, though it must be aware that the index search function is recursive, stack space is limited, and indices can only be children of indices.
- (7) Data I/O is done direct to and from the netfs' s pages. The netfs indicates that page A is at index B of the data-file represented by cookie C, and that it should be read or written. The cache backend may or may not start I/O on that page, but if it does, a netfs callback will be invoked to indicate completion. The I/O may be either synchronous or asynchronous.
- (8) Cookies can be “retired” upon release. At this point FS-Cache will mark them as obsolete and the index hierarchy rooted at that point will get recycled.
- (9) The netfs provides a “match” function for index searches. In addition to saying whether a match was made or not, this can also specify that an entry should be updated or deleted.
- (10) As much as possible is done asynchronously.

FS-Cache maintains a virtual indexing tree in which all indices, files, objects and pages are kept. Bits of this tree may actually reside in one or more caches:



(continues on next page)

Statistical Information

If FS-Cache is compiled with the following options enabled:

```
CONFIG_FSCACHE_STATS=y
CONFIG_FSCACHE_HISTOGRAM=y
```

then it will gather certain statistics and display them through a number of proc files.

/proc/fs/fscache/stats

This shows counts of a number of events that can happen in FS-Cache:

	CLASS	EVENT	MEANING
Cookies	idx=N		Number of index cookies allocated
	dat=N		Number of data storage cookies allocated
	spc=N		Number of special cookies allocated
Objects	alc=N		Number of objects allocated
	nal=N		Number of object allocation failures
	avl=N		Number of objects that reached the available state
	ded=N		Number of objects that reached the dead state
ChkAux	non=N		Number of objects that didn't have a coherency check
	ok=N		Number of objects that passed a coherency check
	upd=N		Number of objects that needed a coherency data update
	obs=N		Number of objects that were declared obsolete
Pages	mrk=N	unc=N	Number of pages marked as being cached Number of uncached pages
Acquire	n=N		Number of acquire cookie requests seen
	nul=N		Number of acq reqs given a NULL parent
	noc=N		Number of acq reqs rejected due to no cache available
	ok=N		Number of acq reqs succeeded
	nbf=N		Number of acq reqs rejected due to error
	oom=N		Number of acq reqs failed on ENOMEM
Lookups	n=N		Number of lookup calls made on cache backends
	neg=N		Number of negative lookups made
	pos=N		Number of positive lookups made
	crt=N		Number of objects created by lookup
	tmo=N		Number of lookups timed out and requeued
Updates	n=N		Number of update cookie requests seen
	nul=N		Number of upd reqs given a NULL parent
	run=N		Number of upd reqs granted CPU time
Relinqs	n=N		Number of relinquish cookie requests seen
	nul=N		Number of rlq reqs given a NULL parent
	wcr=N		Number of rlq reqs waited on completion of creation
AttrChg	n=N		Number of attribute changed requests seen
	ok=N		Number of attr changed requests queued
	nbf=N		Number of attr changed rejected -ENOBUFS
	oom=N		Number of attr changed failed -ENOMEM
	run=N		Number of attr changed ops given CPU time

Continued on next page

Table 1 - continued from previous page

	CLASS	EVENT	MEANING
Allocs	n=N		Number of allocation requests seen
	ok=N		Number of successful alloc reqs
	wt=N		Number of alloc reqs that waited on lookup completion
	nbf=N		Number of alloc reqs rejected -ENOBUFS
	int=N		Number of alloc reqs aborted -ERESTARTSYS
	ops=N		Number of alloc reqs submitted
	owt=N		Number of alloc reqs waited for CPU time
	abt=N		Number of alloc reqs aborted due to object death
Retrvls	n=N		Number of retrieval (read) requests seen
	ok=N		Number of successful retr reqs
	wt=N		Number of retr reqs that waited on lookup completion
	nod=N		Number of retr reqs returned -ENODATA
	nbf=N		Number of retr reqs rejected -ENOBUFS
	int=N		Number of retr reqs aborted -ERESTARTSYS
	oom=N		Number of retr reqs failed -ENOMEM
	ops=N		Number of retr reqs submitted
	owt=N		Number of retr reqs waited for CPU time
	abt=N		Number of retr reqs aborted due to object death
Stores	n=N		Number of storage (write) requests seen
	ok=N		Number of successful store reqs
	agn=N		Number of store reqs on a page already pending storage
	nbf=N		Number of store reqs rejected -ENOBUFS
	oom=N		Number of store reqs failed -ENOMEM
	ops=N		Number of store reqs submitted
	run=N		Number of store reqs granted CPU time
	pgs=N		Number of pages given store req processing time
	rxd=N		Number of store reqs deleted from tracking tree
	olm=N		Number of store reqs over store limit
VmScan	nos=N		Number of release reqs against pages with no pending store
	gon=N		Number of release reqs against pages stored by time lock granted
	bsy=N		Number of release reqs ignored due to in-progress store
	can=N		Number of page stores cancelled due to release req
Ops	pend=N		Number of times async ops added to pending queues
	run=N		Number of times async ops given CPU time
	enq=N		Number of times async ops queued for processing
	can=N		Number of async ops cancelled
	rej=N		Number of async ops rejected due to object lookup/create failure
	ini=N		Number of async ops initialised
	dfr=N		Number of async ops queued for deferred release
	rel=N		Number of async ops released (should equal ini=N when idle)
	gc=N		Number of deferred-release async ops garbage collected
CacheOp	alo=N		Number of in-progress alloc_object() cache ops
	luo=N		Number of in-progress lookup_object() cache ops
	luc=N		Number of in-progress lookup_complete() cache ops
	gro=N		Number of in-progress grab_object() cache ops
	upo=N		Number of in-progress update_object() cache ops
	dro=N		Number of in-progress drop_object() cache ops
	pto=N		Number of in-progress put_object() cache ops

Continued on next page

Table 1 - continued from previous page

	CLASS	EVENT	MEANING
	syn=N		Number of in-progress sync_cache() cache ops
	atc=N		Number of in-progress attr_changed() cache ops
	rap=N		Number of in-progress read_or_alloc_page() cache ops
	ras=N		Number of in-progress read_or_alloc_pages() cache ops
	alp=N		Number of in-progress allocate_page() cache ops
	als=N		Number of in-progress allocate_pages() cache ops
	wrp=N		Number of in-progress write_page() cache ops
	ucp=N		Number of in-progress uncache_page() cache ops
	dsp=N		Number of in-progress dissociate_pages() cache ops
CacheEv	nsp=N		Number of object lookups/creations rejected due to lack of space
	stl=N		Number of stale objects deleted
	rtr=N		Number of objects retired when relinquished
	cul=N		Number of objects culled

/proc/fs/fscache/histogram

```
cat /proc/fs/fscache/histogram
JIFS SECS OBJ INST OP RUNS OBJ RUNS RETRV DLY RETRIEVL
=====
```

This shows the breakdown of the number of times each amount of time between 0 jiffies and HZ-1 jiffies a variety of tasks took to run. The columns are as follows:

COLUMN	TIME MEASUREMENT
OBJ INST	Length of time to instantiate an object
OP RUNS	Length of time a call to process an operation took
OBJ RUNS	Length of time a call to process an object event took
RETRV DLY	Time between an requesting a read and lookup completing
RE-TRIEVLS	Time between beginning and end of a retrieval

Each row shows the number of events that took a particular range of times. Each step is 1 jiffy in size. The JIFS column indicates the particular jiffy range covered, and the SECS field the equivalent number of seconds.

COLUMN	DESCRIPTION
NETFS_COOKIE_DEF	Name of netfs cookie definition
TY	Cookie type (IX - index, DT - data, hex - special)
FL	Cookie flags
NETFS_DATA	Netfs private data stored in the cookie
OBJECT_KEY	Object key } 1 column, with separating comma
AUX_DATA	Object aux data } presence may be configured

The data shown may be filtered by attaching the a key to an appropriate keyring before viewing the file. Something like:

```
keyctl add user fscache:objlist <restrictions> @s
```

where <restrictions> are a selection of the following letters:

K	Show hexdump of object key (don' t show if not given)
A	Show hexdump of object aux data (don' t show if not given)

and the following paired letters:

C	Show objects that have a cookie
c	Show objects that don' t have a cookie
B	Show objects that are busy
b	Show objects that aren' t busy
W	Show objects that have pending writes
w	Show objects that don' t have pending writes
R	Show objects that have outstanding reads
r	Show objects that don' t have outstanding reads
S	Show objects that have work queued
s	Show objects that don' t have work queued

If neither side of a letter pair is given, then both are implied. For example:

```
keyctl add user fscache:objlist KB @s
```

shows objects that are busy, and lists their object keys, but does not dump their auxiliary data. It also implies “CcWwRrSs”, but as ‘B’ is given, ‘b’ is not implied.

By default all objects and all fields will be shown.

Debugging

If CONFIG_FSCACHE_DEBUG is enabled, the FS-Cache facility can have runtime debugging enabled by adjusting the value in:

```
/sys/module/fscache/parameters/debug
```

This is a bitmask of debugging streams to enable:

BIT	VALUE	STREAM	POINT
0	1	Cache management	Function entry trace
1	2		Function exit trace
2	4		General
3	8	Cookie management	Function entry trace
4	16		Function exit trace
5	32		General
6	64	Page handling	Function entry trace
7	128		Function exit trace
8	256		General
9	512	Operation management	Function entry trace
10	1024		Function exit trace
11	2048		General

The appropriate set of values should be OR' d together and the result written to the control file. For example:

```
echo $((1|8|64)) >/sys/module/fscache/parameters/debug
```

will turn on all function entry debugging.

1.21.2 In-Kernel Cache Object Representation and Management

By: David Howells <dhowells@redhat.com>

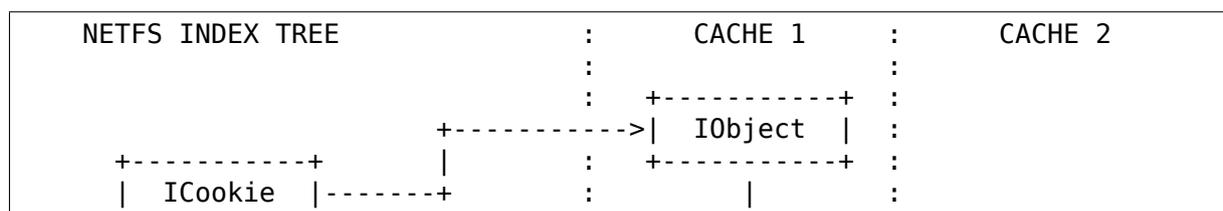
Representation

FS-Cache maintains an in-kernel representation of each object that a netfs is currently interested in. Such objects are represented by the `fscache_cookie` struct and are referred to as cookies.

FS-Cache also maintains a separate in-kernel representation of the objects that a cache backend is currently actively caching. Such objects are represented by the `fscache_object` struct. The cache backends allocate these upon request, and are expected to embed them in their own representations. These are referred to as objects.

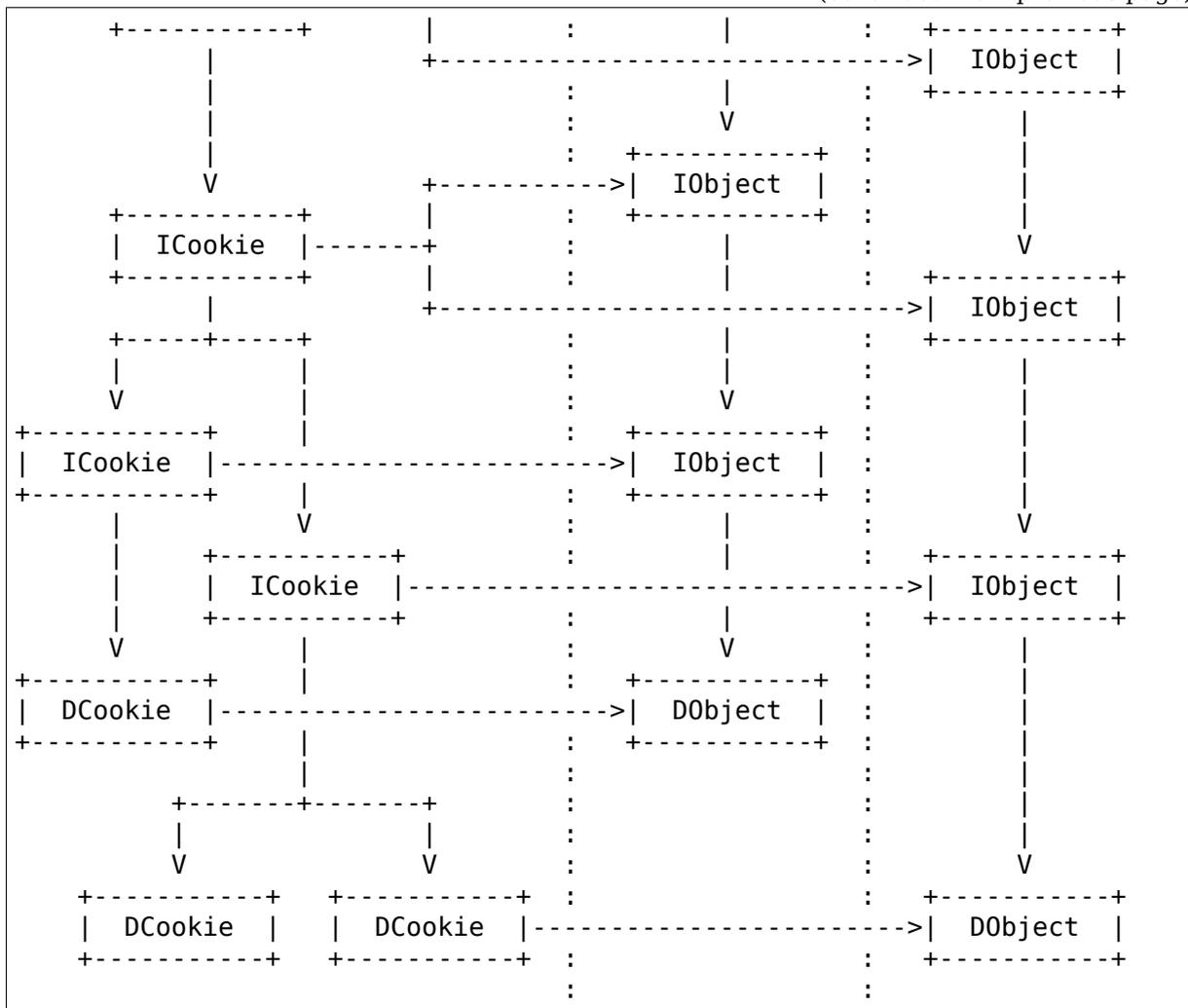
There is a 1:N relationship between cookies and objects. A cookie may be represented by multiple objects - an index may exist in more than one cache - or even by no objects (it may not be cached).

Furthermore, both cookies and objects are hierarchical. The two hierarchies correspond, but the cookies tree is a superset of the union of the object trees of multiple caches:



(continues on next page)

(continued from previous page)



In the above illustration, ICookie and IObject represent indices and DCookie and DObject represent data storage objects. Indices may have representation in multiple caches, but currently, non-index objects may not. Objects of any type may also be entirely unrepresented.

As far as the netfs API goes, the netfs is only actually permitted to see pointers to the cookies. The cookies themselves and any objects attached to those cookies are hidden from it.

Object Management State Machine

Within FS-Cache, each active object is managed by its own individual state machine. The state for an object is kept in the `fscache_object` struct, in `object->state`. A cookie may point to a set of objects that are in different states.

Each state has an action associated with it that is invoked when the machine wakes up in that state. There are four logical sets of states:

- (1) Preparation: states that wait for the parent objects to become ready. The representations are hierarchical, and it is expected that an object must be created or accessed with respect to its parent object.

- (2) Initialisation: states that perform lookups in the cache and validate what's found and that create on disk any missing metadata.
- (3) Normal running: states that allow netfs operations on objects to proceed and that update the state of objects.
- (4) Termination: states that detach objects from their netfs cookies, that delete objects from disk, that handle disk and system errors and that free up in-memory resources.

In most cases, transitioning between states is in response to signalled events. When a state has finished processing, it will usually set the mask of events in which it is interested (`object->event_mask`) and relinquish the worker thread. Then when an event is raised (by calling `fscache_raise_event()`), if the event is not masked, the object will be queued for processing (by calling `fscache_enqueue_object()`).

Provision of CPU Time

The work to be done by the various states was given CPU time by the threads of the slow work facility. This was used in preference to the workqueue facility because:

- (1) Threads may be completely occupied for very long periods of time by a particular work item. These state actions may be doing sequences of synchronous, journalled disk accesses (`lookup`, `mkdir`, `create`, `setxattr`, `getxattr`, `truncate`, `unlink`, `rmdir`, `rename`).
- (2) Threads may do little actual work, but may rather spend a lot of time sleeping on I/O. This means that single-threaded and 1-per-CPU-threaded workqueues don't necessarily have the right numbers of threads.

Locking Simplification

Because only one worker thread may be operating on any particular object's state machine at once, this simplifies the locking, particularly with respect to disconnecting the netfs's representation of a cache object (`fscache_cookie`) from the cache backend's representation (`fscache_object`) - which may be requested from either end.

The Set of States

The object state machine has a set of states that it can be in. There are preparation states in which the object sets itself up and waits for its parent object to transit to a state that allows access to its children:

- (1) State `FSCACHE_OBJECT_INIT`.

Initialise the object and wait for the parent object to become active. In the cache, it is expected that it will not be possible to look an object up from the parent object, until that parent object itself has been looked up.

There are initialisation states in which the object sets itself up and accesses disk for the object metadata:

(2) State `FSCACHE_OBJECT_LOOKING_UP`.

Look up the object on disk, using the parent as a starting point. FS-Cache expects the cache backend to probe the cache to see whether this object is represented there, and if it is, to see if it's valid (coherency management).

The cache should call `fscache_object_lookup_negative()` to indicate lookup failure for whatever reason, and should call `fscache_obtained_object()` to indicate success.

At the completion of lookup, FS-Cache will let the netfs go ahead with read operations, no matter whether the file is yet cached. If not yet cached, read operations will be immediately rejected with `ENODATA` until the first known page is uncached - as to that point there can be no data to be read out of the cache for that file that isn't currently also held in the pagecache.

(3) State `FSCACHE_OBJECT_CREATING`.

Create an object on disk, using the parent as a starting point. This happens if the lookup failed to find the object, or if the object's coherency data indicated what's on disk is out of date. In this state, FS-Cache expects the cache to create

The cache should call `fscache_obtained_object()` if creation completes successfully, `fscache_object_lookup_negative()` otherwise.

At the completion of creation, FS-Cache will start processing write operations the netfs has queued for an object. If creation failed, the write ops will be transparently discarded, and nothing recorded in the cache.

There are some normal running states in which the object spends its time servicing netfs requests:

(4) State `FSCACHE_OBJECT_AVAILABLE`.

A transient state in which pending operations are started, child objects are permitted to advance from `FSCACHE_OBJECT_INIT` state, and temporary lookup data is freed.

(5) State `FSCACHE_OBJECT_ACTIVE`.

The normal running state. In this state, requests the netfs makes will be passed on to the cache.

(6) State `FSCACHE_OBJECT_INVALIDATING`.

The object is undergoing invalidation. When the state comes here, it discards all pending read, write and attribute change operations as it is going to clear out the cache entirely and reinitialise it. It will then continue to the `FSCACHE_OBJECT_UPDATING` state.

(7) State `FSCACHE_OBJECT_UPDATING`.

The state machine comes here to update the object in the cache from the netfs's records. This involves updating the auxiliary data that is used to maintain coherency.

And there are terminal states in which an object cleans itself up, deallocates memory and potentially deletes stuff from disk:

(8) State `FSCACHE_OBJECT_LC_DYING`.

The object comes here if it is dying because of a lookup or creation error. This would be due to a disk error or system error of some sort. Temporary data is cleaned up, and the parent is released.

(9) State `FSCACHE_OBJECT_DYING`.

The object comes here if it is dying due to an error, because its parent cookie has been relinquished by the netfs or because the cache is being withdrawn.

Any child objects waiting on this one are given CPU time so that they too can destroy themselves. This object waits for all its children to go away before advancing to the next state.

(10) State `FSCACHE_OBJECT_ABORT_INIT`.

The object comes to this state if it was waiting on its parent in `FS-CACHE_OBJECT_INIT`, but its parent died. The object will destroy itself so that the parent may proceed from the `FSCACHE_OBJECT_DYING` state.

(11) State `FSCACHE_OBJECT_RELEASING`.

(12) State `FSCACHE_OBJECT_RECYCLING`.

The object comes to one of these two states when dying once it is rid of all its children, if it is dying because the netfs relinquished its cookie. In the first state, the cached data is expected to persist, and in the second it will be deleted.

(13) State `FSCACHE_OBJECT_WITHDRAWING`.

The object transits to this state if the cache decides it wants to withdraw the object from service, perhaps to make space, but also due to error or just because the whole cache is being withdrawn.

(14) State `FSCACHE_OBJECT_DEAD`.

The object transits to this state when the in-memory object record is ready to be deleted. The object processor shouldn't ever see an object in this state.

The Set of Events

There are a number of events that can be raised to an object state machine:

`FSCACHE_OBJECT_EV_UPDATE` The netfs requested that an object be updated. The state machine will ask the cache backend to update the object, and the cache backend will ask the netfs for details of the change through its cookie definition ops.

`FSCACHE_OBJECT_EV_CLEARED` This is signalled in two circumstances:

- (a) when an object's last child object is dropped and
- (b) when the last operation outstanding on an object is completed.

This is used to proceed from the dying state.

FSCACHE_OBJECT_EV_ERROR This is signalled when an I/O error occurs during the processing of some object.

FSCACHE_OBJECT_EV_RELEASE, FSCACHE_OBJECT_EV_RETIRE These are signalled when the netfs relinquishes a cookie it was using. The event selected depends on whether the netfs asks for the backing object to be retired (deleted) or retained.

FSCACHE_OBJECT_EV_WITHDRAW This is signalled when the cache backend wants to withdraw an object. This means that the object will have to be detached from the netfs' s cookie.

Because the withdrawing releasing/retiring events are all handled by the object state machine, it doesn' t matter if there' s a collision with both ends trying to sever the connection at the same time. The state machine can just pick which one it wants to honour, and that effects the other.

1.21.3 FS-Cache Cache backend API

The FS-Cache system provides an API by which actual caches can be supplied to FS-Cache for it to then serve out to network filesystems and other interested parties.

This API is declared in <linux/fscache-cache.h>.

Initialising and Registering a Cache

To start off, a cache definition must be initialised and registered for each cache the backend wants to make available. For instance, CacheFS does this in the `fill_super()` operation on mounting.

The cache definition (`struct fscache_cache`) should be initialised by calling:

```
void fscache_init_cache(struct fscache_cache *cache,
                       struct fscache_cache_ops *ops,
                       const char *idfmt,
                       ...);
```

Where:

- “cache” is a pointer to the cache definition;
- “ops” is a pointer to the table of operations that the backend supports on this cache; and
- “idfmt” is a format and printf-style arguments for constructing a label for the cache.

The cache should then be registered with FS-Cache by passing a pointer to the previously initialised cache definition to:

```
int fscache_add_cache(struct fscache_cache *cache,
                     struct fscache_object *fsdef,
                     const char *tagname);
```

Two extra arguments should also be supplied:

- “fsdef” which should point to the object representation for the FS-Cache master index in this cache. Netfs primary index entries will be created here. FS-Cache keeps the caller’s reference to the index object if successful and will release it upon withdrawal of the cache.
- “tagname” which, if given, should be a text string naming this cache. If this is NULL, the identifier will be used instead. For CacheFS, the identifier is set to name the underlying block device and the tag can be supplied by mount.

This function may return -ENOMEM if it ran out of memory or -EEXIST if the tag is already in use. 0 will be returned on success.

Unregistering a Cache

A cache can be withdrawn from the system by calling this function with a pointer to the cache definition:

```
void fscache_withdraw_cache(struct fscache_cache *cache);
```

In CacheFS’s case, this is called by put_super().

Security

The cache methods are executed one of two contexts:

- (1) that of the userspace process that issued the netfs operation that caused the cache method to be invoked, or
- (2) that of one of the processes in the FS-Cache thread pool.

In either case, this may not be an appropriate context in which to access the cache.

The calling process’s fsuid, fsgid and SELinux security identities may need to be masqueraded for the duration of the cache driver’s access to the cache. This is left to the cache to handle; FS-Cache makes no effort in this regard.

Control and Statistics Presentation

The cache may present data to the outside world through FS-Cache’s interfaces in sysfs and procfs - the former for control and the latter for statistics.

A sysfs directory called /sys/fs/fscache/<cachetag>/ is created if CONFIG_SYSFS is enabled. This is accessible through the kobject struct fscache_cache::kobj and is for use by the cache as it sees fit.

Relevant Data Structures

- Index/Data file FS-Cache representation cookie:

```
struct fscache_cookie {
    struct fscache_object_def    *def;
    struct fscache_netfs        *netfs;
    void                        *netfs_data;
    ...
};
```

The fields that might be of use to the backend describe the object definition, the netfs definition and the netfs' s data for this cookie. The object definition contain functions supplied by the netfs for loading and matching index entries; these are required to provide some of the cache operations.

- In-cache object representation:

```
struct fscache_object {
    int                        debug_id;
    enum {
        FSCACHE_OBJECT_RECYCLING,
        ...
    }
    spinlock_t                lock;
    struct fscache_cache      *cache;
    struct fscache_cookie     *cookie;
    ...
};
```

Structures of this type should be allocated by the cache backend and passed to FS-Cache when requested by the appropriate cache operation. In the case of CacheFS, they' re embedded in CacheFS' s internal object structures.

The `debug_id` is a simple integer that can be used in debugging messages that refer to a particular object. In such a case it should be printed using "OBJ%x" to be consistent with FS-Cache.

Each object contains a pointer to the cookie that represents the object it is backing. An object should be retired when `put_object()` is called if it is in state `FSCACHE_OBJECT_RECYCLING`. The `fscache_object` struct should be initialised by calling `fscache_object_init(object)`.

- FS-Cache operation record:

```
struct fscache_operation {
    atomic_t                usage;
    struct fscache_object    *object;
    unsigned long            flags;
#define FSCACHE_OP_EXCLUSIVE
    void (*processor)(struct fscache_operation *op);
    void (*release)(struct fscache_operation *op);
    ...
};
```

FS-Cache has a pool of threads that it uses to give CPU time to the various asynchronous operations that need to be done as part of driving the cache.

These are represented by the above structure. The processor method is called to give the op CPU time, and the release method to get rid of it when its usage count reaches 0.

An operation can be made exclusive upon an object by setting the appropriate flag before enqueueing it with `fscache_enqueue_operation()`. If an operation needs more processing time, it should be enqueued again.

- FS-Cache retrieval operation record:

```
struct fscache_retrieval {
    struct fscache_operation op;
    struct address_space      *mapping;
    struct list_head          *to_do;
    ...
};
```

A structure of this type is allocated by FS-Cache to record retrieval and allocation requests made by the netfs. This struct is then passed to the backend to do the operation. The backend may get extra refs to it by calling `fscache_get_retrieval()` and refs may be discarded by calling `fscache_put_retrieval()`.

A retrieval operation can be used by the backend to do retrieval work. To do this, the `retrieval->op.processor` method pointer should be set appropriately by the backend and `fscache_enqueue_retrieval()` called to submit it to the thread pool. CacheFiles, for example, uses this to queue page examination when it detects `PG_lock` being cleared.

The `to_do` field is an empty list available for the cache backend to use as it sees fit.

- FS-Cache storage operation record:

```
struct fscache_storage {
    struct fscache_operation op;
    pgoff_t                  store_limit;
    ...
};
```

A structure of this type is allocated by FS-Cache to record outstanding writes to be made. FS-Cache itself enqueues this operation and invokes the `write_page()` method on the object at appropriate times to effect storage.

Cache Operations

The cache backend provides FS-Cache with a table of operations that can be performed on the denizens of the cache. These are held in a structure of type:

```
struct fscache_cache_ops
```

- Name of cache provider [mandatory]:

```
const char *name
```

This isn't strictly an operation, but should be pointed at a string naming the backend.

- Allocate a new object [mandatory]:

```
struct fscache_object *(*alloc_object)(struct fscache_cache_
↳*cache,
                                     struct fscache_cookie_
↳*cookie)
```

This method is used to allocate a cache object representation to back a cookie in a particular cache. `fscache_object_init()` should be called on the object to initialise it prior to returning.

This function may also be used to parse the index key to be used for multiple lookup calls to turn it into a more convenient form. FS-Cache will call the `lookup_complete()` method to allow the cache to release the form once lookup is complete or aborted.

- Look up and create object [mandatory]:

```
void (*lookup_object)(struct fscache_object *object)
```

This method is used to look up an object, given that the object is already allocated and attached to the cookie. This should instantiate that object in the cache if it can.

The method should call `fscache_object_lookup_negative()` as soon as possible if it determines the object doesn't exist in the cache. If the object is found to exist and the netfs indicates that it is valid then `fscache_obtained_object()` should be called once the object is in a position to have data stored in it. Similarly, `fscache_obtained_object()` should also be called once a non-present object has been created.

If a lookup error occurs, `fscache_object_lookup_error()` should be called to abort the lookup of that object.

- Release lookup data [mandatory]:

```
void (*lookup_complete)(struct fscache_object *object)
```

This method is called to ask the cache to release any resources it was using to perform a lookup.

- Increment object refcount [mandatory]:

```
struct fscache_object *(*grab_object)(struct fscache_object_
↳*object)
```

This method is called to increment the reference count on an object. It may fail (for instance if the cache is being withdrawn) by returning NULL. It should return the object pointer if successful.

- Lock/Unlock object [mandatory]:

```
void (*lock_object)(struct fscache_object *object)
void (*unlock_object)(struct fscache_object *object)
```

These methods are used to exclusively lock an object. It must be possible to schedule with the lock held, so a spinlock isn't sufficient.

- Pin/Unpin object [optional]:

```
int (*pin_object)(struct fscache_object *object)
void (*unpin_object)(struct fscache_object *object)
```

These methods are used to pin an object into the cache. Once pinned an object cannot be reclaimed to make space. Return `-ENOSPC` if there's not enough space in the cache to permit this.

- Check coherency state of an object [mandatory]:

```
int (*check_consistency)(struct fscache_object *object)
```

This method is called to have the cache check the saved auxiliary data of the object against the netfs's idea of the state. 0 should be returned if they're consistent and `-ESTALE` otherwise. `-ENOMEM` and `-ERESTARTSYS` may also be returned.

- Update object [mandatory]:

```
int (*update_object)(struct fscache_object *object)
```

This is called to update the index entry for the specified object. The new information should be in `object->cookie->netfs_data`. This can be obtained by calling `object->cookie->def->get_aux()/get_attr()`.

- Invalidate data object [mandatory]:

```
int (*invalidate_object)(struct fscache_operation *op)
```

This is called to invalidate a data object (as pointed to by `op->object`). All the data stored for this object should be discarded and an `attr_changed` operation should be performed. The caller will follow up with an object update operation.

`fscache_op_complete()` must be called on `op` before returning.

- Discard object [mandatory]:

```
void (*drop_object)(struct fscache_object *object)
```

This method is called to indicate that an object has been unbound from its cookie, and that the cache should release the object's resources and retire it if it's in state `FSCACHE_OBJECT_RECYCLING`.

This method should not attempt to release any references held by the caller. The caller will invoke the `put_object()` method as appropriate.

- Release object reference [mandatory]:

```
void (*put_object)(struct fscache_object *object)
```

This method is used to discard a reference to an object. The object may be freed when all the references to it are released.

- Synchronise a cache [mandatory]:

```
void (*sync)(struct fscache_cache *cache)
```

This is called to ask the backend to synchronise a cache with its backing device.

- Dissociate a cache [mandatory]:

```
void (*dissociate_pages)(struct fscache_cache *cache)
```

This is called to ask a cache to perform any page dissociations as part of cache withdrawal.

- Notification that the attributes on a netfs file changed [mandatory]:

```
int (*attr_changed)(struct fscache_object *object);
```

This is called to indicate to the cache that certain attributes on a netfs file have changed (for example the maximum size a file may reach). The cache can read these from the netfs by calling the cookie's `get_attr()` method.

The cache may use the file size information to reserve space on the cache. It should also call `fscache_set_store_limit()` to indicate to FS-Cache the highest byte it's willing to store for an object.

This method may return -ve if an error occurred or the cache object cannot be expanded. In such a case, the object will be withdrawn from service.

This operation is run asynchronously from FS-Cache's thread pool, and storage and retrieval operations from the netfs are excluded during the execution of this operation.

- Reserve cache space for an object's data [optional]:

```
int (*reserve_space)(struct fscache_object *object, loff_t  
↳ size);
```

This is called to request that cache space be reserved to hold the data for an object and the metadata used to track it. Zero size should be taken as request to cancel a reservation.

This should return 0 if successful, -ENOSPC if there isn't enough space available, or -ENOMEM or -EIO on other errors.

The reservation may exceed the current size of the object, thus permitting future expansion. If the amount of space consumed by an object would exceed the reservation, it's permitted to refuse requests to allocate pages, but not required. An object may be pruned down to its reservation size if larger than that already.

- Request page be read from cache [mandatory]:

```
int (*read_or_alloc_page)(struct fscache_retrieval *op,  
                          struct page *page,  
                          gfp_t gfp)
```

This is called to attempt to read a netfs page from the cache, or to reserve a backing block if not. FS-Cache will have done as much checking as it can before calling, but most of the work belongs to the backend.

If there's no page in the cache, then `-ENODATA` should be returned if the backend managed to reserve a backing block; `-ENOBUFFS` or `-ENOMEM` if it didn't.

If there is suitable data in the cache, then a read operation should be queued and 0 returned. When the read finishes, `fscache_end_io()` should be called.

The `fscache_mark_pages_cached()` should be called for the page if any cache metadata is retained. This will indicate to the netfs that the page needs explicit uncaching. This operation takes a `pagevec`, thus allowing several pages to be marked at once.

The retrieval record pointed to by `op` should be retained for each page queued and released when I/O on the page has been formally ended. `fscache_get/put_retrieval()` are available for this purpose.

The retrieval record may be used to get CPU time via the FS-Cache thread pool. If this is desired, the `op->op.processor` should be set to point to the appropriate processing routine, and `fscache_enqueue_retrieval()` should be called at an appropriate point to request CPU time. For instance, the retrieval routine could be enqueued upon the completion of a disk read. The `to_do` field in the retrieval record is provided to aid in this.

If an I/O error occurs, `fscache_io_error()` should be called and `-ENOBUFFS` returned if possible or `fscache_end_io()` called with a suitable error code.

`fscache_put_retrieval()` should be called after a page or pages are dealt with. This will complete the operation when all pages are dealt with.

- Request pages be read from cache [mandatory]:

```
int (*read_or_alloc_pages)(struct fscache_retrieval *op,
                          struct list_head *pages,
                          unsigned *nr_pages,
                          gfp_t gfp)
```

This is like the `read_or_alloc_page()` method, except it is handed a list of pages instead of one page. Any pages on which a read operation is started must be added to the page cache for the specified mapping and also to the LRU. Such pages must also be removed from the pages list and `*nr_pages` decremented per page.

If there was an error such as `-ENOMEM`, then that should be returned; else if one or more pages couldn't be read or allocated, then `-ENOBUFFS` should be returned; else if one or more pages couldn't be read, then `-ENODATA` should be returned. If all the pages are dispatched then 0 should be returned.

- Request page be allocated in the cache [mandatory]:

```
int (*allocate_page)(struct fscache_retrieval *op,
                    struct page *page,
                    gfp_t gfp)
```

This is like the `read_or_alloc_page()` method, except that it shouldn't read from the cache, even if there's data there that could be retrieved. It should, however, set up any internal metadata required such that the `write_page()` method can write to the cache.

If there's no backing block available, then `-ENOBUFFS` should be returned (or `-ENOMEM` if there were other problems). If a block is successfully allocated, then the netfs page should be marked and 0 returned.

- Request pages be allocated in the cache [mandatory]:

```
int (*allocate_pages)(struct fscache_retrieval *op,
                    struct list_head *pages,
                    unsigned *nr_pages,
                    gfp_t gfp)
```

This is an multiple page version of the `allocate_page()` method. `pages` and `nr_pages` should be treated as for the `read_or_alloc_page()` method.

- Request page be written to cache [mandatory]:

```
int (*write_page)(struct fscache_storage *op,
                 struct page *page);
```

This is called to write from a page on which there was a previously successful `read_or_alloc_page()` call or similar. FS-Cache filters out pages that don't have mappings.

This method is called asynchronously from the FS-Cache thread pool. It is not required to actually store anything, provided `-ENODATA` is then returned to the next read of this page.

If an error occurred, then a negative error code should be returned, otherwise zero should be returned. FS-Cache will take appropriate action in response to an error, such as withdrawing this object.

If this method returns success then FS-Cache will inform the netfs appropriately.

- Discard retained per-page metadata [mandatory]:

```
void (*uncache_page)(struct fscache_object *object, struct ↵
                    ↵page *page)
```

This is called when a netfs page is being evicted from the pagecache. The cache backend should tear down any internal representation or tracking it maintains for this page.

FS-Cache Utilities

FS-Cache provides some utilities that a cache backend may make use of:

- Note occurrence of an I/O error in a cache:

```
void fscache_io_error(struct fscache_cache *cache)
```

This tells FS-Cache that an I/O error occurred in the cache. After this has been called, only resource dissociation operations (object and page release) will be passed from the netfs to the cache backend for the specified cache.

This does not actually withdraw the cache. That must be done separately.

- Invoke the retrieval I/O completion function:

```
void fscache_end_io(struct fscache_retrieval *op, struct page *page,  
int error);
```

This is called to note the end of an attempt to retrieve a page. The error value should be 0 if successful and an error otherwise.

- Record that one or more pages being retrieved or allocated have been dealt with:

```
void fscache_retrieval_complete(struct fscache_retrieval *op,  
int n_pages);
```

This is called to record the fact that one or more pages have been dealt with and are no longer the concern of this operation. When the number of pages remaining in the operation reaches 0, the operation will be completed.

- Record operation completion:

```
void fscache_op_complete(struct fscache_operation *op);
```

This is called to record the completion of an operation. This deducts this operation from the parent object's run state, potentially permitting one or more pending operations to start running.

- Set highest store limit:

```
void fscache_set_store_limit(struct fscache_object *object,  
loff_t i_size);
```

This sets the limit FS-Cache imposes on the highest byte it's willing to try and store for a netfs. Any page over this limit is automatically rejected by `fscache_read_alloc_page()` and co with `-ENOBUFFS`.

- Mark pages as being cached:

```
void fscache_mark_pages_cached(struct fscache_retrieval *op,  
struct pagevec *pagevec);
```

This marks a set of pages as being cached. After this has been called, the netfs must call `fscache_uncache_page()` to unmark the pages.

- Perform coherency check on an object:

```
enum fscache_checkaux fscache_check_aux(struct fscache_object *object,
                                         const void *data,
                                         uint16_t datalen);
```

This asks the netfs to perform a coherency check on an object that has just been looked up. The cookie attached to the object will determine the netfs to use. `data` and `datalen` should specify where the auxiliary data retrieved from the cache can be found.

One of three values will be returned:

FSCACHE_CHECKAUX_OKAY The coherency data indicates the object is valid as is.

FSCACHE_CHECKAUX_NEEDS_UPDATE The coherency data needs updating, but otherwise the object is valid.

FSCACHE_CHECKAUX_OBSOLETE The coherency data indicates that the object is obsolete and should be discarded.

- Initialise a freshly allocated object:

```
void fscache_object_init(struct fscache_object *object);
```

This initialises all the fields in an object representation.

- Indicate the destruction of an object:

```
void fscache_object_destroyed(struct fscache_cache *cache);
```

This must be called to inform FS-Cache that an object that belonged to a cache has been destroyed and deallocated. This will allow continuation of the cache withdrawal process when it is stopped pending destruction of all the objects.

- Indicate negative lookup on an object:

```
void fscache_object_lookup_negative(struct fscache_object *object);
```

This is called to indicate to FS-Cache that a lookup process for an object found a negative result.

This changes the state of an object to permit reads pending on lookup completion to go off and start fetching data from the netfs server as it's known at this point that there can't be any data in the cache.

This may be called multiple times on an object. Only the first call is significant - all subsequent calls are ignored.

- Indicate an object has been obtained:

```
void fscache_obtained_object(struct fscache_object *object);
```

This is called to indicate to FS-Cache that a lookup process for an object produced a positive result, or that an object was created. This should only be called once for any particular object.

This changes the state of an object to indicate:

(1) if no call to `fscache_object_lookup_negative()` has been made on this object, that there may be data available, and that reads can now go and look for it; and

(2) that writes may now proceed against this object.

- Indicate that object lookup failed:

```
void fscache_object_lookup_error(struct fscache_object *object);
```

This marks an object as having encountered a fatal error (usually EIO) and causes it to move into a state whereby it will be withdrawn as soon as possible.

- Indicate that a stale object was found and discarded:

```
void fscache_object_retrying_stale(struct fscache_object *object);
```

This is called to indicate that the lookup procedure found an object in the cache that the netfs decided was stale. The object has been discarded from the cache and the lookup will be performed again.

- Indicate that the caching backend killed an object:

```
void fscache_object_mark_killed(struct fscache_object *object,  
                               enum fscache_why_object_killed why);
```

This is called to indicate that the cache backend preemptively killed an object. The `why` parameter should be set to indicate the reason:

FSCACHE_OBJECT_IS_STALE

- the object was stale and needs discarding.

FSCACHE_OBJECT_NO_SPACE

- there was insufficient cache space

FSCACHE_OBJECT_WAS_RETIRED

- the object was retired when relinquished.

FSCACHE_OBJECT_WAS_CULLED

- the object was culled to make space.

- Get and release references on a retrieval record:

```
void fscache_get_retrieval(struct fscache_retrieval *op);  
void fscache_put_retrieval(struct fscache_retrieval *op);
```

These two functions are used to retain a retrieval record while doing asynchronous data retrieval and block allocation.

- Enqueue a retrieval record for processing:

```
void fscache_enqueue_retrieval(struct fscache_retrieval *op);
```

This enqueues a retrieval record for processing by the FS-Cache thread pool. One of the threads in the pool will invoke the retrieval record's `op`

>op.processor callback function. This function may be called from within the callback function.

- List of object state names:

```
const char *fscache_object_states[];
```

For debugging purposes, this may be used to turn the state that an object is in into a text string for display purposes.

1.21.4 CacheFiles: CACHE ON ALREADY MOUNTED FILESYSTEM

Overview

CacheFiles is a caching backend that's meant to use as a cache a directory on an already mounted filesystem of a local type (such as Ext3).

CacheFiles uses a userspace daemon to do some of the cache management - such as reaping stale nodes and culling. This is called `cachefilesd` and lives in `/sbin`.

The filesystem and data integrity of the cache are only as good as those of the filesystem providing the backing services. Note that CacheFiles does not attempt to journal anything since the journalling interfaces of the various filesystems are very specific in nature.

CacheFiles creates a misc character device - `"/dev/cachefiles"` - that is used to communication with the daemon. Only one thing may have this open at once, and while it is open, a cache is at least partially in existence. The daemon opens this and sends commands down it to control the cache.

CacheFiles is currently limited to a single cache.

CacheFiles attempts to maintain at least a certain percentage of free space on the filesystem, shrinking the cache by culling the objects it contains to make space if necessary - see the "Cache Culling" section. This means it can be placed on the same medium as a live set of data, and will expand to make use of spare space and automatically contract when the set of data requires more space.

Requirements

The use of CacheFiles and its daemon requires the following features to be available in the system and in the cache filesystem:

- `dnotify`.
- extended attributes (`xattrs`).
- `openat()` and friends.
- `bmap()` support on files in the filesystem (`FIBMAP` ioctl).
- The use of `bmap()` to detect a partial page at the end of the file.

It is strongly recommended that the `"dir_index"` option is enabled on Ext3 filesystems being used as a cache.

Configuration

The cache is configured by a script in `/etc/cache/filesd.conf`. These commands set up cache ready for use. The following script commands are available:

brun <N>%, bcull <N>%, bstop <N>%, frun <N>%, fcull <N>%, fstop <N>%

Configure the culling limits. Optional. See the section on culling. The defaults are 7% (run), 5% (cull) and 1% (stop) respectively.

The commands beginning with a ‘b’ are file space (block) limits, those beginning with an ‘f’ are file count limits.

dir <path> Specify the directory containing the root of the cache. Mandatory.

tag <name> Specify a tag to FS-Cache to use in distinguishing multiple caches. Optional. The default is “CacheFiles” .

debug <mask> Specify a numeric bitmask to control debugging in the kernel module. Optional. The default is zero (all off). The following values can be OR’ d into the mask to collect various information:

1	Turn on trace of function entry (<code>_enter()</code> macros)
2	Turn on trace of function exit (<code>_leave()</code> macros)
4	Turn on trace of internal debug points (<code>_debug()</code>)

This mask can also be set through sysfs, eg:

```
echo 5 >/sys/modules/cache/files/parameters/debug
```

Starting the Cache

The cache is started by running the daemon. The daemon opens the cache device, configures the cache and tells it to begin caching. At that point the cache binds to `fs-cache` and the cache becomes live.

The daemon is run as follows:

```
/sbin/cache/filesd [-d]* [-s] [-n] [-f <configfile>]
```

The flags are:

- d** Increase the debugging level. This can be specified multiple times and is cumulative with itself.
- s** Send messages to `stderr` instead of `syslog`.
- n** Don’ t daemonise and go into background.
- f <configfile>** Use an alternative configuration file rather than the default one.

Things to Avoid

Do not mount other things within the cache as this will cause problems. The kernel module contains its own very cut-down path walking facility that ignores mount-points, but the daemon can't avoid them.

Do not create, rename or unlink files and directories in the cache while the cache is active, as this may cause the state to become uncertain.

Renaming files in the cache might make objects appear to be other objects (the filename is part of the lookup key).

Do not change or remove the extended attributes attached to cache files by the cache as this will cause the cache state management to get confused.

Do not create files or directories in the cache, lest the cache get confused or serve incorrect data.

Do not chmod files in the cache. The module creates things with minimal permissions to prevent random users being able to access them directly.

Cache Culling

The cache may need culling occasionally to make space. This involves discarding objects from the cache that have been used less recently than anything else. Culling is based on the access time of data objects. Empty directories are culled if not in use.

Cache culling is done on the basis of the percentage of blocks and the percentage of files available in the underlying filesystem. There are six “limits” :

brun, frun If the amount of free space and the number of available files in the cache rises above both these limits, then culling is turned off.

bcull, fcull If the amount of available space or the number of available files in the cache falls below either of these limits, then culling is started.

bstop, fstop If the amount of available space or the number of available files in the cache falls below either of these limits, then no further allocation of disk space or files is permitted until culling has raised things above these limits again.

These must be configured thusly:

```
0 <= bstop < bcull < brun < 100
0 <= fstop < fcull < frun < 100
```

Note that these are percentages of available space and available files, and do `_not_` appear as 100 minus the percentage displayed by the “df” program.

The userspace daemon scans the cache to build up a table of cullable objects. These are then culled in least recently used order. A new scan of the cache is started as soon as space is made in the table. Objects will be skipped if their atimes have changed or if the kernel module says it is still using them.

Cache Structure

The CacheFiles module will create two directories in the directory it was given:

- cache/
- graveyard/

The active cache objects all reside in the first directory. The CacheFiles kernel module moves any retired or culled objects that it can't simply unlink to the graveyard from which the daemon will actually delete them.

The daemon uses dnotify to monitor the graveyard directory, and will delete anything that appears therein.

The module represents index objects as directories with the filename "I..." or "J..." . Note that the "cache/" directory is itself a special index.

Data objects are represented as files if they have no children, or directories if they do. Their filenames all begin "D..." or "E..." . If represented as a directory, data objects will have a file in the directory called "data" that actually holds the data.

Special objects are similar to data objects, except their filenames begin "S..." or "T..." .

If an object has children, then it will be represented as a directory. Immediately in the representative directory are a collection of directories named for hash values of the child object keys with an '@' prepended. Into this directory, if possible, will be placed the representations of the child objects:

```

/INDEX      /INDEX      /INDEX      /DATA FILES
/=====
cache/@4a/I03nfs/@30/Ji0000000000000000 - fHg8hi8400
cache/@4a/I03nfs/@30/Ji0000000000000000 - fHg8hi8400/@75/Es0g000w...DB1ry
cache/@4a/I03nfs/@30/Ji0000000000000000 - fHg8hi8400/@75/Es0g000w...N22ry
cache/@4a/I03nfs/@30/Ji0000000000000000 - fHg8hi8400/@75/Es0g000w...FP1ry
    
```

If the key is so long that it exceeds NAME_MAX with the decorations added on to it, then it will be cut into pieces, the first few of which will be used to make a nest of directories, and the last one of which will be the objects inside the last directory. The names of the intermediate directories will have '+' prepended:

```
J1223/@23/+xy...z/+kl...m/Epqr
```

Note that keys are raw data, and not only may they exceed NAME_MAX in size, they may also contain things like '/' and NUL characters, and so they may not be suitable for turning directly into a filename.

To handle this, CacheFiles will use a suitably printable filename directly and "base-64" encode ones that aren't directly suitable. The two versions of object filenames indicate the encoding:

OBJECT TYPE	PRINTABLE	ENCODED
Index	"I..."	"J..."
Data	"D..."	"E..."
Special	"S..."	"T..."

Intermediate directories are always “@” or “+” as appropriate.

Each object in the cache has an extended attribute label that holds the object type ID (required to distinguish special objects) and the auxiliary data from the netfs. The latter is used to detect stale objects in the cache and update or retire them.

Note that CacheFiles will erase from the cache any file it doesn't recognise or any file of an incorrect type (such as a FIFO file or a device file).

Security Model and SELinux

CacheFiles is implemented to deal properly with the LSM security features of the Linux kernel and the SELinux facility.

One of the problems that CacheFiles faces is that it is generally acting on behalf of a process, and running in that process' s context, and that includes a security context that is not appropriate for accessing the cache - either because the files in the cache are inaccessible to that process, or because if the process creates a file in the cache, that file may be inaccessible to other processes.

The way CacheFiles works is to temporarily change the security context (fsuid, fsgid and actor security label) that the process acts as - without changing the security context of the process when it the target of an operation performed by some other process (so signalling and suchlike still work correctly).

When the CacheFiles module is asked to bind to its cache, it:

- (1) Finds the security label attached to the root cache directory and uses that as the security label with which it will create files. By default, this is:

```
cachefiles_var_t
```

- (2) Finds the security label of the process which issued the bind request (presumed to be the cachefilesd daemon), which by default will be:

```
cachefilesd_t
```

and asks LSM to supply a security ID as which it should act given the daemon' s label. By default, this will be:

```
cachefiles_kernel_t
```

SELinux transitions the daemon' s security ID to the module' s security ID based on a rule of this form in the policy:

```
type_transition <daemon's-ID> kernel_t : process <module's-ID>;
```

For instance:

```
type_transition cachefilesd_t kernel_t : process cachefiles_kernel_t;
```

The module' s security ID gives it permission to create, move and remove files and directories in the cache, to find and access directories and files in the cache, to set and access extended attributes on cache objects, and to read and write files in the cache.

The daemon's security ID gives it only a very restricted set of permissions: it may scan directories, stat files and erase files and directories. It may not read or write files in the cache, and so it is precluded from accessing the data cached therein; nor is it permitted to create new files in the cache.

There are policy source files available in:

<http://people.redhat.com/~dhowells/fscache/cachefilesd-0.8.tar.bz2>

and later versions. In that tarball, see the files:

```
cachefilesd.te
cachefilesd.fc
cachefilesd.if
```

They are built and installed directly by the RPM.

If a non-RPM based system is being used, then copy the above files to their own directory and run:

```
make -f /usr/share/selinux/devel/Makefile
semodule -i cachefilesd.pp
```

You will need checkpolicy and selinux-policy-devel installed prior to the build.

By default, the cache is located in /var/fscache, but if it is desirable that it should be elsewhere, then either the above policy files must be altered, or an auxiliary policy must be installed to label the alternate location of the cache.

For instructions on how to add an auxiliary policy to enable the cache to be located elsewhere when SELinux is in enforcing mode, please see:

```
/usr/share/doc/cachefilesd-*/move-cache.txt
```

When the cachefilesd rpm is installed; alternatively, the document can be found in the sources.

A Note on Security

CacheFiles makes use of the split security in the task_struct. It allocates its own task_security structure, and redirects current->cred to point to it when it acts on behalf of another process, in that process's context.

The reason it does this is that it calls vfs_mkdir() and suchlike rather than bypassing security and calling inode ops directly. Therefore the VFS and LSM may deny the CacheFiles access to the cache data because under some circumstances the caching code is running in the security context of whatever process issued the original syscall on the netfs.

Furthermore, should CacheFiles create a file or directory, the security parameters with that object is created (UID, GID, security label) would be derived from that process that issued the system call, thus potentially preventing other processes from accessing the cache - including CacheFiles's cache management daemon (cachefilesd).

What is required is to temporarily override the security of the process that issued the system call. We can't, however, just do an in-place change of the security data

as that affects the process as an object, not just as a subject. This means it may lose signals or ptrace events for example, and affects what the process looks like in /proc.

So CacheFiles makes use of a logical split in the security between the objective security (task->real_cred) and the subjective security (task->cred). The objective security holds the intrinsic security properties of a process and is never overridden. This is what appears in /proc, and is what is used when a process is the target of an operation by some other process (SIGKILL for example).

The subjective security holds the active security properties of a process, and may be overridden. This is not seen externally, and is used when a process acts upon another object, for example SIGKILLing another process or opening a file.

LSM hooks exist that allow SELinux (or Smack or whatever) to reject a request for CacheFiles to run in a context of a specific security label, or to create files and directories with another security label.

Statistical Information

If FS-Cache is compiled with the following option enabled:

```
CONFIG_CACHEFILES_HISTOGRAM=y
```

then it will gather certain statistics and display them through a proc file.

/proc/fs/cache/files/histogram

```
cat /proc/fs/cache/files/histogram
JIFS SECS LOOKUPS MKDIRS CREATES
=====
```

This shows the breakdown of the number of times each amount of time between 0 jiffies and HZ-1 jiffies a variety of tasks took to run. The columns are as follows:

COL-UMN	TIME MEASUREMENT
LOOKUPS	Length of time to perform a lookup on the backing fs
MKDIRS	Length of time to perform a mkdir on the backing fs
CRE-ATES	Length of time to perform a create on the backing fs

Each row shows the number of events that took a particular range of times. Each step is 1 jiffy in size. The JIFS column indicates the particular jiffy range covered, and the SECS field the equivalent number of seconds.

Debugging

If `CONFIG_CACHEFILES_DEBUG` is enabled, the CacheFiles facility can have run-time debugging enabled by adjusting the value in:

```
/sys/module/cache/files/parameters/debug
```

This is a bitmask of debugging streams to enable:

BIT	VALUE	STREAM	POINT
0	1	General	Function entry trace
1	2		Function exit trace
2	4		General

The appropriate set of values should be OR' d together and the result written to the control file. For example:

```
echo $((1|4|8)) >/sys/module/cache/files/parameters/debug
```

will turn on all function entry debugging.

1.21.5 FS-Cache Network Filesystem API

There' s an API by which a network filesystem can make use of the FS-Cache facilities. This is based around a number of principles:

- (1) Caches can store a number of different object types. There are two main object types: indices and files. The first is a special type used by FS-Cache to make finding objects faster and to make retiring of groups of objects easier.
- (2) Every index, file or other object is represented by a cookie. This cookie may or may not have anything associated with it, but the netfs doesn' t need to care.
- (3) Barring the top-level index (one entry per cached netfs), the index hierarchy for each netfs is structured according the whim of the netfs.

This API is declared in `<linux/fscache.h>`.

Network Filesystem Definition

FS-Cache needs a description of the network filesystem. This is specified using a record of the following structure:

```
struct fscache_netfs {
    uint32_t                version;
    const char              *name;
    struct fscache_cookie   *primary_index;
    ...
};
```

This first two fields should be filled in before registration, and the third will be filled in by the registration function; any other fields should just be ignored and are for internal use only.

The fields are:

- (1) The name of the netfs (used as the key in the toplevel index).
- (2) The version of the netfs (if the name matches but the version doesn't, the entire in-cache hierarchy for this netfs will be scrapped and begun afresh).
- (3) The cookie representing the primary index will be allocated according to another parameter passed into the registration function.

For example, kAfs (linux/fs/afs/) uses the following definitions to describe itself:

```
struct fscache_netfs afs_cache_netfs = {
    .version      = 0,
    .name         = "afs",
};
```

Index Definition

Indices are used for two purposes:

- (1) To aid the finding of a file based on a series of keys (such as AFS' s "cell" , "volume ID" , "vnode ID").
- (2) To make it easier to discard a subset of all the files cached based around a particular key - for instance to mirror the removal of an AFS volume.

However, since it' s unlikely that any two netfs' s are going to want to define their index hierarchies in quite the same way, FS-Cache tries to impose as few restraints as possible on how an index is structured and where it is placed in the tree. The netfs can even mix indices and data files at the same level, but it' s not recommended.

Each index entry consists of a key of indeterminate length plus some auxiliary data, also of indeterminate length.

There are some limits on indices:

- (1) Any index containing non-index objects should be restricted to a single cache. Any such objects created within an index will be created in the first cache only. The cache in which an index is created can be controlled by cache tags (see below).
- (2) The entry data must be atomically journallable, so it is limited to about 400 bytes at present. At least 400 bytes will be available.
- (3) The depth of the index tree should be judged with care as the search function is recursive. Too many layers will run the kernel out of stack.

Object Definition

To define an object, a structure of the following type should be filled out:

```
struct fscache_cookie_def
{
    uint8_t name[16];
    uint8_t type;

    struct fscache_cache_tag *(*select_cache)(
        const void *parent_netfs_data,
        const void *cookie_netfs_data);

    enum fscache_checkaux (*check_aux)(void *cookie_netfs_data,
        const void *data,
        uint16_t datalen,
        loff_t object_size);

    void (*get_context)(void *cookie_netfs_data, void *context);
    void (*put_context)(void *cookie_netfs_data, void *context);

    void (*mark_pages_cached)(void *cookie_netfs_data,
        struct address_space *mapping,
        struct pagevec *cached_pvec);
};
```

This has the following fields:

- (1) The type of the object [mandatory].

This is one of the following values:

FSCACHE_COOKIE_TYPE_INDEX This defines an index, which is a special FS-Cache type.

FSCACHE_COOKIE_TYPE_DATAFILE This defines an ordinary data file.

Any other value between 2 and 255 This defines an extraordinary object such as an XATTR.

- (2) The name of the object type (NUL terminated unless all 16 chars are used) [optional].
- (3) A function to select the cache in which to store an index [optional].

This function is invoked when an index needs to be instantiated in a cache during the instantiation of a non-index object. Only the immediate index parent for the non-index object will be queried. Any indices above that in the hierarchy may be stored in multiple caches. This function does not need to be supplied for any non-index object or any index that will only have index children.

If this function is not supplied or if it returns NULL then the first cache in the parent's list will be chosen, or failing that, the first cache in the master list.

- (4) A function to check the auxiliary data [optional].

This function will be called to check that a match found in the cache for this object is valid. For instance with AFS it could check the auxiliary data against the data version number returned by the server to determine whether the index entry in a cache is still valid.

If this function is absent, it will be assumed that matching objects in a cache are always valid.

The function is also passed the cache's idea of the object size and may use this to manage coherency also.

If present, the function should return one of the following values:

FSCACHE_CHECKAUX_OKAY

- the entry is okay as is

FSCACHE_CHECKAUX_NEEDS_UPDATE

- the entry requires update

FSCACHE_CHECKAUX_OBSOLETE

- the entry should be deleted

This function can also be used to extract data from the auxiliary data in the cache and copy it into the netfs's structures.

- (5) A pair of functions to manage contexts for the completion callback [optional].

The cache read/write functions are passed a context which is then passed to the I/O completion callback function. To ensure this context remains valid until after the I/O completion is called, two functions may be provided: one to get an extra reference on the context, and one to drop a reference to it.

If the context is not used or is a type of object that won't go out of scope, then these functions are not required. These functions are not required for indices as indices may not contain data. These functions may be called in interrupt context and so may not sleep.

- (6) A function to mark a page as retaining cache metadata [optional].

This is called by the cache to indicate that it is retaining in-memory information for this page and that the netfs should uncache the page when it has finished. This does not indicate whether there's data on the disk or not. Note that several pages at once may be presented for marking.

The PG_fscache bit is set on the pages before this function would be called, so the function need not be provided if this is sufficient.

This function is not required for indices as they're not permitted data.

- (7) A function to unmark all the pages retaining cache metadata [mandatory].

This is called by FS-Cache to indicate that a backing store is being unbound from a cookie and that all the marks on the pages should be cleared to prevent confusion. Note that the cache will have torn down all its tracking information so that the pages don't need to be explicitly uncached.

This function is not required for indices as they're not permitted data.

Network Filesystem (Un)registration

The first step is to declare the network filesystem to the cache. This also involves specifying the layout of the primary index (for AFS, this would be the “cell” level).

The registration function is:

```
int fscache_register_netfs(struct fscache_netfs *netfs);
```

It just takes a pointer to the netfs definition. It returns 0 or an error as appropriate.

For kAfs, registration is done as follows:

```
ret = fscache_register_netfs(&afs_cache_netfs);
```

The last step is, of course, unregistration:

```
void fscache_unregister_netfs(struct fscache_netfs *netfs);
```

Cache Tag Lookup

FS-Cache permits the use of more than one cache. To permit particular index subtrees to be bound to particular caches, the second step is to look up cache representation tags. This step is optional; it can be left entirely up to FS-Cache as to which cache should be used. The problem with doing that is that FS-Cache will always pick the first cache that was registered.

To get the representation for a named tag:

```
struct fscache_cache_tag *fscache_lookup_cache_tag(const char *name);
```

This takes a text string as the name and returns a representation of a tag. It will never return an error. It may return a dummy tag, however, if it runs out of memory; this will inhibit caching with this tag.

Any representation so obtained must be released by passing it to this function:

```
void fscache_release_cache_tag(struct fscache_cache_tag *tag);
```

The tag will be retrieved by FS-Cache when it calls the object definition operation `select_cache()`.

Index Registration

The third step is to inform FS-Cache about part of an index hierarchy that can be used to locate files. This is done by requesting a cookie for each index in the path to the file:

```
struct fscache_cookie *
fscache_acquire_cookie(struct fscache_cookie *parent,
                      const struct fscache_object_def *def,
                      const void *index_key,
                      size_t index_key_len,
```

(continues on next page)

(continued from previous page)

```

const void *aux_data,
size_t aux_data_len,
void *netfs_data,
loff_t object_size,
bool enable);

```

This function creates an index entry in the index represented by parent, filling in the index entry by calling the operations pointed to by def.

A unique key that represents the object within the parent must be pointed to by index_key and is of length index_key_len.

An optional blob of auxiliary data that is to be stored within the cache can be pointed to with aux_data and should be of length aux_data_len. This would typically be used for storing coherency data.

The netfs may pass an arbitrary value in netfs_data and this will be presented to it in the event of any calling back. This may also be used in tracing or logging of messages.

The cache tracks the size of the data attached to an object and this set to be object_size. For indices, this should be 0. This value will be passed to the ->check_aux() callback.

Note that this function never returns an error - all errors are handled internally. It may, however, return NULL to indicate no cookie. It is quite acceptable to pass this token back to this function as the parent to another acquisition (or even to the relinquish cookie, read page and write page functions - see below).

Note also that no indices are actually created in a cache until a non-index object needs to be created somewhere down the hierarchy. Furthermore, an index may be created in several different caches independently at different times. This is all handled transparently, and the netfs doesn't see any of it.

A cookie will be created in the disabled state if enabled is false. A cookie must be enabled to do anything with it. A disabled cookie can be enabled by calling fscache_enable_cookie() (see below).

For example, with AFS, a cell would be added to the primary index. This index entry would have a dependent inode containing volume mappings within this cell:

```

cell->cache =
    fscache_acquire_cookie(afs_cache_netfs.primary_index,
                          &afs_cell_cache_index_def,
                          cell->name, strlen(cell->name),
                          NULL, 0,
                          cell, 0, true);

```

And then a particular volume could be added to that index by ID, creating another index for vnodes (AFS inode equivalents):

```

volume->cache =
    fscache_acquire_cookie(volume->cell->cache,
                          &afs_volume_cache_index_def,
                          &volume->vid, sizeof(volume->vid),

```

(continues on next page)

(continued from previous page)

```
NULL, 0,  
volume, 0, true);
```

Data File Registration

The fourth step is to request a data file be created in the cache. This is identical to index cookie acquisition. The only difference is that the type in the object definition should be something other than index type:

```
vnode->cache =  
    fscache_acquire_cookie(volume->cache,  
                           &afs_vnode_cache_object_def,  
                           &key, sizeof(key),  
                           &aux, sizeof(aux),  
                           vnode, vnode->status.size, true);
```

Miscellaneous Object Registration

An optional step is to request an object of miscellaneous type be created in the cache. This is almost identical to index cookie acquisition. The only difference is that the type in the object definition should be something other than index type. While the parent object could be an index, it's more likely it would be some other type of object such as a data file:

```
xattr->cache =  
    fscache_acquire_cookie(vnode->cache,  
                           &afs_xattr_cache_object_def,  
                           &xattr->name, strlen(xattr->name),  
                           NULL, 0,  
                           xattr, strlen(xattr->val), true);
```

Miscellaneous objects might be used to store extended attributes or directory entries for example.

Setting the Data File Size

The fifth step is to set the physical attributes of the file, such as its size. This doesn't automatically reserve any space in the cache, but permits the cache to adjust its metadata for data tracking appropriately:

```
int fscache_attr_changed(struct fscache_cookie *cookie);
```

The cache will return `-ENOBUFFS` if there is no backing cache or if there is no space to allocate any extra metadata required in the cache.

Note that attempts to read or write data pages in the cache over this size may be rebuffed with `-ENOBUFFS`.

This operation schedules an attribute adjustment to happen asynchronously at some point in the future, and as such, it may happen after the function returns to the caller. The attribute adjustment excludes read and write operations.

Page alloc/read/write

And the sixth step is to store and retrieve pages in the cache. There are three functions that are used to do this.

Note:

- (1) A page should not be re-read or re-allocated without uncaching it first.
- (2) A read or allocated page must be uncached when the netfs page is released from the pagecache.
- (3) A page should only be written to the cache if previous read or allocated.

This permits the cache to maintain its page tracking in proper order.

PAGE READ

Firstly, the netfs should ask FS-Cache to examine the caches and read the contents cached for a particular page of a particular file if present, or else allocate space to store the contents if not:

```
typedef
void (*fscache_rw_complete_t)(struct page *page,
                              void *context,
                              int error);

int fscache_read_or_alloc_page(struct fscache_cookie *cookie,
                              struct page *page,
                              fscache_rw_complete_t end_io_func,
                              void *context,
                              gfp_t gfp);
```

The cookie argument must specify a cookie for an object that isn't an index, the page specified will have the data loaded into it (and is also used to specify the page number), and the gfp argument is used to control how any memory allocations made are satisfied.

If the cookie indicates the inode is not cached:

- (1) The function will return -ENOBUFFS.

Else if there's a copy of the page resident in the cache:

- (1) The mark_pages_cached() cookie operation will be called on that page.
- (2) The function will submit a request to read the data from the cache's backing device directly into the page specified.
- (3) The function will return 0.
- (4) When the read is complete, end_io_func() will be invoked with:
 - The netfs data supplied when the cookie was created.
 - The page descriptor.
 - The context argument passed to the above function. This will be maintained with the get_context/put_context functions mentioned above.

- An argument that's 0 on success or negative for an error code.

If an error occurs, it should be assumed that the page contains no usable data. `fscache_readpages_cancel()` may need to be called.

`end_io_func()` will be called in process context if the read results in an error, but it might be called in interrupt context if the read is successful.

Otherwise, if there's not a copy available in cache, but the cache may be able to store the page:

- (1) The `mark_pages_cached()` cookie operation will be called on that page.
- (2) A block may be reserved in the cache and attached to the object at the appropriate place.
- (3) The function will return `-ENODATA`.

This function may also return `-ENOMEM` or `-EINTR`, in which case it won't have read any data from the cache.

Page Allocate

Alternatively, if there's not expected to be any data in the cache for a page because the file has been extended, a block can simply be allocated instead:

```
int fscache_alloc_page(struct fscache_cookie *cookie,
                      struct page *page,
                      gfp_t gfp);
```

This is similar to the `fscache_read_or_alloc_page()` function, except that it never reads from the cache. It will return 0 if a block has been allocated, rather than `-ENODATA` as the other would. One or the other must be performed before writing to the cache.

The `mark_pages_cached()` cookie operation will be called on the page if successful.

Page Write

Secondly, if the netfs changes the contents of the page (either due to an initial download or if a user performs a write), then the page should be written back to the cache:

```
int fscache_write_page(struct fscache_cookie *cookie,
                      struct page *page,
                      loff_t object_size,
                      gfp_t gfp);
```

The cookie argument must specify a data file cookie, the page specified should contain the data to be written (and is also used to specify the page number), `object_size` is the revised size of the object and the `gfp` argument is used to control how any memory allocations made are satisfied.

The page must have first been read or allocated successfully and must not have been uncached before writing is performed.

If the cookie indicates the inode is not cached then:

- (1) The function will return -ENOBUFS.

Else if space can be allocated in the cache to hold this page:

- (1) PG_fscache_write will be set on the page.
- (2) The function will submit a request to write the data to cache's backing device directly from the page specified.
- (3) The function will return 0.
- (4) When the write is complete PG_fscache_write is cleared on the page and anyone waiting for that bit will be woken up.

Else if there's no space available in the cache, -ENOBUFS will be returned. It is also possible for the PG_fscache_write bit to be cleared when no write took place if unforeseen circumstances arose (such as a disk error).

Writing takes place asynchronously.

Multiple Page Read

A facility is provided to read several pages at once, as requested by the `readpages()` address space operation:

```
int fscache_read_or_alloc_pages(struct fscache_cookie *cookie,
                               struct address_space *mapping,
                               struct list_head *pages,
                               int *nr_pages,
                               fscache_rw_complete_t end_io_func,
                               void *context,
                               gfp_t gfp);
```

This works in a similar way to `fscache_read_or_alloc_page()`, except:

- (1) Any page it can retrieve data for is removed from `pages` and `nr_pages` and dispatched for reading to the disk. Reads of adjacent pages on disk may be merged for greater efficiency.
- (2) The `mark_pages_cached()` cookie operation will be called on several pages at once if they're being read or allocated.
- (3) If there was a general error, then that error will be returned.

Else if some pages couldn't be allocated or read, then -ENOBUFS will be returned.

Else if some pages couldn't be read but were allocated, then -ENODATA will be returned.

Otherwise, if all pages had reads dispatched, then 0 will be returned, the list will be empty and `*nr_pages` will be 0.

- (4) `end_io_func` will be called once for each page being read as the reads complete. It will be called in process context if `error != 0`, but it may be called in interrupt context if there is no error.

Note that a return of `-ENODATA`, `-ENOBUFFS` or any other error does not preclude some of the pages being read and some being allocated. Those pages will have been marked appropriately and will need uncaching.

Cancellation of Unread Pages

If one or more pages are passed to `fscache_read_or_alloc_pages()` but not then read from the cache and also not read from the underlying filesystem then those pages will need to have any marks and reservations removed. This can be done by calling:

```
void fscache_readpages_cancel(struct fscache_cookie *cookie,
                             struct list_head *pages);
```

prior to returning to the caller. The cookie argument should be as passed to `fscache_read_or_alloc_pages()`. Every page in the pages list will be examined and any that have `PG_fscache` set will be uncached.

Page Uncaching

To uncache a page, this function should be called:

```
void fscache_uncache_page(struct fscache_cookie *cookie,
                          struct page *page);
```

This function permits the cache to release any in-memory representation it might be holding for this netfs page. This function must be called once for each page on which the read or write page functions above have been called to make sure the cache's in-memory tracking information gets torn down.

Note that pages can't be explicitly deleted from the a data file. The whole data file must be retired (see the `relinquish cookie` function below).

Furthermore, note that this does not cancel the asynchronous read or write operation started by the read/alloc and write functions, so the page invalidation functions must use:

```
bool fscache_check_page_write(struct fscache_cookie *cookie,
                              struct page *page);
```

to see if a page is being written to the cache, and:

```
void fscache_wait_on_page_write(struct fscache_cookie *cookie,
                                struct page *page);
```

to wait for it to finish if it is.

When `releasepage()` is being implemented, a special FS-Cache function exists to manage the heuristics of coping with `vmscan` trying to eject pages, which may conflict with the cache trying to write pages to the cache (which may itself need to allocate memory):

```
bool fscache_maybe_release_page(struct fscache_cookie *cookie,
                               struct page *page,
                               gfp_t gfp);
```

This takes the netfs cookie, and the page and gfp arguments as supplied to `releasepage()`. It will return false if the page cannot be released yet for some reason and if it returns true, the page has been uncached and can now be released.

To make a page available for release, this function may wait for an outstanding storage request to complete, or it may attempt to cancel the storage request - in which case the page will not be stored in the cache this time.

Bulk Image Page Uncache

A convenience routine is provided to perform an uncache on all the pages attached to an inode. This assumes that the pages on the inode correspond on a 1:1 basis with the pages in the cache:

```
void fscache_uncache_all_inode_pages(struct fscache_cookie *cookie,
                                     struct inode *inode);
```

This takes the netfs cookie that the pages were cached with and the inode that the pages are attached to. This function will wait for pages to finish being written to the cache and for the cache to finish with the page generally. No error is returned.

Index and Data File consistency

To find out whether auxiliary data for an object is up to data within the cache, the following function can be called:

```
int fscache_check_consistency(struct fscache_cookie *cookie,
                              const void *aux_data);
```

This will call back to the netfs to check whether the auxiliary data associated with a cookie is correct; if `aux_data` is non-NULL, it will update the auxiliary data buffer first. It returns 0 if it is and `-ESTALE` if it isn't; it may also return `-ENOMEM` and `-ERESTARTSYS`.

To request an update of the index data for an index or other object, the following function should be called:

```
void fscache_update_cookie(struct fscache_cookie *cookie,
                           const void *aux_data);
```

This function will update the cookie's auxiliary data buffer from `aux_data` if that is non-NULL and then schedule this to be stored on disk. The update method in the parent index definition will be called to transfer the data.

Note that partial updates may happen automatically at other times, such as when data blocks are added to a data file object.

Cookie Enablement

Cookies exist in one of two states: enabled and disabled. If a cookie is disabled, it ignores all attempts to acquire child cookies; check, update or invalidate its state; allocate, read or write backing pages - though it is still possible to uncache pages and relinquish the cookie.

The initial enablement state is set by `fscache_acquire_cookie()`, but the cookie can be enabled or disabled later. To disable a cookie, call:

```
void fscache_disable_cookie(struct fscache_cookie *cookie,
                           const void *aux_data,
                           bool invalidate);
```

If the cookie is not already disabled, this locks the cookie against other enable and disable ops, marks the cookie as being disabled, discards or invalidates any backing objects and waits for cessation of activity on any associated object before unlocking the cookie.

All possible failures are handled internally. The caller should consider calling `fscache_uncache_all_inode_pages()` afterwards to make sure all page markings are cleared up.

Cookies can be enabled or reenabled with:

```
void fscache_enable_cookie(struct fscache_cookie *cookie,
                           const void *aux_data,
                           loff_t object_size,
                           bool (*can_enable)(void *data),
                           void *data)
```

If the cookie is not already enabled, this locks the cookie against other enable and disable ops, invokes `can_enable()` and, if the cookie is not an index cookie, will begin the procedure of acquiring backing objects.

The optional `can_enable()` function is passed the data argument and returns a ruling as to whether or not enablement should actually be permitted to begin.

All possible failures are handled internally. The cookie will only be marked as enabled if provisional backing objects are allocated.

The object's data size is updated from `object_size` and is passed to the `->check_aux()` function.

In both cases, the cookie's auxiliary data buffer is updated from `aux_data` if that is non-NULL inside the enablement lock before proceeding.

Miscellaneous Cookie operations

There are a number of operations that can be used to control cookies:

- Cookie pinning:

```
int fscache_pin_cookie(struct fscache_cookie *cookie);
void fscache_unpin_cookie(struct fscache_cookie ↵
↵ *cookie);
```

These operations permit data cookies to be pinned into the cache and to have the pinning removed. They are not permitted on index cookies.

The pinning function will return 0 if successful, -ENOBUFS in the cookie isn't backed by a cache, -EOPNOTSUPP if the cache doesn't support pinning, -ENOSPC if there isn't enough space to honour the operation, -ENOMEM or -EIO if there's any other problem.

- Data space reservation:

```
int fscache_reserve_space(struct fscache_cookie *cookie, loff_
↵ ↵ size);
```

This permits a netfs to request cache space be reserved to store up to the given amount of a file. It is permitted to ask for more than the current size of the file to allow for future file expansion.

If size is given as zero then the reservation will be cancelled.

The function will return 0 if successful, -ENOBUFS in the cookie isn't backed by a cache, -EOPNOTSUPP if the cache doesn't support reservations, -ENOSPC if there isn't enough space to honour the operation, -ENOMEM or -EIO if there's any other problem.

Note that this doesn't pin an object in a cache; it can still be culled to make space if it's not in use.

Cookie Unregistration

To get rid of a cookie, this function should be called:

```
void fscache_relinquish_cookie(struct fscache_cookie *cookie,
                               const void *aux_data,
                               bool retire);
```

If retire is non-zero, then the object will be marked for recycling, and all copies of it will be removed from all active caches in which it is present. Not only that but all child objects will also be retired.

If retire is zero, then the object may be available again when next the acquisition function is called. Retirement here will overrule the pinning on a cookie.

The cookie's auxiliary data will be updated from aux_data if that is non-NULL so that the cache can lazily update it on disk.

One very important note - `relinquish` must NOT be called for a cookie unless all the cookies for “child” indices, objects and pages have been relinquished first.

Index Invalidation

There is no direct way to invalidate an index subtree. To do this, the caller should relinquish and retire the cookie they have, and then acquire a new one.

Data File Invalidation

Sometimes it will be necessary to invalidate an object that contains data. Typically this will be necessary when the server tells the netfs of a foreign change - at which point the netfs has to throw away all the state it had for an inode and reload from the server.

To indicate that a cache object should be invalidated, the following function can be called:

```
void fscache_invalidate(struct fscache_cookie *cookie);
```

This can be called with spinlocks held as it defers the work to a thread pool. All extant storage, retrieval and attribute change ops at this point are cancelled and discarded. Some future operations will be rejected until the cache has had a chance to insert a barrier in the operations queue. After that, operations will be queued again behind the invalidation operation.

The invalidation operation will perform an attribute change operation and an auxiliary data update operation as it is very likely these will have changed.

Using the following function, the netfs can wait for the invalidation operation to have reached a point at which it can start submitting ordinary operations once again:

```
void fscache_wait_on_invalidate(struct fscache_cookie *cookie);
```

FS-cache Specific Page Flag

FS-Cache makes use of a page flag, `PG_private_2`, for its own purpose. This is given the alternative name `PG_fscache`.

`PG_fscache` is used to indicate that the page is known by the cache, and that the cache must be informed if the page is going to go away. It's an indication to the netfs that the cache has an interest in this page, where an interest may be a pointer to it, resources allocated or reserved for it, or I/O in progress upon it.

The netfs can use this information in methods such as `releasepage()` to determine whether it needs to uncache a page or update it.

Furthermore, if this bit is set, `releasepage()` and `invalidatepage()` operations will be called on a page to get rid of it, even if `PG_private` is not set. This allows caching to be attempted on a page before `read_cache_pages()` to be called after `fscache_read_or_alloc_pages()` as the former will try and release pages it was given under certain circumstances.

This bit does not overlap with such as PG_private. This means that FS-Cache can be used with a filesystem that uses the block buffering code.

There are a number of operations defined on this flag:

```
int PageFsCache(struct page *page);
void SetPageFsCache(struct page *page)
void ClearPageFsCache(struct page *page)
int TestSetPageFsCache(struct page *page)
int TestClearPageFsCache(struct page *page)
```

These functions are bit test, bit set, bit clear, bit test and set and bit test and clear operations on PG_fscache.

1.21.6 Asynchronous Operations Handling

By: David Howells <dhowells@redhat.com>

Overview

FS-Cache has an asynchronous operations handling facility that it uses for its data storage and retrieval routines. Its operations are represented by fscache_operation structs, though these are usually embedded into some other structure.

This facility is available to and expected to be used by the cache backends, and FS-Cache will create operations and pass them off to the appropriate cache backend for completion.

To make use of this facility, <linux/fscache-cache.h> should be #included.

Operation Record Initialisation

An operation is recorded in an fscache_operation struct:

```
struct fscache_operation {
    union {
        struct work_struct fast_work;
        struct slow_work slow_work;
    };
    unsigned long flags;
    fscache_operation_processor_t processor;
    ...
};
```

Someone wanting to issue an operation should allocate something with this struct embedded in it. They should initialise it by calling:

```
void fscache_operation_init(struct fscache_operation *op,
                           fscache_operation_release_t release);
```

with the operation to be initialised and the release function to use.

The `op->flags` parameter should be set to indicate the CPU time provision and the exclusivity (see the Parameters section).

The `op->fast_work`, `op->slow_work` and `op->processor` flags should be set as appropriate for the CPU time provision (see the Parameters section).

`FSCACHE_OP_WAITING` may be set in `op->flags` prior to each submission of the operation and waited for afterwards.

Parameters

There are a number of parameters that can be set in the operation record's flag parameter. There are three options for the provision of CPU time in these operations:

- (1) The operation may be done synchronously (`FSCACHE_OP_MYTHREAD`). A thread may decide it wants to handle an operation itself without deferring it to another thread.

This is, for example, used in read operations for calling `readpages()` on the backing filesystem in CacheFiles. Although `readpages()` does an asynchronous data fetch, the determination of whether pages exist is done synchronously - and the netfs does not proceed until this has been determined.

If this option is to be used, `FSCACHE_OP_WAITING` must be set in `op->flags` before submitting the operation, and the operating thread must wait for it to be cleared before proceeding:

```
wait_on_bit(&op->flags, FSCACHE_OP_WAITING,  
            TASK_UNINTERRUPTIBLE);
```

- (2) The operation may be fast asynchronous (`FSCACHE_OP_FAST`), in which case it will be given to `keventd` to process. Such an operation is not permitted to sleep on I/O.

This is, for example, used by CacheFiles to copy data from a backing fs page to a netfs page after the backing fs has read the page in.

If this option is used, `op->fast_work` and `op->processor` must be initialised before submitting the operation:

```
INIT_WORK(&op->fast_work, do_some_work);
```

- (3) The operation may be slow asynchronous (`FSCACHE_OP_SLOW`), in which case it will be given to the slow work facility to process. Such an operation is permitted to sleep on I/O.

This is, for example, used by FS-Cache to handle background writes of pages that have just been fetched from a remote server.

If this option is used, `op->slow_work` and `op->processor` must be initialised before submitting the operation:

```
fscache_operation_init_slow(op, processor)
```

Furthermore, operations may be one of two types:

- (1) Exclusive (FSCACHE_OP_EXCLUSIVE). Operations of this type may not run in conjunction with any other operation on the object being operated upon.

An example of this is the attribute change operation, in which the file being written to may need truncation.

- (2) Shareable. Operations of this type may be running simultaneously. It's up to the operation implementation to prevent interference between other operations running at the same time.

Procedure

Operations are used through the following procedure:

- (1) The submitting thread must allocate the operation and initialise it itself. Normally this would be part of a more specific structure with the generic op embedded within.
- (2) The submitting thread must then submit the operation for processing using one of the following two functions:

```
int fscache_submit_op(struct fscache_object *object,
                    struct fscache_operation *op);

int fscache_submit_exclusive_op(struct fscache_object *object,
                                struct fscache_operation *op);
```

The first function should be used to submit non-exclusive ops and the second to submit exclusive ones. The caller must still set the FSCACHE_OP_EXCLUSIVE flag.

If successful, both functions will assign the operation to the specified object and return 0. -ENOBUFFS will be returned if the object specified is permanently unavailable.

The operation manager will defer operations on an object that is still undergoing lookup or creation. The operation will also be deferred if an operation of conflicting exclusivity is in progress on the object.

If the operation is asynchronous, the manager will retain a reference to it, so the caller should put their reference to it by passing it to:

```
void fscache_put_operation(struct fscache_operation *op);
```

- (3) If the submitting thread wants to do the work itself, and has marked the operation with FSCACHE_OP_MYTHREAD, then it should monitor FSCACHE_OP_WAITING as described above and check the state of the object if necessary (the object might have died while the thread was waiting).

When it has finished doing its processing, it should call fscache_op_complete() and fscache_put_operation() on it.

- (4) The operation holds an effective lock upon the object, preventing other exclusive ops conflicting until it is released. The operation can be enqueued for further immediate asynchronous processing by adjusting the CPU time provisioning option if necessary, eg:

```
op->flags &= ~FSCACHE_OP_TYPE;
op->flags |= ~FSCACHE_OP_FAST;
```

and calling:

```
void fscache_enqueue_operation(struct fscache_operation *op)
```

This can be used to allow other things to have use of the worker thread pools.

Asynchronous Callback

When used in asynchronous mode, the worker thread pool will invoke the processor method with a pointer to the operation. This should then get at the container struct by using `container_of()`:

```
static void fscache_write_op(struct fscache_operation *_op)
{
    struct fscache_storage *op =
        container_of(_op, struct fscache_storage, op);
    ...
}
```

The caller holds a reference on the operation, and will invoke `fscache_put_operation()` when the processor function returns. The processor function is at liberty to call `fscache_enqueue_operation()` or to take extra references.

1.22 Changes since 2.5.0:

—

recommended

New helpers: `sb_bread()`, `sb_getblk()`, `sb_find_get_block()`, `set_bh()`, `sb_set_blocksize()` and `sb_min_blocksize()`.

Use them.

(`sb_find_get_block()` replaces 2.4' s `get_hash_table()`)

—

recommended

New methods: `->alloc_inode()` and `->destroy_inode()`.

Remove `inode->u.foo_inode_i`

Declare:

```
struct foo_inode_info {
    /* fs-private stuff */
    struct inode vfs_inode;
};
static inline struct foo_inode_info *F00_I(struct inode *inode)
```

(continues on next page)

(continued from previous page)

```
{
    return list_entry(inode, struct foo_inode_info, vfs_inode);
}
```

Use `FOO_I(inode)` instead of `&inode->u.foo_inode_i;`

Add `foo_alloc_inode()` and `foo_destroy_inode()` - the former should allocate `foo_inode_info` and return the address of `->vfs_inode`, the latter should free `FOO_I(inode)` (see in-tree filesystems for examples).

Make them `->alloc_inode` and `->destroy_inode` in your `super_operations`.

Keep in mind that now you need explicit initialization of private data typically between calling `iget_locked()` and unlocking the inode.

At some point that will become mandatory.

—

mandatory

Change of `file_system_type` method (`->read_super` to `->get_sb`)

`->read_super()` is no more. Ditto for `DECLARE_FSTYPE` and `DECLARE_FSTYPE_DEV`.

Turn your `foo_read_super()` into a function that would return 0 in case of success and negative number in case of error (`-EINVAL` unless you have more informative error value to report). Call it `foo_fill_super()`. Now declare:

```
int foo_get_sb(struct file_system_type *fs_type,
              int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
    return get_sb_bdev(fs_type, flags, dev_name, data, foo_fill_super,
                      mnt);
}
```

(or similar with `s/bdev/nodev/` or `s/bdev/single/`, depending on the kind of filesystem).

Replace `DECLARE_FSTYPE` with explicit initializer and have `->get_sb` set as `foo_get_sb`.

—

mandatory

Locking change: `->s_vfs_rename_sem` is taken only by cross-directory renames. Most likely there is no need to change anything, but if you relied on global exclusion between renames for some internal purpose - you need to change your internal locking. Otherwise exclusion warranties remain the same (i.e. parents and victim are locked, etc.).

—

informational

Now we have the exclusion between `->lookup()` and directory removal (by `->rmdir()` and `->rename()`). If you used to need that exclusion and do it by internal

locking (most of filesystems couldn't care less) - you can relax your locking.

—

mandatory

->lookup(), ->truncate(), ->create(), ->unlink(), ->mknod(), ->mkdir(), ->rmdir(), ->link(), ->lseek(), ->symlink(), ->rename() and ->readdir() are called without BKL now. Grab it on entry, drop upon return - that will guarantee the same locking you used to have. If your method or its parts do not need BKL - better yet, now you can shift lock_kernel() and unlock_kernel() so that they would protect exactly what needs to be protected.

—

mandatory

BKL is also moved from around sb operations. BKL should have been shifted into individual fs sb_op functions. If you don't need it, remove it.

—

informational

check for ->link() target not being a directory is done by callers. Feel free to drop it...

—

informational

->link() callers hold ->i_mutex on the object we are linking to. Some of your problems might be over...

—

mandatory

new file_system_type method - kill_sb(superblock). If you are converting an existing filesystem, set it according to ->fs_flags:

FS_REQUIRES_DEV	-	kill_block_super
FS_LITTER	-	kill_litter_super
neither	-	kill_anon_super

FS_LITTER is gone - just remove it from fs_flags.

—

mandatory

FS_SINGLE is gone (actually, that had happened back when ->get_sb() went in - and hadn't been documented ;-/). Just remove it from fs_flags (and see ->get_sb() entry for other actions).

—

mandatory

->setattr() is called without BKL now. Caller _always_ holds ->i_mutex, so watch for ->i_mutex-grabbing code that might be used by your ->setattr(). Callers of notify_change() need ->i_mutex now.

—

recommended

New `super_block` field `struct export_operations *s_export_op` for explicit support for exporting, e.g. via NFS. The structure is fully documented at its declaration in `include/linux/fs.h`, and in `Documentation/filesystems/nfs/exporting.rst`.

Briefly it allows for the definition of `decode_fh` and `encode_fh` operations to encode and decode filehandles, and allows the filesystem to use a standard helper function for `decode_fh`, and provide file-system specific support for this helper, particularly `get_parent`.

It is planned that this will be required for exporting once the code settles down a bit.

mandatory

`s_export_op` is now required for exporting a filesystem. `isofs`, `ext2`, `ext3`, `resierfs`, `fat` can be used as examples of very different filesystems.

—

mandatory

`iget4()` and the `read_inode2` callback have been superseded by `iget5_locked()` which has the following prototype:

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

‘test’ is an additional function that can be used when the inode number is not sufficient to identify the actual file object. ‘set’ should be a non-blocking function that initializes those parts of a newly created inode to allow the test function to succeed. ‘data’ is passed as an opaque value to both test and set functions.

When the inode has been created by `iget5_locked()`, it will be returned with the `I_NEW` flag set and will still be locked. The filesystem then needs to finalize the initialization. Once the inode is initialized it must be unlocked by calling `unlock_new_inode()`.

The filesystem is responsible for setting (and possibly testing) `i_ino` when appropriate. There is also a simpler `iget_locked` function that just takes the superblock and inode number as arguments and does the test and set for you.

e.g.:

```
inode = iget_locked(sb, ino);
if (inode->i_state & I_NEW) {
    err = read_inode_from_disk(inode);
    if (err < 0) {
        iget_failed(inode);
        return err;
    }
    unlock_new_inode(inode);
}
```

Note that if the process of setting up a new inode fails, then `iget_failed()` should be called on the inode to render it dead, and an appropriate error should be passed back to the caller.

—

recommended

`->getattr()` finally getting used. See instances in `nfs`, `minix`, etc.

—

mandatory

`->revalidate()` is gone. If your filesystem had it - provide `->getattr()` and let it call whatever you had as `->revalidate()` + (for symlinks that had `->revalidate()`) add calls in `->follow_link()/->readlink()`.

—

mandatory

`->d_parent` changes are not protected by BKL anymore. Read access is safe if at least one of the following is true:

- filesystem has no cross-directory rename()
- we know that parent had been locked (e.g. we are looking at `->d_parent` of `->lookup()` argument).
- we are called from `->rename()`.
- the child's `->d_lock` is held

Audit your code and add locking if needed. Notice that any place that is not protected by the conditions above is risky even in the old tree - you had been relying on BKL and that's prone to screwups. Old tree had quite a few holes of that kind - unprotected access to `->d_parent` leading to anything from oops to silent memory corruption.

—

mandatory

`FS_NOMOUNT` is gone. If you use it - just set `SB_NOUSER` in flags (see `rootfs` for one kind of solution and `bdev/socket/pipe` for another).

—

recommended

Use `bdev_read_only(bdev)` instead of `is_read_only(kdev)`. The latter is still alive, but only because of the mess in `drivers/s390/block/dasd.c`. As soon as it gets fixed `is_read_only()` will die.

—

mandatory

`->permission()` is called without BKL now. Grab it on entry, drop upon return - that will guarantee the same locking you used to have. If your method or its parts do not need BKL - better yet, now you can shift `lock_kernel()` and `unlock_kernel()` so that they would protect exactly what needs to be protected.

—

mandatory

->stats() is now called without BKL held. BKL should have been shifted into individual fs sb_op functions where it's not clear that it's safe to remove it. If you don't need it, remove it.

—

mandatory

is_read_only() is gone; use bdev_read_only() instead.

—

mandatory

destroy_buffers() is gone; use invalidate_bdev().

—

mandatory

fsync_dev() is gone; use fsync_bdev(). NOTE: lvm breakage is deliberate; as soon as struct block_device * is propagated in a reasonable way by that code fixing will become trivial; until then nothing can be done.

mandatory

block truncation on error exit from ->write_begin, and ->direct_IO moved from generic methods (block_write_begin, cont_write_begin, nobh_write_begin, blockdev_direct_IO*) to callers. Take a look at ext2_write_failed and callers for an example.

mandatory

->truncate is gone. The whole truncate sequence needs to be implemented in ->setattr, which is now mandatory for filesystems implementing on-disk size changes. Start with a copy of the old inode_setattr and vmtruncate, and the reorder the vmtruncate + foofs_vmtruncate sequence to be in order of zeroing blocks using block_truncate_page or similar helpers, size update and on finally on-disk truncation which should not fail. setattr_prepare (which used to be inode_change_ok) now includes the size checks for ATTR_SIZE and must be called in the beginning of ->setattr unconditionally.

mandatory

->clear_inode() and ->delete_inode() are gone; ->evict_inode() should be used instead. It gets called whenever the inode is evicted, whether it has remaining links or not. Caller does not evict the pagecache or inode-associated metadata buffers; the method has to use truncate_inode_pages_final() to get rid of those. Caller makes sure async writeback cannot be running for the inode while (or after) ->evict_inode() is called.

->drop_inode() returns int now; it's called on final iput() with inode->i_lock held and it returns true if filesystems wants the inode to be dropped. As before, generic_drop_inode() is still the default and it's been updated appropriately. generic_delete_inode() is also alive and it consists simply of return 1. Note that all actual eviction work is done by caller after ->drop_inode() returns.

As before, `clear_inode()` must be called exactly once on each call of `->evict_inode()` (as it used to be for each call of `->delete_inode()`). Unlike before, if you are using inode-associated metadata buffers (i.e. `mark_buffer_dirty_inode()`), it's your responsibility to call `invalidate_inode_buffers()` before `clear_inode()`.

NOTE: checking `i_nlink` in the beginning of `->write_inode()` and bailing out if it's zero is not and never had been enough. Final `unlink()` and `iput()` may happen while the inode is in the middle of `->write_inode()`; e.g. if you blindly free the on-disk inode, you may end up doing that while `->write_inode()` is writing to it.

—

mandatory

`.d_delete()` now only advises the dcache as to whether or not to cache unreferenced dentries, and is now only called when the dentry refcount goes to 0. Even on 0 refcount transition, it must be able to tolerate being called 0, 1, or more times (eg. constant, idempotent).

—

mandatory

`.d_compare()` calling convention and locking rules are significantly changed. Read updated documentation in `Documentation/filesystems/vfs.rst` (and look at examples of other filesystems) for guidance.

—

mandatory

`.d_hash()` calling convention and locking rules are significantly changed. Read updated documentation in `Documentation/filesystems/vfs.rst` (and look at examples of other filesystems) for guidance.

—

mandatory

`dcache_lock` is gone, replaced by fine grained locks. See `fs/dcache.c` for details of what locks to replace `dcache_lock` with in order to protect particular things. Most of the time, a filesystem only needs `->d_lock`, which protects all the dcache state of a given dentry.

—

mandatory

Filesystems must RCU-free their inodes, if they can have been accessed via `rcu-walk` path walk (basically, if the file can have had a path name in the `vfs` namespace).

Even though `i_dentry` and `i_rcu` share storage in a union, we will initialize the former in `inode_init_always()`, so just leave it alone in the callback. It used to be necessary to clean it there, but not anymore (starting at 3.2).

—

recommended

vfs now tries to do path walking in “rcu-walk mode”, which avoids atomic operations and scalability hazards on dentries and inodes (see Documentation/filesystems/path-lookup.txt). `d_hash` and `d_compare` changes (above) are examples of the changes required to support this. For more complex filesystem callbacks, the vfs drops out of rcu-walk mode before the fs call, so no changes are required to the filesystem. However, this is costly and loses the benefits of rcu-walk mode. We will begin to add filesystem callbacks that are rcu-walk aware, shown below. Filesystems should take advantage of this where possible.

—

mandatory

`d_revalidate` is a callback that is made on every path element (if the filesystem provides it), which requires dropping out of rcu-walk mode. This may now be called in rcu-walk mode (`nd->flags & LOOKUP_RCU`). `-ECHILD` should be returned if the filesystem cannot handle rcu-walk. See Documentation/filesystems/vfs.rst for more details.

`permission` is an inode permission check that is called on many or all directory inodes on the way down a path walk (to check for exec permission). It must now be rcu-walk aware (`mask & MAY_NOT_BLOCK`). See Documentation/filesystems/vfs.rst for more details.

—

mandatory

In `->fallocate()` you must check the mode option passed in. If your filesystem does not support hole punching (deallocating space in the middle of a file) you must return `-EOPNOTSUPP` if `FALLOC_FL_PUNCH_HOLE` is set in mode. Currently you can only have `FALLOC_FL_PUNCH_HOLE` with `FALLOC_FL_KEEP_SIZE` set, so the `i_size` should not change when hole punching, even when punching the end of a file off.

—

mandatory

`->get_sb()` is gone. Switch to use of `->mount()`. Typically it's just a matter of switching from calling `get_sb_...` to `mount_...` and changing the function type. If you were doing it manually, just switch from setting `->mnt_root` to some pointer to returning that pointer. On errors return `ERR_PTR(...)`.

—

mandatory

`->permission()` and `generic_permission()` have lost flags argument; instead of passing `IPERM_FLAG_RCU` we add `MAY_NOT_BLOCK` into mask.

`generic_permission()` has also lost the `check_acl` argument; ACL checking has been taken to VFS and filesystems need to provide a non-NULL `->i_op->get_acl` to read an ACL from disk.

—

mandatory

If you implement your own `->llseek()` you must handle `SEEK_HOLE` and `SEEK_DATA`. You can handle this by returning `-EINVAL`, but it would be nicer to support it in some way. The generic handler assumes that the entire file is data and there is a virtual hole at the end of the file. So if the provided offset is less than `i_size` and `SEEK_DATA` is specified, return the same offset. If the above is true for the offset and you are given `SEEK_HOLE`, return the end of the file. If the offset is `i_size` or greater return `-ENXIO` in either case.

mandatory

If you have your own `->fsync()` you must make sure to call `filemap_write_and_wait_range()` so that all dirty pages are synced out properly. You must also keep in mind that `->fsync()` is not called with `i_mutex` held anymore, so if you require `i_mutex` locking you must make sure to take it and release it yourself.

—

mandatory

`d_alloc_root()` is gone, along with a lot of bugs caused by code misusing it. Replacement: `d_make_root(inode)`. On success `d_make_root(inode)` allocates and returns a new dentry instantiated with the passed in inode. On failure `NULL` is returned and the passed in inode is dropped so the reference to inode is consumed in all cases and failure handling need not do any cleanup for the inode. If `d_make_root(inode)` is passed a `NULL` inode it returns `NULL` and also requires no further error handling. Typical usage is:

```
inode = foofs_new_inode(...);
s->s_root = d_make_root(inode);
if (!s->s_root)
    /* Nothing needed for the inode cleanup */
    return -ENOMEM;
...
```

—

mandatory

The witch is dead! Well, 2/3 of it, anyway. `->d_revalidate()` and `->lookup()` do not take struct `nameidata` anymore; just the flags.

—

mandatory

`->create()` doesn't take struct `nameidata *`; unlike the previous two, it gets "is it an `O_EXCL` or equivalent?" boolean argument. Note that local filesystems can ignore the argument - they are guaranteed that the object doesn't exist. It's remote/distributed ones that might care...

—

mandatory

`FS_REVAL_DOT` is gone; if you used to have it, add `->d_weak_revalidate()` in your dentry operations instead.

—

mandatory

vfs_readdir() is gone; switch to iterate_dir() instead

—

mandatory

->readdir() is gone now; switch to ->iterate()

mandatory

vfs_follow_link has been removed. Filesystems must use nd_set_link from ->follow_link for normal symlinks, or nd_jump_link for magic /proc/<pid> style links.

—

mandatory

iget5_locked()/ilookup5()/ilookup5_nowait() test() callback used to be called with both ->i_lock and inode_hash_lock held; the former is not taken anymore, so verify that your callbacks do not rely on it (none of the in-tree instances did). inode_hash_lock is still held, of course, so they are still serialized wrt removal from inode hash, as well as wrt set() callback of iget5_locked().

—

mandatory

d_materialise_unique() is gone; d_splice_alias() does everything you need now. Remember that they have opposite orders of arguments ;-/

—

mandatory

f_dentry is gone; use f_path.dentry, or, better yet, see if you can avoid it entirely.

—

mandatory

never call ->read() and ->write() directly; use __vfs_{read,write} or wrappers; instead of checking for ->write or ->read being NULL, look for FMODE_CAN_{WRITE,READ} in file->f_mode.

—

mandatory

do_not_use new_sync_{read,write} for ->read/->write; leave it NULL instead.

—

mandatory ->aio_read/->aio_write are gone. Use ->read_iter/->write_iter.

—

recommended

for embedded (“fast”) symlinks just set inode->i_link to wherever the symlink body is and use simple_follow_link() as ->follow_link().

—

mandatory

calling conventions for `->follow_link()` have changed. Instead of returning cookie and using `nd_set_link()` to store the body to traverse, we return the body to traverse and store the cookie using explicit void `**` argument. `nameidata` isn't passed at all - `nd_jump_link()` doesn't need it and `nd_[gs]et_link()` is gone.

—

mandatory

calling conventions for `->put_link()` have changed. It gets `inode` instead of `dentry`, it does not get `nameidata` at all and it gets called only when `cookie` is non-NULL. Note that `link` body isn't available anymore, so if you need it, store it as `cookie`.

—

mandatory

any symlink that might use `page_follow_link_light/page_put_link()` must have `inode_nohighmem(inode)` called before anything might start playing with its page-cache. No `highmem` pages should end up in the pagecache of such symlinks. That includes any preseeding that might be done during symlink creation. `_page_symlink()` will honour the mapping `gfp` flags, so once you've done `inode_nohighmem()` it's safe to use, but if you allocate and insert the page manually, make sure to use the right `gfp` flags.

—

mandatory

`->follow_link()` is replaced with `->get_link()`; same API, except that

- `->get_link()` gets `inode` as a separate argument
- `->get_link()` may be called in RCU mode - in that case NULL `dentry` is passed

—

mandatory

`->get_link()` gets `struct delayed_call *done` now, and should do `set_delayed_call()` where it used to set `*cookie`.

`->put_link()` is gone - just give the destructor to `set_delayed_call()` in `->get_link()`.

—

mandatory

`->getxattr()` and `xattr_handler.get()` get `dentry` and `inode` passed separately. `dentry` might be yet to be attached to `inode`, so do `_not_` use its `->d_inode` in the instances. Rationale: `!@#!@# security_d_instantiate()` needs to be called before we attach `dentry` to `inode`.

—

mandatory

symlinks are no longer the only inodes that do not have `i_bdev/i_cdev/i_pipe/i_link` union zeroed out at `inode` eviction. As the result, you can't assume that non-NULL

value in `->i_nlink` at `->destroy_inode()` implies that it's a symlink. Checking `->i_mode` is really needed now. In-tree we had to fix `shmem_destroy_callback()` that used to take that kind of shortcut; watch out, since that shortcut is no longer valid.

—

mandatory

`->i_mutex` is replaced with `->i_rwsem` now. `inode_lock()` et.al. work as they used to - they just take it exclusive. However, `->lookup()` may be called with parent locked shared. Its instances must not

- use `d_instantiate()` and `d_rehash()` separately - use `d_add()` or `d_splice_alias()` instead.
- use `d_rehash()` alone - call `d_add(new_dentry, NULL)` instead.
- in the unlikely case when (read-only) access to filesystem data structures needs exclusion for some reason, arrange it yourself. None of the in-tree filesystems needed that.
- rely on `->d_parent` and `->d_name` not changing after dentry has been fed to `d_add()` or `d_splice_alias()`. Again, none of the in-tree instances relied upon that.

We are guaranteed that lookups of the same name in the same directory will not happen in parallel ("same" in the sense of your `->d_compare()`). Lookups on different names in the same directory can and do happen in parallel now.

—

recommended

`->iterate_shared()` is added; it's a parallel variant of `->iterate()`. Exclusion on struct file level is still provided (as well as that between it and `lseek` on the same struct file), but if your directory has been opened several times, you can get these called in parallel. Exclusion between that method and all directory-modifying ones is still provided, of course.

Often enough `->iterate()` can serve as `->iterate_shared()` without any changes - it is a read-only operation, after all. If you have any per-inode or per-dentry in-core data structures modified by `->iterate()`, you might need something to serialize the access to them. If you do dcache pre-seeding, you'll need to switch to `d_alloc_parallel()` for that; look for in-tree examples.

Old method is only used if the new one is absent; eventually it will be removed. Switch while you still can; the old one won't stay.

—

mandatory

`->atomic_open()` calls without `O_CREAT` may happen in parallel.

—

mandatory

`->setxattr()` and `xattr_handler.set()` get dentry and inode passed separately. dentry might be yet to be attached to inode, so do `_not_` use its `->d_inode` in the instances.

Rationale: `security_d_instantiate()` needs to be called before we attach dentry to inode and `smack ->d_instantiate()` uses not just `->getxattr()` but `->setxattr()` as well.

—

mandatory

`->d_compare()` doesn't get parent as a separate argument anymore. If you used it for finding the struct super_block involved, `dentry->d_sb` will work just as well; if it's something more complicated, use `dentry->d_parent`. Just be careful not to assume that fetching it more than once will yield the same value - in RCU mode it could change under you.

—

mandatory

`->rename()` has an added flags argument. Any flags not handled by the filesystem should result in `EINVAL` being returned.

—

recommended

`->readlink` is optional for symlinks. Don't set, unless filesystem needs to fake something for `readlink(2)`.

—

mandatory

`->getattr()` is now passed a struct path rather than a `vfsmount` and dentry separately, and it now has `request_mask` and `query_flags` arguments to specify the fields and sync type requested by `statx`. Filesystems not supporting any `statx`-specific features may ignore the new arguments.

—

mandatory

`->atomic_open()` calling conventions have changed. Gone is `int *opened`, along with `FILE_OPENED/FILE_CREATED`. In place of those we have `FMODE_OPENED/FMODE_CREATED`, set in `file->f_mode`. Additionally, return value for 'called `finish_no_open()`, open it yourself' case has become 0, not 1. Since `finish_no_open()` itself is returning 0 now, that part does not need any changes in `->atomic_open()` instances.

—

mandatory

`alloc_file()` has become static now; two wrappers are to be used instead. `alloc_file_pseudo(inode, vfsmount, name, flags, ops)` is for the cases when dentry needs to be created; that's the majority of old `alloc_file()` users. Calling conventions: on success a reference to new struct file is returned and callers reference to inode is subsumed by that. On failure, `ERR_PTR()` is returned and no caller's references are affected, so the caller needs to drop the inode reference it held. `alloc_file_clone(file, flags, ops)` does not affect any caller's references. On success

you get a new struct file sharing the mount/dentry with the original, on failure - ERR_PTR().

—

mandatory

->clone_file_range() and ->dedupe_file_range have been replaced with ->remap_file_range(). See Documentation/filesystems/vfs.rst for more information.

—

recommended

->lookup() instances doing an equivalent of:

```
if (IS_ERR(inode))
    return ERR_CAST(inode);
return d_splice_alias(inode, dentry);
```

don't need to bother with the check - d_splice_alias() will do the right thing when given ERR_PTR(...) as inode. Moreover, passing NULL inode to d_splice_alias() will also do the right thing (equivalent of d_add(dentry, NULL); return NULL;), so that kind of special cases also doesn't need a separate treatment.

—

strongly recommended

take the RCU-delayed parts of ->destroy_inode() into a new method ->free_inode(). If ->destroy_inode() becomes empty - all the better, just get rid of it. Synchronous work (e.g. the stuff that can't be done from an RCU callback, or any WARN_ON() where we want the stack trace) might be movable to ->evict_inode(); however, that goes only for the things that are not needed to balance something done by ->alloc_inode(). IOW, if it's cleaning up the stuff that might have accumulated over the life of in-core inode, ->evict_inode() might be a fit.

Rules for inode destruction:

- if ->destroy_inode() is non-NULL, it gets called
- if ->free_inode() is non-NULL, it gets scheduled by call_rcu()
- combination of NULL ->destroy_inode and NULL ->free_inode is treated as NULL/free_inode_nonrcu, to preserve the compatibility.

Note that the callback (be it via ->free_inode() or explicit call_rcu() in ->destroy_inode()) is NOT ordered wrt superblock destruction; as the matter of fact, the superblock and all associated structures might be already gone. The filesystem driver is guaranteed to be still there, but that's it. Freeing memory in the callback is fine; doing more than that is possible, but requires a lot of care and is best avoided.

—

mandatory

DCACHE_RCUACCESS is gone; having an RCU delay on dentry freeing is the default. DCACHE_NORCU opts out, and only `d_alloc_pseudo()` has any business doing so.

—

mandatory

`d_alloc_pseudo()` is internal-only; uses outside of `alloc_file_pseudo()` are very suspect (and won't work in modules). Such uses are very likely to be misspelled `d_alloc_anon()`.

—

mandatory

[should've been added in 2016] stale comment in `finish_open()` notwithstanding, failure exits in `->atomic_open()` instances should NOT `fput()` the file, no matter what. Everything is handled by the caller.

—

mandatory

`clone_private_mount()` returns a longterm mount now, so the proper destructor of its result is `kern_unmount()` or `kern_unmount_array()`.

FILESYSTEM SUPPORT LAYERS

Documentation for the support code within the filesystem layer for use in filesystem implementations.

2.1 The Linux Journalling API

2.1.1 Overview

Details

The journalling layer is easy to use. You need to first of all create a `journal_t` data structure. There are two calls to do this dependent on how you decide to allocate the physical media on which the journal resides. The `jbd2_journal_init_inode()` call is for journals stored in filesystem inodes, or the `jbd2_journal_init_dev()` call can be used for journal stored on a raw device (in a continuous range of blocks). A `journal_t` is a typedef for a struct pointer, so when you are finally finished make sure you call `jbd2_journal_destroy()` on it to free up any used kernel memory.

Once you have got your `journal_t` object you need to ‘mount’ or load the journal file. The journalling layer expects the space for the journal was already allocated and initialized properly by the userspace tools. When loading the journal you must call `jbd2_journal_load()` to process journal contents. If the client file system detects the journal contents does not need to be processed (or even need not have valid contents), it may call `jbd2_journal_wipe()` to clear the journal contents before calling `jbd2_journal_load()`.

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

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

You still need to actually journal your filesystem changes, this is done by wrapping them into transactions. Additionally you also need to wrap the modification of each of the buffers with calls to the journal layer, so it knows what the modifications you are actually making are. To do this use `jbd2_journal_start()` which returns a transaction handle.

`jbd2_journal_start()` and its counterpart `jbd2_journal_stop()`, which indicates the end of a transaction are nestable calls, so you can reenter a transaction if necessary, but remember you must call `jbd2_journal_stop()` the same number of times as `jbd2_journal_start()` before the transaction is completed (or more accurately leaves the update phase). Ext4/VFS makes use of this feature to simplify handling of inode dirtying, quota support, etc.

Inside each transaction you need to wrap the modifications to the individual buffers (blocks). Before you start to modify a buffer you need to call `jbd2_journal_get_create_access()` / `jbd2_journal_get_write_access()` / `jbd2_journal_get_undo_access()` as appropriate, this allows the journalling layer to copy the unmodified data if it needs to. After all the buffer may be part of a previously uncommitted transaction. At this point you are at last ready to modify a buffer, and once you are have done so you need to call `jbd2_journal_dirty_metadata()`. Or if you've asked for access to a buffer you now know is now longer required to be pushed back on the device you can call `jbd2_journal_forget()` in much the same way as you might have used `bforget()` in the past.

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

Then at umount time , in your `put_super()` you can then call `jbd2_journal_destroy()` to clean up your in-core journal object.

Unfortunately there a couple of ways the journal layer can cause a deadlock. The first thing to note is that each task can only have a single outstanding transaction at any one time, remember nothing commits until the outermost `jbd2_journal_stop()`. This means you must complete the transaction at the end of each file/inode/address etc. operation you perform, so that the journalling system isn't re-entered on another journal. Since transactions can't be nested/batched across differing journals, and another filesystem other than yours (say ext4) may be modified in a later syscall.

The second case to bear in mind is that `jbd2_journal_start()` can block if there isn't enough space in the journal for your transaction (based on the passed `nblocks` param) - when it blocks it merely(!) needs to wait for transactions to complete and be committed from other tasks, so essentially we are waiting for `jbd2_journal_stop()`. So to avoid deadlocks you must treat `jbd2_journal_start()` / `jbd2_journal_stop()` as if they were semaphores and include them in your semaphore ordering rules to prevent deadlocks. Note that `jbd2_journal_extend()` has similar blocking behaviour to `jbd2_journal_start()` so you can deadlock here just as easily as on `jbd2_journal_start()`.

Try to reserve the right number of blocks the first time. ;-). This will be the maximum number of blocks you are going to touch in this transaction. I advise having a look at at least `ext4_jbd.h` to see the basis on which ext4 uses to make these decisions.

Another wriggle to watch out for is your on-disk block allocation strategy. Why? Because, if you do a delete, you need to ensure you haven't reused any of the freed blocks until the transaction freeing these blocks commits. If you reused these blocks and crash happens, there is no way to restore the contents of the reallocated blocks at the end of the last fully committed transaction. One simple

way of doing this is to mark blocks as free in internal in-memory block allocation structures only after the transaction freeing them commits. Ext4 uses journal commit callback for this purpose.

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

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

```
jbd2_journal_lock_updates() //stop new stuff happening..
jbd2_journal_flush()       // checkpoint everything.
..do stuff on stable fs
jbd2_journal_unlock_updates() // carry on with filesystem use.
```

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

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.

2.1.2 Data Types

The journalling layer uses typedefs to ‘hide’ the concrete definitions of the structures used. As a client of the JBD2 layer you can just rely on the using the pointer as a magic cookie of some sort. Obviously the hiding is not enforced as this is ‘C’.

Structures

typedef **handle_t**

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

Description

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

The buffer credits field is used to account for journaled buffers being modified by the running process. To ensure that there is enough log space for all outstanding operations, we need to limit the number of outstanding buffers possible at any time. When the operation completes, any buffer credits not used are credited back

to the transaction, so that at all times we know how many buffers the outstanding updates on a transaction might possibly touch.

This is an opaque datatype.

typedef **journal_t**

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

Description

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

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

This is an opaque datatype.

struct **jbd2_inode**

The `jbd_inode` type is the structure linking inodes in ordered mode present in a transaction so that we can sync them during commit.

Definition

```
struct jbd2_inode {
    transaction_t *i_transaction;
    transaction_t *i_next_transaction;
    struct list_head i_list;
    struct inode *i_vfs_inode;
    unsigned long i_flags;
    loff_t i_dirty_start;
    loff_t i_dirty_end;
};
```

Members

i_transaction Which transaction does this inode belong to? Either the running transaction or the committing one. [`j_list_lock`]

i_next_transaction Pointer to the running transaction modifying inode's data in case there is already a committing transaction touching it. [`j_list_lock`]

i_list List of inodes in the `i_transaction` [`j_list_lock`]

i_vfs_inode VFS inode this inode belongs to [constant for lifetime of structure]

i_flags Flags of inode [`j_list_lock`]

i_dirty_start Offset in bytes where the dirty range for this inode starts. [`j_list_lock`]

i_dirty_end Inclusive offset in bytes where the dirty range for this inode ends. [`j_list_lock`]

struct **jbd2_journal_handle**

The `handle_s` type is the concrete type associated with `handle_t`.

Definition

```

struct jbd2_journal_handle {
    union {
        transaction_t *h_transaction;
        journal_t *h_journal;
    };
    handle_t *h_rsv_handle;
    int h_total_credits;
    int h_revoke_credits;
    int h_revoke_credits_requested;
    int h_ref;
    int h_err;
    unsigned int    h_sync:        1;
    unsigned int    h_jdata:       1;
    unsigned int    h_reserved:    1;
    unsigned int    h_aborted:     1;
    unsigned int    h_type:        8;
    unsigned int    h_line_no:     16;
    unsigned long   h_start_jiffies;
    unsigned int    h_requested_credits;
    unsigned int    saved_alloc_context;
};

```

Members

{unnamed_union} anonymous

h_transaction Which compound transaction is this update a part of?

h_journal Which journal handle belongs to - used iff **h_reserved** set.

h_rsv_handle Handle reserved for finishing the logical operation.

h_total_credits Number of remaining buffers we are allowed to add to journal.
These are dirty buffers and revoke descriptor blocks.

h_revoke_credits Number of remaining revoke records available for handle

h_revoke_credits_requested Holds **h_revoke_credits** after handle is started.

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_reserved Flag for handle for reserved credits.

h_aborted Flag indicating fatal error on handle.

h_type For handle statistics.

h_line_no For handle statistics.

h_start_jiffies Handle Start time.

h_requested_credits Holds **h_total_credits** after handle is started.

saved_alloc_context Saved context while transaction is open.

struct **journal_s**

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

Definition

```
struct journal_s {
    unsigned long          j_flags;
    int j_errno;
    struct mutex           j_abort_mutex;
    struct buffer_head     *j_sb_buffer;
    journal_superblock_t *j_superblock;
    int j_format_version;
    rwlock_t j_state_lock;
    int j_barrier_count;
    struct mutex           j_barrier;
    transaction_t *j_running_transaction;
    transaction_t *j_committing_transaction;
    transaction_t *j_checkpoint_transactions;
    wait_queue_head_t j_wait_transaction_locked;
    wait_queue_head_t j_wait_done_commit;
    wait_queue_head_t j_wait_commit;
    wait_queue_head_t j_wait_updates;
    wait_queue_head_t j_wait_reserved;
    struct mutex           j_checkpoint_mutex;
    struct buffer_head     *j_chkpt_bhs[JBD2_NR_BATCH];
    unsigned long          j_head;
    unsigned long          j_tail;
    unsigned long          j_free;
    unsigned long          j_first;
    unsigned long          j_last;
    struct block_device    *j_dev;
    int j_blocksize;
    unsigned long long     j_blk_offset;
    char j_devname[BDEVNAME_SIZE+24];
    struct block_device    *j_fs_dev;
    unsigned int           j_maxlen;
    atomic_t j_reserved_credits;
    spinlock_t j_list_lock;
    struct inode           *j_inode;
    tid_t j_tail_sequence;
    tid_t j_transaction_sequence;
    tid_t j_commit_sequence;
    tid_t j_commit_request;
    __u8 j_uuid[16];
    struct task_struct     *j_task;
    int j_max_transaction_buffers;
    int j_revoke_records_per_block;
    unsigned long          j_commit_interval;
    struct timer_list      j_commit_timer;
    spinlock_t j_revoke_lock;
    struct jbd2_revoke_table_s *j_revoke;
    struct jbd2_revoke_table_s *j_revoke_table[2];
    struct buffer_head     **j_wbuf;
    int j_wbufsize;
    pid_t j_last_sync_writer;
    u64 j_average_commit_time;
    u32 j_min_batch_time;
    u32 j_max_batch_time;
    void (*j_commit_callback)(journal_t *, transaction_t *);
    spinlock_t j_history_lock;
```

(continues on next page)

(continued from previous page)

```

struct proc_dir_entry  *j_proc_entry;
struct transaction_stats_s j_stats;
unsigned int           j_failed_commit;
void *j_private;
struct crypto_shash *j_chksum_driver;
__u32 j_csum_seed;
#ifdef CONFIG_DEBUG_LOCK_ALLOC;
struct lockdep_map     j_trans_commit_map;
#endif;
};

```

Members

- j_flags** General journaling state flags [j_state_lock]
- j_errno** Is there an outstanding uncleared error on the journal (from a prior abort)? [j_state_lock]
- j_abort_mutex** Lock the whole aborting procedure.
- j_sb_buffer** The first part of the superblock buffer.
- j_superblock** The second part of the 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_state_lock]
- j_barrier** The barrier lock itself.
- j_running_transaction** Transactions: The current running transaction... [j_state_lock] [caller holding open handle]
- j_committing_transaction** the transaction we are pushing to disk [j_state_lock] [caller holding open handle]
- j_checkpoint_transactions** ...and a linked circular list of all transactions waiting for checkpointing. [j_list_lock]
- j_wait_transaction_locked** Wait queue for waiting for a locked transaction to start committing, or for a barrier lock to be released.
- j_wait_done_commit** Wait queue for waiting for commit to complete.
- j_wait_commit** Wait queue to trigger commit.
- j_wait_updates** Wait queue to wait for updates to complete.
- j_wait_reserved** Wait queue to wait for reserved buffer credits to drop.
- j_checkpoint_mutex** Semaphore for locking against concurrent checkpoints.
- j_chkpt_bhs** List of buffer heads used by the checkpoint routine. This was moved from jbd2_log_do_checkpoint() to reduce stack usage. Access to this array is controlled by the **j_checkpoint_mutex**. [j_checkpoint_mutex]
- j_head** Journal head: identifies the first unused block in the journal. [j_state_lock]

- j_tail** Journal tail: identifies the oldest still-used block in the journal. [j_state_lock]
- j_free** Journal free: how many free blocks are there in the journal? [j_state_lock]
- j_first** The block number of the first usable block in the journal [j_state_lock].
- j_last** The block number one beyond the last usable block in the journal [j_state_lock].
- j_dev** Device where we store the journal.
- j_blocksize** Block size for the location where we store the journal.
- j_blk_offset** Starting block offset into the device where we store the journal.
- j_devname** Journal device name.
- j_fs_dev** Device which holds the client fs. For internal journal this will be equal to j_dev.
- j_maxlen** Total maximum capacity of the journal region on disk.
- j_reserved_credits** Number of buffers reserved from the running transaction.
- j_list_lock** Protects the buffer lists and internal buffer state.
- j_inode** Optional inode where we store the journal. If present, all journal block numbers are mapped into this inode via `bmap()`.
- j_tail_sequence** Sequence number of the oldest transaction in the log [j_state_lock]
- j_transaction_sequence** Sequence number of the next transaction to grant [j_state_lock]
- j_commit_sequence** Sequence number of the most recently committed transaction [j_state_lock].
- j_commit_request** Sequence number of the most recent transaction wanting commit [j_state_lock]
- j_uuid** Journal uuid: identifies the object (filesystem, LVM volume etc) backed by this journal. This will eventually be replaced by an array of uuids, allowing us to index multiple devices within a single journal and to perform atomic updates across them.
- 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_revoke_records_per_block** Number of revoke records that fit in one descriptor block.
- j_commit_interval** What is the maximum transaction lifetime before we begin a commit?
- j_commit_timer** The timer used to wakeup the commit thread.
- j_revoke_lock** Protect the revoke table.

- j_revoke** The revoke table - maintains the list of revoked blocks in the current transaction.
- j_revoke_table** Alternate revoke tables for `j_revoke`.
- j_wbuf** Array of bhs for `jbd2_journal_commit_transaction`.
- j_wbufsize** Size of `j_wbuf` array.
- j_last_sync_writer** The pid of the last person to run a synchronous operation through the journal.
- j_average_commit_time** The average amount of time in nanoseconds it takes to commit a transaction to disk. [`j_state_lock`]
- j_min_batch_time** Minimum time that we should wait for additional filesystem operations to get batched into a synchronous handle in microseconds.
- j_max_batch_time** Maximum time that we should wait for additional filesystem operations to get batched into a synchronous handle in microseconds.
- j_commit_callback** This function is called when a transaction is closed.
- j_history_lock** Protect the transactions statistics history.
- j_proc_entry** `procfs` entry for the `jbd` statistics directory.
- j_stats** Overall statistics.
- j_failed_commit** Failed journal commit ID.
- j_private** An opaque pointer to fs-private information. `ext3` puts its superblock pointer here.
- j_chksum_driver** Reference to checksum algorithm driver via `cryptoapi`.
- j_csum_seed** Precomputed journal UUID checksum for seeding other checksums.
- j_trans_commit_map** Lockdep entity to track transaction commit dependencies. Handles hold this “lock” for read, when we wait for commit, we acquire the “lock” for writing. This matches the properties of `jbd2` journalling where the running transaction has to wait for all handles to be dropped to commit that transaction and also acquiring a handle may require transaction commit to finish.

2.1.3 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

Journal Level

int **jbd2_journal_force_commit_nested**(journal_t * journal)

Parameters

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

Description

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

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

Parameters

journal_t * journal journal to force

Description

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

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

Parameters

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

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

unsigned long long start Block nr Start of journal.

int len Length of the journal in blocks.

int blocksize blocksize of journalling device

Return

a newly created journal_t *

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

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

Parameters

struct inode * inode An inode to create the journal in

Description

jbd2_journal_init_inode creates a journal which maps an on-disk inode as the journal. The inode must exist already, must support bmap() and must have all data blocks preallocated.

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

Parameters

journal_t * journal The journal to update.

Description

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

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

Parameters

journal_t * journal Journal to act on.

Description

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

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

Parameters

journal_t * journal Journal to act on.

Description

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

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

Parameters

journal_t * journal Journal to check.

unsigned long compat bitmask of compatible features

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

unsigned long incompat bitmask of incompatible features

Description

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

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

Parameters

journal_t * journal Journal to check.

unsigned long compat bitmask of compatible features

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

unsigned long incompat bitmask of incompatible features

Description

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

```
int jbd2_journal_set_features(journal_t * journal, unsigned
                             long compat, unsigned long ro, unsigned
                             long incompat)
```

Mark a given journal feature in the superblock

Parameters

journal_t * journal Journal to act on.

unsigned long compat bitmask of compatible features

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

unsigned long incompat bitmask of incompatible features

Description

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

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

Parameters

journal_t * journal Journal to act on.

Description

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

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

Parameters

journal_t * journal Journal to act on.

int write flag (see below)

Description

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

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

void **jbd2_journal_abort**(journal_t * journal, int errno)

Shutdown the journal immediately.

Parameters

journal_t * journal the journal to shutdown.

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

Description

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

The `jbd2_journal_abort` function is intended to support higher level error recovery mechanisms such as the `ext2/ext3` remount-readonly error mode.

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

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

Any attempt to get a new transaction handle on a journal which is in ABORT state will just result in an `-EROFS` error return. A `jbd2_journal_stop` on an existing handle will return `-EIO` if we have entered abort state during the update.

Recursive transactions are not disturbed by journal abort until the final `jbd2_journal_stop`, which will receive the `-EIO` error.

Finally, the `jbd2_journal_abort` call allows the caller to supply an `errno` which will be recorded (if possible) in the journal superblock. This allows a client to record failure conditions in the middle of a transaction without having to complete the transaction to record the failure to disk. `ext3_error`, for example, now uses this functionality.

int **jbd2_journal_errno**(journal_t * journal)

returns the journal' s error state.

Parameters

journal_t * journal journal to examine.

Description

This is the `errno` number set with `jbd2_journal_abort()`, the last time the journal was mounted - if the journal was stopped without calling abort this will be 0.

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

int **jbd2_journal_clear_err**(journal_t * journal)

clears the journal' s error state

Parameters

journal_t * journal journal to act on.

Description

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

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

Parameters

journal_t * journal journal to act on.

Description

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

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

Parameters

journal_t * journal the journal to recover

Description

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

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

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

Parameters

journal_t * journal journal to startup

Description

Locate any valid recovery information from the journal and set up the journal structures in memory to ignore it (presumably because the caller has evidence that it is out of date). This function doesn't appear to be exported..

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

Transaction Level

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

Parameters

journal_t * journal Journal to start transaction on.

int nblocks number of block buffer we might modify

Description

We make sure that the transaction can guarantee at least `nblocks` of modified buffers in the log. We block until the log can guarantee that much space. Additionally, if `rsv_blocks > 0`, we also create another handle with `rsv_blocks` reserved blocks in the journal. This handle is stored in `h_rsv_handle`. It is not attached to any particular transaction and thus doesn't block transaction commit. If the caller uses this reserved handle, it has to set `h_rsv_handle` to `NULL` as otherwise `jbd2_journal_stop()` on the parent handle will dispose the reserved one. Reserved handle has to be converted to a normal handle using `jbd2_journal_start_reserved()` before it can be used.

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

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

Parameters

handle_t * handle handle to start

unsigned int type for handle statistics

unsigned int line_no for handle statistics

Description

Start handle that has been previously reserved with `jbd2_journal_reserve()`. This attaches **handle** to the running transaction (or creates one if there's not transaction running). Unlike `jbd2_journal_start()` this function cannot block on journal commit, checkpointing, or similar stuff. It can block on memory allocation or frozen journal though.

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

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

Parameters

handle_t * handle handle to 'extend'

int nblocks nr blocks to try to extend by.

int revoke_records number of revoke records to try to extend by.

Description

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

`jbd2_journal_extend` tries to give the running handle more buffer credits. It does not guarantee that allocation - this is a best-effort only. The calling process **MUST** be able to deal cleanly with a failure to extend here.

Return 0 on success, non-zero on failure.

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

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

Parameters

handle_t * handle handle to restart

int nblocks nr credits requested

int revoke_records number of revoke record credits requested

gfp_t gfp_mask memory allocation flags (for start_this_handle)

Description

Restart a handle for a multi-transaction filesystem operation.

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

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

Parameters

journal_t * journal Journal to establish a barrier on.

Description

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

The journal lock should not be held on entry.

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

Parameters

journal_t * journal Journal to release the barrier on.

Description

Release a transaction barrier obtained with `jbd2_journal_lock_updates()`. Should be called without the journal lock held.

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

Parameters

handle_t * handle transaction to add buffer modifications to

struct buffer_head * bh bh to be used for metadata writes

Return

error code or 0 on success.

Description

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

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

Parameters

handle_t * handle transaction to new buffer to

struct buffer_head * bh new buffer.

Description

Call this if you create a new bh.

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

Parameters

handle_t * handle transaction

struct buffer_head * bh buffer to undo

Description

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

To deal with that, `jbd2_journal_get_undo_access` requests write access to a buffer for parts of non-rewindable operations such as delete operations on the bitmaps. The journalling code must keep a copy of the buffer's contents prior to the `undo_access` call until such time as we know that the buffer has definitely been committed to disk.

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

Returns error number or 0 on success.

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

Parameters

struct buffer_head * bh buffer to trigger on

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

Description

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

Call with `NULL` to clear the triggers.

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

Parameters

handle_t * handle transaction to add buffer to.

struct buffer_head * bh buffer to mark

Description

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

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

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

Returns error number or 0 on success.

Special care needs to be taken if the buffer already belongs to the current committing transaction (in which case we should have frozen data present for that commit). In that case, we don't relink the buffer: that only gets done when the old transaction finally completes its commit.

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

Parameters

handle_t * handle transaction handle

struct buffer_head * bh bh to 'forget'

Description

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

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

Decrements `bh->b_count` by one.

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

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

Parameters

handle_t * handle transaction to complete.

Description

All done for a particular handle.

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

`jbd2_journal_stop` itself will not usually return an error, but it may do so in unusual circumstances. In particular, expect it to return `-EIO` if a `jbd2_journal_abort` has been executed since the transaction began.

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

Parameters

journal_t * journal journal for operation

struct page * page to try and free

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

Description

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

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

This complicates JBD locking somewhat. We aren't protected by the BKL here. We wish to remove the buffer from its committing or running transaction's `->t_datalist` via `__jbd2_journal_unfile_buffer`.

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

Even worse, someone may be doing a `jbd2_journal_dirty_data` on this buffer. So we need to lock against that. `jbd2_journal_dirty_data()` will come out of the lock with the buffer dirty, which makes it ineligible for release here.

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

Return 0 on failure, 1 on success

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

Parameters

journal_t * journal journal to use for flush...

struct page * page page to flush

unsigned int offset start of the range to invalidate

unsigned int length length of the range to invalidate

Description

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

2.1.4 See also

Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen Tweedie

Ext3 Journalling FileSystem, OLS 2000, Dr. Stephen Tweedie

2.2 Filesystem-level encryption (fscrypt)

2.2.1 Introduction

fscrypt is a library which filesystems can hook into to support transparent encryption of files and directories.

Note: “fscrypt” in this document refers to the kernel-level portion, implemented in `fs/crypto/`, as opposed to the userspace tool `fscrypt`. This document only covers the kernel-level portion. For command-line examples of how to use encryption, see the documentation for the userspace tool `fscrypt`. Also, it is recommended to use the fscrypt userspace tool, or other existing userspace tools such as `fscryptctl` or [Android’s key management system](#), over using the kernel’s API directly. Using existing tools reduces the chance of introducing your own security bugs. (Nevertheless, for completeness this documentation covers the kernel’s API anyway.)

Unlike `dm-crypt`, fscrypt operates at the filesystem level rather than at the block device level. This allows it to encrypt different files with different keys and to have unencrypted files on the same filesystem. This is useful for multi-user systems where each user’s data-at-rest needs to be cryptographically isolated from the others. However, except for filenames, fscrypt does not encrypt filesystem metadata.

Unlike `eCryptfs`, which is a stacked filesystem, fscrypt is integrated directly into supported filesystems —currently ext4, F2FS, and UBIFS. This allows encrypted files to be read and written without caching both the decrypted and encrypted pages in the pagecache, thereby nearly halving the memory used and bringing it in line with unencrypted files. Similarly, half as many dentries and inodes are needed. `eCryptfs` also limits encrypted filenames to 143 bytes, causing application compatibility issues; fscrypt allows the full 255 bytes (`NAME_MAX`). Finally, unlike `eCryptfs`, the fscrypt API can be used by unprivileged users, with no need to mount anything.

fscrypt does not support encrypting files in-place. Instead, it supports marking an empty directory as encrypted. Then, after userspace provides the key, all regular

files, directories, and symbolic links created in that directory tree are transparently encrypted.

2.2.2 Threat model

Offline attacks

Provided that userspace chooses a strong encryption key, fscrypt protects the confidentiality of file contents and filenames in the event of a single point-in-time permanent offline compromise of the block device content. fscrypt does not protect the confidentiality of non-filename metadata, e.g. file sizes, file permissions, file timestamps, and extended attributes. Also, the existence and location of holes (unallocated blocks which logically contain all zeroes) in files is not protected.

fscrypt is not guaranteed to protect confidentiality or authenticity if an attacker is able to manipulate the filesystem offline prior to an authorized user later accessing the filesystem.

Online attacks

fscrypt (and storage encryption in general) can only provide limited protection, if any at all, against online attacks. In detail:

Side-channel attacks

fscrypt is only resistant to side-channel attacks, such as timing or electromagnetic attacks, to the extent that the underlying Linux Cryptographic API algorithms are. If a vulnerable algorithm is used, such as a table-based implementation of AES, it may be possible for an attacker to mount a side channel attack against the online system. Side channel attacks may also be mounted against applications consuming decrypted data.

Unauthorized file access

After an encryption key has been added, fscrypt does not hide the plaintext file contents or filenames from other users on the same system. Instead, existing access control mechanisms such as file mode bits, POSIX ACLs, LSMs, or namespaces should be used for this purpose.

(For the reasoning behind this, understand that while the key is added, the confidentiality of the data, from the perspective of the system itself, is not protected by the mathematical properties of encryption but rather only by the correctness of the kernel. Therefore, any encryption-specific access control checks would merely be enforced by kernel code and therefore would be largely redundant with the wide variety of access control mechanisms already available.)

Kernel memory compromise

An attacker who compromises the system enough to read from arbitrary memory, e.g. by mounting a physical attack or by exploiting a kernel security vulnerability, can compromise all encryption keys that are currently in use.

However, fscrypt allows encryption keys to be removed from the kernel, which may protect them from later compromise.

In more detail, the `FS_IOC_REMOVE_ENCRYPTION_KEY` ioctl (or the `FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS` ioctl) can wipe a master encryption key from kernel memory. If it does so, it will also try to evict all cached inodes which had been “unlocked” using the key, thereby wiping their per-file keys and making them once again appear “locked”, i.e. in ciphertext or encrypted form.

However, these ioctls have some limitations:

- Per-file keys for in-use files will not be removed or wiped. Therefore, for maximum effect, userspace should close the relevant encrypted files and directories before removing a master key, as well as kill any processes whose working directory is in an affected encrypted directory.
- The kernel cannot magically wipe copies of the master key(s) that userspace might have as well. Therefore, userspace must wipe all copies of the master key(s) it makes as well; normally this should be done immediately after `FS_IOC_ADD_ENCRYPTION_KEY`, without waiting for `FS_IOC_REMOVE_ENCRYPTION_KEY`. Naturally, the same also applies to all higher levels in the key hierarchy. Userspace should also follow other security precautions such as `mlock()`ing memory containing keys to prevent it from being swapped out.
- In general, decrypted contents and filenames in the kernel VFS caches are freed but not wiped. Therefore, portions thereof may be recoverable from freed memory, even after the corresponding key(s) were wiped. To partially solve this, you can set `CONFIG_PAGE_POISONING=y` in your kernel config and add `page_poison=1` to your kernel command line. However, this has a performance cost.
- Secret keys might still exist in CPU registers, in crypto accelerator hardware (if used by the crypto API to implement any of the algorithms), or in other places not explicitly considered here.

Limitations of v1 policies

v1 encryption policies have some weaknesses with respect to online attacks:

- There is no verification that the provided master key is correct. Therefore, a malicious user can temporarily associate the wrong key with another user’s encrypted files to which they have read-only access. Because of filesystem caching, the wrong key will then be used by the other user’s accesses to those files, even if the other user has the correct key in their own keyring. This violates the meaning of “read-only access”.

- A compromise of a per-file key also compromises the master key from which it was derived.
- Non-root users cannot securely remove encryption keys.

All the above problems are fixed with v2 encryption policies. For this reason among others, it is recommended to use v2 encryption policies on all new encrypted directories.

2.2.3 Key hierarchy

Master Keys

Each encrypted directory tree is protected by a master key. Master keys can be up to 64 bytes long, and must be at least as long as the greater of the key length needed by the contents and filenames encryption modes being used. For example, if AES-256-XTS is used for contents encryption, the master key must be 64 bytes (512 bits). Note that the XTS mode is defined to require a key twice as long as that required by the underlying block cipher.

To “unlock” an encrypted directory tree, userspace must provide the appropriate master key. There can be any number of master keys, each of which protects any number of directory trees on any number of filesystems.

Master keys must be real cryptographic keys, i.e. indistinguishable from random bytestrings of the same length. This implies that users **must not** directly use a password as a master key, zero-pad a shorter key, or repeat a shorter key. Security cannot be guaranteed if userspace makes any such error, as the cryptographic proofs and analysis would no longer apply.

Instead, users should generate master keys either using a cryptographically secure random number generator, or by using a KDF (Key Derivation Function). The kernel does not do any key stretching; therefore, if userspace derives the key from a low-entropy secret such as a passphrase, it is critical that a KDF designed for this purpose be used, such as scrypt, PBKDF2, or Argon2.

Key derivation function

With one exception, fscrypt never uses the master key(s) for encryption directly. Instead, they are only used as input to a KDF (Key Derivation Function) to derive the actual keys.

The KDF used for a particular master key differs depending on whether the key is used for v1 encryption policies or for v2 encryption policies. Users **must not** use the same key for both v1 and v2 encryption policies. (No real-world attack is currently known on this specific case of key reuse, but its security cannot be guaranteed since the cryptographic proofs and analysis would no longer apply.)

For v1 encryption policies, the KDF only supports deriving per-file encryption keys. It works by encrypting the master key with AES-128-ECB, using the file’s 16-byte nonce as the AES key. The resulting ciphertext is used as the derived key. If the ciphertext is longer than needed, then it is truncated to the needed length.

For v2 encryption policies, the KDF is HKDF-SHA512. The master key is passed as the “input keying material”, no salt is used, and a distinct “application-specific information string” is used for each distinct key to be derived. For example, when a per-file encryption key is derived, the application-specific information string is the file’s nonce prefixed with “fscrypt\0” and a context byte. Different context bytes are used for other types of derived keys.

HKDF-SHA512 is preferred to the original AES-128-ECB based KDF because HKDF is more flexible, is nonreversible, and evenly distributes entropy from the master key. HKDF is also standardized and widely used by other software, whereas the AES-128-ECB based KDF is ad-hoc.

Per-file encryption keys

Since each master key can protect many files, it is necessary to “tweak” the encryption of each file so that the same plaintext in two files doesn’t map to the same ciphertext, or vice versa. In most cases, fscrypt does this by deriving per-file keys. When a new encrypted inode (regular file, directory, or symlink) is created, fscrypt randomly generates a 16-byte nonce and stores it in the inode’s encryption xattr. Then, it uses a KDF (as described in Key derivation function) to derive the file’s key from the master key and nonce.

Key derivation was chosen over key wrapping because wrapped keys would require larger xattrs which would be less likely to fit in-line in the filesystem’s inode table, and there didn’t appear to be any significant advantages to key wrapping. In particular, currently there is no requirement to support unlocking a file with multiple alternative master keys or to support rotating master keys. Instead, the master keys may be wrapped in userspace, e.g. as is done by the `fscrypt` tool.

DIRECT_KEY policies

The Adiantum encryption mode (see Encryption modes and usage) is suitable for both contents and filenames encryption, and it accepts long IVs —long enough to hold both an 8-byte logical block number and a 16-byte per-file nonce. Also, the overhead of each Adiantum key is greater than that of an AES-256-XTS key.

Therefore, to improve performance and save memory, for Adiantum a “direct key” configuration is supported. When the user has enabled this by setting `FSCRYPT_POLICY_FLAG_DIRECT_KEY` in the fscrypt policy, per-file encryption keys are not used. Instead, whenever any data (contents or filenames) is encrypted, the file’s 16-byte nonce is included in the IV. Moreover:

- For v1 encryption policies, the encryption is done directly with the master key. Because of this, users **must not** use the same master key for any other purpose, even for other v1 policies.
- For v2 encryption policies, the encryption is done with a per-mode key derived using the KDF. Users may use the same master key for other v2 encryption policies.

IV_INO_LBLK_64 policies

When `FSCRYPT_POLICY_FLAG_IV_INO_LBLK_64` is set in the `fscrypt` policy, the encryption keys are derived from the master key, encryption mode number, and filesystem UUID. This normally results in all files protected by the same master key sharing a single contents encryption key and a single filenames encryption key. To still encrypt different files' data differently, inode numbers are included in the IVs. Consequently, shrinking the filesystem may not be allowed.

This format is optimized for use with inline encryption hardware compliant with the UFS standard, which supports only 64 IV bits per I/O request and may have only a small number of keyslots.

IV_INO_LBLK_32 policies

`IV_INO_LBLK_32` policies work like `IV_INO_LBLK_64`, except that for `IV_INO_LBLK_32`, the inode number is hashed with SipHash-2-4 (where the SipHash key is derived from the master key) and added to the file logical block number mod 2^{32} to produce a 32-bit IV.

This format is optimized for use with inline encryption hardware compliant with the eMMC v5.2 standard, which supports only 32 IV bits per I/O request and may have only a small number of keyslots. This format results in some level of IV reuse, so it should only be used when necessary due to hardware limitations.

Key identifiers

For master keys used for v2 encryption policies, a unique 16-byte “key identifier” is also derived using the KDF. This value is stored in the clear, since it is needed to reliably identify the key itself.

Dirhash keys

For directories that are indexed using a secret-keyed dirhash over the plaintext filenames, the KDF is also used to derive a 128-bit SipHash-2-4 key per directory in order to hash filenames. This works just like deriving a per-file encryption key, except that a different KDF context is used. Currently, only casefolded (“case-insensitive”) encrypted directories use this style of hashing.

2.2.4 Encryption modes and usage

`fscrypt` allows one encryption mode to be specified for file contents and one encryption mode to be specified for filenames. Different directory trees are permitted to use different encryption modes. Currently, the following pairs of encryption modes are supported:

- AES-256-XTS for contents and AES-256-CTS-CBC for filenames
- AES-128-CBC for contents and AES-128-CTS-CBC for filenames
- Adiantum for both contents and filenames

If unsure, you should use the (AES-256-XTS, AES-256-CTS-CBC) pair.

AES-128-CBC was added only for low-powered embedded devices with crypto accelerators such as CAAM or CESA that do not support XTS. To use AES-128-CBC, `CONFIG_CRYPTOTO_ESSIV` and `CONFIG_CRYPTOTO_SHA256` (or another SHA-256 implementation) must be enabled so that ESSIV can be used.

Adiantum is a (primarily) stream cipher-based mode that is fast even on CPUs without dedicated crypto instructions. It's also a true wide-block mode, unlike XTS. It can also eliminate the need to derive per-file encryption keys. However, it depends on the security of two primitives, XChaCha12 and AES-256, rather than just one. See the paper “Adiantum: length-preserving encryption for entry-level processors” (<https://eprint.iacr.org/2018/720.pdf>) for more details. To use Adiantum, `CONFIG_CRYPTOTO_ADIANTUM` must be enabled. Also, fast implementations of ChaCha and NHPoly1305 should be enabled, e.g. `CONFIG_CRYPTOTO_CHACHA20_NEON` and `CONFIG_CRYPTOTO_NHPOLY1305_NEON` for ARM.

New encryption modes can be added relatively easily, without changes to individual filesystems. However, authenticated encryption (AE) modes are not currently supported because of the difficulty of dealing with ciphertext expansion.

Contents encryption

For file contents, each filesystem block is encrypted independently. Starting from Linux kernel 5.5, encryption of filesystems with block size less than system's page size is supported.

Each block's IV is set to the logical block number within the file as a little endian number, except that:

- With CBC mode encryption, ESSIV is also used. Specifically, each IV is encrypted with AES-256 where the AES-256 key is the SHA-256 hash of the file's data encryption key.
- With `DIRECT_KEY` policies, the file's nonce is appended to the IV. Currently this is only allowed with the Adiantum encryption mode.
- With `IV_INO_LBLK_64` policies, the logical block number is limited to 32 bits and is placed in bits 0-31 of the IV. The inode number (which is also limited to 32 bits) is placed in bits 32-63.
- With `IV_INO_LBLK_32` policies, the logical block number is limited to 32 bits and is placed in bits 0-31 of the IV. The inode number is then hashed and added mod 2^{32} .

Note that because file logical block numbers are included in the IVs, filesystems must enforce that blocks are never shifted around within encrypted files, e.g. via “collapse range” or “insert range” .

Filenames encryption

For filenames, each full filename is encrypted at once. Because of the requirements to retain support for efficient directory lookups and filenames of up to 255 bytes, the same IV is used for every filename in a directory.

However, each encrypted directory still uses a unique key, or alternatively has the file's nonce (for DIRECT_KEY policies) or inode number (for IV_INO_LBLK_64 policies) included in the IVs. Thus, IV reuse is limited to within a single directory.

With CTS-CBC, the IV reuse means that when the plaintext filenames share a common prefix at least as long as the cipher block size (16 bytes for AES), the corresponding encrypted filenames will also share a common prefix. This is undesirable. Adiantum does not have this weakness, as it is a wide-block encryption mode.

All supported filenames encryption modes accept any plaintext length ≥ 16 bytes; cipher block alignment is not required. However, filenames shorter than 16 bytes are NUL-padded to 16 bytes before being encrypted. In addition, to reduce leakage of filename lengths via their ciphertexts, all filenames are NUL-padded to the next 4, 8, 16, or 32-byte boundary (configurable). 32 is recommended since this provides the best confidentiality, at the cost of making directory entries consume slightly more space. Note that since NUL ($\backslash 0$) is not otherwise a valid character in filenames, the padding will never produce duplicate plaintexts.

Symbolic link targets are considered a type of filename and are encrypted in the same way as filenames in directory entries, except that IV reuse is not a problem as each symlink has its own inode.

2.2.5 User API

Setting an encryption policy

FS_IOC_SET_ENCRYPTION_POLICY

The FS_IOC_SET_ENCRYPTION_POLICY ioctl sets an encryption policy on an empty directory or verifies that a directory or regular file already has the specified encryption policy. It takes in a pointer to a struct `fscrypt_policy_v1` or a struct `fscrypt_policy_v2`, defined as follows:

```
#define FSCRYPT_POLICY_V1          0
#define FSCRYPT_KEY_DESCRIPTOR_SIZE 8
struct fscrypt_policy_v1 {
    __u8 version;
    __u8 contents_encryption_mode;
    __u8 filenames_encryption_mode;
    __u8 flags;
    __u8 master_key_descriptor[FSCRYPT_KEY_DESCRIPTOR_SIZE];
};
#define fscrypt_policy    fscrypt_policy_v1

#define FSCRYPT_POLICY_V2          2
#define FSCRYPT_KEY_IDENTIFIER_SIZE 16
struct fscrypt_policy_v2 {
```

(continues on next page)

(continued from previous page)

```
    __u8 version;
    __u8 contents_encryption_mode;
    __u8 filenames_encryption_mode;
    __u8 flags;
    __u8 __reserved[4];
    __u8 master_key_identifier[FSCRYPT_KEY_IDENTIFIER_SIZE];
};
```

This structure must be initialized as follows:

- `version` must be `FSCRYPT_POLICY_V1` (0) if the struct is `fscrypt_policy_v1` or `FSCRYPT_POLICY_V2` (2) if the struct is `fscrypt_policy_v2`. (Note: we refer to the original policy version as “v1”, though its version code is really 0.) For new encrypted directories, use v2 policies.
- `contents_encryption_mode` and `filenames_encryption_mode` must be set to constants from `<linux/fscrypt.h>` which identify the encryption modes to use. If unsure, use `FSCRYPT_MODE_AES_256_XTS` (1) for `contents_encryption_mode` and `FSCRYPT_MODE_AES_256_CTS` (4) for `filenames_encryption_mode`.
- `flags` contains optional flags from `<linux/fscrypt.h>`:
 - `FSCRYPT_POLICY_FLAGS_PAD_*`: The amount of NUL padding to use when encrypting filenames. If unsure, use `FSCRYPT_POLICY_FLAGS_PAD_32` (0x3).
 - `FSCRYPT_POLICY_FLAG_DIRECT_KEY`: See `DIRECT_KEY` policies.
 - `FSCRYPT_POLICY_FLAG_IV_INO_LBLK_64`: See `IV_INO_LBLK_64` policies.
 - `FSCRYPT_POLICY_FLAG_IV_INO_LBLK_32`: See `IV_INO_LBLK_32` policies.

v1 encryption policies only support the `PAD_*` and `DIRECT_KEY` flags. The other flags are only supported by v2 encryption policies.

The `DIRECT_KEY`, `IV_INO_LBLK_64`, and `IV_INO_LBLK_32` flags are mutually exclusive.

- For v2 encryption policies, `__reserved` must be zeroed.
- For v1 encryption policies, `master_key_descriptor` specifies how to find the master key in a keyring; see [Adding keys](#). It is up to userspace to choose a unique `master_key_descriptor` for each master key. The `e4crypt` and `fscrypt` tools use the first 8 bytes of SHA-512 (`SHA-512(master_key)`), but this particular scheme is not required. Also, the master key need not be in the keyring yet when `FS_IOC_SET_ENCRYPTION_POLICY` is executed. However, it must be added before any files can be created in the encrypted directory.

For v2 encryption policies, `master_key_descriptor` has been replaced with `master_key_identifier`, which is longer and cannot be arbitrarily chosen. Instead, the key must first be added using `FS_IOC_ADD_ENCRYPTION_KEY`. Then, the `key_spec.u.identifier` the kernel returned in the struct

`fsencrypt_add_key_arg` must be used as the `master_key_identifier` in the `struct fsencrypt_policy_v2`.

If the file is not yet encrypted, then `FS_IOC_SET_ENCRYPTION_POLICY` verifies that the file is an empty directory. If so, the specified encryption policy is assigned to the directory, turning it into an encrypted directory. After that, and after providing the corresponding master key as described in [Adding keys](#), all regular files, directories (recursively), and symlinks created in the directory will be encrypted, inheriting the same encryption policy. The filenames in the directory's entries will be encrypted as well.

Alternatively, if the file is already encrypted, then `FS_IOC_SET_ENCRYPTION_POLICY` validates that the specified encryption policy exactly matches the actual one. If they match, then the `ioctl` returns 0. Otherwise, it fails with `EEXIST`. This works on both regular files and directories, including nonempty directories.

When a v2 encryption policy is assigned to a directory, it is also required that either the specified key has been added by the current user or that the caller has `CAP_FOWNER` in the initial user namespace. (This is needed to prevent a user from encrypting their data with another user's key.) The key must remain added while `FS_IOC_SET_ENCRYPTION_POLICY` is executing. However, if the new encrypted directory does not need to be accessed immediately, then the key can be removed right away afterwards.

Note that the `ext4` filesystem does not allow the root directory to be encrypted, even if it is empty. Users who want to encrypt an entire filesystem with one key should consider using `dm-crypt` instead.

`FS_IOC_SET_ENCRYPTION_POLICY` can fail with the following errors:

- `EACCES`: the file is not owned by the process's `uid`, nor does the process have the `CAP_FOWNER` capability in a namespace with the file owner's `uid` mapped
- `EEXIST`: the file is already encrypted with an encryption policy different from the one specified
- `EINVAL`: an invalid encryption policy was specified (invalid version, mode(s), or flags; or reserved bits were set); or a v1 encryption policy was specified but the directory has the `casefold` flag enabled (casefolding is incompatible with v1 policies).
- `ENOKEY`: a v2 encryption policy was specified, but the key with the specified `master_key_identifier` has not been added, nor does the process have the `CAP_FOWNER` capability in the initial user namespace
- `ENOTDIR`: the file is unencrypted and is a regular file, not a directory
- `ENOTEMPTY`: the file is unencrypted and is a nonempty directory
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for filesystems, or the filesystem superblock has not had encryption enabled on it. (For example, to use encryption on an `ext4` filesystem, `CONFIG_FS_ENCRYPTION` must be enabled in the kernel config, and the su-

perblock must have had the “encrypt” feature flag enabled using tune2fs -O encrypt or mkfs.ext4 -O encrypt.)

- EPERM: this directory may not be encrypted, e.g. because it is the root directory of an ext4 filesystem
- EROFS: the filesystem is readonly

Getting an encryption policy

Two ioctls are available to get a file’s encryption policy:

- FS_IOC_GET_ENCRYPTION_POLICY_EX
- FS_IOC_GET_ENCRYPTION_POLICY

The extended (`_EX`) version of the ioctl is more general and is recommended to use when possible. However, on older kernels only the original ioctl is available. Applications should try the extended version, and if it fails with `ENOTTY` fall back to the original version.

FS_IOC_GET_ENCRYPTION_POLICY_EX

The `FS_IOC_GET_ENCRYPTION_POLICY_EX` ioctl retrieves the encryption policy, if any, for a directory or regular file. No additional permissions are required beyond the ability to open the file. It takes in a pointer to a struct `fsencrypt_get_policy_ex_arg`, defined as follows:

```
struct fsencrypt_get_policy_ex_arg {
    __u64 policy_size; /* input/output */
    union {
        __u8 version;
        struct fsencrypt_policy_v1 v1;
        struct fsencrypt_policy_v2 v2;
    } policy; /* output */
};
```

The caller must initialize `policy_size` to the size available for the policy struct, i.e. `sizeof(arg.policy)`.

On success, the policy struct is returned in `policy`, and its actual size is returned in `policy_size`. `policy.version` should be checked to determine the version of policy returned. Note that the version code for the “v1” policy is actually 0 (`FSCRYPT_POLICY_V1`).

`FS_IOC_GET_ENCRYPTION_POLICY_EX` can fail with the following errors:

- EINVAL: the file is encrypted, but it uses an unrecognized encryption policy version
- ENODATA: the file is not encrypted
- ENOTTY: this type of filesystem does not implement encryption, or this kernel is too old to support `FS_IOC_GET_ENCRYPTION_POLICY_EX` (try `FS_IOC_GET_ENCRYPTION_POLICY` instead)

- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it
- `E_OVERFLOW`: the file is encrypted and uses a recognized encryption policy version, but the policy struct does not fit into the provided buffer

Note: if you only need to know whether a file is encrypted or not, on most filesystems it is also possible to use the `FS_IOC_GETFLAGS` ioctl and check for `FS_ENCRYPT_FL`, or to use the `statx()` system call and check for `STATX_ATTR_ENCRYPTED` in `stx_attributes`.

FS_IOC_GET_ENCRYPTION_POLICY

The `FS_IOC_GET_ENCRYPTION_POLICY` ioctl can also retrieve the encryption policy, if any, for a directory or regular file. However, unlike `FS_IOC_GET_ENCRYPTION_POLICY_EX`, `FS_IOC_GET_ENCRYPTION_POLICY` only supports the original policy version. It takes in a pointer directly to a struct `fscrypt_policy_v1` rather than a struct `fscrypt_get_policy_ex_arg`.

The error codes for `FS_IOC_GET_ENCRYPTION_POLICY` are the same as those for `FS_IOC_GET_ENCRYPTION_POLICY_EX`, except that `FS_IOC_GET_ENCRYPTION_POLICY` also returns `EINVAL` if the file is encrypted using a newer encryption policy version.

Getting the per-filesystem salt

Some filesystems, such as `ext4` and `F2FS`, also support the deprecated ioctl `FS_IOC_GET_ENCRYPTION_PWSALT`. This ioctl retrieves a randomly generated 16-byte value stored in the filesystem superblock. This value is intended to be used as a salt when deriving an encryption key from a passphrase or other low-entropy user credential.

`FS_IOC_GET_ENCRYPTION_PWSALT` is deprecated. Instead, prefer to generate and manage any needed salt(s) in userspace.

Getting a file's encryption nonce

Since Linux v5.7, the ioctl `FS_IOC_GET_ENCRYPTION_NONCE` is supported. On encrypted files and directories it gets the inode's 16-byte nonce. On unencrypted files and directories, it fails with `ENODATA`.

This ioctl can be useful for automated tests which verify that the encryption is being done correctly. It is not needed for normal use of `fscrypt`.

Adding keys

FS_IOC_ADD_ENCRYPTION_KEY

The `FS_IOC_ADD_ENCRYPTION_KEY` ioctl adds a master encryption key to the filesystem, making all files on the filesystem which were encrypted using that key appear “unlocked”, i.e. in plaintext form. It can be executed on any file or directory on the target filesystem, but using the filesystem’s root directory is recommended. It takes in a pointer to a struct `fsencrypt_add_key_arg`, defined as follows:

```
struct fsencrypt_add_key_arg {
    struct fsencrypt_key_specifier key_spec;
    __u32 raw_size;
    __u32 key_id;
    __u32 __reserved[8];
    __u8 raw[];
};

#define FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR      1
#define FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER    2

struct fsencrypt_key_specifier {
    __u32 type; /* one of FSCRYPT_KEY_SPEC_TYPE_* */
    __u32 __reserved;
    union {
        __u8 __reserved[32]; /* reserve some extra space */
        __u8 descriptor[FSCRYPT_KEY_DESCRIPTOR_SIZE];
        __u8 identifier[FSCRYPT_KEY_IDENTIFIER_SIZE];
    } u;
};

struct fsencrypt_provisioning_key_payload {
    __u32 type;
    __u32 __reserved;
    __u8 raw[];
};
```

struct `fsencrypt_add_key_arg` must be zeroed, then initialized as follows:

- If the key is being added for use by v1 encryption policies, then `key_spec.type` must contain `FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR`, and `key_spec.u.descriptor` must contain the descriptor of the key being added, corresponding to the value in the `master_key_descriptor` field of struct `fsencrypt_policy_v1`. To add this type of key, the calling process must have the `CAP_SYS_ADMIN` capability in the initial user namespace.

Alternatively, if the key is being added for use by v2 encryption policies, then `key_spec.type` must contain `FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER`, and `key_spec.u.identifier` is an output field which the kernel fills in with a cryptographic hash of the key. To add this type of key, the calling process does not need any privileges. However, the number of keys that can be added is limited by the user’s quota for the keyrings service (see `Documentation/security/keys/core.rst`).

- `raw_size` must be the size of the raw key provided, in bytes. Alternatively, if `key_id` is nonzero, this field must be 0, since in that case the size is implied

by the specified Linux keyring key.

- `key_id` is 0 if the raw key is given directly in the `raw` field. Otherwise `key_id` is the ID of a Linux keyring key of type “`fscrypt-provisioning`” whose payload is a `struct fscrypt_provisioning_key_payload` whose `raw` field contains the raw key and whose `type` field matches `key_spec.type`. Since `raw` is variable-length, the total size of this key’s payload must be `sizeof(struct fscrypt_provisioning_key_payload)` plus the raw key size. The process must have Search permission on this key.

Most users should leave this 0 and specify the raw key directly. The support for specifying a Linux keyring key is intended mainly to allow re-adding keys after a filesystem is unmounted and re-mounted, without having to store the raw keys in userspace memory.

- `raw` is a variable-length field which must contain the actual key, `raw_size` bytes long. Alternatively, if `key_id` is nonzero, then this field is unused.

For v2 policy keys, the kernel keeps track of which user (identified by effective user ID) added the key, and only allows the key to be removed by that user—or by “root”, if they use `FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS`.

However, if another user has added the key, it may be desirable to prevent that other user from unexpectedly removing it. Therefore, `FS_IOC_ADD_ENCRYPTION_KEY` may also be used to add a v2 policy key again, even if it’s already added by other user(s). In this case, `FS_IOC_ADD_ENCRYPTION_KEY` will just install a claim to the key for the current user, rather than actually add the key again (but the raw key must still be provided, as a proof of knowledge).

`FS_IOC_ADD_ENCRYPTION_KEY` returns 0 if either the key or a claim to the key was either added or already exists.

`FS_IOC_ADD_ENCRYPTION_KEY` can fail with the following errors:

- `EACCES`: `FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR` was specified, but the caller does not have the `CAP_SYS_ADMIN` capability in the initial user namespace; or the raw key was specified by Linux key ID but the process lacks Search permission on the key.
- `EDQUOT`: the key quota for this user would be exceeded by adding the key
- `EINVAL`: invalid key size or key specifier type, or reserved bits were set
- `EKEYREJECTED`: the raw key was specified by Linux key ID, but the key has the wrong type
- `ENOKEY`: the raw key was specified by Linux key ID, but no key exists with that ID
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it

Legacy method

For v1 encryption policies, a master encryption key can also be provided by adding it to a process-subscribed keyring, e.g. to a session keyring, or to a user keyring if the user keyring is linked into the session keyring.

This method is deprecated (and not supported for v2 encryption policies) for several reasons. First, it cannot be used in combination with `FS_IOC_REMOVE_ENCRYPTION_KEY` (see [Removing keys](#)), so for removing a key a workaround such as `keyctl_unlink()` in combination with `sync; echo 2 > /proc/sys/vm/drop_caches` would have to be used. Second, it doesn't match the fact that the locked/unlocked status of encrypted files (i.e. whether they appear to be in plaintext form or in ciphertext form) is global. This mismatch has caused much confusion as well as real problems when processes running under different UIDs, such as a `sudo` command, need to access encrypted files.

Nevertheless, to add a key to one of the process-subscribed keyrings, the `add_key()` system call can be used (see: [Documentation/security/keys/core.rst](#)). The key type must be “logon”; keys of this type are kept in kernel memory and cannot be read back by userspace. The key description must be “fscrypt:” followed by the 16-character lower case hex representation of the `master_key_descriptor` that was set in the encryption policy. The key payload must conform to the following structure:

```
#define FSCRYPT_MAX_KEY_SIZE          64

struct fscrypt_key {
    __u32 mode;
    __u8 raw[FSCRYPT_MAX_KEY_SIZE];
    __u32 size;
};
```

`mode` is ignored; just set it to 0. The actual key is provided in `raw` with `size` indicating its size in bytes. That is, the bytes `raw[0..size-1]` (inclusive) are the actual key.

The key description prefix “fscrypt:” may alternatively be replaced with a filesystem-specific prefix such as “ext4:”. However, the filesystem-specific prefixes are deprecated and should not be used in new programs.

Removing keys

Two `ioctl`s are available for removing a key that was added by `FS_IOC_ADD_ENCRYPTION_KEY`:

- `FS_IOC_REMOVE_ENCRYPTION_KEY`
- `FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS`

These two `ioctl`s differ only in cases where v2 policy keys are added or removed by non-root users.

These `ioctl`s don't work on keys that were added via the legacy process-subscribed keyrings mechanism.

Before using these ioctls, read the Kernel memory compromise section for a discussion of the security goals and limitations of these ioctls.

FS_IOC_REMOVE_ENCRYPTION_KEY

The `FS_IOC_REMOVE_ENCRYPTION_KEY` ioctl removes a claim to a master encryption key from the filesystem, and possibly removes the key itself. It can be executed on any file or directory on the target filesystem, but using the filesystem's root directory is recommended. It takes in a pointer to a struct `fscrypt_remove_key_arg`, defined as follows:

```
struct fscrypt_remove_key_arg {
    struct fscrypt_key_specifier key_spec;
#define FSCRYPT_KEY_REMOVAL_STATUS_FLAG_FILES_BUSY    0x00000001
#define FSCRYPT_KEY_REMOVAL_STATUS_FLAG_OTHER_USERS  0x00000002
    __u32 removal_status_flags; /* output */
    __u32 __reserved[5];
};
```

This structure must be zeroed, then initialized as follows:

- The key to remove is specified by `key_spec`:
 - To remove a key used by v1 encryption policies, set `key_spec.type` to `FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR` and fill in `key_spec.u.descriptor`. To remove this type of key, the calling process must have the `CAP_SYS_ADMIN` capability in the initial user namespace.
 - To remove a key used by v2 encryption policies, set `key_spec.type` to `FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER` and fill in `key_spec.u.identifier`.

For v2 policy keys, this ioctl is usable by non-root users. However, to make this possible, it actually just removes the current user's claim to the key, undoing a single call to `FS_IOC_ADD_ENCRYPTION_KEY`. Only after all claims are removed is the key really removed.

For example, if `FS_IOC_ADD_ENCRYPTION_KEY` was called with uid 1000, then the key will be “claimed” by uid 1000, and `FS_IOC_REMOVE_ENCRYPTION_KEY` will only succeed as uid 1000. Or, if both uids 1000 and 2000 added the key, then for each uid `FS_IOC_REMOVE_ENCRYPTION_KEY` will only remove their own claim. Only once both are removed is the key really removed. (Think of it like unlinking a file that may have hard links.)

If `FS_IOC_REMOVE_ENCRYPTION_KEY` really removes the key, it will also try to “lock” all files that had been unlocked with the key. It won't lock files that are still in-use, so this ioctl is expected to be used in cooperation with userspace ensuring that none of the files are still open. However, if necessary, this ioctl can be executed again later to retry locking any remaining files.

`FS_IOC_REMOVE_ENCRYPTION_KEY` returns 0 if either the key was removed (but may still have files remaining to be locked), the user's claim to the key was removed, or the key was already removed but had files remaining to be the locked so the ioctl retried locking them. In any of these cases, `removal_status_flags` is filled in with the following informational status flags:

- `FSCRYPT_KEY_REMOVAL_STATUS_FLAG_FILES_BUSY`: set if some file(s) are still in-use. Not guaranteed to be set in the case where only the user's claim to the key was removed.
- `FSCRYPT_KEY_REMOVAL_STATUS_FLAG_OTHER_USERS`: set if only the user's claim to the key was removed, not the key itself

`FS_IOC_REMOVE_ENCRYPTION_KEY` can fail with the following errors:

- `EACCES`: The `FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR` key specifier type was specified, but the caller does not have the `CAP_SYS_ADMIN` capability in the initial user namespace
- `EINVAL`: invalid key specifier type, or reserved bits were set
- `ENOKEY`: the key object was not found at all, i.e. it was never added in the first place or was already fully removed including all files locked; or, the user does not have a claim to the key (but someone else does).
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it

FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS

`FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS` is exactly the same as `FS_IOC_REMOVE_ENCRYPTION_KEY`, except that for v2 policy keys, the `ALL_USERS` version of the ioctl will remove all users' claims to the key, not just the current user's. I.e., the key itself will always be removed, no matter how many users have added it. This difference is only meaningful if non-root users are adding and removing keys.

Because of this, `FS_IOC_REMOVE_ENCRYPTION_KEY_ALL_USERS` also requires "root", namely the `CAP_SYS_ADMIN` capability in the initial user namespace. Otherwise it will fail with `EACCES`.

Getting key status

FS_IOC_GET_ENCRYPTION_KEY_STATUS

The `FS_IOC_GET_ENCRYPTION_KEY_STATUS` ioctl retrieves the status of a master encryption key. It can be executed on any file or directory on the target filesystem, but using the filesystem's root directory is recommended. It takes in a pointer to a struct `fscrypt_get_key_status_arg`, defined as follows:

```
struct fscrypt_get_key_status_arg {
    /* input */
    struct fscrypt_key_specifier key_spec;
    __u32 __reserved[6];

    /* output */
#define FSCRYPT_KEY_STATUS_ABSENT 1
#define FSCRYPT_KEY_STATUS_PRESENT 2
```

(continues on next page)

(continued from previous page)

```
#define FSCRYPT_KEY_STATUS_INCOMPLETELY_REMOVED 3
    __u32 status;
#define FSCRYPT_KEY_STATUS_FLAG_ADDED_BY_SELF 0x00000001
    __u32 status_flags;
    __u32 user_count;
    __u32 __out_reserved[13];
};
```

The caller must zero all input fields, then fill in `key_spec`:

- To get the status of a key for v1 encryption policies, set `key_spec.type` to `FSCRYPT_KEY_SPEC_TYPE_DESCRIPTOR` and fill in `key_spec.u.descriptor`.
- To get the status of a key for v2 encryption policies, set `key_spec.type` to `FSCRYPT_KEY_SPEC_TYPE_IDENTIFIER` and fill in `key_spec.u.identifier`.

On success, 0 is returned and the kernel fills in the output fields:

- `status` indicates whether the key is absent, present, or incompletely removed. Incompletely removed means that the master secret has been removed, but some files are still in use; i.e., `FS_IOC_REMOVE_ENCRYPTION_KEY` returned 0 but set the informational status flag `FSCRYPT_KEY_REMOVAL_STATUS_FLAG_FILES_BUSY`.
- `status_flags` can contain the following flags:
 - `FSCRYPT_KEY_STATUS_FLAG_ADDED_BY_SELF` indicates that the key has added by the current user. This is only set for keys identified by identifier rather than by descriptor.
- `user_count` specifies the number of users who have added the key. This is only set for keys identified by identifier rather than by descriptor.

`FS_IOC_GET_ENCRYPTION_KEY_STATUS` can fail with the following errors:

- `EINVAL`: invalid key specifier type, or reserved bits were set
- `ENOTTY`: this type of filesystem does not implement encryption
- `EOPNOTSUPP`: the kernel was not configured with encryption support for this filesystem, or the filesystem superblock has not had encryption enabled on it

Among other use cases, `FS_IOC_GET_ENCRYPTION_KEY_STATUS` can be useful for determining whether the key for a given encrypted directory needs to be added before prompting the user for the passphrase needed to derive the key.

`FS_IOC_GET_ENCRYPTION_KEY_STATUS` can only get the status of keys in the filesystem-level keyring, i.e. the keyring managed by `FS_IOC_ADD_ENCRYPTION_KEY` and `FS_IOC_REMOVE_ENCRYPTION_KEY`. It cannot get the status of a key that has only been added for use by v1 encryption policies using the legacy mechanism involving process-subscribed keyrings.

2.2.6 Access semantics

With the key

With the encryption key, encrypted regular files, directories, and symlinks behave very similarly to their unencrypted counterparts —after all, the encryption is intended to be transparent. However, astute users may notice some differences in behavior:

- Unencrypted files, or files encrypted with a different encryption policy (i.e. different key, modes, or flags), cannot be renamed or linked into an encrypted directory; see Encryption policy enforcement. Attempts to do so will fail with `EXDEV`. However, encrypted files can be renamed within an encrypted directory, or into an unencrypted directory.

Note: “moving” an unencrypted file into an encrypted directory, e.g. with the `mv` program, is implemented in userspace by a copy followed by a delete. Be aware that the original unencrypted data may remain recoverable from free space on the disk; prefer to keep all files encrypted from the very beginning. The `shred` program may be used to overwrite the source files but isn't guaranteed to be effective on all filesystems and storage devices.

- Direct I/O is not supported on encrypted files. Attempts to use direct I/O on such files will fall back to buffered I/O.
- The `fallocate` operations `FALLOC_FL_COLLAPSE_RANGE` and `FALLOC_FL_INSERT_RANGE` are not supported on encrypted files and will fail with `EOPNOTSUPP`.
- Online defragmentation of encrypted files is not supported. The `EXT4_IOC_MOVE_EXT` and `F2FS_IOC_MOVE_RANGE` ioctls will fail with `EOPNOTSUPP`.
- The `ext4` filesystem does not support data journaling with encrypted regular files. It will fall back to ordered data mode instead.
- DAX (Direct Access) is not supported on encrypted files.
- The `st_size` of an encrypted symlink will not necessarily give the length of the symlink target as required by POSIX. It will actually give the length of the ciphertext, which will be slightly longer than the plaintext due to NUL-padding and an extra 2-byte overhead.
- The maximum length of an encrypted symlink is 2 bytes shorter than the maximum length of an unencrypted symlink. For example, on an `EXT4` filesystem with a 4K block size, unencrypted symlinks can be up to 4095 bytes long, while encrypted symlinks can only be up to 4093 bytes long (both lengths excluding the terminating null).

Note that `mmap` is supported. This is possible because the pagecache for an encrypted file contains the plaintext, not the ciphertext.

Without the key

Some filesystem operations may be performed on encrypted regular files, directories, and symlinks even before their encryption key has been added, or after their encryption key has been removed:

- File metadata may be read, e.g. using `stat()`.
- Directories may be listed, in which case the filenames will be listed in an encoded form derived from their ciphertext. The current encoding algorithm is described in [Filename hashing and encoding](#). The algorithm is subject to change, but it is guaranteed that the presented filenames will be no longer than `NAME_MAX` bytes, will not contain the `/` or `\0` characters, and will uniquely identify directory entries.

The `.` and `..` directory entries are special. They are always present and are not encrypted or encoded.

- Files may be deleted. That is, nondirectory files may be deleted with `unlink()` as usual, and empty directories may be deleted with `rmdir()` as usual. Therefore, `rm` and `rm -r` will work as expected.
- Symlink targets may be read and followed, but they will be presented in encrypted form, similar to filenames in directories. Hence, they are unlikely to point to anywhere useful.

Without the key, regular files cannot be opened or truncated. Attempts to do so will fail with `ENOKEY`. This implies that any regular file operations that require a file descriptor, such as `read()`, `write()`, `mmap()`, `fallocate()`, and `ioctl()`, are also forbidden.

Also without the key, files of any type (including directories) cannot be created or linked into an encrypted directory, nor can a name in an encrypted directory be the source or target of a `rename`, nor can an `O_TMPFILE` temporary file be created in an encrypted directory. All such operations will fail with `ENOKEY`.

It is not currently possible to backup and restore encrypted files without the encryption key. This would require special APIs which have not yet been implemented.

2.2.7 Encryption policy enforcement

After an encryption policy has been set on a directory, all regular files, directories, and symbolic links created in that directory (recursively) will inherit that encryption policy. Special files —that is, named pipes, device nodes, and UNIX domain sockets —will not be encrypted.

Except for those special files, it is forbidden to have unencrypted files, or files encrypted with a different encryption policy, in an encrypted directory tree. Attempts to link or rename such a file into an encrypted directory will fail with `EXDEV`. This is also enforced during `->lookup()` to provide limited protection against offline attacks that try to disable or downgrade encryption in known locations where applications may later write sensitive data. It is recommended that systems implementing a form of “verified boot” take advantage of this by validating all top-level encryption policies prior to access.

2.2.8 Implementation details

Encryption context

An encryption policy is represented on-disk by a struct `fsencrypt_context_v1` or a struct `fsencrypt_context_v2`. It is up to individual filesystems to decide where to store it, but normally it would be stored in a hidden extended attribute. It should not be exposed by the xattr-related system calls such as `getxattr()` and `setxattr()` because of the special semantics of the encryption xattr. (In particular, there would be much confusion if an encryption policy were to be added to or removed from anything other than an empty directory.) These structs are defined as follows:

```
#define FS_KEY_DERIVATION_NONCE_SIZE 16

#define FSCRYPT_KEY_DESCRIPTOR_SIZE 8
struct fsencrypt_context_v1 {
    u8 version;
    u8 contents_encryption_mode;
    u8 filenames_encryption_mode;
    u8 flags;
    u8 master_key_descriptor[FSCRYPT_KEY_DESCRIPTOR_SIZE];
    u8 nonce[FS_KEY_DERIVATION_NONCE_SIZE];
};

#define FSCRYPT_KEY_IDENTIFIER_SIZE 16
struct fsencrypt_context_v2 {
    u8 version;
    u8 contents_encryption_mode;
    u8 filenames_encryption_mode;
    u8 flags;
    u8 __reserved[4];
    u8 master_key_identifier[FSCRYPT_KEY_IDENTIFIER_SIZE];
    u8 nonce[FS_KEY_DERIVATION_NONCE_SIZE];
};
```

The context structs contain the same information as the corresponding policy structs (see [Setting an encryption policy](#)), except that the context structs also contain a nonce. The nonce is randomly generated by the kernel and is used as KDF input or as a tweak to cause different files to be encrypted differently; see [Per-file encryption keys and DIRECT_KEY policies](#).

Data path changes

For the read path (`->readpage()`) of regular files, filesystems can read the ciphertext into the page cache and decrypt it in-place. The page lock must be held until decryption has finished, to prevent the page from becoming visible to userspace prematurely.

For the write path (`->writepage()`) of regular files, filesystems cannot encrypt data in-place in the page cache, since the cached plaintext must be preserved. Instead, filesystems must encrypt into a temporary buffer or “bounce page”, then write out the temporary buffer. Some filesystems, such as UBIFS, already use temporary buffers regardless of encryption. Other filesystems, such as ext4 and F2FS, have to allocate bounce pages specially for encryption.

Filename hashing and encoding

Modern filesystems accelerate directory lookups by using indexed directories. An indexed directory is organized as a tree keyed by filename hashes. When a `->lookup()` is requested, the filesystem normally hashes the filename being looked up so that it can quickly find the corresponding directory entry, if any.

With encryption, lookups must be supported and efficient both with and without the encryption key. Clearly, it would not work to hash the plaintext filenames, since the plaintext filenames are unavailable without the key. (Hashing the plaintext filenames would also make it impossible for the filesystem's `fsck` tool to optimize encrypted directories.) Instead, filesystems hash the ciphertext filenames, i.e. the bytes actually stored on-disk in the directory entries. When asked to do a `->lookup()` with the key, the filesystem just encrypts the user-supplied name to get the ciphertext.

Lookups without the key are more complicated. The raw ciphertext may contain the `\0` and `/` characters, which are illegal in filenames. Therefore, `readdir()` must base64-encode the ciphertext for presentation. For most filenames, this works fine; on `->lookup()`, the filesystem just base64-decodes the user-supplied name to get back to the raw ciphertext.

However, for very long filenames, base64 encoding would cause the filename length to exceed `NAME_MAX`. To prevent this, `readdir()` actually presents long filenames in an abbreviated form which encodes a strong “hash” of the ciphertext filename, along with the optional filesystem-specific hash(es) needed for directory lookups. This allows the filesystem to still, with a high degree of confidence, map the filename given in `->lookup()` back to a particular directory entry that was previously listed by `readdir()`. See `struct fscrypt_nokey_name` in the source for more details.

Note that the precise way that filenames are presented to userspace without the key is subject to change in the future. It is only meant as a way to temporarily present valid filenames so that commands like `rm -r` work as expected on encrypted directories.

2.2.9 Tests

To test `fscrypt`, use `xfstests`, which is Linux's de facto standard filesystem test suite. First, run all the tests in the “encrypt” group on the relevant filesystem(s). For example, to test `ext4` and `f2fs` encryption using `kvm-xfstests`:

```
kvm-xfstests -c ext4,f2fs -g encrypt
```

UBIFS encryption can also be tested this way, but it should be done in a separate command, and it takes some time for `kvm-xfstests` to set up emulated UBI volumes:

```
kvm-xfstests -c ubifs -g encrypt
```

No tests should fail. However, tests that use non-default encryption modes (e.g. `generic/549` and `generic/550`) will be skipped if the needed algorithms were not built into the kernel's crypto API. Also, tests that access the raw block device (e.g. `generic/399`, `generic/548`, `generic/549`, `generic/550`) will be skipped on UBIFS.

Besides running the “encrypt” group tests, for ext4 and f2fs it’s also possible to run most xfstests with the “test_dummy_encryption” mount option. This option causes all new files to be automatically encrypted with a dummy key, without having to make any API calls. This tests the encrypted I/O paths more thoroughly. To do this with kvm-xfstests, use the “encrypt” filesystem configuration:

```
kvm-xfstests -c ext4/encrypt,f2fs/encrypt -g auto
```

Because this runs many more tests than “-g encrypt” does, it takes much longer to run; so also consider using [gce-xfstests](#) instead of kvm-xfstests:

```
gce-xfstests -c ext4/encrypt,f2fs/encrypt -g auto
```

2.3 fs-verity: read-only file-based authenticity protection

2.3.1 Introduction

fs-verity (fs/verity/) is a support layer that filesystems can hook into to support transparent integrity and authenticity protection of read-only files. Currently, it is supported by the ext4 and f2fs filesystems. Like fscrypt, not too much filesystem-specific code is needed to support fs-verity.

fs-verity is similar to [dm-verity](#) but works on files rather than block devices. On regular files on filesystems supporting fs-verity, userspace can execute an ioctl that causes the filesystem to build a Merkle tree for the file and persist it to a filesystem-specific location associated with the file.

After this, the file is made readonly, and all reads from the file are automatically verified against the file’s Merkle tree. Reads of any corrupted data, including mmap reads, will fail.

Userspace can use another ioctl to retrieve the root hash (actually the “file measurement”, which is a hash that includes the root hash) that fs-verity is enforcing for the file. This ioctl executes in constant time, regardless of the file size.

fs-verity is essentially a way to hash a file in constant time, subject to the caveat that reads which would violate the hash will fail at runtime.

2.3.2 Use cases

By itself, the base fs-verity feature only provides integrity protection, i.e. detection of accidental (non-malicious) corruption.

However, because fs-verity makes retrieving the file hash extremely efficient, it’s primarily meant to be used as a tool to support authentication (detection of malicious modifications) or auditing (logging file hashes before use).

Trusted userspace code (e.g. operating system code running on a read-only partition that is itself authenticated by dm-verity) can authenticate the contents of an fs-verity file by using the FS_IOC_MEASURE_VERITY ioctl to retrieve its hash, then verifying a digital signature of it.

A standard file hash could be used instead of fs-verity. However, this is inefficient if the file is large and only a small portion may be accessed. This is often the case for Android application package (APK) files, for example. These typically contain many translations, classes, and other resources that are infrequently or even never accessed on a particular device. It would be slow and wasteful to read and hash the entire file before starting the application.

Unlike an ahead-of-time hash, fs-verity also re-verifies data each time it's paged in. This ensures that malicious disk firmware can't undetectably change the contents of the file at runtime.

fs-verity does not replace or obsolete dm-verity. dm-verity should still be used on read-only filesystems. fs-verity is for files that must live on a read-write filesystem because they are independently updated and potentially user-installed, so dm-verity cannot be used.

The base fs-verity feature is a hashing mechanism only; actually authenticating the files is up to userspace. However, to meet some users' needs, fs-verity optionally supports a simple signature verification mechanism where users can configure the kernel to require that all fs-verity files be signed by a key loaded into a keyring; see [Built-in signature verification](#). Support for fs-verity file hashes in IMA (Integrity Measurement Architecture) policies is also planned.

2.3.3 User API

FS_IOC_ENABLE_VERITY

The `FS_IOC_ENABLE_VERITY` ioctl enables fs-verity on a file. It takes in a pointer to a `struct fsverity_enable_arg`, defined as follows:

```
struct fsverity_enable_arg {
    __u32 version;
    __u32 hash_algorithm;
    __u32 block_size;
    __u32 salt_size;
    __u64 salt_ptr;
    __u32 sig_size;
    __u32 __reserved1;
    __u64 sig_ptr;
    __u64 __reserved2[11];
};
```

This structure contains the parameters of the Merkle tree to build for the file, and optionally contains a signature. It must be initialized as follows:

- `version` must be 1.
- `hash_algorithm` must be the identifier for the hash algorithm to use for the Merkle tree, such as `FS_VERITY_HASH_ALG_SHA256`. See `include/uapi/linux/fsverity.h` for the list of possible values.
- `block_size` must be the Merkle tree block size. Currently, this must be equal to the system page size, which is usually 4096 bytes. Other sizes may be supported in the future. This value is not necessarily the same as the filesystem block size.

- `salt_size` is the size of the salt in bytes, or 0 if no salt is provided. The salt is a value that is prepended to every hashed block; it can be used to personalize the hashing for a particular file or device. Currently the maximum salt size is 32 bytes.
- `salt_ptr` is the pointer to the salt, or NULL if no salt is provided.
- `sig_size` is the size of the signature in bytes, or 0 if no signature is provided. Currently the signature is (somewhat arbitrarily) limited to 16128 bytes. See `Built-in signature verification` for more information.
- `sig_ptr` is the pointer to the signature, or NULL if no signature is provided.
- All reserved fields must be zeroed.

`FS_IOC_ENABLE_VERITY` causes the filesystem to build a Merkle tree for the file and persist it to a filesystem-specific location associated with the file, then mark the file as a verity file. This ioctl may take a long time to execute on large files, and it is interruptible by fatal signals.

`FS_IOC_ENABLE_VERITY` checks for write access to the inode. However, it must be executed on an `O_RDONLY` file descriptor and no processes can have the file open for writing. Attempts to open the file for writing while this ioctl is executing will fail with `ETXTBSY`. (This is necessary to guarantee that no writable file descriptors will exist after verity is enabled, and to guarantee that the file's contents are stable while the Merkle tree is being built over it.)

On success, `FS_IOC_ENABLE_VERITY` returns 0, and the file becomes a verity file. On failure (including the case of interruption by a fatal signal), no changes are made to the file.

`FS_IOC_ENABLE_VERITY` can fail with the following errors:

- `EACCES`: the process does not have write access to the file
- `EBADMSG`: the signature is malformed
- `EBUSY`: this ioctl is already running on the file
- `EEXIST`: the file already has verity enabled
- `EFAULT`: the caller provided inaccessible memory
- `EINTR`: the operation was interrupted by a fatal signal
- `EINVAL`: unsupported version, hash algorithm, or block size; or reserved bits are set; or the file descriptor refers to neither a regular file nor a directory.
- `EISDIR`: the file descriptor refers to a directory
- `EKEYREJECTED`: the signature doesn't match the file
- `EMSGSIZE`: the salt or signature is too long
- `ENOKEY`: the fs-verity keyring doesn't contain the certificate needed to verify the signature
- `ENOPKG`: fs-verity recognizes the hash algorithm, but it's not available in the kernel's crypto API as currently configured (e.g. for SHA-512, missing `CONFIG_CRYPT_SHA512`).
- `ENOTTY`: this type of filesystem does not implement fs-verity

- EOPNOTSUPP: the kernel was not configured with fs-verity support; or the filesystem superblock has not had the ‘verity’ feature enabled on it; or the filesystem does not support fs-verity on this file. (See Filesystem support.)
- EPERM: the file is append-only; or, a signature is required and one was not provided.
- EROFS: the filesystem is read-only
- ETXTBSY: someone has the file open for writing. This can be the caller’s file descriptor, another open file descriptor, or the file reference held by a writable memory map.

FS_IOC_MEASURE_VERITY

The `FS_IOC_MEASURE_VERITY` ioctl retrieves the measurement of a verity file. The file measurement is a digest that cryptographically identifies the file contents that are being enforced on reads.

This ioctl takes in a pointer to a variable-length structure:

```
struct fsverity_digest {
    __u16 digest_algorithm;
    __u16 digest_size; /* input/output */
    __u8 digest[];
};
```

`digest_size` is an input/output field. On input, it must be initialized to the number of bytes allocated for the variable-length `digest` field.

On success, 0 is returned and the kernel fills in the structure as follows:

- `digest_algorithm` will be the hash algorithm used for the file measurement. It will match `fsverity_enable_arg::hash_algorithm`.
- `digest_size` will be the size of the digest in bytes, e.g. 32 for SHA-256. (This can be redundant with `digest_algorithm`.)
- `digest` will be the actual bytes of the digest.

`FS_IOC_MEASURE_VERITY` is guaranteed to execute in constant time, regardless of the size of the file.

`FS_IOC_MEASURE_VERITY` can fail with the following errors:

- EFAULT: the caller provided inaccessible memory
- ENODATA: the file is not a verity file
- ENOTTY: this type of filesystem does not implement fs-verity
- EOPNOTSUPP: the kernel was not configured with fs-verity support, or the filesystem superblock has not had the ‘verity’ feature enabled on it. (See Filesystem support.)
- EOVERFLOW: the digest is longer than the specified `digest_size` bytes. Try providing a larger buffer.

FS_IOC_GETFLAGS

The existing ioctl `FS_IOC_GETFLAGS` (which isn't specific to fs-verity) can also be used to check whether a file has fs-verity enabled or not. To do so, check for `FS_VERITY_FL` (0x00100000) in the returned flags.

The verity flag is not settable via `FS_IOC_SETFLAGS`. You must use `FS_IOC_ENABLE_VERITY` instead, since parameters must be provided.

statx

Since Linux v5.5, the `statx()` system call sets `STATX_ATTR_VERITY` if the file has fs-verity enabled. This can perform better than `FS_IOC_GETFLAGS` and `FS_IOC_MEASURE_VERITY` because it doesn't require opening the file, and opening verity files can be expensive.

2.3.4 Accessing verity files

Applications can transparently access a verity file just like a non-verity one, with the following exceptions:

- Verity files are readonly. They cannot be opened for writing or `truncate()`, even if the file mode bits allow it. Attempts to do one of these things will fail with `EPERM`. However, changes to metadata such as owner, mode, timestamps, and xattrs are still allowed, since these are not measured by fs-verity. Verity files can also still be renamed, deleted, and linked to.
- Direct I/O is not supported on verity files. Attempts to use direct I/O on such files will fall back to buffered I/O.
- DAX (Direct Access) is not supported on verity files, because this would circumvent the data verification.
- Reads of data that doesn't match the verity Merkle tree will fail with `EIO` (for `read()`) or `SIGBUS` (for `mmap()` reads).
- If the `sysctl "fs.verity.require_signatures"` is set to 1 and the file's verity measurement is not signed by a key in the fs-verity keyring, then opening the file will fail. See Built-in signature verification.

Direct access to the Merkle tree is not supported. Therefore, if a verity file is copied, or is backed up and restored, then it will lose its “verity”-ness. fs-verity is primarily meant for files like executables that are managed by a package manager.

2.3.5 File measurement computation

This section describes how fs-verity hashes the file contents using a Merkle tree to produce the “file measurement” which cryptographically identifies the file contents. This algorithm is the same for all filesystems that support fs-verity.

Userspace only needs to be aware of this algorithm if it needs to compute the file measurement itself, e.g. in order to sign the file.

Merkle tree

The file contents is divided into blocks, where the block size is configurable but is usually 4096 bytes. The end of the last block is zero-padded if needed. Each block is then hashed, producing the first level of hashes. Then, the hashes in this first level are grouped into ‘blocksize’ -byte blocks (zero-padding the ends as needed) and these blocks are hashed, producing the second level of hashes. This proceeds up the tree until only a single block remains. The hash of this block is the “Merkle tree root hash” .

If the file fits in one block and is nonempty, then the “Merkle tree root hash” is simply the hash of the single data block. If the file is empty, then the “Merkle tree root hash” is all zeroes.

The “blocks” here are not necessarily the same as “filesystem blocks” .

If a salt was specified, then it’ s zero-padded to the closest multiple of the input size of the hash algorithm’ s compression function, e.g. 64 bytes for SHA-256 or 128 bytes for SHA-512. The padded salt is prepended to every data or Merkle tree block that is hashed.

The purpose of the block padding is to cause every hash to be taken over the same amount of data, which simplifies the implementation and keeps open more possibilities for hardware acceleration. The purpose of the salt padding is to make the salting “free” when the salted hash state is precomputed, then imported for each hash.

Example: in the recommended configuration of SHA-256 and 4K blocks, 128 hash values fit in each block. Thus, each level of the Merkle tree is approximately 128 times smaller than the previous, and for large files the Merkle tree’ s size converges to approximately 1/127 of the original file size. However, for small files, the padding is significant, making the space overhead proportionally more.

fs-verity descriptor

By itself, the Merkle tree root hash is ambiguous. For example, it can’ t distinguish a large file from a small second file whose data is exactly the top-level hash block of the first file. Ambiguities also arise from the convention of padding to the next block boundary.

To solve this problem, the verity file measurement is actually computed as a hash of the following structure, which contains the Merkle tree root hash as well as other fields such as the file size:

```

struct fsverity_descriptor {
    __u8 version;           /* must be 1 */
    __u8 hash_algorithm;   /* Merkle tree hash algorithm */
    __u8 log_blocksize;    /* log2 of size of data and tree blocks */
    __u8 salt_size;        /* size of salt in bytes; 0 if none */
    __le32 sig_size;       /* must be 0 */
    __le64 data_size;      /* size of file the Merkle tree is built
↳over */
    __u8 root_hash[64];    /* Merkle tree root hash */
    __u8 salt[32];         /* salt prepended to each hashed block */
    __u8 __reserved[144]; /* must be 0's */
};

```

Note that the `sig_size` field must be set to 0 for the purpose of computing the file measurement, even if a signature was provided (or will be provided) to `FS_IOC_ENABLE_VERITY`.

2.3.6 Built-in signature verification

With `CONFIG_FS_VERITY_BUILTIN_SIGNATURES=y`, fs-verity supports putting a portion of an authentication policy (see Use cases) in the kernel. Specifically, it adds support for:

1. At fs-verity module initialization time, a keyring “.fs-verity” is created. The root user can add trusted X.509 certificates to this keyring using the `add_key()` system call, then (when done) optionally use `keyctl_restrict_keyring()` to prevent additional certificates from being added.
2. `FS_IOC_ENABLE_VERITY` accepts a pointer to a PKCS#7 formatted detached signature in DER format of the file measurement. On success, this signature is persisted alongside the Merkle tree. Then, any time the file is opened, the kernel will verify the file’s actual measurement against this signature, using the certificates in the “.fs-verity” keyring.
3. A new sysctl “fs.verity.require_signatures” is made available. When set to 1, the kernel requires that all verity files have a correctly signed file measurement as described in (2).

File measurements must be signed in the following format, which is similar to the structure used by `FS_IOC_MEASURE_VERITY`:

```

struct fsverity_signed_digest {
    char magic[8];           /* must be "FSVerity" */
    __le16 digest_algorithm;
    __le16 digest_size;
    __u8 digest[];
};

```

fs-verity’s built-in signature verification support is meant as a relatively simple mechanism that can be used to provide some level of authenticity protection for verity files, as an alternative to doing the signature verification in userspace or using IMA-appraisal. However, with this mechanism, userspace programs still need to check that the verity bit is set, and there is no protection against verity files being swapped around.

2.3.7 Filesystem support

fs-verity is currently supported by the ext4 and f2fs filesystems. The `CONFIG_FS_VERITY` kconfig option must be enabled to use fs-verity on either filesystem.

`include/linux/fsverity.h` declares the interface between the `fs/verity/` support layer and filesystems. Briefly, filesystems must provide an `fsverity_operations` structure that provides methods to read and write the verity metadata to a filesystem-specific location, including the Merkle tree blocks and `fsverity_descriptor`. Filesystems must also call functions in `fs/verity/` at certain times, such as when a file is opened or when pages have been read into the pagecache. (See Verifying data.)

ext4

ext4 supports fs-verity since Linux v5.4 and e2fsprogs v1.45.2.

To create verity files on an ext4 filesystem, the filesystem must have been formatted with `-O verity` or had `tune2fs -O verity` run on it. “verity” is an `RO_COMPAT` filesystem feature, so once set, old kernels will only be able to mount the filesystem readonly, and old versions of `e2fsck` will be unable to check the filesystem. Moreover, currently ext4 only supports mounting a filesystem with the “verity” feature when its block size is equal to `PAGE_SIZE` (often 4096 bytes).

ext4 sets the `EXT4_VERITY_FL` on-disk inode flag on verity files. It can only be set by `FS_IOC_ENABLE_VERITY`, and it cannot be cleared.

ext4 also supports encryption, which can be used simultaneously with fs-verity. In this case, the plaintext data is verified rather than the ciphertext. This is necessary in order to make the file measurement meaningful, since every file is encrypted differently.

ext4 stores the verity metadata (Merkle tree and `fsverity_descriptor`) past the end of the file, starting at the first 64K boundary beyond `i_size`. This approach works because (a) verity files are readonly, and (b) pages fully beyond `i_size` aren't visible to userspace but can be read/written internally by ext4 with only some relatively small changes to ext4. This approach avoids having to depend on the `EA_INODE` feature and on rearchitecting ext4's `xattr` support to support paging multi-gigabyte `xattrs` into memory, and to support encrypting `xattrs`. Note that the verity metadata must be encrypted when the file is, since it contains hashes of the plaintext data.

Currently, ext4 verity only supports the case where the Merkle tree block size, filesystem block size, and page size are all the same. It also only supports extent-based files.

f2fs

f2fs supports fs-verity since Linux v5.4 and f2fs-tools v1.11.0.

To create verity files on an f2fs filesystem, the filesystem must have been formatted with `-O verity`.

f2fs sets the `FADVISE_VERITY_BIT` on-disk inode flag on verity files. It can only be set by `FS_IOC_ENABLE_VERITY`, and it cannot be cleared.

Like ext4, f2fs stores the verity metadata (Merkle tree and `fsverity_descriptor`) past the end of the file, starting at the first 64K boundary beyond `i_size`. See explanation for ext4 above. Moreover, f2fs supports at most 4096 bytes of xattr entries per inode which wouldn't be enough for even a single Merkle tree block.

Currently, f2fs verity only supports a Merkle tree block size of 4096. Also, f2fs doesn't support enabling verity on files that currently have atomic or volatile writes pending.

2.3.8 Implementation details

Verifying data

fs-verity ensures that all reads of a verity file's data are verified, regardless of which syscall is used to do the read (e.g. `mmap()`, `read()`, `pread()`) and regardless of whether it's the first read or a later read (unless the later read can return cached data that was already verified). Below, we describe how filesystems implement this.

Pagecache

For filesystems using Linux's pagecache, the `->readpage()` and `->readpages()` methods must be modified to verify pages before they are marked `Uptodate`. Merely hooking `->read_iter()` would be insufficient, since `->read_iter()` is not used for memory maps.

Therefore, `fs/verity/` provides a function `fsverity_verify_page()` which verifies a page that has been read into the pagecache of a verity inode, but is still locked and not `Uptodate`, so it's not yet readable by userspace. As needed to do the verification, `fsverity_verify_page()` will call back into the filesystem to read Merkle tree pages via `fsverity_operations::read_merkle_tree_page()`.

`fsverity_verify_page()` returns false if verification failed; in this case, the filesystem must not set the page `Uptodate`. Following this, as per the usual Linux pagecache behavior, attempts by userspace to `read()` from the part of the file containing the page will fail with `EIO`, and accesses to the page within a memory map will raise `SIGBUS`.

`fsverity_verify_page()` currently only supports the case where the Merkle tree block size is equal to `PAGE_SIZE` (often 4096 bytes).

In principle, `fsverity_verify_page()` verifies the entire path in the Merkle tree from the data page to the root hash. However, for efficiency the filesystem may cache

the hash pages. Therefore, `fsverity_verify_page()` only ascends the tree reading hash pages until an already-verified hash page is seen, as indicated by the `PageChecked` bit being set. It then verifies the path to that page.

This optimization, which is also used by `dm-verity`, results in excellent sequential read performance. This is because usually (e.g. 127 in 128 times for 4K blocks and SHA-256) the hash page from the bottom level of the tree will already be cached and checked from reading a previous data page. However, random reads perform worse.

Block device based filesystems

Block device based filesystems (e.g. `ext4` and `f2fs`) in Linux also use the pagecache, so the above subsection applies too. However, they also usually read many pages from a file at once, grouped into a structure called a “bio”. To make it easier for these types of filesystems to support `fs-verity`, `fs/verity/` also provides a function `fsverity_verify_bio()` which verifies all pages in a bio.

`ext4` and `f2fs` also support encryption. If a verity file is also encrypted, the pages must be decrypted before being verified. To support this, these filesystems allocate a “post-read context” for each bio and store it in `->bi_private`:

```
struct bio_post_read_ctx {
    struct bio *bio;
    struct work_struct work;
    unsigned int cur_step;
    unsigned int enabled_steps;
};
```

`enabled_steps` is a bitmask that specifies whether decryption, verity, or both is enabled. After the bio completes, for each needed postprocessing step the filesystem enqueues the `bio_post_read_ctx` on a workqueue, and then the workqueue work does the decryption or verification. Finally, pages where no decryption or verity error occurred are marked `Uptodate`, and the pages are unlocked.

Files on `ext4` and `f2fs` may contain holes. Normally, `->readpages()` simply zeroes holes and sets the corresponding pages `Uptodate`; no bios are issued. To prevent this case from bypassing `fs-verity`, these filesystems use `fsverity_verify_page()` to verify hole pages.

`ext4` and `f2fs` disable direct I/O on verity files, since otherwise direct I/O would bypass `fs-verity`. (They also do the same for encrypted files.)

2.3.9 Userspace utility

This document focuses on the kernel, but a userspace utility for `fs-verity` can be found at:

<https://git.kernel.org/pub/scm/linux/kernel/git/ebiggers/fsverity-utils.git>

See the `README.md` file in the `fsverity-utils` source tree for details, including examples of setting up `fs-verity` protected files.

2.3.10 Tests

To test fs-verity, use `xfstests`. For example, using `kvm-xfstests`:

```
kvm-xfstests -c ext4,f2fs -g verity
```

2.3.11 FAQ

This section answers frequently asked questions about fs-verity that weren't already directly answered in other parts of this document.

Q Why isn't fs-verity part of IMA?

A fs-verity and IMA (Integrity Measurement Architecture) have different focuses. fs-verity is a filesystem-level mechanism for hashing individual files using a Merkle tree. In contrast, IMA specifies a system-wide policy that specifies which files are hashed and what to do with those hashes, such as log them, authenticate them, or add them to a measurement list.

IMA is planned to support the fs-verity hashing mechanism as an alternative to doing full file hashes, for people who want the performance and security benefits of the Merkle tree based hash. But it doesn't make sense to force all uses of fs-verity to be through IMA. As a standalone filesystem feature, fs-verity already meets many users' needs, and it's testable like other filesystem features e.g. with `xfstests`.

Q Isn't fs-verity useless because the attacker can just modify the hashes in the Merkle tree, which is stored on-disk?

A To verify the authenticity of an fs-verity file you must verify the authenticity of the "file measurement", which is basically the root hash of the Merkle tree. See Use cases.

Q Isn't fs-verity useless because the attacker can just replace a verity file with a non-verity one?

A See Use cases. In the initial use case, it's really trusted userspace code that authenticates the files; fs-verity is just a tool to do this job efficiently and securely. The trusted userspace code will consider non-verity files to be inauthentic.

Q Why does the Merkle tree need to be stored on-disk? Couldn't you store just the root hash?

A If the Merkle tree wasn't stored on-disk, then you'd have to compute the entire tree when the file is first accessed, even if just one byte is being read. This is a fundamental consequence of how Merkle tree hashing works. To verify a leaf node, you need to verify the whole path to the root hash, including the root node (the thing which the root hash is a hash of). But if the root node isn't stored on-disk, you have to compute it by hashing its children, and so on until you've actually hashed the entire file.

That defeats most of the point of doing a Merkle tree-based hash, since if you have to hash the whole file ahead of time anyway, then you could simply do `sha256(file)` instead. That would be much simpler, and a bit faster too.

It's true that an in-memory Merkle tree could still provide the advantage of verification on every read rather than just on the first read. However, it would be inefficient because every time a hash page gets evicted (you can't pin the entire Merkle tree into memory, since it may be very large), in order to restore it you again need to hash everything below it in the tree. This again defeats most of the point of doing a Merkle tree-based hash, since a single block read could trigger re-hashing gigabytes of data.

Q But couldn't you store just the leaf nodes and compute the rest?

A See previous answer; this really just moves up one level, since one could alternatively interpret the data blocks as being the leaf nodes of the Merkle tree. It's true that the tree can be computed much faster if the leaf level is stored rather than just the data, but that's only because each level is less than 1% the size of the level below (assuming the recommended settings of SHA-256 and 4K blocks). For the exact same reason, by storing "just the leaf nodes" you'd already be storing over 99% of the tree, so you might as well simply store the whole tree.

Q Can the Merkle tree be built ahead of time, e.g. distributed as part of a package that is installed to many computers?

A This isn't currently supported. It was part of the original design, but was removed to simplify the kernel UAPI and because it wasn't a critical use case. Files are usually installed once and used many times, and cryptographic hashing is somewhat fast on most modern processors.

Q Why doesn't fs-verity support writes?

A Write support would be very difficult and would require a completely different design, so it's well outside the scope of fs-verity. Write support would require:

- A way to maintain consistency between the data and hashes, including all levels of hashes, since corruption after a crash (especially of potentially the entire file!) is unacceptable. The main options for solving this are data journalling, copy-on-write, and log-structured volume. But it's very hard to retrofit existing filesystems with new consistency mechanisms. Data journalling is available on ext4, but is very slow.
- Rebuilding the the Merkle tree after every write, which would be extremely inefficient. Alternatively, a different authenticated dictionary structure such as an "authenticated skiplist" could be used. However, this would be far more complex.

Compare it to dm-verity vs. dm-integrity. dm-verity is very simple: the kernel just verifies read-only data against a read-only Merkle

tree. In contrast, dm-integrity supports writes but is slow, is much more complex, and doesn't actually support full-device authentication since it authenticates each sector independently, i.e. there is no "root hash". It doesn't really make sense for the same device-mapper target to support these two very different cases; the same applies to fs-verity.

- Q** Since verity files are immutable, why isn't the immutable bit set?
- A** The existing "immutable" bit (`FS_IMMUTABLE_FL`) already has a specific set of semantics which not only make the file contents read-only, but also prevent the file from being deleted, renamed, linked to, or having its owner or mode changed. These extra properties are unwanted for fs-verity, so reusing the immutable bit isn't appropriate.
- Q** Why does the API use `ioctl`s instead of `setxattr()` and `getxattr()`?
- A** Abusing the `xattr` interface for basically arbitrary syscalls is heavily frowned upon by most of the Linux filesystem developers. An `xattr` should really just be an `xattr` on-disk, not an API to e.g. magically trigger construction of a Merkle tree.
- Q** Does fs-verity support remote filesystems?
- A** Only `ext4` and `f2fs` support is implemented currently, but in principle any filesystem that can store per-file verity metadata can support fs-verity, regardless of whether it's local or remote. Some filesystems may have fewer options of where to store the verity metadata; one possibility is to store it past the end of the file and "hide" it from userspace by manipulating `i_size`. The data verification functions provided by `fs/verity/` also assume that the filesystem uses the Linux pagecache, but both local and remote filesystems normally do so.
- Q** Why is anything filesystem-specific at all? Shouldn't fs-verity be implemented entirely at the VFS level?
- A** There are many reasons why this is not possible or would be very difficult, including the following:
- To prevent bypassing verification, pages must not be marked `Uptodate` until they've been verified. Currently, each filesystem is responsible for marking pages `Uptodate` via `->readpages()`. Therefore, currently it's not possible for the VFS to do the verification on its own. Changing this would require significant changes to the VFS and all filesystems.
 - It would require defining a filesystem-independent way to store the verity metadata. Extended attributes don't work for this because (a) the Merkle tree may be gigabytes, but many filesystems assume that all `xattrs` fit into a single 4K filesystem block, and (b) `ext4` and `f2fs` encryption doesn't encrypt `xattrs`, yet the Merkle tree must be encrypted when the file contents are, because it stores hashes of the plaintext file contents.
- So the verity metadata would have to be stored in an actual file. Using a separate file would be very ugly, since the metadata is

fundamentally part of the file to be protected, and it could cause problems where users could delete the real file but not the metadata file or vice versa. On the other hand, having it be in the same file would break applications unless filesystems' notion of `i_size` were divorced from the VFS' s, which would be complex and require changes to all filesystems.

- It's desirable that `FS_IOC_ENABLE_VERITY` uses the filesystem's transaction mechanism so that either the file ends up with verity enabled, or no changes were made. Allowing intermediate states to occur after a crash may cause problems.

Documentation for filesystem implementations.

3.1 v9fs: Plan 9 Resource Sharing for Linux

3.1.1 About

v9fs is a Unix implementation of the Plan 9 9p remote filesystem protocol.

This software was originally developed by Ron Minnich <rminnich@sandia.gov> and Maya Gokhale. Additional development by Greg Watson <gwatson@lanl.gov> and most recently Eric Van Hensbergen <ericvh@gmail.com>, Latchesar Ionkov <lucho@ionkov.net> and Russ Cox <rsc@swtch.com>.

The best detailed explanation of the Linux implementation and applications of the 9p client is available in the form of a USENIX paper:

<http://www.usenix.org/events/usenix05/tech/freenix/hensbergen.html>

Other applications are described in the following papers:

- XCPU & Clustering <http://xcpu.org/papers/xcpu-talk.pdf>
- KVMFS: control file system for KVM <http://xcpu.org/papers/kvmfs.pdf>
- CellFS: A New Programming Model for the Cell BE <http://xcpu.org/papers/cellfs-talk.pdf>
- PROSE I/O: Using 9p to enable Application Partitions http://plan9.escet.urjc.es/iwp9/cready/PROSE_iwp9_2006.pdf
- VirtFS: A Virtualization Aware File System pass-through <http://goo.gl/3WPDg>

3.1.2 Usage

For remote file server:

```
mount -t 9p 10.10.1.2 /mnt/9
```

For Plan 9 From User Space applications (<http://swtch.com/plan9>):

```
mount -t 9p `namespace`/acme /mnt/9 -o trans=unix,uname=$USER
```

For server running on QEMU host with virtio transport:

```
mount -t 9p -o trans=virtio <mount_tag> /mnt/9
```

where `mount_tag` is the tag associated by the server to each of the exported mount points. Each 9P export is seen by the client as a virtio device with an associated “`mount_tag`” property. Available mount tags can be seen by reading `/sys/bus/virtio/drivers/9pnet_virtio/virtio<n>/mount_tag` files.

3.1.3 Options

trans=name	<p>select an alternative transport. Valid options are currently:</p> <table border="1" data-bbox="874 389 1246 1084"> <tr> <td data-bbox="874 389 954 501">unix</td> <td data-bbox="954 389 1246 501">specifying a named pipe mount point</td> </tr> <tr> <td data-bbox="874 501 954 613">tcp</td> <td data-bbox="954 501 1246 613">specifying a normal TCP/IP connection</td> </tr> <tr> <td data-bbox="874 613 954 757">fd</td> <td data-bbox="954 613 1246 757">used passed file descriptors for connection (see rfdno and wfdno)</td> </tr> <tr> <td data-bbox="874 757 954 972">virtio</td> <td data-bbox="954 757 1246 972">connect to the next virtio channel available (from QEMU with trans_virtio module)</td> </tr> <tr> <td data-bbox="874 972 954 1084">rdma</td> <td data-bbox="954 972 1246 1084">connect to a specified RDMA channel</td> </tr> </table>	unix	specifying a named pipe mount point	tcp	specifying a normal TCP/IP connection	fd	used passed file descriptors for connection (see rfdno and wfdno)	virtio	connect to the next virtio channel available (from QEMU with trans_virtio module)	rdma	connect to a specified RDMA channel
unix	specifying a named pipe mount point										
tcp	specifying a normal TCP/IP connection										
fd	used passed file descriptors for connection (see rfdno and wfdno)										
virtio	connect to the next virtio channel available (from QEMU with trans_virtio module)										
rdma	connect to a specified RDMA channel										
uname=name	user name to attempt mount as on the remote server. The server may override or ignore this value. Certain user names may require authentication.										
aname=name	aname specifies the file tree to access when the server is offering several exported file systems.										
cache=mode	<p>specifies a caching policy. By default, no caches are used.</p> <p>none default no cache policy, metadata and data alike are synchronous.</p> <p>loose no attempts are made at consistency, intended for exclusive, read-only mounts</p> <p>fscache use FS-Cache for a persistent, read-only cache backend.</p> <p>mmap minimal cache that is only used for read-write mmap. Nothing</p>										
336	<p>Chapter 3. Filesystems</p> <p>else cache=none</p>										
debug=n	specifies debug level. The debug										

3.1.4 Behavior

This section aims at describing 9p ‘quirks’ that can be different from a local filesystem behaviors.

- Setting `O_NONBLOCK` on a file will make client reads return as early as the server returns some data instead of trying to fill the read buffer with the requested amount of bytes or end of file is reached.

3.1.5 Resources

Protocol specifications are maintained on github: <http://ericvh.github.com/9p-rfc/>
9p client and server implementations are listed on <http://9p.cat-v.org/implementations>

A 9p2000.L server is being developed by LLNL and can be found at <http://code.google.com/p/diod/>

There are user and developer mailing lists available through the v9fs project on sourceforge (<http://sourceforge.net/projects/v9fs>).

News and other information is maintained on a Wiki. (<http://sf.net/apps/mediawiki/v9fs/index.php>).

Bug reports are best issued via the mailing list.

For more information on the Plan 9 Operating System check out <http://plan9.bell-labs.com/plan9>

For information on Plan 9 from User Space (Plan 9 applications and libraries ported to Linux/BSD/OSX/etc) check out <https://9fans.github.io/plan9port/>

3.2 Acorn Disc Filing System - ADFS

3.2.1 Filesystems supported by ADFS

The ADFS module supports the following Filecore formats which have:

- new maps
- new directories or big directories

In terms of the named formats, this means we support:

- E and E+, with or without boot block
- F and F+

We fully support reading files from these filesystems, and writing to existing files within their existing allocation. Essentially, we do not support changing any of the filesystem metadata.

This is intended to support loopback mounted Linux native filesystems on a RISC OS Filecore filesystem, but will allow the data within files to be changed.

If write support (ADFS_FS_RW) is configured, we allow rudimentary directory updates, specifically updating the access mode and timestamp.

3.2.2 Mount options for ADFS

uid=nnn	All files in the partition will be owned by user id nnn. Default 0 (root).
gid=nnn	All files in the partition will be in group nnn. Default 0 (root).
own-mask=nnnn	The permission mask for ADFS 'owner' permissions will be nnnn. Default 0700.
oth-mask=nnnn	The permission mask for ADFS 'other' permissions will be nnnn. Default 0077.
ftsuffix=n	When ftsuffix=0, no file type suffix will be applied. When ftsuffix=1, a hexadecimal suffix corresponding to the RISC OS file type will be added. Default 0.

3.2.3 Mapping of ADFS permissions to Linux permissions

ADFS permissions consist of the following:

- Owner read
- Owner write
- Other read
- Other write

(In older versions, an 'execute' permission did exist, but this does not hold the same meaning as the Linux 'execute' permission and is now obsolete).

The mapping is performed as follows:

Owner read	-> -r--r--r--
Owner write	-> --w--w---w
Owner read and filetype UnixExec	-> ---x--x--x
These are then masked by ownmask, eg 700	-> -rwx-----
Possible owner mode permissions	-> -rwx-----
Other read	-> -r--r--r--
Other write	-> --w--w---w
Other read and filetype UnixExec	-> ---x--x--x
These are then masked by othmask, eg 077	-> ----rwxrwx
Possible other mode permissions	-> ----rwxrwx

Hence, with the default masks, if a file is owner read/write, and not a UnixExec filetype, then the permissions will be:

```
-rw-----
```

However, if the masks were ownmask=0770,othmask=0007, then this would be modified to:

```
-rw-rw----
```

There is no restriction on what you can do with these masks. You may wish that either read bits give read access to the file for all, but keep the default write protection (ownmask=0755,othmask=0577):

```
-rw-r--r--
```

You can therefore tailor the permission translation to whatever you desire the permissions should be under Linux.

3.2.4 RISC OS file type suffix

RISC OS file types are stored in bits 19..8 of the file load address.

To enable non-RISC OS systems to be used to store files without losing file type information, a file naming convention was devised (initially for use with NFS) such that a hexadecimal suffix of the form ,xyz denoted the file type: e.g. BasicFile,ffb is a BASIC (0xffb) file. This naming convention is now also used by RISC OS emulators such as RPCEmu.

Mounting an ADFS disc with option ftsuffix=1 will cause appropriate file type suffixes to be appended to file names read from a directory. If the ftsuffix option is zero or omitted, no file type suffixes will be added.

3.3 Overview of Amiga Filesystems

Not all varieties of the Amiga filesystems are supported for reading and writing. The Amiga currently knows six different filesystems:

DOS0	The old or original filesystem, not really suited for hard disks and normally not used on them, either. Supported read/write.
DOS1	The original Fast File System. Supported read/write.
DOS2	The old “international” filesystem. International means that a bug has been fixed so that accented (“international”) letters in file names are case-insensitive, as they ought to be. Supported read/write.
DOS3	The “international” Fast File System. Supported read/write.
DOS4	The original filesystem with directory cache. The directory cache speeds up directory accesses on floppies considerably, but slows down file creation/deletion. Doesn’ t make much sense on hard disks. Supported read only.
DOS5	The Fast File System with directory cache. Supported read only.

All of the above filesystems allow block sizes from 512 to 32K bytes. Supported block sizes are: 512, 1024, 2048 and 4096 bytes. Larger blocks speed up almost everything at the expense of wasted disk space. The speed gain above 4K seems not really worth the price, so you don’ t lose too much here, either.

The muFS (multi user File System) equivalents of the above file systems are supported, too.

3.3.1 Mount options for the AFFS

protect If this option is set, the protection bits cannot be altered.

setuid[=uid] This sets the owner of all files and directories in the file system to uid or the uid of the current user, respectively.

setgid[=gid] Same as above, but for gid.

mode=mode Sets the mode flags to the given (octal) value, regardless of the original permissions. Directories will get an x permission if the corresponding r bit is set. This is useful since most of the plain AmigaOS files will map to 600.

nofilenametruncate The file system will return an error when filename exceeds standard maximum filename length (30 characters).

reserved=num Sets the number of reserved blocks at the start of the partition to num. You should never need this option. Default is 2.

root=block Sets the block number of the root block. This should never be necessary.

bs=blksize Sets the blocksize to blksize. Valid block sizes are 512, 1024, 2048 and 4096. Like the root option, this should never be necessary, as the affs can figure it out itself.

quiet The file system will not return an error for disallowed mode changes.

verbose The volume name, file system type and block size will be written to the syslog when the filesystem is mounted.

mufs The filesystem is really a muFS, also it doesn't identify itself as one. This option is necessary if the filesystem wasn't formatted as muFS, but is used as one.

prefix=path Path will be prefixed to every absolute path name of symbolic links on an AFFS partition. Default = `"/`. (See below.)

volume=name When symbolic links with an absolute path are created on an AFFS partition, name will be prepended as the volume name. Default = `""` (empty string). (See below.)

3.3.2 Handling of the Users/Groups and protection flags

Amiga -> Linux:

The Amiga protection flags RWEDRWEDHSPARWED are handled as follows:

- R maps to r for user, group and others. On directories, R implies x.
- If both W and D are allowed, w will be set.
- E maps to x.
- H and P are always retained and ignored under Linux.
- A is always reset when a file is written to.

User id and group id will be used unless set[gu]id are given as mount options. Since most of the Amiga file systems are single user systems they will be owned by root. The root directory (the mount point) of the Amiga filesystem will be owned by the user who actually mounts the filesystem (the root directory doesn't have uid/gid fields).

Linux -> Amiga:

The Linux rwxrwxrwx file mode is handled as follows:

- r permission will set R for user, group and others.
- w permission will set W and D for user, group and others.
- x permission of the user will set E for plain files.
- All other flags (suid, sgid, ...) are ignored and will not be retained.

Newly created files and directories will get the user and group ID of the current user and a mode according to the umask.

3.3.3 Symbolic links

Although the Amiga and Linux file systems resemble each other, there are some, not always subtle, differences. One of them becomes apparent with symbolic links. While Linux has a file system with exactly one root directory, the Amiga has a separate root directory for each file system (for example, partition, floppy disk, ...). With the Amiga, these entities are called "volumes". They have symbolic names which can be used to access them. Thus, symbolic links can point to a different volume. AFS turns the volume name into a directory name and prepends the prefix path (see prefix option) to it.

Example: You mount all your Amiga partitions under /amiga/<volume> (where <volume> is the name of the volume), and you give the option "prefix=/amiga/" when mounting all your AFS partitions. (They might be "User", "WB" and "Graphics", the mount points /amiga/User, /amiga/WB and /amiga/Graphics). A symbolic link referring to "User:sc/include/dos/dos.h" will be followed to "/amiga/User/sc/include/dos/dos.h".

3.3.4 Examples

Command line:

```
mount Archive/Amiga/Workbench3.1.adf /mnt -t affs -o loop,verbose
mount /dev/sda3 /Amiga -t affs
```

/etc/fstab entry:

```
/dev/sdb5 /amiga/Workbench affs noauto,user,exec,verbose 0 0
```

3.3.5 IMPORTANT NOTE

If you boot Windows 95 (don't know about 3.x, 98 and NT) while you have an Amiga harddisk connected to your PC, it will overwrite the bytes 0x00dc..0x00df of block 0 with garbage, thus invalidating the Rigid Disk Block. Sheer luck has it that this is an unused area of the RDB, so only the checksum doesn't match anymore. Linux will ignore this garbage and recognize the RDB anyway, but before you connect that drive to your Amiga again, you must restore or repair your RDB. So please do make a backup copy of it before booting Windows!

If the damage is already done, the following should fix the RDB (where <disk> is the device name).

DO AT YOUR OWN RISK:

```
dd if=/dev/<disk> of=rdb.tmp count=1
cp rdb.tmp rdb.fixed
dd if=/dev/zero of=rdb.fixed bs=1 seek=220 count=4
dd if=rdb.fixed of=/dev/<disk>
```

3.3.6 Bugs, Restrictions, Caveats

Quite a few things may not work as advertised. Not everything is tested, though several hundred MB have been read and written using this fs. For a most up-to-date list of bugs please consult fs/affs/Changes.

By default, filenames are truncated to 30 characters without warning. 'no-file-nametruncate' mount option can change that behavior.

Case is ignored by the affs in filename matching, but Linux shells do care about the case. Example (with /wb being an affs mounted fs):

```
rm /wb/WRONGCASE
```

will remove /mnt/wrongcase, but:

```
rm /wb/WR*
```

will not since the names are matched by the shell.

The block allocation is designed for hard disk partitions. If more than 1 process writes to a (small) diskette, the blocks are allocated in an ugly way (but the real AFFS doesn't do much better). This is also true when space gets tight.

You cannot execute programs on an OFS (Old File System), since the program files cannot be memory mapped due to the 488 byte blocks. For the same reason you cannot mount an image on such a filesystem via the loopback device.

The bitmap valid flag in the root block may not be accurate when the system crashes while an affs partition is mounted. There's currently no way to fix a garbled filesystem without an Amiga (disk validator) or manually (who would do this?). Maybe later.

If you mount affs partitions on system startup, you may want to tell fsck that the fs should not be checked (place a '0' in the sixth field of /etc/fstab).

It's not possible to read floppy disks with a normal PC or workstation due to an incompatibility with the Amiga floppy controller.

If you are interested in an Amiga Emulator for Linux, look at

<http://web.archive.org/web/%2E/http://www.freiburg.linux.de/~uae/>

3.4 kAFS: AFS FILESYSTEM

3.4.1 Overview

This filesystem provides a fairly simple secure AFS filesystem driver. It is under development and does not yet provide the full feature set. The features it does support include:

- (*) Security (currently only AFS kaserver and KerberosIV tickets).
- (*) File reading and writing.
- (*) Automounting.
- (*) Local caching (via fscache).

It does not yet support the following AFS features:

- (*) `pioctl()` system call.

3.4.2 Compilation

The filesystem should be enabled by turning on the kernel configuration options:

<code>CONFIG_AF_RXRPC</code>	- The RxRPC protocol transport
<code>CONFIG_RXKAD</code>	- The RxRPC Kerberos security handler
<code>CONFIG_AFS</code>	- The AFS filesystem

Additionally, the following can be turned on to aid debugging:

<code>CONFIG_AF_RXRPC_DEBUG</code>	- Permit AF_RXRPC debugging to be enabled
<code>CONFIG_AFS_DEBUG</code>	- Permit AFS debugging to be enabled

They permit the debugging messages to be turned on dynamically by manipulating the masks in the following files:

<code>/sys/module/af_rxrpc/parameters/debug</code>
<code>/sys/module/kafs/parameters/debug</code>

3.4.3 Usage

When inserting the driver modules the root cell must be specified along with a list of volume location server IP addresses:

```
modprobe rxrpc
modprobe kafs rootcell=cambridge.redhat.com:172.16.18.73:172.16.18.91
```

The first module is the AF_RXRPC network protocol driver. This provides the RxRPC remote operation protocol and may also be accessed from userspace. See:

Documentation/networking/rxrpc.rst

The second module is the kerberos RxRPC security driver, and the third module is the actual filesystem driver for the AFS filesystem.

Once the module has been loaded, more modules can be added by the following procedure:

```
echo add grand.central.org 18.9.48.14:128.2.203.61:130.237.48.87 >/proc/fs/
↪afs/cells
```

Where the parameters to the “add” command are the name of a cell and a list of volume location servers within that cell, with the latter separated by colons.

Filesystems can be mounted anywhere by commands similar to the following:

```
mount -t afs "%cambridge.redhat.com:root.afs." /afs
mount -t afs "#cambridge.redhat.com:root.cell." /afs/cambridge
mount -t afs "#root.afs." /afs
mount -t afs "#root.cell." /afs/cambridge
```

Where the initial character is either a hash or a percent symbol depending on whether you definitely want a R/W volume (percent) or whether you’d prefer a R/O volume, but are willing to use a R/W volume instead (hash).

The name of the volume can be suffixed with “.backup” or “.readonly” to specify connection to only volumes of those types.

The name of the cell is optional, and if not given during a mount, then the named volume will be looked up in the cell specified during modprobe.

Additional cells can be added through /proc (see later section).

3.4.4 Mountpoints

AFS has a concept of mountpoints. In AFS terms, these are specially formatted symbolic links (of the same form as the “device name” passed to mount). kAFS presents these to the user as directories that have a follow-link capability (ie: symbolic link semantics). If anyone attempts to access them, they will automatically cause the target volume to be mounted (if possible) on that site.

Automatically mounted filesystems will be automatically unmounted approximately twenty minutes after they were last used. Alternatively they can be unmounted directly with the umount() system call.

Manually unmounting an AFS volume will cause any idle submounts upon it to be culled first. If all are culled, then the requested volume will also be unmounted, otherwise error EBUSY will be returned.

This can be used by the administrator to attempt to unmount the whole AFS tree mounted on /afs in one go by doing:

```
umount /afs
```

3.4.5 Dynamic Root

A mount option is available to create a serverless mount that is only usable for dynamic lookup. Creating such a mount can be done by, for example:

```
mount -t afs none /afs -o dyn
```

This creates a mount that just has an empty directory at the root. Attempting to look up a name in this directory will cause a mountpoint to be created that looks up a cell of the same name, for example:

```
ls /afs/grand.central.org/
```

3.4.6 Proc Filesystem

The AFS modules creates a “/proc/fs/afs/” directory and populates it:

- (*) A “cells” file that lists cells currently known to the afs module and their usage counts:

```
[root@andromeda ~]# cat /proc/fs/afs/cells
USE NAME
 3 cambridge.redhat.com
```

- (*) A directory per cell that contains files that list volume location servers, volumes, and active servers known within that cell:

```
[root@andromeda ~]# cat /proc/fs/afs/cambridge.redhat.com/
↪servers
USE ADDR          STATE
 4 172.16.18.91    0
[root@andromeda ~]# cat /proc/fs/afs/cambridge.redhat.com/
↪vlservers
ADDRESS
172.16.18.91
[root@andromeda ~]# cat /proc/fs/afs/cambridge.redhat.com/
↪volumes
USE STT VLID[0]  VLID[1]  VLID[2]  NAME
 1 Val 20000000 20000001 20000002 root.afs
```

3.4.7 The Cell Database

The filesystem maintains an internal database of all the cells it knows and the IP addresses of the volume location servers for those cells. The cell to which the system belongs is added to the database when modprobe is performed by the “rootcell=” argument or, if compiled in, using a “kafs.rootcell=” argument on the kernel command line.

Further cells can be added by commands similar to the following:

```
echo add CELLNAME VLADDR[:VLADDR][:VLADDR]... >/proc/fs/afs/cells
echo add grand.central.org 18.9.48.14:128.2.203.61:130.237.48.87 >/proc/fs/
↪afs/cells
```

No other cell database operations are available at this time.

3.4.8 Security

Secure operations are initiated by acquiring a key using the klog program. A very primitive klog program is available at:

<http://people.redhat.com/~dhowells/rxrpc/klog.c>

This should be compiled by:

```
make klog LDLIBS="-lcrypto -lcrypt -lkrb4 -lkeyutils"
```

And then run as:

```
./klog
```

Assuming it's successful, this adds a key of type RxRPC, named for the service and cell, eg: “afs@<cellname>”. This can be viewed with the keyctl program or by cat'ing /proc/keys:

```
[root@andromeda ~]# keyctl show
Session Keyring
   -3 --alswrv      0      0  keyring: _ses.3268
    2 --alswrv      0      0  \_  keyring: _uid.0
111416553 --als--v   0      0  \_  rxrpc: afs@CAMBRIDGE.REDHAT.COM
```

Currently the username, realm, password and proposed ticket lifetime are compiled in to the program.

It is not required to acquire a key before using AFS facilities, but if one is not acquired then all operations will be governed by the anonymous user parts of the ACLs.

If a key is acquired, then all AFS operations, including mounts and automounts, made by a possessor of that key will be secured with that key.

If a file is opened with a particular key and then the file descriptor is passed to a process that doesn't have that key (perhaps over an AF_UNIX socket), then the operations on the file will be made with key that was used to open the file.

3.4.9 The @sys Substitution

The list of up to 16 @sys substitutions for the current network namespace can be configured by writing a list to `/proc/fs/afs/sysname`:

```
[root@andromeda ~]# echo foo amd64_linux_26 >/proc/fs/afs/sysname
```

or cleared entirely by writing an empty list:

```
[root@andromeda ~]# echo >/proc/fs/afs/sysname
```

The current list for current network namespace can be retrieved by:

```
[root@andromeda ~]# cat /proc/fs/afs/sysname
foo
amd64_linux_26
```

When @sys is being substituted for, each element of the list is tried in the order given.

By default, the list will contain one item that conforms to the pattern “<arch>_linux_26” , amd64 being the name for x86_64.

3.5 autofs - how it works

3.5.1 Purpose

The goal of autofs is to provide on-demand mounting and race free automatic un-mounting of various other filesystems. This provides two key advantages:

1. There is no need to delay boot until all filesystems that might be needed are mounted. Processes that try to access those slow filesystems might be delayed but other processes can continue freely. This is particularly important for network filesystems (e.g. NFS) or filesystems stored on media with a media-changing robot.
2. The names and locations of filesystems can be stored in a remote database and can change at any time. The content in that data base at the time of access will be used to provide a target for the access. The interpretation of names in the filesystem can even be programmatic rather than database-backed, allowing wildcards for example, and can vary based on the user who first accessed a name.

3.5.2 Context

The “autofs” filesystem module is only one part of an autofs system. There also needs to be a user-space program which looks up names and mounts filesystems. This will often be the “automount” program, though other tools including “systemd” can make use of “autofs” . This document describes only the kernel module and the interactions required with any user-space program. Subsequent text refers to this as the “automount daemon” or simply “the daemon” .

“autofs” is a Linux kernel module which provides the “autofs” filesystem type. Several “autofs” filesystems can be mounted and they can each be managed separately, or all managed by the same daemon.

3.5.3 Content

An autofs filesystem can contain 3 sorts of objects: directories, symbolic links and mount traps. Mount traps are directories with extra properties as described in the next section.

Objects can only be created by the automount daemon: symlinks are created with a regular symlink system call, while directories and mount traps are created with mkdir. The determination of whether a directory should be a mount trap is based on a master map. This master map is consulted by autofs to determine which directories are mount points. Mount points can be direct/indirect/offset. On most systems, the default master map is located at /etc/auto.master.

If neither the direct or offset mount options are given (so the mount is considered to be indirect), then the root directory is always a regular directory, otherwise it is a mount trap when it is empty and a regular directory when not empty. Note that direct and offset are treated identically so a concise summary is that the root directory is a mount trap only if the filesystem is mounted direct and the root is empty.

Directories created in the root directory are mount traps only if the filesystem is mounted indirect and they are empty.

Directories further down the tree depend on the maxproto mount option and particularly whether it is less than five or not. When maxproto is five, no directories further down the tree are ever mount traps, they are always regular directories. When the maxproto is four (or three), these directories are mount traps precisely when they are empty.

So: non-empty (i.e. non-leaf) directories are never mount traps. Empty directories are sometimes mount traps, and sometimes not depending on where in the tree they are (root, top level, or lower), the maxproto, and whether the mount was indirect or not.

3.5.4 Mount Traps

A core element of the implementation of autofs is the Mount Traps which are provided by the Linux VFS. Any directory provided by a filesystem can be designated as a trap. This involves two separate features that work together to allow autofs to do its job.

DCACHE_NEED_AUTOMOUNT

If a dentry has the `DCACHE_NEED_AUTOMOUNT` flag set (which gets set if the inode has `S_AUTOMOUNT` set, or can be set directly) then it is (potentially) a mount trap. Any access to this directory beyond a “stat” will (normally) cause the `d_op->d_automount()` dentry operation to be called. The task of this method is to find the filesystem that should be mounted on the directory and to return it. The VFS is responsible for actually mounting the root of this filesystem on the directory.

autofs doesn't find the filesystem itself but sends a message to the automount daemon asking it to find and mount the filesystem. The autofs `d_automount` method then waits for the daemon to report that everything is ready. It will then return “NULL” indicating that the mount has already happened. The VFS doesn't try to mount anything but follows down the mount that is already there.

This functionality is sufficient for some users of mount traps such as NFS which creates traps so that mountpoints on the server can be reflected on the client. However it is not sufficient for autofs. As mounting onto a directory is considered to be “beyond a stat”, the automount daemon would not be able to mount a filesystem on the ‘trap’ directory without some way to avoid getting caught in the trap. For that purpose there is another flag.

DCACHE_MANAGE_TRANSIT

If a dentry has `DCACHE_MANAGE_TRANSIT` set then two very different but related behaviours are invoked, both using the `d_op->d_manage()` dentry operation.

Firstly, before checking to see if any filesystem is mounted on the directory, `d_manage()` will be called with the `rcu_walk` parameter set to false. It may return one of three things:

- A return value of zero indicates that there is nothing special about this dentry and normal checks for mounts and automounts should proceed.

autofs normally returns zero, but first waits for any expiry (automatic unmounting of the mounted filesystem) to complete. This avoids races.

- A return value of `-EISDIR` tells the VFS to ignore any mounts on the directory and to not consider calling `->d_automount()`. This effectively disables the **DCACHE_NEED_AUTOMOUNT** flag causing the directory not be a mount trap after all.

autofs returns this if it detects that the process performing the lookup is the automount daemon and that the mount has been requested but has not yet completed. How it determines this is discussed later. This allows the automount daemon not to get caught in the mount trap.

There is a subtlety here. It is possible that a second autofs filesystem can be mounted below the first and for both of them to be managed by the same

daemon. For the daemon to be able to mount something on the second it must be able to “walk” down past the first. This means that `d_manage` cannot always return `-EISDIR` for the automount daemon. It must only return it when a mount has been requested, but has not yet completed.

`d_manage` also returns `-EISDIR` if the dentry shouldn't be a mount trap, either because it is a symbolic link or because it is not empty.

- Any other negative value is treated as an error and returned to the caller.

autofs can return

- `-ENOENT` if the automount daemon failed to mount anything,
- `-ENOMEM` if it ran out of memory,
- `-EINTR` if a signal arrived while waiting for expiry to complete
- or any other error sent down by the automount daemon.

The second use case only occurs during an “RCU-walk” and so `rcu_walk` will be set.

An RCU-walk is a fast and lightweight process for walking down a filename path (i.e. it is like running on tip-toes). RCU-walk cannot cope with all situations so when it finds a difficulty it falls back to “REF-walk”, which is slower but more robust.

RCU-walk will never call `->d_automount`; the filesystems must already be mounted or RCU-walk cannot handle the path. To determine if a mount-trap is safe for RCU-walk mode it calls `->d_manage()` with `rcu_walk` set to true.

In this case `d_manage()` must avoid blocking and should avoid taking spinlocks if at all possible. Its sole purpose is to determine if it would be safe to follow down into any mounted directory and the only reason that it might not be is if an expiry of the mount is underway.

In the `rcu_walk` case, `d_manage()` cannot return `-EISDIR` to tell the VFS that this is a directory that doesn't require `d_automount`. If `rcu_walk` sees a dentry with `DCACHE_NEED_AUTOMOUNT` set but nothing mounted, it will fall back to REF-walk. `d_manage()` cannot make the VFS remain in RCU-walk mode, but can only tell it to get out of RCU-walk mode by returning `-ECHILD`.

So `d_manage()`, when called with `rcu_walk` set, should either return `-ECHILD` if there is any reason to believe it is unsafe to enter the mounted filesystem, otherwise it should return 0.

autofs will return `-ECHILD` if an expiry of the filesystem has been initiated or is being considered, otherwise it returns 0.

3.5.5 Mountpoint expiry

The VFS has a mechanism for automatically expiring unused mounts, much as it can expire any unused dentry information from the dcache. This is guided by the `MNT_SHRINKABLE` flag. This only applies to mounts that were created by `d_automount()` returning a filesystem to be mounted. As `autofs` doesn't return such a filesystem but leaves the mounting to the automount daemon, it must involve the automount daemon in unmounting as well. This also means that `autofs` has more control over expiry.

The VFS also supports “expiry” of mounts using the `MNT_EXPIRE` flag to the `umount` system call. Unmounting with `MNT_EXPIRE` will fail unless a previous attempt had been made, and the filesystem has been inactive and untouched since that previous attempt. `autofs` does not depend on this but has its own internal tracking of whether filesystems were recently used. This allows individual names in the `autofs` directory to expire separately.

With version 4 of the protocol, the automount daemon can try to unmount any filesystems mounted on the `autofs` filesystem or remove any symbolic links or empty directories any time it likes. If the unmount or removal is successful the filesystem will be returned to the state it was before the mount or creation, so that any access of the name will trigger normal auto-mount processing. In particular, `rmdir` and `unlink` do not leave negative entries in the dcache as a normal filesystem would, so an attempt to access a recently-removed object is passed to `autofs` for handling.

With version 5, this is not safe except for unmounting from top-level directories. As lower-level directories are never mount traps, other processes will see an empty directory as soon as the filesystem is unmounted. So it is generally safest to use the `autofs` expiry protocol described below.

Normally the daemon only wants to remove entries which haven't been used for a while. For this purpose `autofs` maintains a “`last_used`” time stamp on each directory or symlink. For symlinks it genuinely does record the last time the symlink was “used” or followed to find out where it points to. For directories the field is used slightly differently. The field is updated at mount time and during expire checks if it is found to be in use (ie. open file descriptor or process working directory) and during path walks. The update done during path walks prevents frequent expire and immediate mount of frequently accessed automounts. But in the case where a GUI continually access or an application frequently scans an `autofs` directory tree there can be an accumulation of mounts that aren't actually being used. To cater for this case the “`strictexpire`” `autofs` mount option can be used to avoid the “`last_used`” update on path walk thereby preventing this apparent inability to expire mounts that aren't really in use.

The daemon is able to ask `autofs` if anything is due to be expired, using an `ioctl` as discussed later. For a direct mount, `autofs` considers if the entire mount-tree can be unmounted or not. For an indirect mount, `autofs` considers each of the names in the top level directory to determine if any of those can be unmounted and cleaned up.

There is an option with indirect mounts to consider each of the leaves that has been mounted on instead of considering the top-level names. This was originally intended for compatibility with version 4 of `autofs` and should be considered as

deprecated for Sun Format automount maps. However, it may be used again for amd format mount maps (which are generally indirect maps) because the amd automounter allows for the setting of an expire timeout for individual mounts. But there are some difficulties in making the needed changes for this.

When autofs considers a directory it checks the `last_used` time and compares it with the “timeout” value set when the filesystem was mounted, though this check is ignored in some cases. It also checks if the directory or anything below it is in use. For symbolic links, only the `last_used` time is ever considered.

If both appear to support expiring the directory or symlink, an action is taken.

There are two ways to ask autofs to consider expiry. The first is to use the **AUTOFD_IOC_EXPIRE** ioctl. This only works for indirect mounts. If it finds something in the root directory to expire it will return the name of that thing. Once a name has been returned the automount daemon needs to unmount any filesystems mounted below the name normally. As described above, this is unsafe for non-toplevel mounts in a version-5 autofs. For this reason the current automount(8) does not use this ioctl.

The second mechanism uses either the **AUTOFD_DEV_IOCTL_EXPIRE_CMD** or the **AUTOFD_IOC_EXPIRE_MULTI** ioctl. This will work for both direct and indirect mounts. If it selects an object to expire, it will notify the daemon using the notification mechanism described below. This will block until the daemon acknowledges the expiry notification. This implies that the “EXPIRE” ioctl must be sent from a different thread than the one which handles notification.

While the ioctl is blocking, the entry is marked as “expiring” and `d_manage` will block until the daemon affirms that the unmount has completed (together with removing any directories that might have been necessary), or has been aborted.

3.5.6 Communicating with autofs: detecting the daemon

There are several forms of communication between the automount daemon and the filesystem. As we have already seen, the daemon can create and remove directories and symlinks using normal filesystem operations. autofs knows whether a process requesting some operation is the daemon or not based on its process-group id number (see `getpgid(1)`).

When an autofs filesystem is mounted the pgid of the mounting processes is recorded unless the “pgrp=” option is given, in which case that number is recorded instead. Any request arriving from a process in that process group is considered to come from the daemon. If the daemon ever has to be stopped and restarted a new pgid can be provided through an ioctl as will be described below.

3.5.7 Communicating with autofs: the event pipe

When an autofs filesystem is mounted, the ‘write’ end of a pipe must be passed using the ‘fd=’ mount option. autofs will write notification messages to this pipe for the daemon to respond to. For version 5, the format of the message is:

```
struct autofs_v5_packet {
    struct autofs_packet_hdr hdr;
    autofs_wqt_t wait_queue_token;
    __u32 dev;
    __u64 ino;
    __u32 uid;
    __u32 gid;
    __u32 pid;
    __u32 tgid;
    __u32 len;
    char name[NAME_MAX+1];
};
```

And the format of the header is:

```
struct autofs_packet_hdr {
    int proto_version;           /* Protocol version */
    int type;                    /* Type of packet */
};
```

where the type is one of

```
autofs_ptype_missing_indirect
autofs_ptype_expire_indirect
autofs_ptype_missing_direct
autofs_ptype_expire_direct
```

so messages can indicate that a name is missing (something tried to access it but it isn't there) or that it has been selected for expiry.

The pipe will be set to “packet mode” (equivalent to passing O_DIRECT) to `_pipe2(2)` so that a read from the pipe will return at most one packet, and any unread portion of a packet will be discarded.

The `wait_queue_token` is a unique number which can identify a particular request to be acknowledged. When a message is sent over the pipe the affected dentry is marked as either “active” or “expiring” and other accesses to it block until the message is acknowledged using one of the ioctls below with the relevant `wait_queue_token`.

3.5.8 Communicating with autofs: root directory ioctls

The root directory of an autofs filesystem will respond to a number of ioctls. The process issuing the ioctl must have the `CAP_SYS_ADMIN` capability, or must be the automount daemon.

The available ioctl commands are:

- **AUTOFS_IOC_READY:** a notification has been handled. The argument to the ioctl command is the “wait_queue_token” number corresponding to the notification being acknowledged.
- **AUTOFS_IOC_FAIL:** similar to above, but indicates failure with the error code `ENOENT`.
- **AUTOFS_IOC_CATATONIC:** Causes the autofs to enter “catatonic” mode meaning that it stops sending notifications to the daemon. This mode is also entered if a write to the pipe fails.
- **AUTOFS_IOC_PROTOVER:** This returns the protocol version in use.
- **AUTOFS_IOC_PROTOSUBVER:** Returns the protocol sub-version which is really a version number for the implementation.
- **AUTOFS_IOC_SETTIMEOUT:** This passes a pointer to an unsigned long. The value is used to set the timeout for expiry, and the current timeout value is stored back through the pointer.
- **AUTOFS_IOC_ASKUMOUNT:** Returns, in the pointed-to int, 1 if the filesystem could be unmounted. This is only a hint as the situation could change at any instant. This call can be used to avoid a more expensive full unmount attempt.
- **AUTOFS_IOC_EXPIRE:** as described above, this asks if there is anything suitable to expire. A pointer to a packet:

```
struct autofs_packet_expire_multi {
    struct autofs_packet_hdr hdr;
    autofs_wqt_t wait_queue_token;
    int len;
    char name[NAME_MAX+1];
};
```

is required. This is filled in with the name of something that can be unmounted or removed. If nothing can be expired, `errno` is set to `EAGAIN`. Even though a `wait_queue_token` is present in the structure, no “wait queue” is established and no acknowledgment is needed.

- **AUTOFS_IOC_EXPIRE_MULTI:** This is similar to **AUTOFS_IOC_EXPIRE** except that it causes notification to be sent to the daemon, and it blocks until the daemon acknowledges. The argument is an integer which can contain two different flags.

AUTOFS_EXP_IMMEDIATE causes `last_used` time to be ignored and objects are expired if they are not in use.

AUTOFS_EXP_FORCED causes the in use status to be ignored and objects are expired even if they are in use. This assumes that the daemon

has requested this because it is capable of performing the umount.

AUTOFS_EXP_LEAVES will select a leaf rather than a top-level name to expire. This is only safe when maxproto is 4.

3.5.9 Communicating with autofs: char-device ioctls

It is not always possible to open the root of an autofs filesystem, particularly a direct mounted filesystem. If the automount daemon is restarted there is no way for it to regain control of existing mounts using any of the above communication channels. To address this need there is a “miscellaneous” character device (major 10, minor 235) which can be used to communicate directly with the autofs filesystem. It requires CAP_SYS_ADMIN for access.

The ‘ioctl’ s that can be used on this device are described in a separate document autofs-mount-control.txt, and are summarised briefly here. Each ioctl is passed a pointer to an autofs_dev_ioctl structure:

```
struct autofs_dev_ioctl {
    __u32 ver_major;
    __u32 ver_minor;
    __u32 size;           /* total size of data passed in
                          * including this struct */
    __s32 ioctlfid;      /* automount command fd */

    /* Command parameters */
    union {
        struct args_protover          protover;
        struct args_protosubver       protosubver;
        struct args_openmount         openmount;
        struct args_ready              ready;
        struct args_fail               fail;
        struct args_setpipefd         setpipefd;
        struct args_timeout            timeout;
        struct args_requester         requester;
        struct args_expire             expire;
        struct args_askumount         askumount;
        struct args_ismountpoint      ismountpoint;
    };

    char path[0];
};
```

For the **OPEN_MOUNT** and **IS_MOUNTPOINT** commands, the target filesystem is identified by the path. All other commands identify the filesystem by the ioctlfid which is a file descriptor open on the root, and which can be returned by **OPEN_MOUNT**.

The ver_major and ver_minor are in/out parameters which check that the requested version is supported, and report the maximum version that the kernel module can support.

Commands are:

- **AUTOFS_DEV_IOCTL_VERSION_CMD**: does nothing, except validate and set version numbers.

- **AUTOFS_DEV_IOCTL_OPENMOUNT_CMD:** return an open file descriptor on the root of an autofs filesystem. The filesystem is identified by name and device number, which is stored in `openmount.devid`. Device numbers for existing filesystems can be found in `/proc/self/mountinfo`.
- **AUTOFS_DEV_IOCTL_CLOSEMOUNT_CMD:** same as `close(ioctlfid)`.
- **AUTOFS_DEV_IOCTL_SETPIPEFD_CMD:** if the filesystem is in catatonic mode, this can provide the write end of a new pipe in `setpipefd.pipefd` to re-establish communication with a daemon. The process group of the calling process is used to identify the daemon.
- **AUTOFS_DEV_IOCTL_REQUESTER_CMD:** `path` should be a name within the filesystem that has been auto-mounted on. On successful return, `requester.uid` and `requester.gid` will be the UID and GID of the process which triggered that mount.
- **AUTOFS_DEV_IOCTL_ISMOUNTPOINT_CMD:** Check if `path` is a mount-point of a particular type - see separate documentation for details.
- **AUTOFS_DEV_IOCTL_PROTOVER_CMD**
- **AUTOFS_DEV_IOCTL_PROTOSUBVER_CMD**
- **AUTOFS_DEV_IOCTL_READY_CMD**
- **AUTOFS_DEV_IOCTL_FAIL_CMD**
- **AUTOFS_DEV_IOCTL_CATATONIC_CMD**
- **AUTOFS_DEV_IOCTL_TIMEOUT_CMD**
- **AUTOFS_DEV_IOCTL_EXPIRE_CMD**
- **AUTOFS_DEV_IOCTL_ASKUMOUNT_CMD**

These all have the same function as the similarly named **AUTOFS_IOC** ioctls, except that **FAIL** can be given an explicit error number in `fail.status` instead of assuming `ENOENT`, and this **EXPIRE** command corresponds to **AUTOFS_IOC_EXPIRE_MULTI**.

3.5.10 Catatonic mode

As mentioned, an autofs mount can enter “catatonic” mode. This happens if a write to the notification pipe fails, or if it is explicitly requested by an ioctl.

When entering catatonic mode, the pipe is closed and any pending notifications are acknowledged with the error `ENOENT`.

Once in catatonic mode attempts to access non-existing names will result in `ENOENT` while attempts to access existing directories will be treated in the same way as if they came from the daemon, so mount traps will not fire.

When the filesystem is mounted a `_uid_` and `_gid_` can be given which set the ownership of directories and symbolic links. When the filesystem is in catatonic mode, any process with a matching UID can create directories or symlinks in the root directory, but not in other directories.

Catatonic mode can only be left via the `AUT-OFS_DEV_IOCTL_OPENMOUNT_CMD` ioctl on the `/dev/autofs`.

3.5.11 The “ignore” mount option

The “ignore” mount option can be used to provide a generic indicator to applications that the mount entry should be ignored when displaying mount information.

In other OSes that provide autofs and that provide a mount list to user space based on the kernel mount list a no-op mount option (“ignore” is the one use on the most common OSes) is allowed so that autofs file system users can optionally use it.

This is intended to be used by user space programs to exclude autofs mounts from consideration when reading the mounts list.

3.5.12 autofs, name spaces, and shared mounts

With bind mounts and name spaces it is possible for an autofs filesystem to appear at multiple places in one or more filesystem name spaces. For this to work sensibly, the autofs filesystem should always be mounted “shared” . e.g.

```
mount --make-shared /autofs/mount/point
```

The automount daemon is only able to manage a single mount location for an autofs filesystem and if mounts on that are not ‘shared’ , other locations will not behave as expected. In particular access to those other locations will likely result in the ELOOP error

```
Too many levels of symbolic links
```

3.6 Miscellaneous Device control operations for the autofs kernel module

3.6.1 The problem

There is a problem with active restarts in autofs (that is to say restarting autofs when there are busy mounts).

During normal operation autofs uses a file descriptor opened on the directory that is being managed in order to be able to issue control operations. Using a file descriptor gives ioctl operations access to autofs specific information stored in the super block. The operations are things such as setting an autofs mount catatonic, setting the expire timeout and requesting expire checks. As is explained below, certain types of autofs triggered mounts can end up covering an autofs mount itself which prevents us being able to use `open(2)` to obtain a file descriptor for these operations if we don’ t already have one open.

Currently autofs uses “`umount -l`” (lazy umount) to clear active mounts at restart. While using lazy umount works for most cases, anything that needs to walk back

up the mount tree to construct a path, such as `getcwd(2)` and the `proc` file system `/proc/<pid>/cwd`, no longer works because the point from which the path is constructed has been detached from the mount tree.

The actual problem with `autofs` is that it can't reconnect to existing mounts. Immediately one thinks of just adding the ability to remount `autofs` file systems would solve it, but alas, that can't work. This is because `autofs` direct mounts and the implementation of "on demand mount and expire" of nested mount trees have the file system mounted directly on top of the mount trigger directory dentry.

For example, there are two types of automount maps, direct (in the kernel module source you will see a third type called an offset, which is just a direct mount in disguise) and indirect.

Here is a master map with direct and indirect map entries:

```
/-      /etc/auto.direct
/test   /etc/auto.indirect
```

and the corresponding map files:

```
/etc/auto.direct:

/automount/dparse/g6  budgie:/autofs/export1
/automount/dparse/g1  shark:/autofs/export1
and so on.
```

`/etc/auto.indirect:`

```
g1      shark:/autofs/export1
g6      budgie:/autofs/export1
and so on.
```

For the above indirect map an `autofs` file system is mounted on `/test` and mounts are triggered for each sub-directory key by the inode lookup operation. So we see a mount of `shark:/autofs/export1` on `/test/g1`, for example.

The way that direct mounts are handled is by making an `autofs` mount on each full path, such as `/automount/dparse/g1`, and using it as a mount trigger. So when we walk on the path we mount `shark:/autofs/export1` "on top of this mount point". Since these are always directories we can use the `follow_link` inode operation to trigger the mount.

But, each entry in direct and indirect maps can have offsets (making them multi-mount map entries).

For example, an indirect mount map entry could also be:

```
g1  \
/    shark:/autofs/export5/testing/test \
/s1  shark:/autofs/export/testing/test/s1 \
/s2  shark:/autofs/export5/testing/test/s2 \
/s1/ss1 shark:/autofs/export1 \
/s2/ss2 shark:/autofs/export2
```

and a similarly a direct mount map entry could also be:

```
/automount/dparse/g1 \  
/      shark:/autofs/export5/testing/test \  
/s1    shark:/autofs/export/testing/test/s1 \  
/s2    shark:/autofs/export5/testing/test/s2 \  
/s1/ss1 shark:/autofs/export2 \  
/s2/ss2 shark:/autofs/export2
```

One of the issues with version 4 of autofs was that, when mounting an entry with a large number of offsets, possibly with nesting, we needed to mount and umount all of the offsets as a single unit. Not really a problem, except for people with a large number of offsets in map entries. This mechanism is used for the well known “hosts” map and we have seen cases (in 2.4) where the available number of mounts are exhausted or where the number of privileged ports available is exhausted.

In version 5 we mount only as we go down the tree of offsets and similarly for expiring them which resolves the above problem. There is somewhat more detail to the implementation but it isn’ t needed for the sake of the problem explanation. The one important detail is that these offsets are implemented using the same mechanism as the direct mounts above and so the mount points can be covered by a mount.

The current autofs implementation uses an ioctl file descriptor opened on the mount point for control operations. The references held by the descriptor are accounted for in checks made to determine if a mount is in use and is also used to access autofs file system information held in the mount super block. So the use of a file handle needs to be retained.

3.6.2 The Solution

To be able to restart autofs leaving existing direct, indirect and offset mounts in place we need to be able to obtain a file handle for these potentially covered autofs mount points. Rather than just implement an isolated operation it was decided to re-implement the existing ioctl interface and add new operations to provide this functionality.

In addition, to be able to reconstruct a mount tree that has busy mounts, the uid and gid of the last user that triggered the mount needs to be available because these can be used as macro substitution variables in autofs maps. They are recorded at mount request time and an operation has been added to retrieve them.

Since we’ re re-implementing the control interface, a couple of other problems with the existing interface have been addressed. First, when a mount or expire operation completes a status is returned to the kernel by either a “send ready” or a “send fail” operation. The “send fail” operation of the ioctl interface could only ever send ENOENT so the re-implementation allows user space to send an actual status. Another expensive operation in user space, for those using very large maps, is discovering if a mount is present. Usually this involves scanning /proc/mounts and since it needs to be done quite often it can introduce significant overhead when there are many entries in the mount table. An operation to lookup the mount status of a mount point dentry (covered or not) has also been added.

Current kernel development policy recommends avoiding the use of the ioctl mechanism in favor of systems such as Netlink. An implementation using this system

was attempted to evaluate its suitability and it was found to be inadequate, in this case. The Generic Netlink system was used for this as raw Netlink would lead to a significant increase in complexity. There's no question that the Generic Netlink system is an elegant solution for common case ioctl functions but it's not a complete replacement probably because its primary purpose in life is to be a message bus implementation rather than specifically an ioctl replacement. While it would be possible to work around this there is one concern that led to the decision to not use it. This is that the autofs expire in the daemon has become far too complex because unmount candidates are enumerated, almost for no other reason than to "count" the number of times to call the expire ioctl. This involves scanning the mount table which has proved to be a big overhead for users with large maps. The best way to improve this is to try and get back to the way the expire was done long ago. That is, when an expire request is issued for a mount (file handle) we should continually call back to the daemon until we can't unmount any more mounts, then return the appropriate status to the daemon. At the moment we just expire one mount at a time. A Generic Netlink implementation would exclude this possibility for future development due to the requirements of the message bus architecture.

3.6.3 autofs Miscellaneous Device mount control interface

The control interface is opening a device node, typically `/dev/autofs`.

All the ioctls use a common structure to pass the needed parameter information and return operation results:

```
struct autofs_dev_ioctl {
    __u32 ver_major;
    __u32 ver_minor;
    __u32 size;           /* total size of data passed in
                          * including this struct */
    __s32 ioctlfd;      /* automount command fd */

    /* Command parameters */
    union {
        struct args_protover          protover;
        struct args_protosubver       protosubver;
        struct args_openmount         openmount;
        struct args_ready              ready;
        struct args_fail               fail;
        struct args_setpipefd         setpipefd;
        struct args_timeout            timeout;
        struct args_requester         requester;
        struct args_expire             expire;
        struct args_askumount          askumount;
        struct args_ismountpoint       ismountpoint;
    };

    char path[0];
};
```

The `ioctlfd` field is a mount point file descriptor of an autofs mount point. It is returned by the open call and is used by all calls except the check for whether a given path is a mount point, where it may optionally be used to check a specific mount corresponding to a given mount point file descriptor, and when requesting

the uid and gid of the last successful mount on a directory within the autofs file system.

The union is used to communicate parameters and results of calls made as described below.

The path field is used to pass a path where it is needed and the size field is used account for the increased structure length when translating the structure sent from user space.

This structure can be initialized before setting specific fields by using the void function call `init_autofs_dev_ioctl(struct autofs_dev_ioctl *)`.

All of the ioctls perform a copy of this structure from user space to kernel space and return `-EINVAL` if the size parameter is smaller than the structure size itself, `-ENOMEM` if the kernel memory allocation fails or `-EFAULT` if the copy itself fails. Other checks include a version check of the compiled in user space version against the module version and a mismatch results in a `-EINVAL` return. If the size field is greater than the structure size then a path is assumed to be present and is checked to ensure it begins with a `“/”` and is `NULL` terminated, otherwise `-EINVAL` is returned. Following these checks, for all ioctl commands except `AUTOFS_DEV_IOCTL_VERSION_CMD`, `AUTOFS_DEV_IOCTL_OPENMOUNT_CMD` and `AUTOFS_DEV_IOCTL_CLOSEMOUNT_CMD` the `ioctlfd` is validated and if it is not a valid descriptor or doesn't correspond to an autofs mount point an error of `-EBADF`, `-ENOTTY` or `-EINVAL` (not an autofs descriptor) is returned.

3.6.4 The ioctls

An example of an implementation which uses this interface can be seen in autofs version 5.0.4 and later in file `lib/dev-ioctl-lib.c` of the distribution tar available for download from kernel.org in directory `/pub/linux/daemons/autofs/v5`.

The device node ioctl operations implemented by this interface are:

AUTOFS_DEV_IOCTL_VERSION

Get the major and minor version of the autofs device ioctl kernel module implementation. It requires an initialized `struct autofs_dev_ioctl` as an input parameter and sets the version information in the passed in structure. It returns 0 on success or the error `-EINVAL` if a version mismatch is detected.

AUTOFS_DEV_IOCTL_PROTOVER_CMD and AUTOFS_DEV_IOCTL_PROTOSUBVER_CMD

Get the major and minor version of the autofs protocol version understood by loaded module. This call requires an initialized `struct autofs_dev_ioctl` with the `ioctlfd` field set to a valid autofs mount point descriptor and sets the requested version number in `version` field of `struct args_protover` or `sub_version` field of `struct args_protosubver`. These commands return 0 on success or one of the negative error codes if validation fails.

AUTOFS_DEV_IOCTL_OPENMOUNT and AUTOFS_DEV_IOCTL_CLOSEMOUNT

Obtain and release a file descriptor for an autofs managed mount point path. The open call requires an initialized struct `autofs_dev_ioctl` with the `path` field set and the `size` field adjusted appropriately as well as the `devid` field of struct `args_openmount` set to the device number of the autofs mount. The device number can be obtained from the mount options shown in `/proc/mounts`. The close call requires an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call. The release of the file descriptor can also be done with `close(2)` so any open descriptors will also be closed at process exit. The close call is included in the implemented operations largely for completeness and to provide for a consistent user space implementation.

AUTOFS_DEV_IOCTL_READY_CMD and AUTOFS_DEV_IOCTL_FAIL_CMD

Return mount and expire result status from user space to the kernel. Both of these calls require an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call and the `token` field of struct `args_ready` or struct `args_fail` set to the wait queue token number, received by user space in the foregoing mount or expire request. The `status` field of struct `args_fail` is set to the `errno` of the operation. It is set to 0 on success.

AUTOFS_DEV_IOCTL_SETPIPEFD_CMD

Set the pipe file descriptor used for kernel communication to the daemon. Normally this is set at mount time using an option but when reconnecting to an existing mount we need to use this to tell the autofs mount about the new kernel pipe descriptor. In order to protect mounts against incorrectly setting the pipe descriptor we also require that the autofs mount be `catatonic` (see next call).

The call requires an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call and the `pipefd` field of struct `args_setpipefd` set to descriptor of the pipe. On success the call also sets the process group id used to identify the controlling process (eg. the owning `automount(8)` daemon) to the process group of the caller.

AUTOFS_DEV_IOCTL_CATATONIC_CMD

Make the autofs mount point `catatonic`. The autofs mount will no longer issue mount requests, the kernel communication pipe descriptor is released and any remaining waits in the queue released.

The call requires an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call.

AUTOFS_DEV_IOCTL_TIMEOUT_CMD

Set the expire timeout for mounts within an autofs mount point.

The call requires an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call.

AUTOFS_DEV_IOCTL_REQUESTER_CMD

Return the uid and gid of the last process to successfully trigger a the mount on the given path dentry.

The call requires an initialized struct `autofs_dev_ioctl` with the `path` field set to the mount point in question and the `size` field adjusted appropriately. Upon return the `uid` field of struct `args_requester` contains the uid and `gid` field the gid.

When reconstructing an autofs mount tree with active mounts we need to re-connect to mounts that may have used the original process uid and gid (or string variations of them) for mount lookups within the map entry. This call provides the ability to obtain this uid and gid so they may be used by user space for the mount map lookups.

AUTOFS_DEV_IOCTL_EXPIRE_CMD

Issue an expire request to the kernel for an autofs mount. Typically this ioctl is called until no further expire candidates are found.

The call requires an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call. In addition an immediate expire that's independent of the mount timeout, and a forced expire that's independent of whether the mount is busy, can be requested by setting the `how` field of struct `args_expire` to `AUTOFS_EXP_IMMEDIATE` or `AUTOFS_EXP_FORCED`, respectively. If no expire candidates can be found the ioctl returns -1 with `errno` set to `EAGAIN`.

This call causes the kernel module to check the mount corresponding to the given `ioctlfd` for mounts that can be expired, issues an expire request back to the daemon and waits for completion.

AUTOFS_DEV_IOCTL_ASKMOUNT_CMD

Checks if an autofs mount point is in use.

The call requires an initialized struct `autofs_dev_ioctl` with the `ioctlfd` field set to the descriptor obtained from the open call and it returns the result in the `may_umount` field of struct `args_askumount`, 1 for busy and 0 otherwise.

AUTOFS_DEV_IOCTL_ISMOUNTPOINT_CMD

Check if the given path is a mountpoint.

The call requires an initialized struct `autofs_dev_ioctl`. There are two possible variations. Both use the `path` field set to the path of the mount point to check and the `size` field adjusted appropriately. One uses the `ioctlfd` field to identify a specific mount point to check while the other variation uses the `path` and optionally `in.type` field of struct `args_ismountpoint` set to an `autofs` mount type. The call returns 1 if this is a mount point and sets `out.devid` field to the device number of the mount and `out.magic` field to the relevant super block magic number (described below) or 0 if it isn't a mountpoint. In both cases the device number (as returned by `new_encode_dev()`) is returned in `out.devid` field.

If supplied with a file descriptor we're looking for a specific mount, not necessarily at the top of the mounted stack. In this case the path the descriptor corresponds to is considered a mountpoint if it is itself a mountpoint or contains a mount, such as a multi-mount without a root mount. In this case we return 1 if the descriptor corresponds to a mount point and also returns the super magic of the covering mount if there is one or 0 if it isn't a mountpoint.

If a path is supplied (and the `ioctlfd` field is set to -1) then the path is looked up and is checked to see if it is the root of a mount. If a type is also given we are looking for a particular `autofs` mount and if a match isn't found a fail is returned. If the located path is the root of a mount 1 is returned along with the super magic of the mount or 0 otherwise.

3.7 BeOS filesystem for Linux

Document last updated: Dec 6, 2001

3.7.1 Warning

Make sure you understand that this is alpha software. This means that the implementation is neither complete nor well-tested.

I DISCLAIM ALL RESPONSIBILITY FOR ANY POSSIBLE BAD EFFECTS OF THIS CODE!

3.7.2 License

This software is covered by the GNU General Public License. See the file `COPYING` for the complete text of the license. Or the GNU website: <<http://www.gnu.org/licenses/licenses.html>>

3.7.3 Author

The largest part of the code written by Will Dyson <will_dyson@pobox.com> He has been working on the code since Aug 13, 2001. See the changelog for details.

Original Author: Makoto Kato <m_kato@ga2.so-net.ne.jp>

His original code can still be found at: <<http://hp.vector.co.jp/authors/VA008030/bfs/>>

Does anyone know of a more current email address for Makoto? He doesn't respond to the address given above...

This filesystem doesn't have a maintainer.

3.7.4 What is this Driver?

This module implements the native filesystem of BeOS <http://www.beincorporated.com/> for the linux 2.4.1 and later kernels. Currently it is a read-only implementation.

3.7.5 Which is it, BFS or BEFS?

Be, Inc said, "BeOS Filesystem is officially called BFS, not BeFS" . But Unixware Boot Filesystem is called bfs, too. And they are already in the kernel. Because of this naming conflict, on Linux the BeOS filesystem is called befs.

3.7.6 How to Install

step 1. Install the BeFS patch into the source code tree of linux.

Apply the patchfile to your kernel source tree. Assuming that your kernel source is in /foo/bar/linux and the patchfile is called patch-befs-xxx, you would do the following:

```
cd /foo/bar/linux patch -p1 < /path/to/patch-befs-xxx
```

if the patching step fails (i.e. there are rejected hunks), you can try to figure it out yourself (it shouldn't be hard), or mail the maintainer (Will Dyson <will_dyson@pobox.com>) for help.

step 2. Configuration & make kernel

The linux kernel has many compile-time options. Most of them are beyond the scope of this document. I suggest the Kernel-HOWTO document as a good general reference on this topic. <http://www.linuxdocs.org/HOWTOs/Kernel-HOWTO-4.html>

However, to use the BeFS module, you must enable it at configure time:

```
cd /foo/bar/linux
make menuconfig (or xconfig)
```

The BeFS module is not a standard part of the linux kernel, so you must first enable support for experimental code under the “Code maturity level” menu.

Then, under the “Filesystems” menu will be an option called “BeFS filesystem (experimental)”, or something like that. Enable that option (it is fine to make it a module).

Save your kernel configuration and then build your kernel.

step 3. Install

See the kernel howto <<http://www.linux.com/howto/Kernel-HOWTO.html>> for instructions on this critical step.

3.7.7 Using BFS

To use the BeOS filesystem, use filesystem type ‘befs’ .

ex:

```
mount -t befs /dev/fd0 /beos
```

3.7.8 Mount Options

uid=nnn	All files in the partition will be owned by user id nnn.
gid=nnn	All files in the partition will be in group nnn.
iocharset=xxx	Use xxx as the name of the NLS translation table.
debug	The driver will output debugging information to the syslog.

3.7.9 How to Get Lastest Version

The latest version is currently available at: <<http://befs-driver.sourceforge.net/>>

3.7.10 Any Known Bugs?

As of Jan 20, 2002:

None

3.7.11 Special Thanks

Dominic Giampalo ···Writing “Practical file system design with Be filesystem”

Hiroyuki Yamada ···Testing LinuxPPC.

3.8 BFS Filesystem for Linux

The BFS filesystem is used by SCO UnixWare OS for the /stand slice, which usually contains the kernel image and a few other files required for the boot process.

In order to access /stand partition under Linux you obviously need to know the partition number and the kernel must support UnixWare disk slices (CONFIG_UNIXWARE_DISKLABEL config option). However BFS support does not depend on having UnixWare disklabel support because one can also mount BFS filesystem via loopback:

```
# losetup /dev/loop0 stand.img
# mount -t bfs /dev/loop0 /mnt/stand
```

where stand.img is a file containing the image of BFS filesystem. When you have finished using it and unmounted you need to also deallocate /dev/loop0 device by:

```
# losetup -d /dev/loop0
```

You can simplify mounting by just typing:

```
# mount -t bfs -o loop stand.img /mnt/stand
```

this will allocate the first available loopback device (and load loop.o kernel module if necessary) automatically. If the loopback driver is not loaded automatically, make sure that you have compiled the module and that modprobe is functioning. Beware that umount will not deallocate /dev/loopN device if /etc/mtab file on your system is a symbolic link to /proc/mounts. You will need to do it manually using “-d” switch of losetup(8). Read losetup(8) manpage for more info.

To create the BFS image under UnixWare you need to find out first which slice contains it. The command prtvtoc(1M) is your friend:

```
# prtvtoc /dev/rdisk/c0b0t0d0s0
```

(assuming your root disk is on target=0, lun=0, bus=0, controller=0). Then you look for the slice with tag “STAND”, which is usually slice 10. With this information you can use dd(1) to create the BFS image:

```
# umount /stand
# dd if=/dev/rdisk/c0b0t0d0sa of=stand.img bs=512
```

Just in case, you can verify that you have done the right thing by checking the magic number:

```
# od -Ad -tx4 stand.img | more
```

The first 4 bytes should be 0x1badface.

If you have any patches, questions or suggestions regarding this BFS implementation please contact the author:

Tigran Aivazian <aivazian.tigran@gmail.com>

3.9 BTRFS

Btrfs is a copy on write filesystem for Linux aimed at implementing advanced features while focusing on fault tolerance, repair and easy administration. Jointly developed by several companies, licensed under the GPL and open for contribution from anyone.

The main Btrfs features include:

- Extent based file storage (2⁶⁴ max file size)
- Space efficient packing of small files
- Space efficient indexed directories
- Dynamic inode allocation
- Writable snapshots
- Subvolumes (separate internal filesystem roots)
- Object level mirroring and striping
- Checksums on data and metadata (multiple algorithms available)
- Compression
- Integrated multiple device support, with several raid algorithms
- Offline filesystem check
- Efficient incremental backup and FS mirroring
- Online filesystem defragmentation

For more information please refer to the wiki

<https://btrfs.wiki.kernel.org>

that maintains information about administration tasks, frequently asked questions, use cases, mount options, comprehensible changelogs, features, manual pages, source code repositories, contacts etc.

3.10 Mounting root file system via SMB (cifs.ko)

Written 2019 by Paulo Alcantara <palcantara@suse.de>

Written 2019 by Aurelien Aptel <aaptel@suse.com>

The CONFIG_CIFS_ROOT option enables experimental root file system support over the SMB protocol via cifs.ko.

It introduces a new kernel command-line option called ‘cifsroot=’ which will tell the kernel to mount the root file system over the network by utilizing SMB or CIFS protocol.

In order to mount, the network stack will also need to be set up by using ‘ip=’ config option. For more details, see Documentation/admin-guide/nfs/nfsroot.rst.

A CIFS root mount currently requires the use of SMB1+UNIX Extensions which is only supported by the Samba server. SMB1 is the older deprecated version of the protocol but it has been extended to support POSIX features (See [1]). The equivalent extensions for the newer recommended version of the protocol (SMB3) have not been fully implemented yet which means SMB3 doesn't support some required POSIX file system objects (e.g. block devices, pipes, sockets).

As a result, a CIFS root will default to SMB1 for now but the version to use can nonetheless be changed via the 'vers=' mount option. This default will change once the SMB3 POSIX extensions are fully implemented.

3.10.1 Server configuration

To enable SMB1+UNIX extensions you will need to set these global settings in Samba smb.conf:

```
[global]
server min protocol = NT1
unix extension = yes      # default
```

3.10.2 Kernel command line

```
root=/dev/cifs
```

This is just a virtual device that basically tells the kernel to mount the root file system via SMB protocol.

```
cifsroot=//<server-ip>/<share>[,options]
```

Enables the kernel to mount the root file system via SMB that are located in the <server-ip> and <share> specified in this option.

The default mount options are set in fs/cifs/cifsroot.c.

server-ip IPv4 address of the server.

share Path to SMB share (rootfs).

options Optional mount options. For more information, see mount.cifs(8).

3.10.3 Examples

Export root file system as a Samba share in smb.conf file:

```
...
[linux]
    path = /path/to/rootfs
    read only = no
    guest ok = yes
    force user = root
    force group = root
    browseable = yes
```

(continues on next page)

(continued from previous page)

```
writeable = yes
admin users = root
public = yes
create mask = 0777
directory mask = 0777
...
```

Restart smb service:

```
# systemctl restart smb
```

Test it under QEMU on a kernel built with CONFIG_CIFS_ROOT and CONFIG_IP_PNP options enabled:

```
# qemu-system-x86_64 -enable-kvm -cpu host -m 1024 \
-kernel /path/to/linux/arch/x86/boot/bzImage -nographic \
-append "root=/dev/cifs rw ip=dhcp cifsroot=//10.0.2.2/linux,username=foo,
↪password=bar console=ttyS0 3"
```

1: https://wiki.samba.org/index.php/UNIX_Extensions

3.11 Ceph Distributed File System

Ceph is a distributed network file system designed to provide good performance, reliability, and scalability.

Basic features include:

- POSIX semantics
- Seamless scaling from 1 to many thousands of nodes
- High availability and reliability. No single point of failure.
- N-way replication of data across storage nodes
- Fast recovery from node failures
- Automatic rebalancing of data on node addition/removal
- Easy deployment: most FS components are userspace daemons

Also,

- Flexible snapshots (on any directory)
- Recursive accounting (nested files, directories, bytes)

In contrast to cluster filesystems like GFS, OCFS2, and GPFS that rely on symmetric access by all clients to shared block devices, Ceph separates data and metadata management into independent server clusters, similar to Lustre. Unlike Lustre, however, metadata and storage nodes run entirely as user space daemons. File data is striped across storage nodes in large chunks to distribute workload and facilitate high throughputs. When storage nodes fail, data is re-replicated in a distributed fashion by the storage nodes themselves (with some minimal coordination from a cluster monitor), making the system extremely efficient and scalable.

Metadata servers effectively form a large, consistent, distributed in-memory cache above the file namespace that is extremely scalable, dynamically redistributes metadata in response to workload changes, and can tolerate arbitrary (well, non-Byzantine) node failures. The metadata server takes a somewhat unconventional approach to metadata storage to significantly improve performance for common workloads. In particular, inodes with only a single link are embedded in directories, allowing entire directories of dentries and inodes to be loaded into its cache with a single I/O operation. The contents of extremely large directories can be fragmented and managed by independent metadata servers, allowing scalable concurrent access.

The system offers automatic data rebalancing/migration when scaling from a small cluster of just a few nodes to many hundreds, without requiring an administrator carve the data set into static volumes or go through the tedious process of migrating data between servers. When the file system approaches full, new nodes can be easily added and things will “just work.”

Ceph includes flexible snapshot mechanism that allows a user to create a snapshot on any subdirectory (and its nested contents) in the system. Snapshot creation and deletion are as simple as `'mkdir .snap/foo'` and `'rmdir .snap/foo'` .

Ceph also provides some recursive accounting on directories for nested files and bytes. That is, a `'getfattr -d foo'` on any directory in the system will reveal the total number of nested regular files and subdirectories, and a summation of all nested file sizes. This makes the identification of large disk space consumers relatively quick, as no `'du'` or similar recursive scan of the file system is required.

Finally, Ceph also allows quotas to be set on any directory in the system. The quota can restrict the number of bytes or the number of files stored beneath that point in the directory hierarchy. Quotas can be set using extended attributes `'ceph.quota.max_files'` and `'ceph.quota.max_bytes'` , eg:

```
setfattr -n ceph.quota.max_bytes -v 1000000000 /some/dir
getfattr -n ceph.quota.max_bytes /some/dir
```

A limitation of the current quotas implementation is that it relies on the cooperation of the client mounting the file system to stop writers when a limit is reached. A modified or adversarial client cannot be prevented from writing as much data as it needs.

3.11.1 Mount Syntax

The basic mount syntax is:

```
# mount -t ceph monip[:port][,monip2[:port]...]:/[subdir] mnt
```

You only need to specify a single monitor, as the client will get the full list when it connects. (However, if the monitor you specify happens to be down, the mount won't succeed.) The port can be left off if the monitor is using the default. So if the monitor is at 1.2.3.4:

```
# mount -t ceph 1.2.3.4:/ /mnt/ceph
```

is sufficient. If `/sbin/mount.ceph` is installed, a hostname can be used instead of an IP address.

3.11.2 Mount Options

ip=A.B.C.D[:N] Specify the IP and/or port the client should bind to locally. There is normally not much reason to do this. If the IP is not specified, the client's IP address is determined by looking at the address its connection to the monitor originates from.

wsizex Specify the maximum write size in bytes. Default: 64 MB.

rsizex Specify the maximum read size in bytes. Default: 64 MB.

rasizex Specify the maximum readahead size in bytes. Default: 8 MB.

mount_timeout=X Specify the timeout value for mount (in seconds), in the case of a non-responsive Ceph file system. The default is 60 seconds.

caps_max=X Specify the maximum number of caps to hold. Unused caps are released when number of caps exceeds the limit. The default is 0 (no limit)

rbytes When `stat()` is called on a directory, set `st_size` to 'rbytes', the summation of file sizes over all files nested beneath that directory. This is the default.

norbytes When `stat()` is called on a directory, set `st_size` to the number of entries in that directory.

nocrc Disable CRC32C calculation for data writes. If set, the storage node must rely on TCP's error correction to detect data corruption in the data payload.

dcache Use the dcache contents to perform negative lookups and `readdir` when the client has the entire directory contents in its cache. (This does not change correctness; the client uses cached metadata only when a lease or capability ensures it is valid.)

nocache Do not use the dcache as above. This avoids a significant amount of complex code, sacrificing performance without affecting correctness, and is useful for tracking down bugs.

noasyncreaddir Do not use the dcache as above for `readdir`.

noquotadf Report overall filesystem usage in `statfs` instead of using the root directory quota.

nocopyfrom Don't use the RADOS 'copy-from' operation to perform remote object copies. Currently, it's only used in `copy_file_range`, which will revert to the default VFS implementation if this option is used.

recover_session=<no|clean> Set auto reconnect mode in the case where the client is blacklisted. The available modes are "no" and "clean". The default is "no".

- no: never attempt to reconnect when client detects that it has been blacklisted. Operations will generally fail after being blacklisted.
- clean: client reconnects to the ceph cluster automatically when it detects that it has been blacklisted. During reconnect, client drops dirty data/metadata, invalidates page caches and writable file handles. After reconnect, file locks become stale because the MDS loses track of them. If an inode contains any stale file locks, read/write on the inode is not allowed until applications release all stale file locks.

3.11.3 More Information

For more information on Ceph, see the home page at <https://ceph.com/>

The Linux kernel client source tree is available at

- <https://github.com/ceph/ceph-client.git>
- <git://git.kernel.org/pub/scm/linux/kernel/git/sage/ceph-client.git>

and the source for the full system is at <https://github.com/ceph/ceph.git>

3.12 Coda Kernel-Venus Interface

Note: This is one of the technical documents describing a component of Coda - this document describes the client kernel-Venus interface.

For more information:

<http://www.coda.cs.cmu.edu>

For user level software needed to run Coda:

<ftp://ftp.coda.cs.cmu.edu>

To run Coda you need to get a user level cache manager for the client, named Venus, as well as tools to manipulate ACLs, to log in, etc. The client needs to have the Coda filesystem selected in the kernel configuration.

The server needs a user level server and at present does not depend on kernel support.

The Venus kernel interface

Peter J. Braam

v1.0, Nov 9, 1997

This document describes the communication between Venus and kernel level filesystem code needed for the operation of the Coda file system. This document version is meant to describe the current interface (version 1.0) as well as improvements we envisage.

3.12.1 1. Introduction

A key component in the Coda Distributed File System is the cache manager, Venus.

When processes on a Coda enabled system access files in the Coda filesystem, requests are directed at the filesystem layer in the operating system. The operating system will communicate with Venus to service the request for the process. Venus manages a persistent client cache and makes remote procedure calls to Coda file servers and related servers (such as authentication servers) to service these requests it receives from the operating system. When Venus has serviced a request it replies to the operating system with appropriate return codes, and other data related to the request. Optionally the kernel support for Coda may maintain a minicache of recently processed requests to limit the number of interactions with Venus. Venus possesses the facility to inform the kernel when elements from its minicache are no longer valid.

This document describes precisely this communication between the kernel and Venus. The definitions of so called upcalls and downcalls will be given with the format of the data they handle. We shall also describe the semantic invariants resulting from the calls.

Historically Coda was implemented in a BSD file system in Mach 2.6. The interface between the kernel and Venus is very similar to the BSD VFS interface. Similar functionality is provided, and the format of the parameters and returned data is very similar to the BSD VFS. This leads to an almost natural environment for implementing a kernel-level filesystem driver for Coda in a BSD system. However, other operating systems such as Linux and Windows 95 and NT have virtual filesystem with different interfaces.

To implement Coda on these systems some reverse engineering of the Venus/Kernel protocol is necessary. Also it came to light that other systems could profit significantly from certain small optimizations and modifications to the protocol. To facilitate this work as well as to make future ports easier, communication between Venus and the kernel should be documented in great detail. This is the aim of this document.

3.12.2 2. Servicing Coda filesystem calls

The service of a request for a Coda file system service originates in a process P which accessing a Coda file. It makes a system call which traps to the OS kernel. Examples of such calls trapping to the kernel are read, write, open, close, create, mkdir, rmdir, chmod in a Unix ontext. Similar calls exist in the Win32 environment, and are named CreateFile.

Generally the operating system handles the request in a virtual filesystem (VFS) layer, which is named I/O Manager in NT and IFS manager in Windows 95. The VFS is responsible for partial processing of the request and for locating the specific filesystem(s) which will service parts of the request. Usually the information in the path assists in locating

the correct FS drivers. Sometimes after extensive pre-processing, the VFS starts invoking exported routines in the FS driver. This is the point where the FS specific processing of the request starts, and here the Coda specific kernel code comes into play.

The FS layer for Coda must expose and implement several interfaces. First and foremost the VFS must be able to make all necessary calls to the Coda FS layer, so the Coda FS driver must expose the VFS interface as applicable in the operating system. These differ very significantly among operating systems, but share features such as facilities to read/write and create and remove objects. The Coda FS layer services such VFS requests by invoking one or more well defined services offered by the cache manager Venus. When the replies from Venus have come back to the FS driver, servicing of the VFS call continues and finishes with a reply to the kernel's VFS. Finally the VFS layer returns to the process.

As a result of this design a basic interface exposed by the FS driver must allow Venus to manage message traffic. In particular Venus must be able to retrieve and place messages and to be notified of the arrival of a new message. The notification must be through a mechanism which does not block Venus since Venus must attend to other tasks even when no messages are waiting or being processed.

Interfaces of the Coda FS Driver

Furthermore the FS layer provides for a special path of communication between a user process and Venus, called the `pioctl` interface. The `pioctl` interface is used for Coda specific services, such as requesting detailed information about the persistent cache managed by Venus. Here the involvement of the kernel is minimal. It identifies the calling process and passes the information on to Venus. When Venus replies the response is passed back to the caller in unmodified form.

Finally Venus allows the kernel FS driver to cache the results from certain services. This is done to avoid excessive context switches and results in an efficient system. However, Venus may acquire information, for example from the network which implies that cached information must be flushed or replaced. Venus then makes a downcall to the Coda FS layer to request flushes or updates in the cache. The kernel FS driver handles such requests synchronously.

Among these interfaces the VFS interface and the facility to place, receive and be notified of messages are platform specific. We will not go into the calls exported to the VFS layer but we will state the requirements of the message exchange mechanism.

3.12.3 3. The message layer

At the lowest level the communication between Venus and the FS driver proceeds through messages. The synchronization between processes requesting Coda file service and Venus relies on blocking and waking up processes. The Coda FS driver processes VFS- and piocctl-requests on behalf of a process P, creates messages for Venus, awaits replies and finally returns to the caller. The implementation of the exchange of messages is platform specific, but the semantics have (so far) appeared to be generally applicable. Data buffers are created by the FS Driver in kernel memory on behalf of P and copied to user memory in Venus.

The FS Driver while servicing P makes upcalls to Venus. Such an upcall is dispatched to Venus by creating a message structure. The structure contains the identification of P, the message sequence number, the size of the request and a pointer to the data in kernel memory for the request. Since the data buffer is re-used to hold the reply from Venus, there is a field for the size of the reply. A flags field is used in the message to precisely record the status of the message. Additional platform dependent structures involve pointers to determine the position of the message on queues and pointers to synchronization objects. In the upcall routine the message structure is filled in, flags are set to 0, and it is placed on the pending queue. The routine calling upcall is responsible for allocating the data buffer; its structure will be described in the next section.

A facility must exist to notify Venus that the message has been created, and implemented using available synchronization objects in the OS. This notification is done in the upcall context of the process P. When the message is on the pending queue, process P cannot proceed in upcall. The (kernel mode) processing of P in the filesystem request routine must be suspended until Venus has replied. Therefore the calling thread in P is blocked in upcall. A pointer in the message structure will locate the synchronization object on which P is sleeping.

Venus detects the notification that a message has arrived, and the FS driver allow Venus to retrieve the message with a `getmsg_from_kernel` call. This action finishes in the kernel by putting the message on the queue of processing messages and setting flags to READ. Venus is passed the contents of the data buffer. The `getmsg_from_kernel` call now returns and Venus processes the request.

At some later point the FS driver receives a message from Venus, namely when Venus calls `sendmsg_to_kernel`. At this moment the Coda FS driver looks at the contents of the message and decides if:

- the message is a reply for a suspended thread P. If so it removes the message from the processing queue and marks the message as WRITTEN. Finally, the FS driver unblocks P (still in the kernel mode context of Venus) and the `sendmsg_to_kernel` call returns to Venus. The process P will be scheduled at some point and continues processing its upcall with the data buffer replaced with the reply from Venus.
- The message is a downcall. A downcall is a request from Venus

to the FS Driver. The FS driver processes the request immediately (usually a cache eviction or replacement) and when it finishes `sendmsg_to_kernel` returns.

Now P awakes and continues processing upcall. There are some subtleties to take account of. First P will determine if it was woken up in upcall by a signal from some other source (for example an attempt to terminate P) or as is normally the case by Venus in its `sendmsg_to_kernel` call. In the normal case, the upcall routine will deallocate the message structure and return. The FS routine can proceed with its processing.

Sleeping and IPC arrangements

In case P is woken up by a signal and not by Venus, it will first look at the flags field. If the message is not yet READ, the process P can handle its signal without notifying Venus. If Venus has READ, and the request should not be processed, P can send Venus a signal message to indicate that it should disregard the previous message. Such signals are put in the queue at the head, and read first by Venus. If the message is already marked as WRITTEN it is too late to stop the processing. The VFS routine will now continue. (- If a VFS request involves more than one upcall, this can lead to complicated state, an extra field “handle_signals” could be added in the message structure to indicate points of no return have been passed.-)

3.1. Implementation details

The Unix implementation of this mechanism has been through the implementation of a character device associated with Coda. Venus retrieves messages by doing a read on the device, replies are sent with a write and notification is through the select system call on the file descriptor for the device. The process P is kept waiting on an interruptible wait queue object.

In Windows NT and the DPMI Windows 95 implementation a `DeviceIoControl` call is used. The `DeviceIoControl` call is designed to copy buffers from user memory to kernel memory with `OPCODES`. The `sendmsg_to_kernel` is issued as a synchronous call, while the `getmsg_from_kernel` call is asynchronous. Windows `EventObjects` are used for notification of message arrival. The process P is kept waiting on a `KernelEvent` object in NT and a semaphore in Windows 95.

3.12.4 4. The interface at the call level

This section describes the upcalls a Coda FS driver can make to Venus. Each of these upcalls make use of two structures: `inputArgs` and `outputArgs`. In pseudo BNF form the structures take the following form:

```
struct inputArgs {
    u_long opcode;
    u_long unique;      /* Keep multiple outstanding msgs distinct.
↳*/
```

(continues on next page)

(continued from previous page)

```

    u_short pid;           /* Common to all */
    u_short pgid;         /* Common to all */
    struct CodaCred cred; /* Common to all */

    <union "in" of call dependent parts of inputArgs>
};

struct outputArgs {
    u_long opcode;
    u_long unique;      /* Keep multiple outstanding msgs.
↳distinct */
    u_long result;

    <union "out" of call dependent parts of inputArgs>
};

```

Before going on let us elucidate the role of the various fields. The inputArgs start with the opcode which defines the type of service requested from Venus. There are approximately 30 upcalls at present which we will discuss. The unique field labels the inputArg with a unique number which will identify the message uniquely. A process and process group id are passed. Finally the credentials of the caller are included.

Before delving into the specific calls we need to discuss a variety of data structures shared by the kernel and Venus.

4.1. Data structures shared by the kernel and Venus

The CodaCred structure defines a variety of user and group ids as they are set for the calling process. The `vuid_t` and `vgid_t` are 32 bit unsigned integers. It also defines group membership in an array. On Unix the CodaCred has proven sufficient to implement good security semantics for Coda but the structure may have to undergo modification for the Windows environment when these mature:

```

struct CodaCred {
    vuid_t cr_uid, cr_euid, cr_suid, cr_fsuid; /* Real, effective,
↳set, fs uid */
    vgid_t cr_gid, cr_egid, cr_sgid, cr_fsgid; /* same for groups.
↳*/
    vgid_t cr_groups[NGROUPS]; /* Group membership for
↳caller */
};

```

Note: It is questionable if we need CodaCredits in Venus. Finally Venus doesn't know about groups, although it does create files with the default uid/gid. Perhaps the list of group membership is superfluous.

The next item is the fundamental identifier used to identify Coda files, the ViceFid. A fid of a file uniquely defines a file or directory in the Coda

filesystem within a cell¹:

```
typedef struct ViceFid {
    VolumeId Volume;
    VnodeId Vnode;
    Unique_t Unique;
} ViceFid;
```

Each of the constituent fields: VolumeId, VnodeId and Unique_t are unsigned 32 bit integers. We envisage that a further field will need to be prefixed to identify the Coda cell; this will probably take the form of a Ipv6 size IP address naming the Coda cell through DNS.

The next important structure shared between Venus and the kernel is the attributes of the file. The following structure is used to exchange information. It has room for future extensions such as support for device files (currently not present in Coda):

```
struct coda_timespec {
    int64_t      tv_sec;          /* seconds */
    long         tv_nsec;        /* nanoseconds */
};

struct coda_vattr {
    enum coda_vtype va_type;     /* vnode type (for
↪ create) */
    u_short       va_mode;       /* files access mode and
↪ type */
    short         va_nlink;      /* number of references
↪ to file */
    void_t       va_uid;         /* owner user id */
    void_t       va_gid;         /* owner group id */
    long         va_fsid;        /* file system id (dev
↪ for now) */
    long         va_fileid;      /* file id */
    u_quad_t     va_size;        /* file size in bytes */
    long         va_blocksize;   /* blocksize preferred
↪ for i/o */
    struct coda_timespec va_atime; /* time of last access */
    struct coda_timespec va_mtime; /* time of last
↪ modification */
    struct coda_timespec va_ctime; /* time file changed */
    u_long       va_gen;         /* generation number of
↪ file */
    u_long       va_flags;       /* flags defined for file
↪ */
    dev_t        va_rdev;        /* device special file
↪ represents */
    u_quad_t     va_bytes;       /* bytes of disk space
↪ held by file */
    u_quad_t     va_filerev;     /* file modification
↪ number */
    u_int        va_vaflags;     /* operations flags, see
↪ below */
};
```

(continues on next page)

¹ A cell is a group of Coda servers acting under the aegis of a single system control machine or SCM. See the Coda Administration manual for a detailed description of the role of the SCM.

(continued from previous page)

```
};      long          va_spare;      /* remain quad aligned */
```

4.2. The pioctl interface

Coda specific requests can be made by application through the pioctl interface. The pioctl is implemented as an ordinary ioctl on a fictitious file /coda/CONTROL. The pioctl call opens this file, gets a file handle and makes the ioctl call. Finally it closes the file.

The kernel involvement in this is limited to providing the facility to open and close and pass the ioctl message and to verify that a path in the pioctl data buffers is a file in a Coda filesystem.

The kernel is handed a data packet of the form:

```
struct {
    const char *path;
    struct ViceIoctl vidata;
    int follow;
} data;
```

where:

```
struct ViceIoctl {
    caddr_t in, out;          /* Data to be transferred in, or ↵
↵out */
    short in_size;          /* Size of input buffer <= 2K */
    short out_size;        /* Maximum size of output buffer,
↵<= 2K */
};
```

The path must be a Coda file, otherwise the ioctl upcall will not be made.

Note: The data structures and code are a mess. We need to clean this up.

We now proceed to document the individual calls:

4.3. root

Arguments in

empty

out:

```
struct cfs_root_out {
    ViceFid VFid;
} cfs_root;
```

Description This call is made to Venus during the initialization of the Coda filesystem. If the result is zero, the `cfs_root` structure contains the ViceFid of the root of the Coda filesystem. If a non-zero result is generated, its value is a platform dependent error code indicating the difficulty Venus encountered in locating the root of the Coda filesystem.

4.4. lookup

Summary Find the ViceFid and type of an object in a directory if it exists.

Arguments in:

```
struct cfs_lookup_in {
    ViceFid    VFid;
    char      *name;           /* Place holder for data. */
} cfs_lookup;
```

out:

```
struct cfs_lookup_out {
    ViceFid VFid;
    int vtype;
} cfs_lookup;
```

Description This call is made to determine the ViceFid and filetype of a directory entry. The directory entry requested carries name `name` and Venus will search the directory identified by `cfs_lookup_in.VFid`. The result may indicate that the name does not exist, or that difficulty was encountered in finding it (e.g. due to disconnection). If the result is zero, the field `cfs_lookup_out.VFid` contains the targets ViceFid and `cfs_lookup_out.vtype` the `coda_vtype` giving the type of object the name designates.

The name of the object is an 8 bit character string of maximum length `CFS_MAXNAMLEN`, currently set to 256 (including a 0 terminator.)

It is extremely important to realize that Venus bitwise ors the field `cfs_lookup.vtype` with `CFS_NOCACHE` to indicate that the object should not be put in the kernel name cache.

Note: The type of the `vtype` is currently wrong. It should be `coda_vtype`. Linux does not take note of `CFS_NOCACHE`. It should.

4.5. getattr

Summary Get the attributes of a file.

Arguments in:

```
struct cfs_getattr_in {
    ViceFid VFid;
    struct coda_vattr attr; /* XXXXX */
} cfs_getattr;
```

out:

```
struct cfs_getattr_out {
    struct coda_vattr attr;
} cfs_getattr;
```

Description This call returns the attributes of the file identified by fid.

Errors Errors can occur if the object with fid does not exist, is unaccessible or if the caller does not have permission to fetch attributes.

Note: Many kernel FS drivers (Linux, NT and Windows 95) need to acquire the attributes as well as the Fid for the instantiation of an internal “inode” or “FileHandle” . A significant improvement in performance on such systems could be made by combining the lookup and getattr calls both at the Venus/kernel interaction level and at the RPC level.

The vattr structure included in the input arguments is superfluous and should be removed.

4.6. setattr

Summary Set the attributes of a file.

Arguments in:

```
struct cfs_setattr_in {
    ViceFid VFid;
    struct coda_vattr attr;
} cfs_setattr;
```

out

empty

Description The structure attr is filled with attributes to be changed in BSD style. Attributes not to be changed are set to -1, apart from vtype which is set to VNON. Other are set to the value to be assigned. The only attributes which the FS driver may request to change are the mode, owner, groupid, atime, mtime and ctime. The return value indicates success or failure.

Errors A variety of errors can occur. The object may not exist, may be inaccessible, or permission may not be granted by Venus.

4.7. access

Arguments in:

```
struct cfs_access_in {
    ViceFid    VFid;
    int flags;
} cfs_access;
```

out

empty

Description Verify if access to the object identified by VFid for operations described by flags is permitted. The result indicates if access will be granted. It is important to remember that Coda uses ACLs to enforce protection and that ultimately the servers, not the clients enforce the security of the system. The result of this call will depend on whether a token is held by the user.

Errors The object may not exist, or the ACL describing the protection may not be accessible.

4.8. create

Summary Invoked to create a file

Arguments in:

```
struct cfs_create_in {
    ViceFid VFid;
    struct coda_vattr attr;
    int excl;
    int mode;
    char      *name;          /* Place holder for data. */
} cfs_create;
```

out:

```
struct cfs_create_out {
    ViceFid VFid;
    struct coda_vattr attr;
} cfs_create;
```

Description This upcall is invoked to request creation of a file. The file will be created in the directory identified by VFid, its name will be name, and the mode will be mode. If excl is set an error will be returned if the file already exists. If the size field in attr is set to zero the file will be truncated. The uid and gid of the file are set by converting the CodaCred to a uid using a macro CRTOUID (this macro is platform dependent). Upon success the VFid and attributes of the file are returned. The Coda FS Driver will normally instantiate a vnode, inode or file handle at kernel level for the new object.

Errors A variety of errors can occur. Permissions may be insufficient. If the object exists and is not a file the error EISDIR is returned under Unix.

Note: The packing of parameters is very inefficient and appears to indicate confusion between the system call `creat` and the VFS operation `create`. The VFS operation `create` is only called to create new objects. This `create` call differs from the Unix one in that it is not invoked to return a file descriptor. The `truncate` and `exclusive` options, together with the `mode`, could simply be part of the `mode` as it is under Unix. There should be no `flags` argument; this is used in `open (2)` to return a file descriptor for `READ` or `WRITE` mode.

The attributes of the directory should be returned too, since the size and `mtime` changed.

4.9. `mkdir`

Summary Create a new directory.

Arguments in:

```
struct cfs_mkdir_in {
    ViceFid    VFid;
    struct coda_vattr attr;
    char      *name;           /* Place holder for data. */
} cfs_mkdir;
```

out:

```
struct cfs_mkdir_out {
    ViceFid VFid;
    struct coda_vattr attr;
} cfs_mkdir;
```

Description This call is similar to `create` but creates a directory. Only the `mode` field in the input parameters is used for creation. Upon successful creation, the `attr` returned contains the attributes of the new directory.

Errors As for `create`.

Note: The input parameter should be changed to `mode` instead of `attributes`.

The attributes of the parent should be returned since the size and `mtime` changes.

4.10. link

Summary Create a link to an existing file.

Arguments in:

```
struct cfs_link_in {
    ViceFid sourceFid;          /* cnode to link *to* */
    ViceFid destFid;           /* Directory in which to
↳ place link */
    char      *tname;          /* Place holder for data. */
} cfs_link;
```

out

empty

Description This call creates a link to the sourceFid in the directory identified by destFid with name tname. The source must reside in the target's parent, i.e. the source must be have parent destFid, i.e. Coda does not support cross directory hard links. Only the return value is relevant. It indicates success or the type of failure.

Errors The usual errors can occur.

4.11. symlink

Summary create a symbolic link

Arguments in:

```
struct cfs_symlink_in {
    ViceFid VFid;              /* Directory to put symlink in
↳ */
    char      *srcname;
    struct coda_vattr attr;
    char      *tname;
} cfs_symlink;
```

out

none

Description Create a symbolic link. The link is to be placed in the directory identified by VFid and named tname. It should point to the pathname srcname. The attributes of the newly created object are to be set to attr.

Note: The attributes of the target directory should be returned since its size changed.

4.12. remove

Summary Remove a file

Arguments in:

```
struct cfs_remove_in {
    ViceFid    VFid;
    char       *name;           /* Place holder for data. */
} cfs_remove;
```

out

none

Description Remove file named `cfs_remove_in.name` in directory identified by `VFid`.

Note: The attributes of the directory should be returned since its `mtime` and `size` may change.

4.13. rmdir

Summary Remove a directory

Arguments in:

```
struct cfs_rmdir_in {
    ViceFid    VFid;
    char       *name;           /* Place holder for data. */
} cfs_rmdir;
```

out

none

Description Remove the directory with name `name` from the directory identified by `VFid`.

Note: The attributes of the parent directory should be returned since its `mtime` and `size` may change.

4.14. readlink

Summary Read the value of a symbolic link.

Arguments in:

```
struct cfs_readlink_in {
    ViceFid VFid;
} cfs_readlink;
```

out:

```
struct cfs_readlink_out {
    int count;
    caddr_t data;          /* Place holder for data. */
} cfs_readlink;
```

Description This routine reads the contents of symbolic link identified by VFid into the buffer data. The buffer data must be able to hold any name up to CFS_MAXNAMLEN (PATH or NAM??).

Errors No unusual errors.

4.15. open

Summary Open a file.

Arguments in:

```
struct cfs_open_in {
    ViceFid VFid;
    int flags;
} cfs_open;
```

out:

```
struct cfs_open_out {
    dev_t dev;
    ino_t inode;
} cfs_open;
```

Description This request asks Venus to place the file identified by VFid in its cache and to note that the calling process wishes to open it with flags as in open(2). The return value to the kernel differs for Unix and Windows systems. For Unix systems the Coda FS Driver is informed of the device and inode number of the container file in the fields dev and inode. For Windows the path of the container file is returned to the kernel.

Note: Currently the cfs_open_out structure is not properly adapted to deal with the Windows case. It might be best to implement two upcalls, one to open aiming at a container file name, the other at a container file inode.

4.16. close

Summary Close a file, update it on the servers.

Arguments in:

```
struct cfs_close_in {
    ViceFid    VFid;
    int flags;
} cfs_close;
```

out

none

Description Close the file identified by VFid.

Note: The flags argument is bogus and not used. However, Venus' code has room to deal with an excec input field, probably this field should be used to inform Venus that the file was closed but is still memory mapped for execution. There are comments about fetching versus not fetching the data in Venus vproc_vfscalls. This seems silly. If a file is being closed, the data in the container file is to be the new data. Here again the excec flag might be in play to create confusion: currently Venus might think a file can be flushed from the cache when it is still memory mapped. This needs to be understood.

4.17. ioctl

Summary Do an ioctl on a file. This includes the pioctl interface.

Arguments in:

```
struct cfs_ioctl_in {
    ViceFid VFid;
    int cmd;
    int len;
    int rwflag;
    char *data;                /* Place holder for data. */
} cfs_ioctl;
```

out:

```
struct cfs_ioctl_out {
    int len;
    caddr_t data;             /* Place holder for data. */
} cfs_ioctl;
```

Description Do an ioctl operation on a file. The command, len and data arguments are filled as usual. flags is not used by Venus.

Note: Another bogus parameter. flags is not used. What is the business

about PREFETCHING in the Venus code?

4.18. rename

Summary Rename a fid.

Arguments in:

```
struct cfs_rename_in {
    ViceFid    sourceFid;
    char       *srcname;
    ViceFid    destFid;
    char       *destname;
} cfs_rename;
```

out

none

Description Rename the object with name srcname in directory sourceFid to destname in destFid. It is important that the names srcname and destname are 0 terminated strings. Strings in Unix kernels are not always null terminated.

4.19. readdir

Summary Read directory entries.

Arguments in:

```
struct cfs_readdir_in {
    ViceFid    VFid;
    int count;
    int offset;
} cfs_readdir;
```

out:

```
struct cfs_readdir_out {
    int size;
    caddr_t    data;          /* Place holder for data. */
} cfs_readdir;
```

Description Read directory entries from VFid starting at offset and read at most count bytes. Returns the data in data and returns the size in size.

Note: This call is not used. Readdir operations exploit container files. We will re-evaluate this during the directory revamp which is about to take place.

4.20. vget

Summary instructs Venus to do an FSDB->Get.

Arguments in:

```
struct cfs_vget_in {
    ViceFid VFid;
} cfs_vget;
```

out:

```
struct cfs_vget_out {
    ViceFid VFid;
    int vtype;
} cfs_vget;
```

Description This upcall asks Venus to do a get operation on an fsobj labelled by VFid.

Note: This operation is not used. However, it is extremely useful since it can be used to deal with read/write memory mapped files. These can be “pinned” in the Venus cache using vget and released with inactive.

4.21. fsync

Summary Tell Venus to update the RVM attributes of a file.

Arguments in:

```
struct cfs_fsync_in {
    ViceFid VFid;
} cfs_fsync;
```

out

none

Description Ask Venus to update RVM attributes of object VFid. This should be called as part of kernel level fsync type calls. The result indicates if the syncing was successful.

Note: Linux does not implement this call. It should.

4.22. inactive

Summary Tell Venus a vnode is no longer in use.

Arguments in:

```
struct cfs_inactive_in {
    ViceFid VFid;
} cfs_inactive;
```

out

none

Description This operation returns EOPNOTSUPP.

Note: This should perhaps be removed.

4.23. rdwr

Summary Read or write from a file

Arguments in:

```
struct cfs_rdwr_in {
    ViceFid VFid;
    int rwflag;
    int count;
    int offset;
    int ioflag;
    caddr_t data; /* Place holder for data. */
} cfs_rdwr;
```

out:

```
struct cfs_rdwr_out {
    int rwflag;
    int count;
    caddr_t data; /* Place holder for data. */
} cfs_rdwr;
```

Description This upcall asks Venus to read or write from a file.

Note: It should be removed since it is against the Coda philosophy that read/write operations never reach Venus. I have been told the operation does not work. It is not currently used.

4.24. ody_mount

Summary Allows mounting multiple Coda “filesystems” on one Unix mount point.

Arguments in:

```
struct ody_mount_in {
    char      *name;           /* Place holder for data. */
} ody_mount;
```

out:

```
struct ody_mount_out {
    ViceFid VFid;
} ody_mount;
```

Description Asks Venus to return the rootfid of a Coda system named name. The fid is returned in VFid.

Note: This call was used by David for dynamic sets. It should be removed since it causes a jungle of pointers in the VFS mounting area. It is not used by Coda proper. Call is not implemented by Venus.

4.25. ody_lookup

Summary Looks up something.

Arguments in

irrelevant

out

irrelevant

Note: Gut it. Call is not implemented by Venus.

4.26. ody_expand

Summary expands something in a dynamic set.

Arguments in

irrelevant

out

irrelevant

Note: Gut it. Call is not implemented by Venus.

4.27. prefetch

Summary Prefetch a dynamic set.

Arguments

in

Not documented.

out

Not documented.

Description Venus worker.cc has support for this call, although it is noted that it doesn't work. Not surprising, since the kernel does not have support for it. (ODY_PREFETCH is not a defined operation).

Note: Gut it. It isn't working and isn't used by Coda.

4.28. signal

Summary Send Venus a signal about an upcall.

Arguments in

none

out

not applicable.

Description This is an out-of-band upcall to Venus to inform Venus that the calling process received a signal after Venus read the message from the input queue. Venus is supposed to clean up the operation.

Errors No reply is given.

Note: We need to better understand what Venus needs to clean up and if it is doing this correctly. Also we need to handle multiple upcall per system call situations correctly. It would be important to know what state changes in Venus take place after an upcall for which the kernel is responsible for notifying Venus to clean up (e.g. open definitely is such a state change, but many others are maybe not).

3.12.5 5. The minicache and downcalls

The Coda FS Driver can cache results of lookup and access upcalls, to limit the frequency of upcalls. Upcalls carry a price since a process context switch needs to take place. The counterpart of caching the information is that Venus will notify the FS Driver that cached entries must be flushed or renamed.

The kernel code generally has to maintain a structure which links the internal file handles (called vnodes in BSD, inodes in Linux and FileHandles in Windows) with the ViceFid's which Venus maintains. The reason is that frequent translations back and forth are needed in order to make upcalls and use the results of upcalls. Such linking objects are called cnodes.

The current minicache implementations have cache entries which record the following:

1. the name of the file
2. the cnode of the directory containing the object
3. a list of CodaCred's for which the lookup is permitted.
4. the cnode of the object

The lookup call in the Coda FS Driver may request the cnode of the desired object from the cache, by passing its name, directory and the CodaCred's of the caller. The cache will return the cnode or indicate that it cannot be found. The Coda FS Driver must be careful to invalidate cache entries when it modifies or removes objects.

When Venus obtains information that indicates that cache entries are no longer valid, it will make a downcall to the kernel. Downcalls are intercepted by the Coda FS Driver and lead to cache invalidations of the kind described below. The Coda FS Driver does not return an error unless the downcall data could not be read into kernel memory.

5.1. INVALIDATE

No information is available on this call.

5.2. FLUSH

Arguments None

Summary Flush the name cache entirely.

Description Venus issues this call upon startup and when it dies. This is to prevent stale cache information being held. Some operating systems allow the kernel name cache to be switched off dynamically. When this is done, this downcall is made.

5.3. PURGEUSER

Arguments

```
struct cfs_purgeuser_out { /* CFS_PURGEUSER is a venus->kernel
↪call */
    struct CodaCred cred;
} cfs_purgeuser;
```

Description Remove all entries in the cache carrying the Cred. This call is issued when tokens for a user expire or are flushed.

5.4. ZAPFILE

Arguments

```
struct cfs_zapfile_out { /* CFS_ZAPFILE is a venus->kernel
↪call */
    ViceFid CodaFid;
} cfs_zapfile;
```

Description Remove all entries which have the (dir vnode, name) pair. This is issued as a result of an invalidation of cached attributes of a vnode.

Note: Call is not named correctly in NetBSD and Mach. The minicache zapfile routine takes different arguments. Linux does not implement the invalidation of attributes correctly.

5.5. ZAPDIR

Arguments

```
struct cfs_zapdir_out { /* CFS_ZAPDIR is a venus->kernel
↪call */
    ViceFid CodaFid;
} cfs_zapdir;
```

Description Remove all entries in the cache lying in a directory CodaFid, and all children of this directory. This call is issued when Venus receives a callback on the directory.

5.6. ZAPVNODE

Arguments

```
struct cfs_zapvnode_out { /* CFS_ZAPVNODE is a venus->kernel_
↪call */
    struct CodaCred cred;
    ViceFid VFid;
} cfs_zapvnode;
```

Description Remove all entries in the cache carrying the cred and VFid as in the arguments. This downcall is probably never issued.

5.7. PURGEFID

Arguments

```
struct cfs_purgefid_out { /* CFS_PURGEFID is a venus->kernel_
↪call */
    ViceFid CodaFid;
} cfs_purgefid;
```

Description Flush the attribute for the file. If it is a dir (odd vnode), purge its children from the namecache and remove the file from the namecache.

5.8. REPLACE

Summary Replace the Fid' s for a collection of names.

Arguments

```
struct cfs_replace_out { /* cfs_replace is a venus->kernel_
↪call */
    ViceFid NewFid;
    ViceFid OldFid;
} cfs_replace;
```

Description This routine replaces a ViceFid in the name cache with another. It is added to allow Venus during reintegration to replace locally allocated temp fids while disconnected with global fids even when the reference counts on those fids are not zero.

3.12.6 6. Initialization and cleanup

This section gives brief hints as to desirable features for the Coda FS Driver at startup and upon shutdown or Venus failures. Before entering the discussion it is useful to repeat that the Coda FS Driver maintains the following data:

1. message queues
2. cnodes

3. name cache entries

The name cache entries are entirely private to the driver, so they can easily be manipulated. The message queues will generally have clear points of initialization and destruction. The cnodes are much more delicate. User processes hold reference counts in Coda filesystems and it can be difficult to clean up the cnodes.

It can expect requests through:

1. the message subsystem
2. the VFS layer
3. pioctl interface

Currently the pioctl passes through the VFS for Coda so we can treat these similarly.

6.1. Requirements

The following requirements should be accommodated:

1. The message queues should have open and close routines. On Unix the opening of the character devices are such routines.
 - Before opening, no messages can be placed.
 - Opening will remove any old messages still pending.
 - Close will notify any sleeping processes that their upcall cannot be completed.
 - Close will free all memory allocated by the message queues.
2. At open the namecache shall be initialized to empty state.
3. Before the message queues are open, all VFS operations will fail. Fortunately this can be achieved by making sure than mounting the Coda filesystem cannot succeed before opening.
4. After closing of the queues, no VFS operations can succeed. Here one needs to be careful, since a few operations (lookup, read/write, readdir) can proceed without upcalls. These must be explicitly blocked.
5. Upon closing the namecache shall be flushed and disabled.
6. All memory held by cnodes can be freed without relying on upcalls.
7. Unmounting the file system can be done without relying on upcalls.
8. Mounting the Coda filesystem should fail gracefully if Venus cannot get the rootfid or the attributes of the rootfid. The latter is best implemented by Venus fetching these objects before attempting to mount.

Note: NetBSD in particular but also Linux have not implemented the above requirements fully. For smooth operation this needs to be corrected.

3.13 Configfs - Userspace-driven Kernel Object Configuration

Joel Becker <joel.becker@oracle.com>

Updated: 31 March 2005

Copyright (c) 2005 Oracle Corporation, Joel
<joel.becker@oracle.com>

Becker

3.13.1 What is configfs?

configfs is a ram-based filesystem that provides the converse of sysfs' s functionality. Where sysfs is a filesystem-based view of kernel objects, configfs is a filesystem-based manager of kernel objects, or config_items.

With sysfs, an object is created in kernel (for example, when a device is discovered) and it is registered with sysfs. Its attributes then appear in sysfs, allowing userspace to read the attributes via readdir(3)/read(2). It may allow some attributes to be modified via write(2). The important point is that the object is created and destroyed in kernel, the kernel controls the lifecycle of the sysfs representation, and sysfs is merely a window on all this.

A configfs config_item is created via an explicit userspace operation: mkdir(2). It is destroyed via rmdir(2). The attributes appear at mkdir(2) time, and can be read or modified via read(2) and write(2). As with sysfs, readdir(3) queries the list of items and/or attributes. symlink(2) can be used to group items together. Unlike sysfs, the lifetime of the representation is completely driven by userspace. The kernel modules backing the items must respond to this.

Both sysfs and configfs can and should exist together on the same system. One is not a replacement for the other.

3.13.2 Using configfs

configfs can be compiled as a module or into the kernel. You can access it by doing:

```
mount -t configfs none /config
```

The configfs tree will be empty unless client modules are also loaded. These are modules that register their item types with configfs as subsystems. Once a client subsystem is loaded, it will appear as a subdirectory (or more than one) under /config. Like sysfs, the configfs tree is always there, whether mounted on /config or not.

An item is created via `mkdir(2)`. The item's attributes will also appear at this time. `readdir(3)` can determine what the attributes are, `read(2)` can query their default values, and `write(2)` can store new values. Don't mix more than one attribute in one attribute file.

There are two types of configfs attributes:

- Normal attributes, which similar to sysfs attributes, are small ASCII text files, with a maximum size of one page (`PAGE_SIZE`, 4096 on i386). Preferably only one value per file should be used, and the same caveats from sysfs apply. Configfs expects `write(2)` to store the entire buffer at once. When writing to normal configfs attributes, userspace processes should first read the entire file, modify the portions they wish to change, and then write the entire buffer back.
- Binary attributes, which are somewhat similar to sysfs binary attributes, but with a few slight changes to semantics. The `PAGE_SIZE` limitation does not apply, but the whole binary item must fit in single kernel `vmalloc`'ed buffer. The `write(2)` calls from user space are buffered, and the attributes' `write_bin_attribute` method will be invoked on the final close, therefore it is imperative for user-space to check the return code of `close(2)` in order to verify that the operation finished successfully. To avoid a malicious user OOMing the kernel, there's a per-binary attribute maximum buffer value.

When an item needs to be destroyed, remove it with `rmdir(2)`. An item cannot be destroyed if any other item has a link to it (via `symlink(2)`). Links can be removed via `unlink(2)`.

3.13.3 Configuring FakeNBD: an Example

Imagine there's a Network Block Device (NBD) driver that allows you to access remote block devices. Call it FakeNBD. FakeNBD uses configfs for its configuration. Obviously, there will be a nice program that sysadmins use to configure FakeNBD, but somehow that program has to tell the driver about it. Here's where configfs comes in.

When the FakeNBD driver is loaded, it registers itself with configfs. `readdir(3)` sees this just fine:

```
# ls /config
fakenbd
```

A `fakenbd` connection can be created with `mkdir(2)`. The name is arbitrary, but likely the tool will make some use of the name. Perhaps it is a uuid or a disk name:

```
# mkdir /config/fakenbd/disk1
# ls /config/fakenbd/disk1
target device rw
```

The `target` attribute contains the IP address of the server FakeNBD will connect to. The `device` attribute is the device on the server. Predictably, the `rw` attribute determines whether the connection is read-only or read-write:

```
# echo 10.0.0.1 > /config/fakenbd/disk1/target
# echo /dev/sda1 > /config/fakenbd/disk1/device
# echo 1 > /config/fakenbd/disk1/rw
```

That's it. That's all there is. Now the device is configured, via the shell no less.

3.13.4 Coding With configs

Every object in configs is a `config_item`. A `config_item` reflects an object in the subsystem. It has attributes that match values on that object. `configs` handles the filesystem representation of that object and its attributes, allowing the subsystem to ignore all but the basic show/store interaction.

Items are created and destroyed inside a `config_group`. A group is a collection of items that share the same attributes and operations. Items are created by `mkdir(2)` and removed by `rmdir(2)`, but `configs` handles that. The group has a set of operations to perform these tasks

A subsystem is the top level of a client module. During initialization, the client module registers the subsystem with `configs`, the subsystem appears as a directory at the top of the `configs` filesystem. A subsystem is also a `config_group`, and can do everything a `config_group` can.

3.13.5 struct config_item

```
struct config_item {
    char                *ci_name;
    char                ci_namebuf[UOBJ_NAME_LEN];
    struct kref         ci_kref;
    struct list_head    ci_entry;
    struct config_item *ci_parent;
    struct config_group *ci_group;
    struct config_item_type *ci_type;
    struct dentry       *ci_dentry;
};

void config_item_init(struct config_item *);
void config_item_init_type_name(struct config_item *,
                                const char *name,
                                struct config_item_type *type);
struct config_item *config_item_get(struct config_item *);
void config_item_put(struct config_item *);
```

Generally, `struct config_item` is embedded in a container structure, a structure that actually represents what the subsystem is doing. The `config_item` portion of that structure is how the object interacts with `configs`.

Whether statically defined in a source file or created by a parent `config_group`, a `config_item` must have one of the `_init()` functions called on it. This initializes the reference count and sets up the appropriate fields.

All users of a `config_item` should have a reference on it via `config_item_get()`, and drop the reference when they are done via `config_item_put()`.

By itself, a `config_item` cannot do much more than appear in configs. Usually a subsystem wants the item to display and/or store attributes, among other things. For that, it needs a type.

3.13.6 struct config_item_type

```
struct configfs_item_operations {
    void (*release)(struct config_item *);
    int (*allow_link)(struct config_item *src,
                     struct config_item *target);
    void (*drop_link)(struct config_item *src,
                     struct config_item *target);
};

struct config_item_type {
    struct module                               *ct_owner;
    struct configfs_item_operations            *ct_item_ops;
    struct configfs_group_operations          *ct_group_ops;
    struct configfs_attribute                 **ct_attrs;
    struct configfs_bin_attribute              **ct_bin_attrs;
};
```

The most basic function of a `config_item_type` is to define what operations can be performed on a `config_item`. All items that have been allocated dynamically will need to provide the `ct_item_ops->release()` method. This method is called when the `config_item`'s reference count reaches zero.

3.13.7 struct configfs_attribute

```
struct configfs_attribute {
    char *ca_name;
    struct module *ca_owner;
    umode_t ca_mode;
    ssize_t (*show)(struct config_item *, char *);
    ssize_t (*store)(struct config_item *, const char *, size_t);
};
```

When a `config_item` wants an attribute to appear as a file in the item's `configfs` directory, it must define a `configfs_attribute` describing it. It then adds the attribute to the NULL-terminated array `config_item_type->ct_attrs`. When the item appears in configs, the attribute file will appear with the `configfs_attribute->ca_name` filename. `configfs_attribute->ca_mode` specifies the file permissions.

If an attribute is readable and provides a `->show` method, that method will be called whenever userspace asks for a `read(2)` on the attribute. If an attribute is writable and provides a `->store` method, that method will be called whenever userspace asks for a `write(2)` on the attribute.

3.13.8 struct configfs_bin_attribute

```
struct configfs_bin_attribute {
    struct configfs_attribute    cb_attr;
    void                        *cb_private;
    size_t                      cb_max_size;
};
```

The binary attribute is used when the one needs to use binary blob to appear as the contents of a file in the item's configfs directory. To do so add the binary attribute to the NULL-terminated array `config_item_type->ct_bin_attrs`, and the item appears in configfs, the attribute file will appear with the `configfs_bin_attribute->cb_attr.ca_name` filename. `configfs_bin_attribute->cb_attr.ca_mode` specifies the file permissions. The `cb_private` member is provided for use by the driver, while the `cb_max_size` member specifies the maximum amount of vmalloc buffer to be used.

If binary attribute is readable and the `config_item` provides a `ct_item_ops->read_bin_attribute()` method, that method will be called whenever userspace asks for a `read(2)` on the attribute. The converse will happen for `write(2)`. The reads/writes are buffered so only a single read/write will occur; the attributes' need not concern itself with it.

3.13.9 struct config_group

A `config_item` cannot live in a vacuum. The only way one can be created is via `mkdir(2)` on a `config_group`. This will trigger creation of a child item:

```
struct config_group {
    struct config_item    cg_item;
    struct list_head      cg_children;
    struct configfs_subsystem *cg_subsys;
    struct list_head      default_groups;
    struct list_head      group_entry;
};

void config_group_init(struct config_group *group);
void config_group_init_type_name(struct config_group *group,
                                const char *name,
                                struct config_item_type *type);
```

The `config_group` structure contains a `config_item`. Properly configuring that item means that a group can behave as an item in its own right. However, it can do more: it can create child items or groups. This is accomplished via the group operations specified on the group's `config_item_type`:

```
struct configfs_group_operations {
    struct config_item *(*make_item)(struct config_group *group,
                                     const char *name);
    struct config_group *(*make_group)(struct config_group *group,
                                       const char *name);
    int (*commit_item)(struct config_item *item);
    void (*disconnect_notify)(struct config_group *group,
```

(continues on next page)

(continued from previous page)

```

                                struct config_item *item);
void (*drop_item)(struct config_group *group,
                  struct config_item *item);
};

```

A group creates child items by providing the `ct_group_ops->make_item()` method. If provided, this method is called from `mkdir(2)` in the group's directory. The subsystem allocates a new `config_item` (or more likely, its container structure), initializes it, and returns it to configfs. Configfs will then populate the filesystem tree to reflect the new item.

If the subsystem wants the child to be a group itself, the subsystem provides `ct_group_ops->make_group()`. Everything else behaves the same, using the `group_init()` functions on the group.

Finally, when userspace calls `rmdir(2)` on the item or group, `ct_group_ops->drop_item()` is called. As a `config_group` is also a `config_item`, it is not necessary for a separate `drop_group()` method. The subsystem must `config_item_put()` the reference that was initialized upon item allocation. If a subsystem has no work to do, it may omit the `ct_group_ops->drop_item()` method, and configfs will call `config_item_put()` on the item on behalf of the subsystem.

Important: `drop_item()` is void, and as such cannot fail. When `rmdir(2)` is called, configfs WILL remove the item from the filesystem tree (assuming that it has no children to keep it busy). The subsystem is responsible for responding to this. If the subsystem has references to the item in other threads, the memory is safe. It may take some time for the item to actually disappear from the subsystem's usage. But it is gone from configfs.

When `drop_item()` is called, the item's linkage has already been torn down. It no longer has a reference on its parent and has no place in the item hierarchy. If a client needs to do some cleanup before this teardown happens, the subsystem can implement the `ct_group_ops->disconnect_notify()` method. The method is called after configfs has removed the item from the filesystem view but before the item is removed from its parent group. Like `drop_item()`, `disconnect_notify()` is void and cannot fail. Client subsystems should not drop any references here, as they still must do it in `drop_item()`.

A `config_group` cannot be removed while it still has child items. This is implemented in the configfs `rmdir(2)` code. `->drop_item()` will not be called, as the item has not been dropped. `rmdir(2)` will fail, as the directory is not empty.

3.13.10 struct configfs_subsystem

A subsystem must register itself, usually at `module_init` time. This tells configfs to make the subsystem appear in the file tree:

```

struct configfs_subsystem {
    struct config_group    su_group;
    struct mutex           su_mutex;
};

```

(continues on next page)

(continued from previous page)

```
int configfs_register_subsystem(struct configfs_subsystem *subsys);
void configfs_unregister_subsystem(struct configfs_subsystem *subsys);
```

A subsystem consists of a toplevel `config_group` and a mutex. The group is where child `config_items` are created. For a subsystem, this group is usually defined statically. Before calling `configfs_register_subsystem()`, the subsystem must have initialized the group via the usual `group_init()` functions, and it must also have initialized the mutex.

When the register call returns, the subsystem is live, and it will be visible via `configfs`. At that point, `mkdir(2)` can be called and the subsystem must be ready for it.

3.13.11 An Example

The best example of these basic concepts is the `simple_children` subsystem/group and the `simple_child` item in `samples/configfs/configfs_sample.c`. It shows a trivial object displaying and storing an attribute, and a simple group creating and destroying these children.

3.13.12 Hierarchy Navigation and the Subsystem Mutex

There is an extra bonus that `configfs` provides. The `config_groups` and `config_items` are arranged in a hierarchy due to the fact that they appear in a filesystem. A subsystem is NEVER to touch the filesystem parts, but the subsystem might be interested in this hierarchy. For this reason, the hierarchy is mirrored via the `config_group->cg_children` and `config_item->ci_parent` structure members.

A subsystem can navigate the `cg_children` list and the `ci_parent` pointer to see the tree created by the subsystem. This can race with `configfs`' management of the hierarchy, so `configfs` uses the subsystem mutex to protect modifications. Whenever a subsystem wants to navigate the hierarchy, it must do so under the protection of the subsystem mutex.

A subsystem will be prevented from acquiring the mutex while a newly allocated item has not been linked into this hierarchy. Similarly, it will not be able to acquire the mutex while a dropping item has not yet been unlinked. This means that an item's `ci_parent` pointer will never be NULL while the item is in `configfs`, and that an item will only be in its parent's `cg_children` list for the same duration. This allows a subsystem to trust `ci_parent` and `cg_children` while they hold the mutex.

3.13.13 Item Aggregation Via symlink(2)

configs provides a simple group via the group->item parent/child relationship. Often, however, a larger environment requires aggregation outside of the parent/child connection. This is implemented via symlink(2).

A config_item may provide the ct_item_ops->allow_link() and ct_item_ops->drop_link() methods. If the ->allow_link() method exists, symlink(2) may be called with the config_item as the source of the link. These links are only allowed between configs config_items. Any symlink(2) attempt outside the configs filesystem will be denied.

When symlink(2) is called, the source config_item's ->allow_link() method is called with itself and a target item. If the source item allows linking to target item, it returns 0. A source item may wish to reject a link if it only wants links to a certain type of object (say, in its own subsystem).

When unlink(2) is called on the symbolic link, the source item is notified via the ->drop_link() method. Like the ->drop_item() method, this is a void function and cannot return failure. The subsystem is responsible for responding to the change.

A config_item cannot be removed while it links to any other item, nor can it be removed while an item links to it. Dangling symlinks are not allowed in configs.

3.13.14 Automatically Created Subgroups

A new config_group may want to have two types of child config_items. While this could be codified by magic names in ->make_item(), it is much more explicit to have a method whereby userspace sees this divergence.

Rather than have a group where some items behave differently than others, configs provides a method whereby one or many subgroups are automatically created inside the parent at its creation. Thus, mkdir("parent") results in "parent" , "parent/subgroup1" , up through "parent/subgroupN" . Items of type 1 can now be created in "parent/subgroup1" , and items of type N can be created in "parent/subgroupN" .

These automatic subgroups, or default groups, do not preclude other children of the parent group. If ct_group_ops->make_group() exists, other child groups can be created on the parent group directly.

A configs subsystem specifies default groups by adding them using the configs_add_default_group() function to the parent config_group structure. Each added group is populated in the configs tree at the same time as the parent group. Similarly, they are removed at the same time as the parent. No extra notification is provided. When a ->drop_item() method call notifies the subsystem the parent group is going away, it also means every default group child associated with that parent group.

As a consequence of this, default groups cannot be removed directly via rmdir(2). They also are not considered when rmdir(2) on the parent group is checking for children.

3.13.15 Dependent Subsystems

Sometimes other drivers depend on particular configfs items. For example, ocfs2 mounts depend on a heartbeat region item. If that region item is removed with `rmdir(2)`, the ocfs2 mount must BUG or go readonly. Not happy.

configfs provides two additional API calls: `configfs_depend_item()` and `configfs_undepend_item()`. A client driver can call `configfs_depend_item()` on an existing item to tell configfs that it is depended on. configfs will then return `-EBUSY` from `rmdir(2)` for that item. When the item is no longer depended on, the client driver calls `configfs_undepend_item()` on it.

These API cannot be called underneath any configfs callbacks, as they will conflict. They can block and allocate. A client driver probably shouldn't call them of its own gumption. Rather it should be providing an API that external subsystems call.

How does this work? Imagine the ocfs2 mount process. When it mounts, it asks for a heartbeat region item. This is done via a call into the heartbeat code. Inside the heartbeat code, the region item is looked up. Here, the heartbeat code calls `configfs_depend_item()`. If it succeeds, then heartbeat knows the region is safe to give to ocfs2. If it fails, it was being torn down anyway, and heartbeat can gracefully pass up an error.

3.13.16 Committable Items

Note: Committable items are currently unimplemented.

Some `config_items` cannot have a valid initial state. That is, no default values can be specified for the item's attributes such that the item can do its work. Userspace must configure one or more attributes, after which the subsystem can start whatever entity this item represents.

Consider the FakeNBD device from above. Without a target address and a target device, the subsystem has no idea what block device to import. The simple example assumes that the subsystem merely waits until all the appropriate attributes are configured, and then connects. This will, indeed, work, but now every attribute store must check if the attributes are initialized. Every attribute store must fire off the connection if that condition is met.

Far better would be an explicit action notifying the subsystem that the `config_item` is ready to go. More importantly, an explicit action allows the subsystem to provide feedback as to whether the attributes are initialized in a way that makes sense. configfs provides this as committable items.

configfs still uses only normal filesystem operations. An item is committed via `rename(2)`. The item is moved from a directory where it can be modified to a directory where it cannot.

Any group that provides the `ct_group_ops->commit_item()` method has committable items. When this group appears in configfs, `mkdir(2)` will not work directly in the group. Instead, the group will have two subdirectories: "live" and "pending". The "live" directory does not support `mkdir(2)` or `rmdir(2)` either. It only allows `rename(2)`. The "pending" directory does allow `mkdir(2)` and `rmdir(2)`. An item is created in the "pending" directory. Its attributes can be modified at will.

Userspace commits the item by renaming it into the “live” directory. At this point, the subsystem receives the `->commit_item()` callback. If all required attributes are filled to satisfaction, the method returns zero and the item is moved to the “live” directory.

As `rmdir(2)` does not work in the “live” directory, an item must be shutdown, or “uncommitted”. Again, this is done via `rename(2)`, this time from the “live” directory back to the “pending” one. The subsystem is notified by the `ct_group_ops->uncommit_object()` method.

3.14 Cramfs - cram a filesystem onto a small ROM

cramfs is designed to be simple and small, and to compress things well.

It uses the zlib routines to compress a file one page at a time, and allows random page access. The meta-data is not compressed, but is expressed in a very terse representation to make it use much less disk space than traditional filesystems.

You can't write to a cramfs filesystem (making it compressible and compact also makes it `_very_ hard` to update on-the-fly), so you have to create the disk image with the “mkcramfs” utility.

3.14.1 Usage Notes

File sizes are limited to less than 16MB.

Maximum filesystem size is a little over 256MB. (The last file on the filesystem is allowed to extend past 256MB.)

Only the low 8 bits of gid are stored. The current version of mkcramfs simply truncates to 8 bits, which is a potential security issue.

Hard links are supported, but hard linked files will still have a link count of 1 in the cramfs image.

Cramfs directories have no `.` or `..` entries. Directories (like every other file on cramfs) always have a link count of 1. (There's no need to use `-noleaf` in `find`, btw.)

No timestamps are stored in a cramfs, so these default to the epoch (1970 GMT). Recently-accessed files may have updated timestamps, but the update lasts only as long as the inode is cached in memory, after which the timestamp reverts to 1970, i.e. moves backwards in time.

Currently, cramfs must be written and read with architectures of the same endianness, and can be read only by kernels with `PAGE_SIZE == 4096`. At least the latter of these is a bug, but it hasn't been decided what the best fix is. For the moment if you have larger pages you can just change the `#define` in `mkcramfs.c`, so long as you don't mind the filesystem becoming unreadable to future kernels.

3.14.2 Memory Mapped cramfs image

The `CRAMFS_MTD` Kconfig option adds support for loading data directly from a physical linear memory range (usually non volatile memory like Flash) instead of going through the block device layer. This saves some memory since no intermediate buffering is necessary to hold the data before decompressing.

And when data blocks are kept uncompressed and properly aligned, they will automatically be mapped directly into user space whenever possible providing eXecute-In-Place (XIP) from ROM of read-only segments. Data segments mapped read-write (hence they have to be copied to RAM) may still be compressed in the cramfs image in the same file along with non compressed read-only segments. Both MMU and no-MMU systems are supported. This is particularly handy for tiny embedded systems with very tight memory constraints.

The location of the cramfs image in memory is system dependent. You must know the proper physical address where the cramfs image is located and configure an MTD device for it. Also, that MTD device must be supported by a map driver that implements the “point” method. Examples of such MTD drivers are `cfi_cmdset_0001` (Intel/Sharp CFI flash) or `physmap` (Flash device in physical memory map). MTD partitions based on such devices are fine too. Then that device should be specified with the “mtd:” prefix as the mount device argument. For example, to mount the MTD device named “fs_partition” on the /mnt directory:

```
$ mount -t cramfs mtd:fs_partition /mnt
```

To boot a kernel with this as root filesystem, suffice to specify something like “root=mtd:fs_partition” on the kernel command line.

3.14.3 Tools

A version of `mkcramfs` that can take advantage of the latest capabilities described above can be found here:

<https://github.com/npitre/cramfs-tools>

3.14.4 For /usr/share/magic

0	ulelong 0x28cd3d45	Linux cramfs offset 0
>4	ulelong x	size %d
>8	ulelong x	flags 0x%x
>12	ulelong x	future 0x%x
>16	string >0	signature "%.16s"
>32	ulelong x	fsid.crc 0x%x
>36	ulelong x	fsid.edition %d
>40	ulelong x	fsid.blocks %d
>44	ulelong x	fsid.files %d
>48	string >0	name "%.16s"
512	ulelong 0x28cd3d45	Linux cramfs offset 512
>516	ulelong x	size %d
>520	ulelong x	flags 0x%x
>524	ulelong x	future 0x%x
>528	string >0	signature "%.16s"
>544	ulelong x	fsid.crc 0x%x
>548	ulelong x	fsid.edition %d
>552	ulelong x	fsid.blocks %d
>556	ulelong x	fsid.files %d
>560	string >0	name "%.16s"

3.14.5 Hacker Notes

See fs/cramfs/README for filesystem layout and implementation notes.

3.15 DebugFS

Copyright © 2009 Jonathan Corbet <corbet@lwn.net>

Debugfs exists as a simple way for kernel developers to make information available to user space. Unlike /proc, which is only meant for information about a process, or sysfs, which has strict one-value-per-file rules, debugfs has no rules at all. Developers can put any information they want there. The debugfs filesystem is also intended to not serve as a stable ABI to user space; in theory, there are no stability constraints placed on files exported there. The real world is not always so simple, though¹; even debugfs interfaces are best designed with the idea that they will need to be maintained forever.

Debugfs is typically mounted with a command like:

```
mount -t debugfs none /sys/kernel/debug
```

(Or an equivalent /etc/fstab line). The debugfs root directory is accessible only to the root user by default. To change access to the tree the “uid”, “gid” and “mode” mount options can be used.

¹ <http://lwn.net/Articles/309298/>

Note that the debugfs API is exported GPL-only to modules.

Code using debugfs should include `<linux/debugfs.h>`. Then, the first order of business will be to create at least one directory to hold a set of debugfs files:

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

This call, if successful, will make a directory called `name` underneath the indicated parent directory. If `parent` is `NULL`, the directory will be created in the debugfs root. On success, the return value is a struct dentry pointer which can be used to create files in the directory (and to clean it up at the end). An `ERR_PTR(-ERROR)` return value indicates that something went wrong. If `ERR_PTR(-ENODEV)` is returned, that is an indication that the kernel has been built without debugfs support and none of the functions described below will work.

The most general way to create a file within a debugfs directory is with:

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

Here, `name` is the name of the file to create, `mode` describes the access permissions the file should have, `parent` indicates the directory which should hold the file, `data` will be stored in the `i_private` field of the resulting inode structure, and `fops` is a set of file operations which implement the file's behavior. At a minimum, the `read()` and/or `write()` operations should be provided; others can be included as needed. Again, the return value will be a dentry pointer to the created file, `ERR_PTR(-ERROR)` on error, or `ERR_PTR(-ENODEV)` if debugfs support is missing.

Create a file with an initial size, the following function can be used instead:

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

`file_size` is the initial file size. The other parameters are the same as the function `debugfs_create_file`.

In a number of cases, the creation of a set of file operations is not actually necessary; the debugfs code provides a number of helper functions for simple situations. Files containing a single integer value can be created with any of:

```
void debugfs_create_u8(const char *name, umode_t mode,
                      struct dentry *parent, u8 *value);
void debugfs_create_u16(const char *name, umode_t mode,
                       struct dentry *parent, u16 *value);
void debugfs_create_u32(const char *name, umode_t mode,
                       struct dentry *parent, u32 *value);
void debugfs_create_u64(const char *name, umode_t mode,
                       struct dentry *parent, u64 *value);
```

These files support both reading and writing the given value; if a specific file should not be written to, simply set the mode bits accordingly. The values in these files are in decimal; if hexadecimal is more appropriate, the following functions can be used instead:

```
void debugfs_create_x8(const char *name, umode_t mode,
                      struct dentry *parent, u8 *value);
void debugfs_create_x16(const char *name, umode_t mode,
                        struct dentry *parent, u16 *value);
void debugfs_create_x32(const char *name, umode_t mode,
                        struct dentry *parent, u32 *value);
void debugfs_create_x64(const char *name, umode_t mode,
                        struct dentry *parent, u64 *value);
```

These functions are useful as long as the developer knows the size of the value to be exported. Some types can have different widths on different architectures, though, complicating the situation somewhat. There are functions meant to help out in such special cases:

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

As might be expected, this function will create a debugfs file to represent a variable of type `size_t`.

Similarly, there are helpers for variables of type unsigned long, in decimal and hexadecimal:

```
struct dentry *debugfs_create_ulong(const char *name, umode_t mode,
                                    struct dentry *parent,
                                    unsigned long *value);
void debugfs_create_xul(const char *name, umode_t mode,
                        struct dentry *parent, unsigned long *value);
```

Boolean values can be placed in debugfs with:

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

A read on the resulting file will yield either Y (for non-zero values) or N, followed by a newline. If written to, it will accept either upper- or lower-case values, or 1 or 0. Any other input will be silently ignored.

Also, `atomic_t` values can be placed in debugfs with:

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

A read of this file will get `atomic_t` values, and a write of this file will set `atomic_t` values.

Another option is exporting a block of arbitrary binary data, with this structure and function:

```
struct debugfs_blob_wrapper {
    void *data;
    unsigned long size;
};

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
```

(continues on next page)

(continued from previous page)

```
struct dentry *parent,  
struct debugfs_blob_wrapper *blob);
```

A read of this file will return the data pointed to by the `debugfs_blob_wrapper` structure. Some drivers use “blobs” as a simple way to return several lines of (static) formatted text output. This function can be used to export binary information, but there does not appear to be any code which does so in the mainline. Note that all files created with `debugfs_create_blob()` are read-only.

If you want to dump a block of registers (something that happens quite often during development, even if little such code reaches mainline. `Debugfs` offers two functions: one to make a registers-only file, and another to insert a register block in the middle of another sequential file:

```
struct debugfs_reg32 {  
    char *name;  
    unsigned long offset;  
};  
  
struct debugfs_regset32 {  
    const struct debugfs_reg32 *regs;  
    int nregs;  
    void __iomem *base;  
    struct device *dev;    /* Optional device for Runtime PM */  
};  
  
debugfs_create_regset32(const char *name, umode_t mode,  
                        struct dentry *parent,  
                        struct debugfs_regset32 *regset);  
  
void debugfs_print_regs32(struct seq_file *s, const struct debugfs_reg32_  
↪ *regs,  
                        int nregs, void __iomem *base, char *prefix);
```

The “base” argument may be 0, but you may want to build the `reg32` array using `__stringify`, and a number of register names (macros) are actually byte offsets over a base for the register block.

If you want to dump an `u32` array in `debugfs`, you can create file with:

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

The “array” argument provides data, and the “elements” argument is the number of elements in the array. Note: Once array is created its size can not be changed.

There is a helper function to create device related `seq_file`:

```
struct dentry *debugfs_create_devm_seqfile(struct device *dev,  
                                           const char *name,  
                                           struct dentry *parent,  
                                           int (*read_fn)(struct seq_file *s,  
                                                           void *data));
```

The “dev” argument is the device related to this debugfs file, and the “read_fn” is a function pointer which to be called to print the seq_file content.

There are a couple of other directory-oriented helper functions:

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

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

A call to `debugfs_rename()` will give a new name to an existing debugfs file, possibly in a different directory. The `new_name` must not exist prior to the call; the return value is `old_dentry` with updated information. Symbolic links can be created with `debugfs_create_symlink()`.

There is one important thing that all debugfs users must take into account: there is no automatic cleanup of any directories created in debugfs. If a module is unloaded without explicitly removing debugfs entries, the result will be a lot of stale pointers and no end of highly antisocial behavior. So all debugfs users - at least those which can be built as modules - must be prepared to remove all files and directories they create there. A file can be removed with:

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

The `dentry` value can be `NULL` or an error value, in which case nothing will be removed.

Once upon a time, debugfs users were required to remember the `dentry` pointer for every debugfs file they created so that all files could be cleaned up. We live in more civilized times now, though, and debugfs users can call:

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

If this function is passed a pointer for the `dentry` corresponding to the top-level directory, the entire hierarchy below that directory will be removed.

3.16 DLMFS

A minimal DLM userspace interface implemented via a virtual file system.

`dlmfs` is built with `OCFS2` as it requires most of its infrastructure.

Project web page <http://ocfs2.wiki.kernel.org>

Tools web page <https://github.com/markfasheh/ocfs2-tools>

OCFS2 mailing lists <http://oss.oracle.com/projects/ocfs2/mailman/>

All code copyright 2005 Oracle except when otherwise noted.

3.16.1 Credits

Some code taken from ramfs which is Copyright © 2000 Linus Torvalds and Transmeta Corp.

Mark Fasheh <mark.fasheh@oracle.com>

3.16.2 Caveats

- Right now it only works with the OCFS2 DLM, though support for other DLM implementations should not be a major issue.

3.16.3 Mount options

None

3.16.4 Usage

If you're just interested in OCFS2, then please see `ocfs2.txt`. The rest of this document will be geared towards those who want to use `dlmfs` for easy to setup and easy to use clustered locking in userspace.

3.16.5 Setup

`dlmfs` requires that the OCFS2 cluster infrastructure be in place. Please download `ocfs2-tools` from the above url and configure a cluster.

You'll want to start heartbeating on a volume which all the nodes in your lockspace can access. The easiest way to do this is via `ocfs2_hb_ctl` (distributed with `ocfs2-tools`). Right now it requires that an OCFS2 file system be in place so that it can automatically find its heartbeat area, though it will eventually support heartbeat against raw disks.

Please see the `ocfs2_hb_ctl` and `mkfs.ocfs2` manual pages distributed with `ocfs2-tools`.

Once you're heartbeating, DLM lock 'domains' can be easily created / destroyed and locks within them accessed.

3.16.6 Locking

Users may access `dlmfs` via standard file system calls, or they can use 'libo2dlm' (distributed with `ocfs2-tools`) which abstracts the file system calls and presents a more traditional locking api.

`dlmfs` handles lock caching automatically for the user, so a lock request for an already acquired lock will not generate another DLM call. Userspace programs are assumed to handle their own local locking.

Two levels of locks are supported - Shared Read, and Exclusive. Also supported is a Trylock operation.

For information on the libo2dlm interface, please see o2dlm.h, distributed with ocfs2-tools.

Lock value blocks can be read and written to a resource via read(2) and write(2) against the fd obtained via your open(2) call. The maximum currently supported LVB length is 64 bytes (though that is an OCFS2 DLM limitation). Through this mechanism, users of dlmfs can share small amounts of data amongst their nodes.

mkdir(2) signals dlmfs to join a domain (which will have the same name as the resulting directory)

rmdir(2) signals dlmfs to leave the domain

Locks for a given domain are represented by regular inodes inside the domain directory. Locking against them is done via the open(2) system call.

The open(2) call will not return until your lock has been granted or an error has occurred, unless it has been instructed to do a trylock operation. If the lock succeeds, you'll get an fd.

open(2) with O_CREAT to ensure the resource inode is created - dlmfs does not automatically create inodes for existing lock resources.

Open Flag	Lock Request Type
O_RDONLY	Shared Read
O_RDWR	Exclusive

Open Flag	Resulting Locking Behavior
O_NONBLOCK	Trylock operation

You must provide exactly one of O_RDONLY or O_RDWR.

If O_NONBLOCK is also provided and the trylock operation was valid but could not lock the resource then open(2) will return ETEXTBUSY.

close(2) drops the lock associated with your fd.

Modes passed to mkdir(2) or open(2) are adhered to locally. Chown is supported locally as well. This means you can use them to restrict access to the resources via dlmfs on your local node only.

The resource LVB may be read from the fd in either Shared Read or Exclusive modes via the read(2) system call. It can be written via write(2) only when open in Exclusive mode.

Once written, an LVB will be visible to other nodes who obtain Read Only or higher level locks on the resource.

3.16.7 See Also

http://opendlm.sourceforge.net/cvsmirror/opendlm/docs/dlmbook_final.pdf

For more information on the VMS distributed locking API.

3.17 eCryptfs: A stacked cryptographic filesystem for Linux

eCryptfs is free software. Please see the file COPYING for details. For documentation, please see the files in the doc/ subdirectory. For building and installation instructions please see the INSTALL file.

Maintainer Phillip Hellewell

Lead developer Michael A. Halcrow <mhalcrow@us.ibm.com>

Developers Michael C. Thompson Kent Yoder

Web Site <http://ecryptfs.sf.net>

This software is currently undergoing development. Make sure to maintain a backup copy of any data you write into eCryptfs.

eCryptfs requires the userspace tools downloadable from the SourceForge site:

<http://sourceforge.net/projects/ecryptfs/>

Userspace requirements include:

- David Howells' userspace keyring headers and libraries (version 1.0 or higher), obtainable from <http://people.redhat.com/~dhowells/keyutils/>
- Libgcrypt

Note: In the beta/experimental releases of eCryptfs, when you upgrade eCryptfs, you should copy the files to an unencrypted location and then copy the files back into the new eCryptfs mount to migrate the files.

3.17.1 Mount-wide Passphrase

Create a new directory into which eCryptfs will write its encrypted files (i.e., /root/crypt). Then, create the mount point directory (i.e., /mnt/crypt). Now it's time to mount eCryptfs:

```
mount -t ecryptfs /root/crypt /mnt/crypt
```

You should be prompted for a passphrase and a salt (the salt may be blank).

Try writing a new file:

```
echo "Hello, World" > /mnt/crypt/hello.txt
```

The operation will complete. Notice that there is a new file in `/root/crypt` that is at least 12288 bytes in size (depending on your host page size). This is the encrypted underlying file for what you just wrote. To test reading, from start to finish, you need to clear the user session keyring:

```
keyctl clear @u
```

Then unmount `/mnt/crypt` and mount again per the instructions given above.

```
cat /mnt/crypt/hello.txt
```

3.17.2 Notes

eCryptfs version 0.1 should only be mounted on (1) empty directories or (2) directories containing files only created by eCryptfs. If you mount a directory that has pre-existing files not created by eCryptfs, then behavior is undefined. Do not run eCryptfs in higher verbosity levels unless you are doing so for the sole purpose of debugging or development, since secret values will be written out to the system log in that case.

Mike Halcrow mhalcrow@us.ibm.com

3.18 efivarfs - a (U)EFI variable filesystem

The efivarfs filesystem was created to address the shortcomings of using entries in sysfs to maintain EFI variables. The old sysfs EFI variables code only supported variables of up to 1024 bytes. This limitation existed in version 0.99 of the EFI specification, but was removed before any full releases. Since variables can now be larger than a single page, sysfs isn't the best interface for this.

Variables can be created, deleted and modified with the efivarfs filesystem.

efivarfs is typically mounted like this:

```
mount -t efivarfs none /sys/firmware/efi/efivars
```

Due to the presence of numerous firmware bugs where removing non-standard UEFI variables causes the system firmware to fail to POST, efivarfs files that are not well-known standardized variables are created as immutable files. This doesn't prevent removal - "chattr -i" will work - but it does prevent this kind of failure from being accomplished accidentally.

Warning: When a content of an UEFI variable in `/sys/firmware/efi/efivars` is displayed, for example using "hexdump", pay attention that the first 4 bytes of the output represent the UEFI variable attributes, in little-endian format.

Practically the output of each efivar is composed of:

```
4_bytes_of_attributes + efivar_data
```

See also:

- Documentation/admin-guide/acpi/ssdt-overlays.rst
- Documentation/ABI/stable/sysfs-firmware-efi-vars

3.19 Enhanced Read-Only File System - EROFS

3.19.1 Overview

EROFS file-system stands for Enhanced Read-Only File System. Different from other read-only file systems, it aims to be designed for flexibility, scalability, but be kept simple and high performance.

It is designed as a better filesystem solution for the following scenarios:

- read-only storage media or
- part of a fully trusted read-only solution, which means it needs to be immutable and bit-for-bit identical to the official golden image for their releases due to security and other considerations and
- hope to save some extra storage space with guaranteed end-to-end performance by using reduced metadata and transparent file compression, especially for those embedded devices with limited memory (ex, smartphone);

Here is the main features of EROFS:

- Little endian on-disk design;
- Currently 4KB block size (nobh) and therefore maximum 16TB address space;
- Metadata & data could be mixed by design;
- 2 inode versions for different requirements:

Inode metadata size	32 bytes	64 bytes
Max file size	4 GB	16 EB (also limited by max. vol size)
Max uids/gids	65536	4294967296
File change time	no	yes (64 + 32-bit timestamp)
Max hardlinks	65536	4294967296
Metadata reserved	4 bytes	14 bytes

- Support extended attributes (xattrs) as an option;
- Support xattr inline and tail-end data inline for all files;
- Support POSIX.1e ACLs by using xattrs;
- Support transparent file compression as an option: LZ4 algorithm with 4 KB fixed-sized output compression for high performance.

The following git tree provides the file system user-space tools under development (ex, formatting tool mkfs.erofs):

- [git://git.kernel.org/pub/scm/linux/kernel/git/xiang/erofs-utils.git](https://git.kernel.org/pub/scm/linux/kernel/git/xiang/erofs-utils.git)

Bugs and patches are welcome, please kindly help us and send to the following linux-erofs mailing list:

- linux-erofs mailing list <linux-erofs@lists.ozlabs.org>

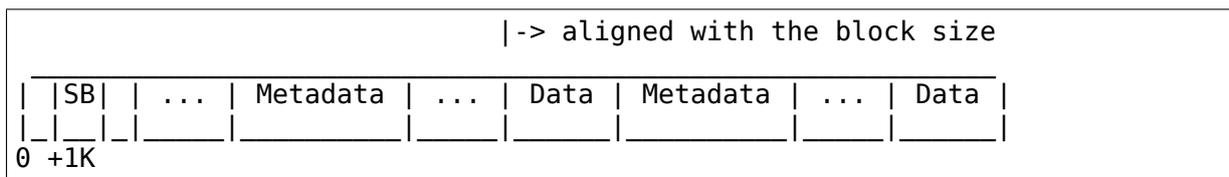
3.19.2 Mount options

(no)user_xattr	Setup Extended User Attributes. Note: xattr is enabled by default if CONFIG_EROFS_FS_XATTR is selected.						
(no)acl	Setup POSIX Access Control List. Note: acl is enabled by default if CONFIG_EROFS_FS_POSIX_ACL is selected.						
cache_strategy=%s	<p>Select a strategy for cached decompression from now on:</p> <table border="1"> <tr> <td>disabled</td> <td>In-place I/O decompression only;</td> </tr> <tr> <td>readahead</td> <td>Cache the last incomplete compressed physical cluster for further reading. It still does in-place I/O decompression for the rest compressed physical clusters;</td> </tr> <tr> <td>readahead</td> <td>Cache the both ends of incomplete compressed physical clusters for further reading. It still does in-place I/O decompression for the rest compressed physical clusters.</td> </tr> </table>	disabled	In-place I/O decompression only;	readahead	Cache the last incomplete compressed physical cluster for further reading. It still does in-place I/O decompression for the rest compressed physical clusters;	readahead	Cache the both ends of incomplete compressed physical clusters for further reading. It still does in-place I/O decompression for the rest compressed physical clusters.
disabled	In-place I/O decompression only;						
readahead	Cache the last incomplete compressed physical cluster for further reading. It still does in-place I/O decompression for the rest compressed physical clusters;						
readahead	Cache the both ends of incomplete compressed physical clusters for further reading. It still does in-place I/O decompression for the rest compressed physical clusters.						

3.19.3 On-disk details

Summary

Different from other read-only file systems, an EROFS volume is designed to be as simple as possible:

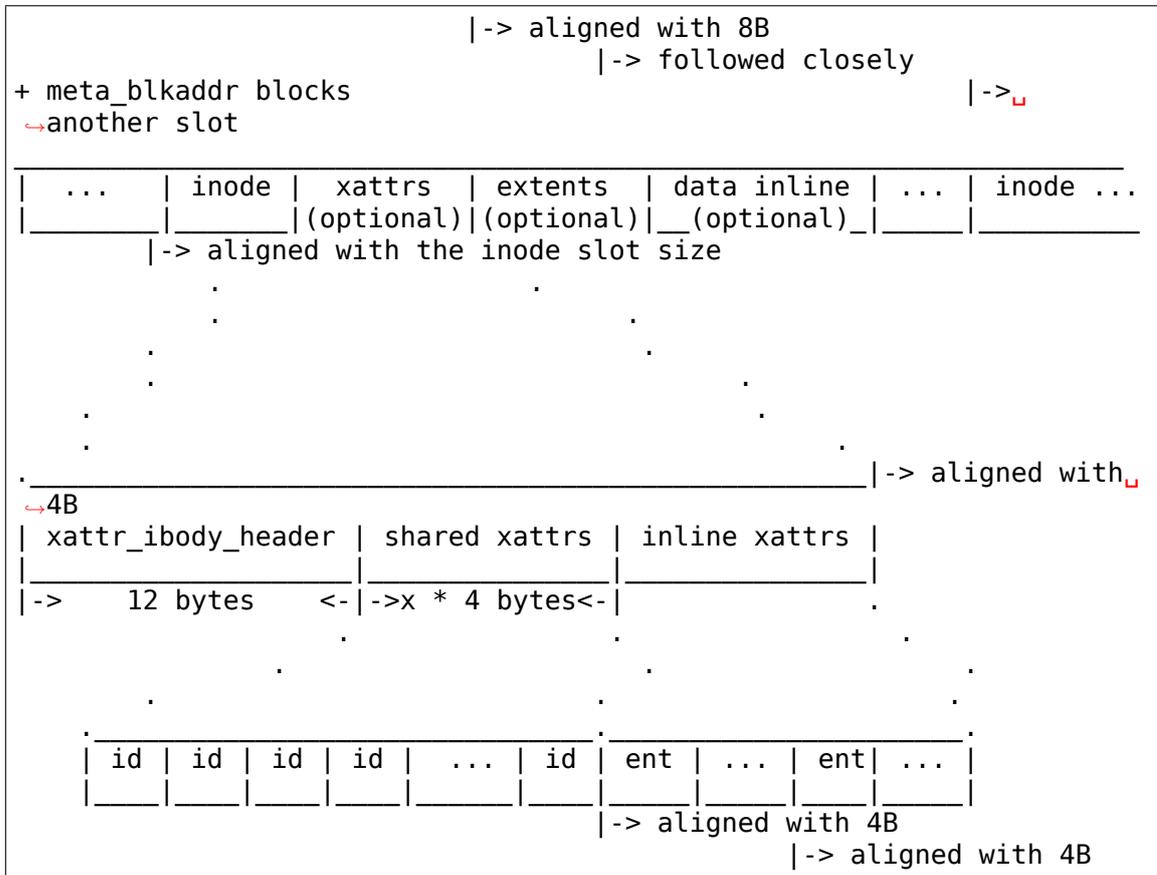


All data areas should be aligned with the block size, but metadata areas may not. All metadata can be now observed in two different spaces (views):

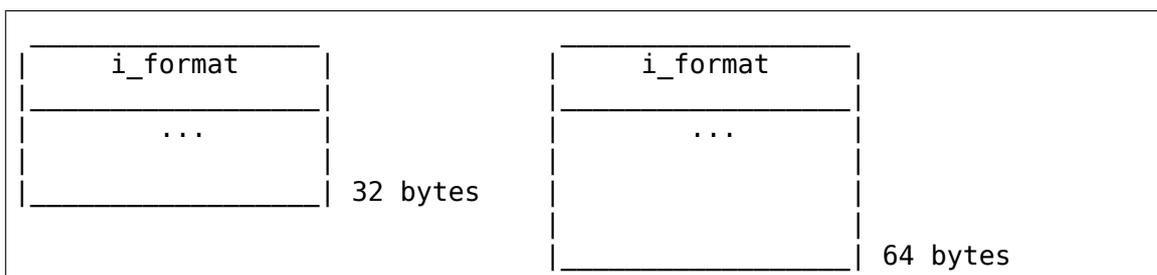
1. Inode metadata space

Each valid inode should be aligned with an inode slot, which is a fixed value (32 bytes) and designed to be kept in line with compact inode size.

Each inode can be directly found with the following formula: $inode\ offset = meta_blkaddr * block_size + 32 * nid$



Inode could be 32 or 64 bytes, which can be distinguished from a common field which all inode versions have - `i_format`:



Xattrs, extents, data inline are followed by the corresponding inode with proper alignment, and they could be optional for different data mappings. `_currently_ total 4 valid data mappings are supported:`

0	flat file data without data inline (no extent);
1	fixed-sized output data compression (with non-compacted indexes);
2	flat file data with tail packing data inline (no extent);
3	fixed-sized output data compression (with compacted indexes, v5.3+).

The size of the optional xattrs is indicated by `i_xattr_count` in inode header.

bsddf	(*)	Makes df act like BSD.
minixdf		Makes df act like Minix.
check=none nocheck	(*)	Don' t do extra checking of bitmaps on mount (check=normal and check=strict options removed)
dax		Use direct access (no page cache). See Documentation/filesystems/dax.txt.
debug		Extra debugging information is sent to the kernel syslog. Useful for developers.
er- rors=continue		Keep going on a filesystem error.
errors=remount- ro		Remount the filesystem read-only on an error.
er- rors=panic		Panic and halt the machine if an error occurs.
grpuid, bs- dgroups		Give objects the same group ID as their parent.
nogrpuid, sysv- groups		New objects have the group ID of their creator.
nouid32		Use 16-bit UIDs and GIDs.
oldalloc		Enable the old block allocator. Orlov should have better performance, we' d like to get some feedback if it' s the contrary for you.
orlov	(*)	Use the Orlov block allocator. (See http://lwn.net/Articles/14633/ and http://lwn.net/Articles/14446/ .)
resuid=n		The user ID which may use the reserved blocks.
resgid=n		The group ID which may use the reserved blocks.
sb=n		Use alternate superblock at this location.
user_xattr		Enable "user." POSIX Extended Attributes (requires CONFIG_EXT2_FS_XATTR).
nouser_xattr		Don' t support "user." extended attributes.
acl		Enable POSIX Access Control Lists support (requires CONFIG_EXT2_FS_POSIX_ACL).
noacl		Don' t support POSIX ACLs.
nobh		Do not attach buffer_heads to file pagecache.
quota, us- rquota		Enable user disk quota support (requires CONFIG_QUOTA).
grpquota		Enable group disk quota support (requires CONFIG_QUOTA).

noquota option is silently ignored by ext2.

3.22 Specification

ext2 shares many properties with traditional Unix filesystems. It has the concepts of blocks, inodes and directories. It has space in the specification for Access Control Lists (ACLs), fragments, undeletion and compression though these are not yet implemented (some are available as separate patches). There is also a versioning mechanism to allow new features (such as journalling) to be added in a maximally compatible manner.

3.22.1 Blocks

The space in the device or file is split up into blocks. These are a fixed size, of 1024, 2048 or 4096 bytes (8192 bytes on Alpha systems), which is decided when the filesystem is created. Smaller blocks mean less wasted space per file, but require slightly more accounting overhead, and also impose other limits on the size of files and the filesystem.

3.22.2 Block Groups

Blocks are clustered into block groups in order to reduce fragmentation and minimise the amount of head seeking when reading a large amount of consecutive data. Information about each block group is kept in a descriptor table stored in the block(s) immediately after the superblock. Two blocks near the start of each group are reserved for the block usage bitmap and the inode usage bitmap which show which blocks and inodes are in use. Since each bitmap is limited to a single block, this means that the maximum size of a block group is 8 times the size of a block.

The block(s) following the bitmaps in each block group are designated as the inode table for that block group and the remainder are the data blocks. The block allocation algorithm attempts to allocate data blocks in the same block group as the inode which contains them.

3.22.3 The Superblock

The superblock contains all the information about the configuration of the filing system. The primary copy of the superblock is stored at an offset of 1024 bytes from the start of the device, and it is essential to mounting the filesystem. Since it is so important, backup copies of the superblock are stored in block groups throughout the filesystem. The first version of ext2 (revision 0) stores a copy at the start of every block group, along with backups of the group descriptor block(s). Because this can consume a considerable amount of space for large filesystems, later revisions can optionally reduce the number of backup copies by only putting backups in specific groups (this is the sparse superblock feature). The groups chosen are 0, 1 and powers of 3, 5 and 7.

The information in the superblock contains fields such as the total number of inodes and blocks in the filesystem and how many are free, how many inodes and blocks are in each block group, when the filesystem was mounted (and if it was

cleanly unmounted), when it was modified, what version of the filesystem it is (see the Revisions section below) and which OS created it.

If the filesystem is revision 1 or higher, then there are extra fields, such as a volume name, a unique identification number, the inode size, and space for optional filesystem features to store configuration info.

All fields in the superblock (as in all other ext2 structures) are stored on the disc in little endian format, so a filesystem is portable between machines without having to know what machine it was created on.

3.22.4 Inodes

The inode (index node) is a fundamental concept in the ext2 filesystem. Each object in the filesystem is represented by an inode. The inode structure contains pointers to the filesystem blocks which contain the data held in the object and all of the metadata about an object except its name. The metadata about an object includes the permissions, owner, group, flags, size, number of blocks used, access time, change time, modification time, deletion time, number of links, fragments, version (for NFS) and extended attributes (EAs) and/or Access Control Lists (ACLs).

There are some reserved fields which are currently unused in the inode structure and several which are overloaded. One field is reserved for the directory ACL if the inode is a directory and alternately for the top 32 bits of the file size if the inode is a regular file (allowing file sizes larger than 2GB). The translator field is unused under Linux, but is used by the HURD to reference the inode of a program which will be used to interpret this object. Most of the remaining reserved fields have been used up for both Linux and the HURD for larger owner and group fields, The HURD also has a larger mode field so it uses another of the remaining fields to store the extra more bits.

There are pointers to the first 12 blocks which contain the file's data in the inode. There is a pointer to an indirect block (which contains pointers to the next set of blocks), a pointer to a doubly-indirect block (which contains pointers to indirect blocks) and a pointer to a trebly-indirect block (which contains pointers to doubly-indirect blocks).

The flags field contains some ext2-specific flags which aren't catered for by the standard chmod flags. These flags can be listed with lsattr and changed with the chattr command, and allow specific filesystem behaviour on a per-file basis. There are flags for secure deletion, undeletable, compression, synchronous updates, immutability, append-only, dumpable, no-atime, indexed directories, and data-journaling. Not all of these are supported yet.

3.22.5 Directories

A directory is a filesystem object and has an inode just like a file. It is a specially formatted file containing records which associate each name with an inode number. Later revisions of the filesystem also encode the type of the object (file, directory, symlink, device, fifo, socket) to avoid the need to check the inode itself for this information (support for taking advantage of this feature does not yet exist in Glibc 2.2).

The inode allocation code tries to assign inodes which are in the same block group as the directory in which they are first created.

The current implementation of ext2 uses a singly-linked list to store the filenames in the directory; a pending enhancement uses hashing of the filenames to allow lookup without the need to scan the entire directory.

The current implementation never removes empty directory blocks once they have been allocated to hold more files.

3.22.6 Special files

Symbolic links are also filesystem objects with inodes. They deserve special mention because the data for them is stored within the inode itself if the symlink is less than 60 bytes long. It uses the fields which would normally be used to store the pointers to data blocks. This is a worthwhile optimisation as it we avoid allocating a full block for the symlink, and most symlinks are less than 60 characters long.

Character and block special devices never have data blocks assigned to them. Instead, their device number is stored in the inode, again reusing the fields which would be used to point to the data blocks.

3.22.7 Reserved Space

In ext2, there is a mechanism for reserving a certain number of blocks for a particular user (normally the super-user). This is intended to allow for the system to continue functioning even if non-privileged users fill up all the space available to them (this is independent of filesystem quotas). It also keeps the filesystem from filling up entirely which helps combat fragmentation.

3.22.8 Filesystem check

At boot time, most systems run a consistency check (e2fsck) on their filesystems. The superblock of the ext2 filesystem contains several fields which indicate whether fsck should actually run (since checking the filesystem at boot can take a long time if it is large). fsck will run if the filesystem was not cleanly unmounted, if the maximum mount count has been exceeded or if the maximum time between checks has been exceeded.

3.22.9 Feature Compatibility

The compatibility feature mechanism used in ext2 is sophisticated. It safely allows features to be added to the filesystem, without unnecessarily sacrificing compatibility with older versions of the filesystem code. The feature compatibility mechanism is not supported by the original revision 0 (`EXT2_GOOD_OLD_REV`) of ext2, but was introduced in revision 1. There are three 32-bit fields, one for compatible features (`COMPAT`), one for read-only compatible (`RO_COMPAT`) features and one for incompatible (`INCOMPAT`) features.

These feature flags have specific meanings for the kernel as follows:

A `COMPAT` flag indicates that a feature is present in the filesystem, but the on-disk format is 100% compatible with older on-disk formats, so a kernel which didn't know anything about this feature could read/write the filesystem without any chance of corrupting the filesystem (or even making it inconsistent). This is essentially just a flag which says "this filesystem has a (hidden) feature" that the kernel or `e2fsck` may want to be aware of (more on `e2fsck` and feature flags later). The ext3 `HAS_JOURNAL` feature is a `COMPAT` flag because the ext3 journal is simply a regular file with data blocks in it so the kernel does not need to take any special notice of it if it doesn't understand ext3 journaling.

An `RO_COMPAT` flag indicates that the on-disk format is 100% compatible with older on-disk formats for reading (i.e. the feature does not change the visible on-disk format). However, an old kernel writing to such a filesystem would/could corrupt the filesystem, so this is prevented. The most common such feature, `SPARSE_SUPER`, is an `RO_COMPAT` feature because sparse groups allow file data blocks where superblock/group descriptor backups used to live, and `ext2_free_blocks()` refuses to free these blocks, which would lead to inconsistent bitmaps. An old kernel would also get an error if it tried to free a series of blocks which crossed a group boundary, but this is a legitimate layout in a `SPARSE_SUPER` filesystem.

An `INCOMPAT` flag indicates the on-disk format has changed in some way that makes it unreadable by older kernels, or would otherwise cause a problem if an old kernel tried to mount it. `FILETYPE` is an `INCOMPAT` flag because older kernels would think a filename was longer than 256 characters, which would lead to corrupt directory listings. The `COMPRESSION` flag is an obvious `INCOMPAT` flag - if the kernel doesn't understand compression, you would just get garbage back from `read()` instead of it automatically decompressing your data. The ext3 `RECOVER` flag is needed to prevent a kernel which does not understand the ext3 journal from mounting the filesystem without replaying the journal.

For `e2fsck`, it needs to be more strict with the handling of these flags than the kernel. If it doesn't understand ANY of the `COMPAT`, `RO_COMPAT`, or `INCOMPAT` flags it will refuse to check the filesystem, because it has no way of verifying whether a given feature is valid or not. Allowing `e2fsck` to succeed on a filesystem with an unknown feature is a false sense of security for the user. Refusing to check a filesystem with unknown features is a good incentive for the user to update to the latest `e2fsck`. This also means that anyone adding feature flags to ext2 also needs to update `e2fsck` to verify these features.

3.22.10 Metadata

It is frequently claimed that the ext2 implementation of writing asynchronous metadata is faster than the ffs synchronous metadata scheme but less reliable. Both methods are equally resolvable by their respective fsck programs.

If you're exceptionally paranoid, there are 3 ways of making metadata writes synchronous on ext2:

- per-file if you have the program source: use the `O_SYNC` flag to `open()`
- per-file if you don't have the source: use “`chattr +S`” on the file
- per-filesystem: add the “`sync`” option to `mount` (or in `/etc/fstab`)

the first and last are not ext2 specific but do force the metadata to be written synchronously. See also Journaling below.

3.22.11 Limitations

There are various limits imposed by the on-disk layout of ext2. Other limits are imposed by the current implementation of the kernel code. Many of the limits are determined at the time the filesystem is first created, and depend upon the block size chosen. The ratio of inodes to data blocks is fixed at filesystem creation time, so the only way to increase the number of inodes is to increase the size of the filesystem. No tools currently exist which can change the ratio of inodes to blocks.

Most of these limits could be overcome with slight changes in the on-disk format and using a compatibility flag to signal the format change (at the expense of some compatibility).

Filesystem block size	1kB	2kB	4kB	8kB
File size limit	16GB	256GB	2048GB	2048GB
Filesystem size limit	2047GB	8192GB	16384GB	32768GB

There is a 2.4 kernel limit of 2048GB for a single block device, so no filesystem larger than that can be created at this time. There is also an upper limit on the block size imposed by the page size of the kernel, so 8kB blocks are only allowed on Alpha systems (and other architectures which support larger pages).

There is an upper limit of 32000 subdirectories in a single directory.

There is a “soft” upper limit of about 10-15k files in a single directory with the current linear linked-list directory implementation. This limit stems from performance problems when creating and deleting (and also finding) files in such large directories. Using a hashed directory index (under development) allows 100k-1M+ files in a single directory without performance problems (although RAM size becomes an issue at this point).

The (meaningless) absolute upper limit of files in a single directory (imposed by the file size, the realistic limit is obviously much less) is over 130 trillion files. It would be higher except there are not enough 4-character names to make up unique directory entries, so they have to be 8 character filenames, even then we are fairly close to running out of unique filenames.

3.22.12 Journaling

A journaling extension to the ext2 code has been developed by Stephen Tweedie. It avoids the risks of metadata corruption and the need to wait for e2fsck to complete after a crash, without requiring a change to the on-disk ext2 layout. In a nutshell, the journal is a regular file which stores whole metadata (and optionally data) blocks that have been modified, prior to writing them into the filesystem. This means it is possible to add a journal to an existing ext2 filesystem without the need for data conversion.

When changes to the filesystem (e.g. a file is renamed) they are stored in a transaction in the journal and can either be complete or incomplete at the time of a crash. If a transaction is complete at the time of a crash (or in the normal case where the system does not crash), then any blocks in that transaction are guaranteed to represent a valid filesystem state, and are copied into the filesystem. If a transaction is incomplete at the time of the crash, then there is no guarantee of consistency for the blocks in that transaction so they are discarded (which means any filesystem changes they represent are also lost). Check Documentation/filesystems/ext4/ if you want to read more about ext4 and journaling.

3.23 References

The kernel source	file:/usr/src/linux/fs/ext2/
e2fsprogs (e2fsck)	http://e2fsprogs.sourceforge.net/
Design & Implementation	http://e2fsprogs.sourceforge.net/ext2intro.html
Journaling (ext3)	ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/
Filesystem Resizing	http://ext2resize.sourceforge.net/
Compression ¹	http://e2compr.sourceforge.net/

Implementations for:

Windows 95/98/NT/2000	http://www.chrysocome.net/explore2fs
Windows 95 ¹	http://www.yipton.net/content.html#FSDEXT2
DOS client ¹	ftp://metalab.unc.edu/pub/Linux/system/filesystems/ext2/
OS/2 ²	ftp://metalab.unc.edu/pub/Linux/system/filesystems/ext2/
RISC OS client	http://www.esw-heim.tu-clausthal.de/~marco/smorbrod/IscaFS/

¹ no longer actively developed/supported (as of Apr 2001)

² no longer actively developed/supported (as of Mar 2009)

3.24 Ext3 Filesystem

Ext3 was originally released in September 1999. Written by Stephen Tweedie for the 2.2 branch, and ported to 2.4 kernels by Peter Braam, Andreas Dilger, Andrew Morton, Alexander Viro, Ted Ts'o and Stephen Tweedie.

Ext3 is the ext2 filesystem enhanced with journalling capabilities. The filesystem is a subset of ext4 filesystem so use ext4 driver for accessing ext3 filesystems.

3.25 WHAT IS Flash-Friendly File System (F2FS)?

NAND flash memory-based storage devices, such as SSD, eMMC, and SD cards, have been equipped on a variety systems ranging from mobile to server systems. Since they are known to have different characteristics from the conventional rotating disks, a file system, an upper layer to the storage device, should adapt to the changes from the sketch in the design level.

F2FS is a file system exploiting NAND flash memory-based storage devices, which is based on Log-structured File System (LFS). The design has been focused on addressing the fundamental issues in LFS, which are snowball effect of wandering tree and high cleaning overhead.

Since a NAND flash memory-based storage device shows different characteristic according to its internal geometry or flash memory management scheme, namely FTL, F2FS and its tools support various parameters not only for configuring on-disk layout, but also for selecting allocation and cleaning algorithms.

The following git tree provides the file system formatting tool (mkfs.f2fs), a consistency checking tool (fsck.f2fs), and a debugging tool (dump.f2fs).

- [git://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs-tools.git](https://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs-tools.git)

For reporting bugs and sending patches, please use the following mailing list:

- linux-f2fs-devel@lists.sourceforge.net

3.25.1 Background and Design issues

Log-structured File System (LFS)

“A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into segments and use a segment cleaner to compress the live information from heavily fragmented segments.” from Rosenblum, M. and Ousterhout, J. K., 1992, “The design and implementation of a log-structured file system” , ACM Trans. Computer Systems 10, 1, 26-52.

Wandering Tree Problem

In LFS, when a file data is updated and written to the end of log, its direct pointer block is updated due to the changed location. Then the indirect pointer block is also updated due to the direct pointer block update. In this manner, the upper index structures such as inode, inode map, and checkpoint block are also updated recursively. This problem is called as wandering tree problem [1], and in order to enhance the performance, it should eliminate or relax the update propagation as much as possible.

[1] Bityutskiy, A. 2005. JFFS3 design issues. <http://www.linux-mtd.infradead.org/>

Cleaning Overhead

Since LFS is based on out-of-place writes, it produces so many obsolete blocks scattered across the whole storage. In order to serve new empty log space, it needs to reclaim these obsolete blocks seamlessly to users. This job is called as a cleaning process.

The process consists of three operations as follows.

1. A victim segment is selected through referencing segment usage table.
2. It loads parent index structures of all the data in the victim identified by segment summary blocks.
3. It checks the cross-reference between the data and its parent index structure.
4. It moves valid data selectively.

This cleaning job may cause unexpected long delays, so the most important goal is to hide the latencies to users. And also definitely, it should reduce the amount of valid data to be moved, and move them quickly as well.

3.25.2 Key Features

Flash Awareness

- Enlarge the random write area for better performance, but provide the high spatial locality
- Align FS data structures to the operational units in FTL as best efforts

Wandering Tree Problem

- Use a term, “node” , that represents inodes as well as various pointer blocks
- Introduce Node Address Table (NAT) containing the locations of all the “node” blocks; this will cut off the update propagation.

Cleaning Overhead

- Support a background cleaning process
- Support greedy and cost-benefit algorithms for victim selection policies
- Support multi-head logs for static/dynamic hot and cold data separation
- Introduce adaptive logging for efficient block allocation

3.25.3 Mount Options

3.25.4 Debugfs Entries

`/sys/kernel/debug/f2fs/` contains information about all the partitions mounted as f2fs. Each file shows the whole f2fs information.

`/sys/kernel/debug/f2fs/status` includes:

- major file system information managed by f2fs currently
- average SIT information about whole segments
- current memory footprint consumed by f2fs.

3.25.5 Sysfs Entries

Information about mounted f2fs file systems can be found in `/sys/fs/f2fs`. Each mounted filesystem will have a directory in `/sys/fs/f2fs` based on its device name (i.e., `/sys/fs/f2fs/sda`). The files in each per-device directory are shown in table below.

Files in `/sys/fs/f2fs/<devname>` (see also [Documentation/ABI/testing/sysfs-fs-f2fs](#))

3.25.6 Usage

1. Download userland tools and compile them.
2. Skip, if f2fs was compiled statically inside kernel. Otherwise, insert the f2fs.ko module:

```
# insmod f2fs.ko
```

3. Create a directory trying to mount:

```
# mkdir /mnt/f2fs
```

4. Format the block device, and then mount as f2fs:

```
# mkfs.f2fs -l label /dev/block_device  
# mount -t f2fs /dev/block_device /mnt/f2fs
```

mkfs.f2fs

The `mkfs.f2fs` is for the use of formatting a partition as the `f2fs` filesystem, which builds a basic on-disk layout.

The options consist of:

<code>-l [label]</code>	Give a volume label, up to 512 unicode name.
<code>-a [0 or 1]</code>	Split start location of each area for heap-based allocation. 1 is set by default, which performs this.
<code>-o [int]</code>	Set overprovision ratio in percent over volume size. 5 is set by default.
<code>-s [int]</code>	Set the number of segments per section. 1 is set by default.
<code>-z [int]</code>	Set the number of sections per zone. 1 is set by default.
<code>-e [str]</code>	Set basic extension list. e.g. "mp3,gif,mov"
<code>-t [0 or 1]</code>	Disable discard command or not. 1 is set by default, which conducts discard.

fsck.f2fs

The `fsck.f2fs` is a tool to check the consistency of an `f2fs`-formatted partition, which examines whether the filesystem metadata and user-made data are cross-referenced correctly or not. Note that, initial version of the tool does not fix any inconsistency.

The options consist of:

<code>-d debug level [default:0]</code>

dump.f2fs

The `dump.f2fs` shows the information of specific inode and dumps SSA and SIT to file. Each file is `dump_ssa` and `dump_sit`.

The `dump.f2fs` is used to debug on-disk data structures of the `f2fs` filesystem. It shows on-disk inode information recognized by a given inode number, and is able to dump all the SSA and SIT entries into predefined files, `./dump_ssa` and `./dump_sit` respectively.

The options consist of:

<code>-d debug level [default:0]</code>
<code>-i inode no (hex)</code>
<code>-s [SIT dump segno from #1~#2 (decimal), for all 0~-1]</code>
<code>-a [SSA dump segno from #1~#2 (decimal), for all 0~-1]</code>

Examples:

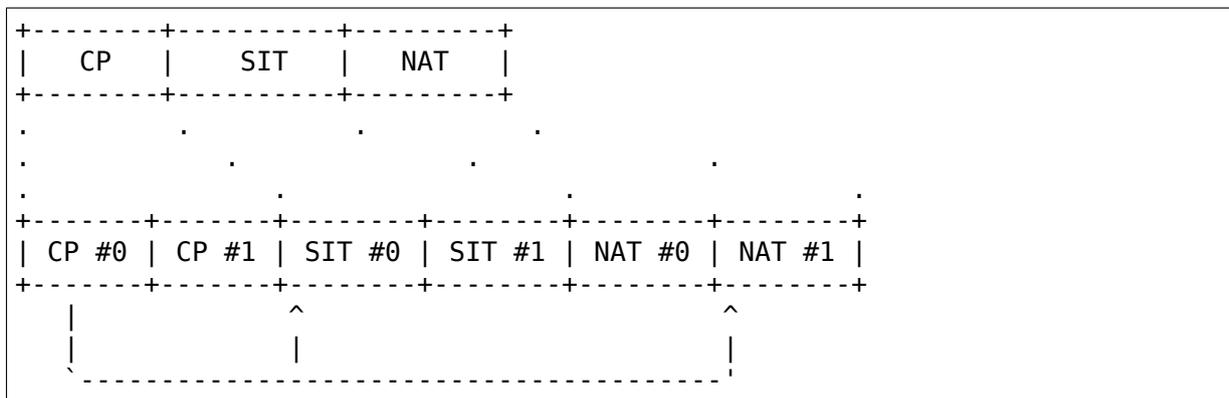
In order to avoid misalignment between file system and flash-based storage, F2FS aligns the start block address of CP with the segment size. Also, it aligns the start block address of Main area with the zone size by reserving some segments in SSA area.

Reference the following survey for additional technical details. <https://wiki.linaro.org/WorkingGroups/Kernel/Projects/FlashCardSurvey>

File System Metadata Structure

F2FS adopts the checkpointing scheme to maintain file system consistency. At mount time, F2FS first tries to find the last valid checkpoint data by scanning CP area. In order to reduce the scanning time, F2FS uses only two copies of CP. One of them always indicates the last valid data, which is called as shadow copy mechanism. In addition to CP, NAT and SIT also adopt the shadow copy mechanism.

For file system consistency, each CP points to which NAT and SIT copies are valid, as shown as below:



Index Structure

The key data structure to manage the data locations is a “node” . Similar to traditional file structures, F2FS has three types of node: inode, direct node, indirect node. F2FS assigns 4KB to an inode block which contains 923 data block indices, two direct node pointers, two indirect node pointers, and one double indirect node pointer as described below. One direct node block contains 1018 data blocks, and one indirect node block contains also 1018 node blocks. Thus, one inode block (i.e., a file) covers:

$$4\text{KB} * (923 + 2 * 1018 + 2 * 1018 * 1018 + 1018 * 1018 * 1018) := 3.94\text{TB}.$$

```

Inode block (4KB)
|- data (923)
|- direct node (2)
   `-- data (1018)
|- indirect node (2)
   `-- direct node (1018)
       `-- data (1018)
`-- double indirect node (1)

```

(continues on next page)

(continued from previous page)

```

    ` - indirect node (1018)
        ` - direct node (1018)
            ` - data (1018)
    
```

Note that, all the node blocks are mapped by NAT which means the location of each node is translated by the NAT table. In the consideration of the wandering tree problem, F2FS is able to cut off the propagation of node updates caused by leaf data writes.

Directory Structure

A directory entry occupies 11 bytes, which consists of the following attributes.

- hash hash value of the file name
- ino inode number
- len the length of file name
- type file type such as directory, symlink, etc

A dentry block consists of 214 dentry slots and file names. Therein a bitmap is used to represent whether each dentry is valid or not. A dentry block occupies 4KB with the following composition.

```

Dentry Block(4 K) = bitmap (27 bytes) + reserved (3 bytes) +
                   dentries(11 * 214 bytes) + file name (8 * 214 bytes)

    [Bucket]
    +-----+
    |dentry block 1 | dentry block 2 |
    +-----+
    .
    .
    .
    [Dentry Block Structure: 4KB]
    +-----+-----+-----+-----+
    | bitmap | reserved | dentries | file names |
    +-----+-----+-----+-----+
    [Dentry Block: 4KB] . . .
    .
    .
    +-----+-----+-----+
    | hash | ino | len | type |
    +-----+-----+-----+
    [Dentry Structure: 11 bytes]
    
```

F2FS implements multi-level hash tables for directory structure. Each level has a hash table with dedicated number of hash buckets as shown below. Note that “A(2B)” means a bucket includes 2 data blocks.

```

-----
A : bucket
B : block
N : MAX_DIR_HASH_DEPTH
-----
    
```

(continues on next page)

(continued from previous page)

```

level #0 | A(2B)
level #1 | A(2B) - A(2B)
level #2 | A(2B) - A(2B) - A(2B) - A(2B)
level #N/2 | A(2B) - A(2B) - A(2B) - A(2B) - A(2B) - ... - A(2B)
level #N | A(4B) - A(4B) - A(4B) - A(4B) - A(4B) - ... - A(4B)

```

The number of blocks and buckets are determined by:

```

# of blocks in level #n = { - 2, if n < MAX_DIR_HASH_DEPTH / 2,
                          - 4, otherwise

# of buckets in level #n = { - 2^(n + dir_level),
                             if n + dir_level < MAX_DIR_HASH_DEPTH /
→ 2,
                             - 2^((MAX_DIR_HASH_DEPTH / 2) - 1),
                             otherwise

```

When F2FS finds a file name in a directory, at first a hash value of the file name is calculated. Then, F2FS scans the hash table in level #0 to find the dentry consisting of the file name and its inode number. If not found, F2FS scans the next hash table in level #1. In this way, F2FS scans hash tables in each levels incrementally from 1 to N. In each levels F2FS needs to scan only one bucket determined by the following equation, which shows $O(\log(\# \text{ of files}))$ complexity:

```

bucket number to scan in level #n = (hash value) % (# of buckets in level
→ #n)

```

In the case of file creation, F2FS finds empty consecutive slots that cover the file name. F2FS searches the empty slots in the hash tables of whole levels from 1 to N in the same way as the lookup operation.

The following figure shows an example of two cases holding children:

```

-----> Dir <-----
|
child                    child
child - child            [hole] - child
child - child - child    [hole] - [hole] - child

Case 1:                    Case 2:
Number of children = 6,    Number of children = 3,
File size = 7              File size = 7

```

Default Block Allocation

At runtime, F2FS manages six active logs inside “Main” area: Hot/Warm/Cold node and Hot/Warm/Cold data.

- Hot node contains direct node blocks of directories.
- Warm node contains direct node blocks except hot node blocks.
- Cold node contains indirect node blocks
- Hot data contains dentry blocks
- Warm data contains data blocks except hot and cold data blocks
- Cold data contains multimedia data or migrated data blocks

LFS has two schemes for free space management: threaded log and copy-and-compaction. The copy-and-compaction scheme which is known as cleaning, is well-suited for devices showing very good sequential write performance, since free segments are served all the time for writing new data. However, it suffers from cleaning overhead under high utilization. Contrarily, the threaded log scheme suffers from random writes, but no cleaning process is needed. F2FS adopts a hybrid scheme where the copy-and-compaction scheme is adopted by default, but the policy is dynamically changed to the threaded log scheme according to the file system status.

In order to align F2FS with underlying flash-based storage, F2FS allocates a segment in a unit of section. F2FS expects that the section size would be the same as the unit size of garbage collection in FTL. Furthermore, with respect to the mapping granularity in FTL, F2FS allocates each section of the active logs from different zones as much as possible, since FTL can write the data in the active logs into one allocation unit according to its mapping granularity.

Cleaning process

F2FS does cleaning both on demand and in the background. On-demand cleaning is triggered when there are not enough free segments to serve VFS calls. Background cleaner is operated by a kernel thread, and triggers the cleaning job when the system is idle.

F2FS supports two victim selection policies: greedy and cost-benefit algorithms. In the greedy algorithm, F2FS selects a victim segment having the smallest number of valid blocks. In the cost-benefit algorithm, F2FS selects a victim segment according to the segment age and the number of valid blocks in order to address log block thrashing problem in the greedy algorithm. F2FS adopts the greedy algorithm for on-demand cleaner, while background cleaner adopts cost-benefit algorithm.

In order to identify whether the data in the victim segment are valid or not, F2FS manages a bitmap. Each bit represents the validity of a block, and the bitmap is composed of a bit stream covering whole blocks in main area.

Write-hint Policy

- 1) `whint_mode=off`. F2FS only passes down `WRITE_LIFE_NOT_SET`.
- 2) `whint_mode=user-based`. F2FS tries to pass down hints given by users.

User	F2FS	Block
<code>ioctl(COLD)</code>	<code>COLD_DATA</code>	<code>WRITE_LIFE_EXTREME</code>
extension list	“	“
- buffered io		
<code>WRITE_LIFE_EXTREME</code>	<code>COLD_DATA</code>	<code>WRITE_LIFE_EXTREME</code>
<code>WRITE_LIFE_SHORT</code>	<code>HOT_DATA</code>	<code>WRITE_LIFE_SHORT</code>
<code>WRITE_LIFE_NOT_SET</code>	<code>WARM_DATA</code>	<code>WRITE_LIFE_NOT_SET</code>
<code>WRITE_LIFE_NONE</code>	“	“
<code>WRITE_LIFE_MEDIUM</code>	“	“
<code>WRITE_LIFE_LONG</code>	“	“
- direct io		
<code>WRITE_LIFE_EXTREME</code>	<code>COLD_DATA</code>	<code>WRITE_LIFE_EXTREME</code>
<code>WRITE_LIFE_SHORT</code>	<code>HOT_DATA</code>	<code>WRITE_LIFE_SHORT</code>
<code>WRITE_LIFE_NOT_SET</code>	<code>WARM_DATA</code>	<code>WRITE_LIFE_NOT_SET</code>
<code>WRITE_LIFE_NONE</code>	“	<code>WRITE_LIFE_NONE</code>
<code>WRITE_LIFE_MEDIUM</code>	“	<code>WRITE_LIFE_MEDIUM</code>
<code>WRITE_LIFE_LONG</code>	“	<code>WRITE_LIFE_LONG</code>

- 3) `whint_mode=fs-based`. F2FS passes down hints with its policy.

User	F2FS	Block
<code>ioctl(COLD)</code>	<code>COLD_DATA</code>	<code>WRITE_LIFE_EXTREME</code>
extension list	“	“
- buffered io		
<code>WRITE_LIFE_EXTREME</code>	<code>COLD_DATA</code>	<code>WRITE_LIFE_EXTREME</code>
<code>WRITE_LIFE_SHORT</code>	<code>HOT_DATA</code>	<code>WRITE_LIFE_SHORT</code>
<code>WRITE_LIFE_NOT_SET</code>	<code>WARM_DATA</code>	<code>WRITE_LIFE_LONG</code>
<code>WRITE_LIFE_NONE</code>	“	“
<code>WRITE_LIFE_MEDIUM</code>	“	“
<code>WRITE_LIFE_LONG</code>	“	“
- direct io		
<code>WRITE_LIFE_EXTREME</code>	<code>COLD_DATA</code>	<code>WRITE_LIFE_EXTREME</code>
<code>WRITE_LIFE_SHORT</code>	<code>HOT_DATA</code>	<code>WRITE_LIFE_SHORT</code>
<code>WRITE_LIFE_NOT_SET</code>	<code>WARM_DATA</code>	<code>WRITE_LIFE_NOT_SET</code>
<code>WRITE_LIFE_NONE</code>	“	<code>WRITE_LIFE_NONE</code>
<code>WRITE_LIFE_MEDIUM</code>	“	<code>WRITE_LIFE_MEDIUM</code>
<code>WRITE_LIFE_LONG</code>	“	<code>WRITE_LIFE_LONG</code>

Fallocate(2) Policy

The default policy follows the below posix rule.

Allocating disk space The default operation (i.e., mode is zero) of `fallocate()` allocates the disk space within the range specified by `offset` and `len`. The file size (as reported by `stat(2)`) will be changed if `offset+len` is greater than the file size. Any subregion within the range specified by `offset` and `len` that did not contain data before the call will be initialized to zero. This default behavior closely resembles the behavior of the `posix_fallocate(3)` library function, and is intended as a method of optimally implementing that function.

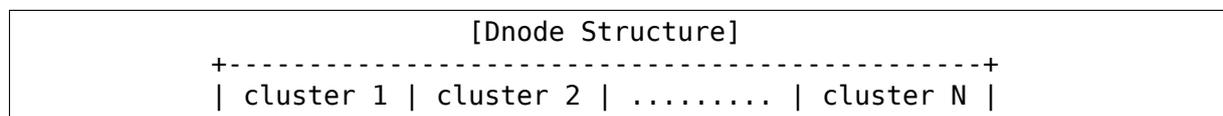
However, once F2FS receives `ioctl(fd, F2FS_IOC_SET_PIN_FILE)` in prior to `fallocate(fd, DEFAULT_MODE)`, it allocates on-disk blocks addresses having zero or random data, which is useful to the below scenario where:

1. `create(fd)`
2. `ioctl(fd, F2FS_IOC_SET_PIN_FILE)`
3. `fallocate(fd, 0, 0, size)`
4. `address = fibmap(fd, offset)`
5. `open(blkdev)`
6. `write(blkdev, address)`

Compression implementation

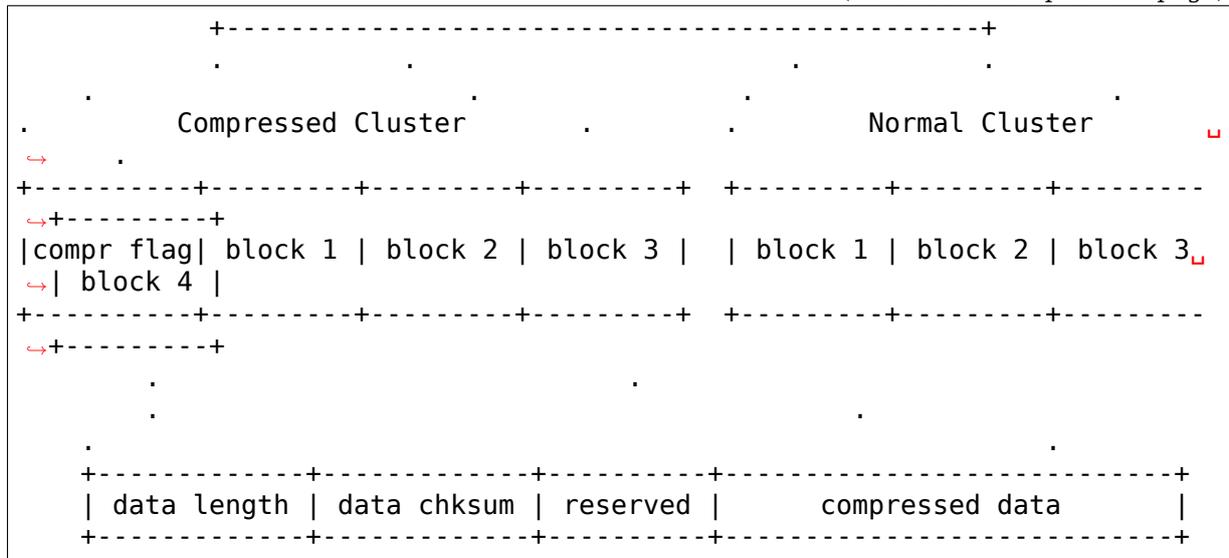
- New term named cluster is defined as basic unit of compression, file can be divided into multiple clusters logically. One cluster includes $4 \ll n$ ($n \geq 0$) logical pages, compression size is also cluster size, each of cluster can be compressed or not.
- In cluster metadata layout, one special block address is used to indicate cluster is compressed one or normal one, for compressed cluster, following metadata maps cluster to $[1, 4 \ll n - 1]$ physical blocks, in where f2fs stores data including compress header and compressed data.
- In order to eliminate write amplification during overwrite, F2FS only support compression on write-once file, data can be compressed only when all logical blocks in file are valid and cluster compress ratio is lower than specified threshold.
- To enable compression on regular inode, there are three ways:
 - `chattr +c file`
 - `chattr +c dir; touch dir/file`
 - `mount w/ -o compress_extension=ext; touch file.ext`

Compress metadata layout:



(continues on next page)

(continued from previous page)



3.26 Global File System

<https://fedorahosted.org/cluster/wiki/HomePage>

GFS is a cluster file system. It allows a cluster of computers to simultaneously use a block device that is shared between them (with FC, iSCSI, NBD, etc). GFS reads and writes to the block device like a local file system, but also uses a lock module to allow the computers coordinate their I/O so file system consistency is maintained. One of the nifty features of GFS is perfect consistency - changes made to the file system on one machine show up immediately on all other machines in the cluster.

GFS uses interchangeable inter-node locking mechanisms, the currently supported mechanisms are:

lock_nolock

- allows gfs to be used as a local file system

lock_dlm

- uses a distributed lock manager (dlm) for inter-node locking.

The dlm is found at `linux/fs/dlm/`

Lock_dlm depends on user space cluster management systems found at the URL above.

To use gfs as a local file system, no external clustering systems are needed, simply:

```
$ mkfs -t gfs2 -p lock_nolock -j 1 /dev/block_device
$ mount -t gfs2 /dev/block_device /dir
```

If you are using Fedora, you need to install the `gfs2-utils` package and, for `lock_dlm`, you will also need to install the `cman` package and write a `cluster.conf` as per the documentation. For F17 and above `cman` has been replaced by the `dlm` package.

GFS2 is not on-disk compatible with previous versions of GFS, but it is pretty close.

The following man pages can be found at the URL above:

fsck.gfs2	to repair a filesystem
gfs2_grow	to expand a filesystem online
gfs2_jadd	to add journals to a filesystem online
tunegfs2	to manipulate, examine and tune a filesystem
gfs2_convert	to convert a gfs filesystem to gfs2 in-place
mkfs.gfs2	to make a filesystem

3.27 uevents and GFS2

During the lifetime of a GFS2 mount, a number of uevents are generated. This document explains what the events are and what they are used for (by `gfs_controld` in `gfs2-utils`).

3.27.1 A list of GFS2 uevents

1. ADD

The ADD event occurs at mount time. It will always be the first uevent generated by the newly created filesystem. If the mount is successful, an ONLINE uevent will follow. If it is not successful then a REMOVE uevent will follow.

The ADD uevent has two environment variables: `SPECTATOR=[0|1]` and `RONLY=[0|1]` that specify the spectator status (a read-only mount with no journal assigned), and read-only (with journal assigned) status of the filesystem respectively.

2. ONLINE

The ONLINE uevent is generated after a successful mount or remount. It has the same environment variables as the ADD uevent. The ONLINE uevent, along with the two environment variables for spectator and RONLY are a relatively recent addition (2.6.32-rc+) and will not be generated by older kernels.

3. CHANGE

The CHANGE uevent is used in two places. One is when reporting the successful mount of the filesystem by the first node (`FIRSTMOUNT=Done`). This is used as a signal by `gfs_controld` that it is then ok for other nodes in the cluster to mount the filesystem.

The other CHANGE uevent is used to inform of the completion of journal recovery for one of the filesystems journals. It has two environment variables, `JID=` which specifies the journal id which has just been recovered, and `RECOVERY=[Done|Failed]` to indicate the success (or otherwise) of the operation. These uevents are generated for every journal recovered, whether it is during the initial mount process or as the result of `gfs_controld` requesting a specific journal recovery via the `/sys/fs/gfs2/<fsname>/lock_module/recovery` file.

Because the CHANGE uevent was used (in early versions of gfs_controld) without checking the environment variables to discover the state, we cannot add any more functions to it without running the risk of someone using an older version of the user tools and breaking their cluster. For this reason the ONLINE uevent was used when adding a new uevent for a successful mount or remount.

4. OFFLINE

The OFFLINE uevent is only generated due to filesystem errors and is used as part of the “withdraw” mechanism. Currently this doesn’t give any information about what the error is, which is something that needs to be fixed.

5. REMOVE

The REMOVE uevent is generated at the end of an unsuccessful mount or at the end of a umount of the filesystem. All REMOVE uevents will have been preceded by at least an ADD uevent for the same filesystem, and unlike the other uevents is generated automatically by the kernel’s kobject subsystem.

3.27.2 Information common to all GFS2 uevents (uevent environment variables)

1. LOCKTABLE=

The LOCKTABLE is a string, as supplied on the mount command line (locktable=) or via fstab. It is used as a filesystem label as well as providing the information for a lock_dlm mount to be able to join the cluster.

2. LOCKPROTO=

The LOCKPROTO is a string, and its value depends on what is set on the mount command line, or via fstab. It will be either lock_nolock or lock_dlm. In the future other lock managers may be supported.

3. JOURNALID=

If a journal is in use by the filesystem (journals are not assigned for spectator mounts) then this will give the numeric journal id in all GFS2 uevents.

4. UUID=

With recent versions of gfs2-utils, mkfs.gfs2 writes a UUID into the filesystem superblock. If it exists, this will be included in every uevent relating to the filesystem.

3.28 Glock internal locking rules

This documents the basic principles of the glock state machine internals. Each glock (struct gfs2_glock in fs/gfs2/incore.h) has two main (internal) locks:

1. A spinlock (gl_lockref.lock) which protects the internal state such as gl_state, gl_target and the list of holders (gl_holders)
2. A non-blocking bit lock, GLF_LOCK, which is used to prevent other threads from making calls to the DLM, etc. at the same time. If a thread takes this lock, it must then call run_queue (usually via the workqueue) when it releases it in order to ensure any pending tasks are completed.

The gl_holders list contains all the queued lock requests (not just the holders) associated with the glock. If there are any held locks, then they will be contiguous entries at the head of the list. Locks are granted in strictly the order that they are queued, except for those marked LM_FLAG_PRIORITY which are used only during recovery, and even then only for journal locks.

There are three lock states that users of the glock layer can request, namely shared (SH), deferred (DF) and exclusive (EX). Those translate to the following DLM lock modes:

Glock mode	DLM	lock mode
UN	IV/NL	Unlocked (no DLM lock associated with glock) or NL
SH	PR	(Protected read)
DF	CW	(Concurrent write)
EX	EX	(Exclusive)

Thus DF is basically a shared mode which is incompatible with the “normal” shared lock mode, SH. In GFS2 the DF mode is used exclusively for direct I/O operations. The glocks are basically a lock plus some routines which deal with cache management. The following rules apply for the cache:

Glock mode	Cache data	Cache Metadata	Dirty Data	Dirty Metadata
UN	No	No	No	No
SH	Yes	Yes	No	No
DF	No	Yes	No	No
EX	Yes	Yes	Yes	Yes

These rules are implemented using the various glock operations which are defined for each type of glock. Not all types of glocks use all the modes. Only inode glocks use the DF mode for example.

Table of glock operations and per type constants:

Field	Purpose
go_xmote	Called before remote state change (e.g. to sync dirty data)
go_xmote	Called after remote state change (e.g. to refill cache)
go_inval	Called if remote state change requires invalidating the cache
go_demote	Returns boolean value of whether its ok to demote a glock (e.g. checks timeout, and that there is no cached data)
go_lock	Called for the first local holder of a lock
go_unlock	Called on the final local unlock of a lock
go_dump	Called to print content of object for debugfs file, or on error to dump glock to the log.
go_type	The type of the glock, LM_TYPE_*
go_callback	Called if the DLM sends a callback to drop this lock
go_flags	GLOF_ASPACE is set, if the glock has an address space associated with it

The minimum hold time for each lock is the time after a remote lock grant for which we ignore remote demote requests. This is in order to prevent a situation where locks are being bounced around the cluster from node to node with none of the nodes making any progress. This tends to show up most with shared mmaped files which are being written to by multiple nodes. By delaying the demotion in response to a remote callback, that gives the userspace program time to make some progress before the pages are unmapped.

There is a plan to try and remove the go_lock and go_unlock callbacks if possible, in order to try and speed up the fast path though the locking. Also, eventually we hope to make the glock “EX” mode locally shared such that any local locking will be done with the i_mutex as required rather than via the glock.

Locking rules for glock operations:

Operation	GLF_LOCK bit lock held	gl_lockref.lock spinlock held
go_xmote_th	Yes	No
go_xmote_bh	Yes	No
go_inval	Yes	No
go_demote_ok	Sometimes	Yes
go_lock	Yes	No
go_unlock	Yes	No
go_dump	Sometimes	Yes
go_callback	Sometimes (N/A)	Yes

Note: Operations must not drop either the bit lock or the spinlock if its held on entry. go_dump and do_demote_ok must never block. Note that go_dump will only be called if the glock’s state indicates that it is caching uptodate data.

Glock locking order within GFS2:

1. i_rwsem (if required)
2. Rename glock (for rename only)

3. Inode glock(s) (Parents before children, inodes at “same level” with same parent in lock number order)
4. Rgrp glock(s) (for (de)allocation operations)
5. Transaction glock (via `gfs2_trans_begin`) for non-read operations
6. `i_rw_mutex` (if required)
7. Page lock (always last, very important!)

There are two glocks per inode. One deals with access to the inode itself (locking order as above), and the other, known as the `iopen` glock is used in conjunction with the `i_nlink` field in the inode to determine the lifetime of the inode in question. Locking of inodes is on a per-inode basis. Locking of rgrps is on a per rgrp basis. In general we prefer to lock local locks prior to cluster locks.

3.28.1 Glock Statistics

The stats are divided into two sets: those relating to the super block and those relating to an individual glock. The super block stats are done on a per cpu basis in order to try and reduce the overhead of gathering them. They are also further divided by glock type. All timings are in nanoseconds.

In the case of both the super block and glock statistics, the same information is gathered in each case. The super block timing statistics are used to provide default values for the glock timing statistics, so that newly created glocks should have, as far as possible, a sensible starting point. The per-glock counters are initialised to zero when the glock is created. The per-glock statistics are lost when the glock is ejected from memory.

The statistics are divided into three pairs of mean and variance, plus two counters. The mean/variance pairs are smoothed exponential estimates and the algorithm used is one which will be very familiar to those used to calculation of round trip times in network code. See “TCP/IP Illustrated, Volume 1” , W. Richard Stevens, sect 21.3, “Round-Trip Time Measurement” , p. 299 and onwards. Also, Volume 2, Sect. 25.10, p. 838 and onwards. Unlike the TCP/IP Illustrated case, the mean and variance are not scaled, but are in units of integer nanoseconds.

The three pairs of mean/variance measure the following things:

1. DLM lock time (non-blocking requests)
2. DLM lock time (blocking requests)
3. Inter-request time (again to the DLM)

A non-blocking request is one which will complete right away, whatever the state of the DLM lock in question. That currently means any requests when (a) the current state of the lock is exclusive, i.e. a lock demotion (b) the requested state is either null or unlocked (again, a demotion) or (c) the “try lock” flag is set. A blocking request covers all the other lock requests.

There are two counters. The first is there primarily to show how many lock requests have been made, and thus how much data has gone into the mean/variance calculations. The other counter is counting queuing of holders at the top layer of

the glock code. Hopefully that number will be a lot larger than the number of dlm lock requests issued.

So why gather these statistics? There are several reasons we'd like to get a better idea of these timings:

1. To be able to better set the glock "min hold time"
2. To spot performance issues more easily
3. To improve the algorithm for selecting resource groups for allocation (to base it on lock wait time, rather than blindly using a "try lock")

Due to the smoothing action of the updates, a step change in some input quantity being sampled will only fully be taken into account after 8 samples (or 4 for the variance) and this needs to be carefully considered when interpreting the results.

Knowing both the time it takes a lock request to complete and the average time between lock requests for a glock means we can compute the total percentage of the time for which the node is able to use a glock vs. time that the rest of the cluster has its share. That will be very useful when setting the lock min hold time.

Great care has been taken to ensure that we measure exactly the quantities that we want, as accurately as possible. There are always inaccuracies in any measuring system, but I hope this is as accurate as we can reasonably make it.

Per sb stats can be found here:

```
/sys/kernel/debug/gfs2/<fsname>/sbstats
```

Per glock stats can be found here:

```
/sys/kernel/debug/gfs2/<fsname>/glstats
```

Assuming that debugfs is mounted on /sys/kernel/debug and also that <fsname> is replaced with the name of the gfs2 filesystem in question.

The abbreviations used in the output as are follows:

srtt	Smoothed round trip time for non blocking dlm requests
srttvar	Variance estimate for srtt
srttb	Smoothed round trip time for (potentially) blocking dlm requests
srttvarb	Variance estimate for srttb
sirt	Smoothed inter request time (for dlm requests)
sirtvar	Variance estimate for sirt
dlm	Number of dlm requests made (dcnt in glstats file)
queue	Number of glock requests queued (qcnt in glstats file)

The sbstats file contains a set of these stats for each glock type (so 8 lines for each type) and for each cpu (one column per cpu). The glstats file contains a set of these stats for each glock in a similar format to the glocks file, but using the format mean/variance for each of the timing stats.

The gfs2_glock_lock_time tracepoint prints out the current values of the stats for the glock in question, along with some addition information on each dlm reply that is received:

status	The status of the dlm request
flags	The dlm request flags
tdiff	The time taken by this specific request

(remaining fields as per above list)

3.29 Macintosh HFS Filesystem for Linux

Note: This filesystem doesn't have a maintainer.

HFS stands for Hierarchical File System and is the filesystem used by the Mac Plus and all later Macintosh models. Earlier Macintosh models used MFS (Macintosh File System), which is not supported, MacOS 8.1 and newer support a filesystem called HFS+ that's similar to HFS but is extended in various areas. Use the hfsplus filesystem driver to access such filesystems from Linux.

3.29.1 Mount options

When mounting an HFS filesystem, the following options are accepted:

creator=cccc, type=cccc Specifies the creator/type values as shown by the MacOS finder used for creating new files. Default values: '????' .

uid=n, gid=n Specifies the user/group that owns all files on the filesystems. Default: user/group id of the mounting process.

dir_umask=n, file_umask=n, umask=n Specifies the umask used for all files , all directories or all files and directories. Defaults to the umask of the mounting process.

session=n Select the CDROM session to mount as HFS filesystem. Defaults to leaving that decision to the CDROM driver. This option will fail with anything but a CDROM as underlying devices.

part=n Select partition number n from the devices. Does only makes sense for CDROMS because they can't be partitioned under Linux. For disk devices the generic partition parsing code does this for us. Defaults to not parsing the partition table at all.

quiet Ignore invalid mount options instead of complaining.

3.29.2 Writing to HFS Filesystems

HFS is not a UNIX filesystem, thus it does not have the usual features you'd expect:

- You can't modify the set-uid, set-gid, sticky or executable bits or the uid and gid of files.
- You can't create hard- or symlinks, device files, sockets or FIFOs.

HFS does on the other have the concepts of multiple forks per file. These non-standard forks are represented as hidden additional files in the normal filesystems namespace which is kind of a kludge and makes the semantics for the a little strange:

- You can't create, delete or rename resource forks of files or the Finder's metadata.
- They are however created (with default values), deleted and renamed along with the corresponding data fork or directory.
- Copying files to a different filesystem will lose those attributes that are essential for MacOS to work.

3.29.3 Creating HFS filesystems

The hfsutils package from Robert Leslie contains a program called hformat that can be used to create HFS filesystem. See <<http://www.mars.org/home/rob/proj/hfs/>> for details.

3.29.4 Credits

The HFS drivers was written by Paul H. Hargrovea (hargrove@sccm.Stanford.EDU). Roman Zippel (roman@ardistech.com) rewrote large parts of the code and brought in btree routines derived from Brad Boyer's hfsplus driver.

3.30 Macintosh HFSPlus Filesystem for Linux

HFSPlus is a filesystem first introduced in MacOS 8.1. HFSPlus has several extensions to HFS, including 32-bit allocation blocks, 255-character unicode filenames, and file sizes of 2^{63} bytes.

3.30.1 Mount options

When mounting an HFSPlus filesystem, the following options are accepted:

creator=cccc, type=cccc Specifies the creator/type values as shown by the MacOS finder used for creating new files. Default values: '????' .

uid=n, gid=n Specifies the user/group that owns all files on the filesystem that have uninitialized permissions structures. Default: user/group id of the mounting process.

umask=n Specifies the umask (in octal) used for files and directories that have uninitialized permissions structures. Default: umask of the mounting process.

session=n Select the CDROM session to mount as HFSPlus filesystem. Defaults to leaving that decision to the CDROM driver. This option will fail with anything but a CDROM as underlying devices.

part=n Select partition number n from the devices. This option only makes sense for CDROMs because they can't be partitioned under Linux. For disk devices the generic partition parsing code does this for us. Defaults to not parsing the partition table at all.

decompose Decompose file name characters.

nodecompose Do not decompose file name characters.

force Used to force write access to volumes that are marked as jour-nalled or locked. Use at your own risk.

nls=cccc Encoding to use when presenting file names.

3.30.2 References

kernel source: <[file:fs/hfsplus](#)>

Apple Technote 1150 <https://developer.apple.com/legacy/library/technotes/tn/tn1150.html>

3.31 Read/Write HPFS 2.09

1998-2004, Mikulas Patocka

email mikulas@artax.karlin.mff.cuni.cz

homepage <http://artax.karlin.mff.cuni.cz/~mikulas/vyplody/hpfs/index-e.cgi>

3.31.1 Credits

Chris Smith, 1993, original read-only HPFS, some code and hpfs structures file
is taken from it

Jacques Gelinas, MSDos mmap, Inspired by fs/nfs/mmap.c (Jon Tombs 15 Aug 1993)

Werner Almesberger, 1992, 1993, MSDos option parser & CR/LF conversion

Mount options

uid=xxx,gid=xxx,umask=xxx (default uid=gid=0 umask=default_system_umask)

Set owner/group/mode for files that do not have it specified in extended attributes. Mode is inverted umask - for example umask 027 gives owner all permission, group read permission and anybody else no access. Note that for files mode is anded with 0666. If you want files to have 'x' rights, you must use extended attributes.

case=lower,asis (default asis) File name lowercasing in readdir.

conv=binary,text,auto (default binary) CR/LF -> LF conversion, if auto, decision is made according to extension - there is a list of text extensions (I think it's better to not convert text file than to damage binary file). If you want to change that list, change it in the source. Original readonly HPFS contained some strange heuristic algorithm that I removed. I think it's danger to let the computer decide whether file is text or binary. For example, DJGPP binaries contain small text message at the beginning and they could be misidentified and damaged under some circumstances.

check=none,normal,strict (default normal) Check level. Selecting none will cause only little speedup and big danger. I tried to write it so that it won't crash if check=normal on corrupted filesystems. check=strict means many superfluous checks - used for debugging (for example it checks if file is allocated in bitmaps when accessing it).

errors=continue,remount-ro,panic (default remount-ro) Behaviour when filesystem errors found.

chkdsk=no,errors,always (default errors) When to mark filesystem dirty so that OS/2 checks it.

eas=no,ro,rw (default rw) What to do with extended attributes. 'no' - ignore them and use always values specified in uid/gid/mode options. 'ro' - read extended attributes but do not create them. 'rw' - create extended attributes when you use chmod/chown/chgrp/mknod/ln -s on the filesystem.

timeshift=(-)nnn (default 0) Shifts the time by nnn seconds. For example, if you see under linux one hour more, than under os/2, use timeshift=-3600.

3.31.2 File names

As in OS/2, filenames are case insensitive. However, shell thinks that names are case sensitive, so for example when you create a file FOO, you can use 'cat FOO', 'cat Foo', 'cat foo' or 'cat F*' but not 'cat f*'. Note, that you also won't be able to compile linux kernel (and maybe other things) on HPFS because kernel creates different files with names like bootsect.S and bootsect.s. When searching for file that's name has characters ≥ 128 , codepages are used - see below. OS/2 ignores dots and spaces at the end of file name, so this driver does as well. If you create 'a. ...', the file 'a' will be created, but you can still access it under names 'a.', 'a..' , 'a...' etc.

3.31.3 Extended attributes

On HPFS partitions, OS/2 can associate to each file a special information called extended attributes. Extended attributes are pairs of (key,value) where key is an ascii string identifying that attribute and value is any string of bytes of variable length. OS/2 stores window and icon positions and file types there. So why not use it for unix-specific info like file owner or access rights? This driver can do it. If you chown/chgrp/chmod on a hpfs partition, extended attributes with keys "UID", "GID" or "MODE" and 2-byte values are created. Only that extended attributes whose value differs from defaults specified in mount options are created. Once created, the extended attributes are never deleted, they're just changed. It means that when your default uid=0 and you type something like 'chown luser file; chown root file' the file will contain extended attribute UID=0. And when you umount the fs and mount it again with uid=luser_uid, the file will be still owned by root! If you chmod file to 444, extended attribute "MODE" will not be set, this special case is done by setting read-only flag. When you mknod a block or char device, besides "MODE", the special 4-byte extended attribute "DEV" will be created containing the device number. Currently this driver cannot resize extended attributes - it means that if somebody (I don't know who?) has set "UID", "GID", "MODE" or "DEV" attributes with different sizes, they won't be rewritten and changing these values doesn't work.

3.31.4 Symlinks

You can do symlinks on HPFS partition, symlinks are achieved by setting extended attribute named "SYMLINK" with symlink value. Like on ext2, you can chown and chgrp symlinks but I don't know what is it good for. chmoding symlink results in chmoding file where symlink points. These symlinks are just for Linux use and incompatible with OS/2. OS/2 PmShell symlinks are not supported because they are stored in very crazy way. They tried to do it so that link changes when file is moved ...sometimes it works. But the link is partly stored in directory extended attributes and partly in OS2SYS.INI. I don't want (and don't know how) to analyze or change OS2SYS.INI.

3.31.5 Codepages

HPFS can contain several uppercasing tables for several codepages and each file has a pointer to codepage its name is in. However OS/2 was created in America where people don't care much about codepages and so multiple codepages support is quite buggy. I have Czech OS/2 working in codepage 852 on my disk. Once I booted English OS/2 working in cp 850 and I created a file on my 852 partition. It marked file name codepage as 850 - good. But when I again booted Czech OS/2, the file was completely inaccessible under any name. It seems that OS/2 uppercases the search pattern with its system code page (852) and file name it's comparing to with its code page (850). These could never match. Is it really what IBM developers wanted? But problems continued. When I created in Czech OS/2 another file in that directory, that file was inaccessible too. OS/2 probably uses different uppercasing method when searching where to place a file (note, that files in HPFS directory must be sorted) and when searching for a file. Finally when I opened this directory in PmShell, PmShell crashed (the funny thing was that, when rebooted, PmShell tried to reopen this directory again :-). `chkdsk` happily ignores these errors and only low-level disk modification saved me. Never mix different language versions of OS/2 on one system although HPFS was designed to allow that. OK, I could implement complex codepage support to this driver but I think it would cause more problems than benefit with such buggy implementation in OS/2. So this driver simply uses first codepage it finds for uppercasing and lowercasing no matter what's file codepage index. Usually all file names are in this codepage - if you don't try to do what I described above :-)

3.31.6 Known bugs

HPFS386 on OS/2 server is not supported. HPFS386 installed on normal OS/2 client should work. If you have OS/2 server, use only read-only mode. I don't know how to handle some HPFS386 structures like access control list or extended perm list, I don't know how to delete them when file is deleted and how to not overwrite them with extended attributes. Send me some info on these structures and I'll make it. However, this driver should detect presence of HPFS386 structures, remount read-only and not destroy them (I hope).

When there's not enough space for extended attributes, they will be truncated and no error is returned.

OS/2 can't access files if the path is longer than about 256 chars but this driver allows you to do it. `chkdsk` ignores such errors.

Sometimes you won't be able to delete some files on a very full filesystem (returning error `ENOSPC`). That's because file in non-leaf node in directory tree (one directory, if it's large, has `dirents` in tree on HPFS) must be replaced with another node when deleted. And that new file might have larger name than the old one so the new name doesn't fit in directory node (`dnode`). And that would result in directory tree splitting, that takes disk space. Workaround is to delete other files that are leaf (probability that the file is non-leaf is about 1/50) or to truncate file first to make some space. You encounter this problem only if you have many directories so that preallocated directory band is full i.e.:

```
number_of_directories / size_of_filesystem_in_mb > 4.
```

You can't delete open directories.

You can't rename over directories (what is it good for?).

Renaming files so that only case changes doesn't work. This driver supports it but vfs doesn't. Something like 'mv file FILE' won't work.

All atimes and directory mtimes are not updated. That's because of performance reasons. If you extremely wish to update them, let me know, I'll write it (but it will be slow).

When the system is out of memory and swap, it may slightly corrupt filesystem (lost files, unbalanced directories). (I guess all filesystem may do it).

When compiled, you get warning: function declaration isn't a prototype. Does anybody know what does it mean?

3.31.7 What does “unbalanced tree” message mean?

Old versions of this driver created sometimes unbalanced dnode trees. OS/2 chkdsk doesn't scream if the tree is unbalanced (and sometimes creates unbalanced trees too :-)) but both HPFS and HPFS386 contain bug that it rarely crashes when the tree is not balanced. This driver handles unbalanced trees correctly and writes warning if it finds them. If you see this message, this is probably because of directories created with old version of this driver. Workaround is to move all files from that directory to another and then back again. Do it in Linux, not OS/2! If you see this message in directory that is whole created by this driver, it is BUG - let me know about it.

3.31.8 Bugs in OS/2

When you have two (or more) lost directories pointing each to other, chkdsk locks up when repairing filesystem.

Sometimes (I think it's random) when you create a file with one-char name under OS/2, OS/2 marks it as 'long'. chkdsk then removes this flag saying “Minor fs error corrected”.

File names like “a .b” are marked as 'long' by OS/2 but chkdsk “corrects” it and marks them as short (and writes “minor fs error corrected”). This bug is not in HPFS386.

3.31.9 Codepage bugs described above

If you don't install fixpacks, there are many, many more...

3.31.10 History

0.90	First public release
0.91	Fixed bug that caused shooting to memory when write_inode was called on open inode (rarely happened)
0.92	Fixed a little memory leak in freeing directory inodes
0.93	Fixed bug that locked up the machine when there were too many filenames with first 15 characters same Fixed write_file to zero file when writing behind file end
0.94	Fixed a little memory leak when trying to delete busy file or directory
0.95	Fixed a bug that i_hpfs_parent_dir was not updated when moving files
1.90	First version for 2.1.1xx kernels
1.91	Fixed a bug that chk_sectors failed when sectors were at the end of disk Fixed a race-condition when write_inode is called while deleting file Fixed a bug that could possibly happen (with very low probability) when using 0xff in filenames. Rewritten locking to avoid race-conditions Mount option 'eas' now works Fsync no longer returns error Files beginning with '.' are marked hidden Remount support added Alloc is not so slow when filesystem becomes full Atimes are no more updated because it slows down operation Code cleanup (removed all commented debug prints)
1.92	Corrected a bug when sync was called just before closing file
1.93	Modified, so that it works with kernels >= 2.1.131, I don't know if it works with previous versions Fixed a possible problem with disks > 64G (but I don't have one, so I can't test it) Fixed a file overflow at 2G Added new option 'timeshift' Changed behaviour on HPFS386: It is now possible to operate on HPFS386 in read-only mode Fixed a bug that slowed down alloc and prevented allocating 100% space (this bug was not destructive)
1.94	Added workaround for one bug in Linux Fixed one buffer leak Fixed some incompatibilities with large extended attributes (but it's still not 100% ok, I have no info on it and OS/2 doesn't want to create them) Rewritten allocation Fixed a bug with i_blocks (du sometimes didn't display correct values) Directories have no longer archive attribute set (some programs don't like it) Fixed a bug that it set badly one flag in large anode tree (it was not destructive)
1.95	Fixed one buffer leak, that could happen on corrupted filesystem Fixed one bug in allocation in 1.94
1.96	Added workaround for one bug in OS/2 (HPFS locked up, HPFS386 reported error sometimes when opening directories in PMSHELL) Fixed a possible bitmap race Fixed possible problem on large disks You can now delete open files
3.31.10	Read/Write HPFS 2.09 457
1.97	Support for HPFS v3 (on large partitions) Fixed a bug that it didn't allow creation of files > 128M (it should be 2G)
1.97	Changed names of global symbols

3.32 FUSE

3.32.1 Definitions

Userspace filesystem: A filesystem in which data and metadata are provided by an ordinary userspace process. The filesystem can be accessed normally through the kernel interface.

Filesystem daemon: The process(es) providing the data and metadata of the filesystem.

Non-privileged mount (or user mount): A userspace filesystem mounted by a non-privileged (non-root) user. The filesystem daemon is running with the privileges of the mounting user. NOTE: this is not the same as mounts allowed with the “user” option in `/etc/fstab`, which is not discussed here.

Filesystem connection: A connection between the filesystem daemon and the kernel. The connection exists until either the daemon dies, or the filesystem is unmounted. Note that detaching (or lazy unmounting) the filesystem does not break the connection, in this case it will exist until the last reference to the filesystem is released.

Mount owner: The user who does the mounting.

User: The user who is performing filesystem operations.

3.32.2 What is FUSE?

FUSE is a userspace filesystem framework. It consists of a kernel module (`fuse.ko`), a userspace library (`libfuse.*`) and a mount utility (`fusermount`).

One of the most important features of FUSE is allowing secure, non-privileged mounts. This opens up new possibilities for the use of filesystems. A good example is `sshfs`: a secure network filesystem using the `sftp` protocol.

The userspace library and utilities are available from the [FUSE homepage](#):

3.32.3 Filesystem type

The filesystem type given to `mount(2)` can be one of the following:

fuse This is the usual way to mount a FUSE filesystem. The first argument of the mount system call may contain an arbitrary string, which is not interpreted by the kernel.

fuseblk The filesystem is block device based. The first argument of the mount system call is interpreted as the name of the device.

3.32.4 Mount options

fd=N The file descriptor to use for communication between the userspace filesystem and the kernel. The file descriptor must have been obtained by opening the FUSE device (`‘/dev/fuse’`).

rootmode=M The file mode of the filesystem’s root in octal representation.

user_id=N The numeric user id of the mount owner.

group_id=N The numeric group id of the mount owner.

default_permissions By default FUSE doesn’t check file access permissions, the filesystem is free to implement its access policy or leave it to the underlying file access mechanism (e.g. in case of network filesystems). This option enables permission checking, restricting access based on file mode. It is usually useful together with the `‘allow_other’` mount option.

allow_other This option overrides the security measure restricting file access to the user mounting the filesystem. This option is by default only allowed to root, but this restriction can be removed with a (userspace) configuration option.

max_read=N With this option the maximum size of read operations can be set. The default is infinite. Note that the size of read requests is limited anyway to 32 pages (which is 128kbyte on i386).

blksize=N Set the block size for the filesystem. The default is 512. This option is only valid for `‘fuseblk’` type mounts.

3.32.5 Control filesystem

There’s a control filesystem for FUSE, which can be mounted by:

```
mount -t fusectl none /sys/fs/fuse/connections
```

Mounting it under the `‘/sys/fs/fuse/connections’` directory makes it backwards compatible with earlier versions.

Under the fuse control filesystem each connection has a directory named by a unique number.

For each connection the following files exist within this directory:

waiting The number of requests which are waiting to be transferred to userspace or being processed by the filesystem daemon. If there is no filesystem activity and `‘waiting’` is non-zero, then the filesystem is hung or deadlocked.

abort Writing anything into this file will abort the filesystem connection. This means that all waiting requests will be aborted an error returned for all aborted and new requests.

Only the owner of the mount may read or write these files.

Interrupting filesystem operations

If a process issuing a FUSE filesystem request is interrupted, the following will happen:

- If the request is not yet sent to userspace AND the signal is fatal (SIGKILL or unhandled fatal signal), then the request is dequeued and returns immediately.
- If the request is not yet sent to userspace AND the signal is not fatal, then an interrupted flag is set for the request. When the request has been successfully transferred to userspace and this flag is set, an INTERRUPT request is queued.
- If the request is already sent to userspace, then an INTERRUPT request is queued.

INTERRUPT requests take precedence over other requests, so the userspace filesystem will receive queued INTERRUPTs before any others.

The userspace filesystem may ignore the INTERRUPT requests entirely, or may honor them by sending a reply to the original request, with the error set to EINTR.

It is also possible that there's a race between processing the original request and its INTERRUPT request. There are two possibilities:

1. The INTERRUPT request is processed before the original request is processed
2. The INTERRUPT request is processed after the original request has been answered

If the filesystem cannot find the original request, it should wait for some timeout and/or a number of new requests to arrive, after which it should reply to the INTERRUPT request with an EAGAIN error. In case 1) the INTERRUPT request will be requeued. In case 2) the INTERRUPT reply will be ignored.

3.32.6 Aborting a filesystem connection

It is possible to get into certain situations where the filesystem is not responding. Reasons for this may be:

- a) Broken userspace filesystem implementation
- b) Network connection down
- c) Accidental deadlock
- d) Malicious deadlock

(For more on c) and d) see later sections)

In either of these cases it may be useful to abort the connection to the filesystem. There are several ways to do this:

- Kill the filesystem daemon. Works in case of a) and b)
- Kill the filesystem daemon and all users of the filesystem. Works in all cases except some malicious deadlocks

- Use forced unmount (umount -f). Works in all cases but only if filesystem is still attached (it hasn't been lazy unmounted)
- Abort filesystem through the FUSE control filesystem. Most powerful method, always works.

3.32.7 How do non-privileged mounts work?

Since the mount() system call is a privileged operation, a helper program (fusermount) is needed, which is installed setuid root.

The implication of providing non-privileged mounts is that the mount owner must not be able to use this capability to compromise the system. Obvious requirements arising from this are:

- A) mount owner should not be able to get elevated privileges with the help of the mounted filesystem
- B) mount owner should not get illegitimate access to information from other users' and the super user's processes
- C) mount owner should not be able to induce undesired behavior in other users' or the super user's processes

3.32.8 How are requirements fulfilled?

- A) The mount owner could gain elevated privileges by either:
 1. creating a filesystem containing a device file, then opening this device
 2. creating a filesystem containing a suid or sgid application, then executing this application

The solution is not to allow opening device files and ignore setuid and setgid bits when executing programs. To ensure this fusermount always adds "nosuid" and "nodev" to the mount options for non-privileged mounts.

- B) If another user is accessing files or directories in the filesystem, the filesystem daemon serving requests can record the exact sequence and timing of operations performed. This information is otherwise inaccessible to the mount owner, so this counts as an information leak.

The solution to this problem will be presented in point 2) of C).

- C) There are several ways in which the mount owner can induce undesired behavior in other users' processes, such as:
 - 1) mounting a filesystem over a file or directory which the mount owner could otherwise not be able to modify (or could only make limited modifications).

This is solved in fusermount, by checking the access permissions on the mountpoint and only allowing the mount if the mount owner can do unlimited modification (has write access to the mountpoint, and mountpoint is not a "sticky" directory)

- 2) Even if 1) is solved the mount owner can change the behavior of other users' processes.
 - i) It can slow down or indefinitely delay the execution of a filesystem operation creating a DoS against the user or the whole system. For example a suid application locking a system file, and then accessing a file on the mount owner' s filesystem could be stopped, and thus causing the system file to be locked forever.
 - ii) It can present files or directories of unlimited length, or directory structures of unlimited depth, possibly causing a system process to eat up disk space, memory or other resources, again causing DoS.

The solution to this as well as B) is not to allow processes to access the filesystem, which could otherwise not be monitored or manipulated by the mount owner. Since if the mount owner can ptrace a process, it can do all of the above without using a FUSE mount, the same criteria as used in ptrace can be used to check if a process is allowed to access the filesystem or not.

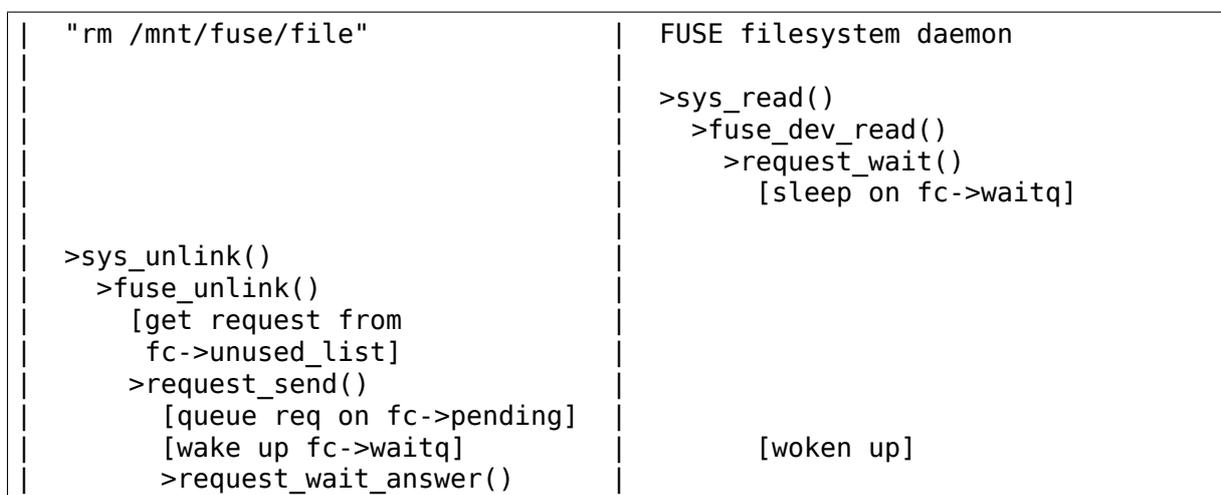
Note that the ptrace check is not strictly necessary to prevent B/2/i, it is enough to check if mount owner has enough privilege to send signal to the process accessing the filesystem, since SIGSTOP can be used to get a similar effect.

3.32.9 I think these limitations are unacceptable?

If a sysadmin trusts the users enough, or can ensure through other measures, that system processes will never enter non-privileged mounts, it can relax the last limitation with a 'user_allow_other' config option. If this config option is set, the mounting user can add the 'allow_other' mount option which disables the check for other users' processes.

3.32.10 Kernel - userspace interface

The following diagram shows how a filesystem operation (in this example unlink) is performed in FUSE.



(continues on next page)

(continued from previous page)

<pre> [sleep on req->waitq] [awoken up] <request_wait_answer() request_send() [add request to fc->unused_list] <fuse_unlink() <sys_unlink() </pre>	<pre> <request_wait() [remove req from fc->pending] [copy req to read buffer] [add req to fc->processing] <fuse_dev_read() <sys_read() [perform unlink] >sys_write() >fuse_dev_write() [look up req in fc->processing] [remove from fc->processing] [copy write buffer to req] [wake up req->waitq] <fuse_dev_write() <sys_write() </pre>
---	---

Note: Everything in the description above is greatly simplified

There are a couple of ways in which to deadlock a FUSE filesystem. Since we are talking about unprivileged userspace programs, something must be done about these.

Scenario 1 - Simple deadlock:

<pre> "rm /mnt/fuse/file" >sys_unlink("/mnt/fuse/file") [acquire inode semaphore for "file"] >fuse_unlink() [sleep on req->waitq] </pre>	<pre> FUSE filesystem daemon <sys_read() >sys_unlink("/mnt/fuse/file") [acquire inode semaphore for "file"] *DEADLOCK* </pre>
--	--

The solution for this is to allow the filesystem to be aborted.

Scenario 2 - Tricky deadlock

This one needs a carefully crafted filesystem. It's a variation on the above, only the call back to the filesystem is not explicit, but is caused by a pagefault.

<pre> Kamikaze filesystem thread 1 [fd = open("/mnt/fuse/file")] [mmap fd to 'addr'] [close fd] [read a byte from addr] >do_page_fault() [find or create page] [lock page] >fuse_readpage() [queue READ request] [sleep on req->waitq] </pre>	<pre> Kamikaze filesystem thread 2 [request served normally] [FLUSH triggers 'magic' flag] [read request to buffer] [create reply header before addr] >sys_write(addr - headerlength) >fuse_dev_write() [look up req in fc->processing] [remove from fc->processing] [copy write buffer to req] >do_page_fault() [find or create page] [lock page] * DEADLOCK * </pre>
--	---

The solution is basically the same as above.

An additional problem is that while the write buffer is being copied to the request, the request must not be interrupted/aborted. This is because the destination address of the copy may not be valid after the request has returned.

This is solved with doing the copy atomically, and allowing abort while the page(s) belonging to the write buffer are faulted with `get_user_pages()`. The `'req->locked'` flag indicates when the copy is taking place, and abort is delayed until this flag is unset.

3.33 Fuse I/O Modes

Fuse supports the following I/O modes:

- direct-io
- cached + write-through + writeback-cache

The direct-io mode can be selected with the `FOPEN_DIRECT_IO` flag in the `FUSE_OPEN` reply.

In direct-io mode the page cache is completely bypassed for reads and writes. No read-ahead takes place. Shared mmap is disabled.

In cached mode reads may be satisfied from the page cache, and data may be read-ahead by the kernel to fill the cache. The cache is always kept consistent after any writes to the file. All mmap modes are supported.

The cached mode has two sub modes controlling how writes are handled. The write-through mode is the default and is supported on all kernels. The writeback-

cache mode may be selected by the `FUSE_WRITEBACK_CACHE` flag in the `FUSE_INIT` reply.

In write-through mode each write is immediately sent to userspace as one or more `WRITE` requests, as well as updating any cached pages (and caching previously uncached, but fully written pages). No `READ` requests are ever sent for writes, so when an uncached page is partially written, the page is discarded.

In writeback-cache mode (enabled by the `FUSE_WRITEBACK_CACHE` flag) writes go to the cache only, which means that the `write(2)` syscall can often complete very fast. Dirty pages are written back implicitly (background writeback or page reclaim on memory pressure) or explicitly (invoked by `close(2)`, `fsync(2)` and when the last ref to the file is being released on `munmap(2)`). This mode assumes that all changes to the filesystem go through the FUSE kernel module (size and `atime/ctime/mtime` attributes are kept up-to-date by the kernel), so it's generally not suitable for network filesystems. If a partial page is written, then the page needs to be first read from userspace. This means, that even for files opened for `O_WRONLY` it is possible that `READ` requests will be generated by the kernel.

3.34 Inotify - A Powerful yet Simple File Change Notification System

Document started 15 Mar 2005 by Robert Love <rml@novell.com>

Document updated 4 Jan 2015 by Zhang Zhen <zhenzhang.zhang@huawei.com>

- Deleted obsoleted interface, just refer to manpages for user interface.

(i) Rationale

Q: What is the design decision behind not tying the watch to the open fd of the watched object?

A: Watches are associated with an open inotify device, not an open file. This solves the primary problem with `dnotify`: keeping the file open pins the file and thus, worse, pins the mount. `Dnotify` is therefore infeasible for use on a desktop system with removable media as the media cannot be unmounted. Watching a file should not require that it be open.

Q: What is the design decision behind using `an-fd-per-instance` as opposed to an `fd-per-watch`?

A: An `fd-per-watch` quickly consumes more file descriptors than are allowed, more `fd`'s than are feasible to manage, and more `fd`'s than are optimally `select()`-able. Yes, root can bump the per-process fd limit and yes, users can use `epoll`, but requiring both is a silly and extraneous requirement. A watch consumes less memory than an open file, separating the number spaces is thus sensible. The current design is what user-space developers want: Users initialize inotify, once, and add `n` watches, requiring but one fd and no twiddling with fd limits. Initializing an inotify instance two thousand times is silly. If we can implement user-space's preferences cleanly—and we can, the `idr` layer makes stuff like this trivial—then we should.

There are other good arguments. With a single fd, there is a single item to block on, which is mapped to a single queue of events. The single fd returns

all watch events and also any potential out-of-band data. If every fd was a separate watch,

- There would be no way to get event ordering. Events on file foo and file bar would pop poll() on both fd' s, but there would be no way to tell which happened first. A single queue trivially gives you ordering. Such ordering is crucial to existing applications such as Beagle. Imagine “mv a b ; mv b a” events without ordering.
- We' d have to maintain n fd' s and n internal queues with state, versus just one. It is a lot messier in the kernel. A single, linear queue is the data structure that makes sense.
- User-space developers prefer the current API. The Beagle guys, for example, love it. Trust me, I asked. It is not a surprise: Who' d want to manage and block on 1000 fd' s via select?
- No way to get out of band data.
- 1024 is still too low. ;-)

When you talk about designing a file change notification system that scales to 1000s of directories, juggling 1000s of fd' s just does not seem the right interface. It is too heavy.

Additionally, it is possible to more than one instance and juggle more than one queue and thus more than one associated fd. There need not be a one-fd-per-process mapping; it is one-fd-per-queue and a process can easily want more than one queue.

Q: Why the system call approach?

A: The poor user-space interface is the second biggest problem with dnotify. Signals are a terrible, terrible interface for file notification. Or for anything, for that matter. The ideal solution, from all perspectives, is a file descriptor-based one that allows basic file I/O and poll/select. Obtaining the fd and managing the watches could have been done either via a device file or a family of new system calls. We decided to implement a family of system calls because that is the preferred approach for new kernel interfaces. The only real difference was whether we wanted to use open(2) and ioctl(2) or a couple of new system calls. System calls beat ioctls.

3.35 ISO9660 Filesystem

Mount options that are the same as for msdos and vfat partitions.

gid=nnn	All files in the partition will be in group nnn.
uid=nnn	All files in the partition will be owned by user id nnn.
umask=nnn	The permission mask (see umask(1)) for the partition.

Mount options that are the same as vfat partitions. These are only useful when using discs encoded using Microsoft' s Joliet extensions.

iocharset	Character set to use for converting from Unicode to ASCII. Joliet filenames are stored in Unicode format, but Unix for the most part doesn't know how to deal with Unicode. There is also an option of doing UTF-8 translations with the utf8 option.
utf8	Encode Unicode names in UTF-8 format. Default is no.

Mount options unique to the iso9660 filesystem.

block=512	Set the block size for the disk to 512 bytes
block=1024	Set the block size for the disk to 1024 bytes
block=2048	Set the block size for the disk to 2048 bytes
check=relaxed	Matches filenames with different cases
check=strict	Matches only filenames with the exact same case
cruft	Try to handle badly formatted CDs.
map=off	Do not map non-Rock Ridge filenames to lower case
map=normal	Map non-Rock Ridge filenames to lower case
map=acorn	As map=normal but also apply Acorn extensions if present
mode=xxx	Sets the permissions on files to xxx unless Rock Ridge extensions set the permissions otherwise
dmode=xxx	Sets the permissions on directories to xxx unless Rock Ridge extensions set the permissions otherwise
override-rock-perm	Set permissions on files and directories according to 'mode' and 'dmode' even though Rock Ridge extensions are present.
nojoliet	Ignore Joliet extensions if they are present.
norock	Ignore Rock Ridge extensions if they are present.
hide	Completely strip hidden files from the file system.
showassoc	Show files marked with the 'associated' bit
unhide	Deprecated; showing hidden files is now default; If given, it is a synonym for 'showassoc' which will recreate previous unhide behavior
session=x	Select number of session on multisession CD
sbsector=xxx	Session begins from sector xxx

Recommended documents about ISO 9660 standard are located at:

- <http://www.y-adagio.com/>
- <ftp://ftp.ecma.ch/ecma-st/Ecma-119.pdf>

Quoting from the PDF “This 2nd Edition of Standard ECMA-119 is technically identical with ISO 9660.” , so it is a valid and gratis substitute of the official ISO specification.

3.36 NILFS2

NILFS2 is a log-structured file system (LFS) supporting continuous snapshotting. In addition to versioning capability of the entire file system, users can even restore files mistakenly overwritten or destroyed just a few seconds ago. Since NILFS2 can keep consistency like conventional LFS, it achieves quick recovery after system crashes.

NILFS2 creates a number of checkpoints every few seconds or per synchronous write basis (unless there is no change). Users can select significant versions among continuously created checkpoints, and can change them into snapshots which will be preserved until they are changed back to checkpoints.

There is no limit on the number of snapshots until the volume gets full. Each snapshot is mountable as a read-only file system concurrently with its writable mount, and this feature is convenient for online backup.

The userland tools are included in `nilfs-utils` package, which is available from the following download page. At least “`mkfs.nilfs2`”, “`mount.nilfs2`”, “`umount.nilfs2`”, and “`nilfs_cleanerd`” (so called cleaner or garbage collector) are required. Details on the tools are described in the man pages included in the package.

Project web page <https://nilfs.sourceforge.io/>

Download page <https://nilfs.sourceforge.io/en/download.html>

List info <http://vger.kernel.org/vger-lists.html#linux-nilfs>

3.36.1 Caveats

Features which NILFS2 does not support yet:

- `atime`
- extended attributes
- POSIX ACLs
- `quotas`
- `fsck`
- defragmentation

3.36.2 Mount options

NILFS2 supports the following mount options: (*) == default

barrier(*)	This enables/disables the use of write barriers. This requires an IO stack which can support barriers, and if nilfs gets an error on a barrier write, it will disable again with a warning.
no-barrier	
errors=continue	Keep going on a filesystem error.
errors=remount-ro(*)	Remount the filesystem read-only on an error.
errors=panic	Panic and halt the machine if an error occurs.
cp=n	Specify the checkpoint-number of the snapshot to be mounted. Checkpoints and snapshots are listed by lscp user command. Only the checkpoints marked as snapshot are mountable with this option. Snapshot is read-only, so a read-only mount option must be specified together.
order=relaxed(*)	Apply relaxed order semantics that allows modified data blocks to be written to disk without making a checkpoint if no metadata update is going. This mode is equivalent to the ordered data mode of the ext3 filesystem except for the updates on data blocks still conserve atomicity. This will improve synchronous write performance for overwriting.
order=strict	Apply strict in-order semantics that preserves sequence of all file operations including overwriting of data blocks. That means, it is guaranteed that no overtaking of events occurs in the recovered file system after a crash.
norecovery	Disable recovery of the filesystem on mount. This disables every write access on the device for read-only mounts or snapshots. This option will fail for r/w mounts on an unclean volume.
discard	This enables/disables the use of discard/TRIM commands.
nodiscard(*)	The discard/TRIM commands are sent to the underlying block device when blocks are freed. This is useful for SSD devices and sparse/thinly-provisioned LUNs.

3.36.3 ioctl

There is some NILFS2 specific functionality which can be accessed by applications through the system call interfaces. The list of all NILFS2 specific ioctls are shown in the table below.

Table of NILFS2 specific ioctls:

ioctl	Description
NILFS_IOCTL_CHANGE_CPMODE	Change checkpoint between checkpoint and snapshot state. This ioctl is used in chcp and mkcp utilities.
NILFS_IOCTL_DELETE_CHECKPOINT	Remove the checkpoint in NILFS2 file system. This ioctl is used in rmcp utility.
NILFS_IOCTL_GET_CPINFO	Return info of requested checkpoints. This ioctl is used in lscp utility and by nilfs_cleanerd daemon.
NILFS_IOCTL_GET_CESTAT	Return checkpoints statistics. This ioctl is used by lscp, rmcp utilities and by nilfs_cleanerd daemon.
NILFS_IOCTL_GET_SEGMENT_USAGE	Return segment usage info about requested segments. This ioctl is used in lssu, nilfs_resize utilities and by nilfs_cleanerd daemon.
NILFS_IOCTL_GET_SEGMENT_USAGE	Modify segment usage info of requested segments. This ioctl is used by nilfs_cleanerd daemon to skip unnecessary cleaning operation of segments and reduce performance penalty or wear of flash device due to redundant move of in-use blocks.
NILFS_IOCTL_GET_SEGMENT_USAGE	Return segment usage statistics. This ioctl is used in lssu, nilfs_resize utilities and by nilfs_cleanerd daemon.
NILFS_IOCTL_GET_MAPPING	Return mapping on virtual block addresses. This ioctl is used by nilfs_cleanerd daemon.
NILFS_IOCTL_GET_DESCRIPTOR	Return info about descriptors of disk block numbers. This ioctl is used by nilfs_cleanerd daemon.
NILFS_IOCTL_GET_SEGMENT_PARAMS	Do garbage collection operation in the environment of requested parameters from userspace. This ioctl is used by nilfs_cleanerd daemon.
NILFS_IOCTL_SYNC	Make sync checkpoint. This ioctl is used in mkcp utility.
NILFS_IOCTL_RESIZE	Resize NILFS2 volume. This ioctl is used by nilfs_resize utility.
NILFS_IOCTL_SET_ALLOC_RANGE	Define lower and upper limit of segments in bytes and upper limit of segments in bytes. This ioctl is used by nilfs_resize utility.

3.36.4 NILFS2 usage

To use nilfs2 as a local file system, simply:

```
# mkfs -t nilfs2 /dev/block_device
# mount -t nilfs2 /dev/block_device /dir
```

This will also invoke the cleaner through the mount helper program (mount.nilfs2).

Checkpoints and snapshots are managed by the following commands. Their manpages are included in the nilfs-utils package above.

lscp	list checkpoints or snapshots.
mkcp	make a checkpoint or a snapshot.
chcp	change an existing checkpoint to a snapshot or vice versa.
rmcp	invalidate specified checkpoint(s).

The logs include regular files, directory files, symbolic link files and several meta data files. The meta data files are the files used to maintain file system meta data. The current version of NILFS2 uses the following meta data files:

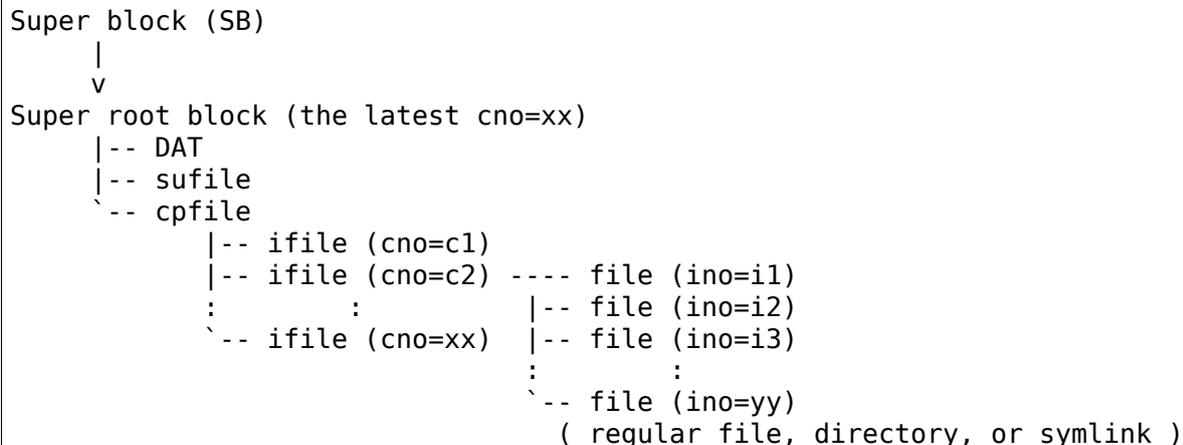
- | | |
|--|--|
| 1) Inode file (ifile) | -- Stores on-disk inodes |
| 2) Checkpoint file (cpfile) | -- Stores checkpoints |
| 3) Segment usage file (sufile) | -- Stores allocation state of segments |
| 4) Data address translation file (DAT) | -- Maps virtual block numbers to usual block numbers. This file serves to make on-disk blocks relocatable. |

The following figure shows a typical organization of the logs:



To stride over segment boundaries, this sequence of files may be split into multiple logs. The sequence of logs that should be treated as logically one log, is delimited with flags marked in the segment summary. The recovery code of nilfs2 looks this boundary information to ensure atomicity of updates.

The super root block is inserted for every checkpoints. It includes three special inodes, inodes for the DAT, cpfile, and sufile. Inodes of regular files, directories, symlinks and other special files, are included in the ifile. The inode of ifile itself is included in the corresponding checkpoint entry in the cpfile. Thus, the hierarchy among NILFS2 files can be depicted as follows:



For detail on the format of each file, please see `nilfs2_ondisk.h` located at `include/uapi/linux` directory.

There are no patents or other intellectual property that we protect with regard to the design of NILFS2. It is allowed to replicate the design in hopes that other operating systems could share (mount, read, write, etc.) data stored in this format.

3.37 NFS

3.37.1 Reference counting in pnfs

There are several inter-related caches. We have layouts which can reference multiple devices, each of which can reference multiple data servers. Each data server can be referenced by multiple devices. Each device can be referenced by multiple layouts. To keep all of this straight, we need to reference count.

struct pnfs_layout_hdr

The on-the-wire command `LAYOUTGET` corresponds to `struct pnfs_layout_segment`, usually referred to by the variable name `lseg`. Each `nfs_inode` may hold a pointer to a cache of these layout segments in `nfsi->layout`, of type `struct pnfs_layout_hdr`.

We reference the header for the inode pointing to it, across each outstanding RPC call that references it (`LAYOUTGET`, `LAYOUTRETURN`, `LAYOUTCOMMIT`), and for each `lseg` held within.

Each header is also (when non-empty) put on a list associated with `struct nfs_client` (`cl_layouts`). Being put on this list does not bump the reference count, as the layout is kept around by the `lseg` that keeps it in the list.

deviceid_cache

`lsegs` reference device ids, which are resolved per `nfs_client` and layout driver type. The device ids are held in a RCU cache (`struct nfs4_deviceid_cache`). The cache itself is referenced across each mount. The entries (`struct nfs4_deviceid`) themselves are held across the lifetime of each `lseg` referencing them.

RCU is used because the `deviceid` is basically a write once, read many data structure. The `hlist` size of 32 buckets needs better justification, but seems reasonable given that we can have multiple `deviceid`'s per filesystem, and multiple filesystems per `nfs_client`.

The hash code is copied from the `nfsd` code base. A discussion of hashing and variations of this algorithm can be found [here](#).

data server cache

file driver devices refer to data servers, which are kept in a module level cache. Its reference is held over the lifetime of the `deviceid` pointing to it.

lseg

lseg maintains an extra reference corresponding to the NFS_LSEG_VALID bit which holds it in the `pnfs_layout_hdr`'s list. When the final lseg is removed from the `pnfs_layout_hdr`'s list, the NFS_LAYOUT_DESTROYED bit is set, preventing any new lsegs from being added.

layout drivers

PNFS utilizes what is called layout drivers. The STD defines 4 basic layout types: "files", "objects", "blocks", and "flexfiles". For each of these types there is a layout-driver with a common function-vectors table which are called by the nfs-client `pnfs-core` to implement the different layout types.

Files-layout-driver code is in: `fs/nfs/filelayout/..` directory
Blocks-layout-driver code is in: `fs/nfs/blocklayout/..` directory
Flexfiles-layout-driver code is in: `fs/nfs/flexfilelayout/..` directory

blocks-layout setup

TODO: Document the setup needs of the blocks layout driver

3.37.2 RPC Cache

This document gives a brief introduction to the caching mechanisms in the `sunrpc` layer that is used, in particular, for NFS authentication.

Caches

The caching replaces the old exports table and allows for a wide variety of values to be caches.

There are a number of caches that are similar in structure though quite possibly very different in content and use. There is a corpus of common code for managing these caches.

Examples of caches that are likely to be needed are:

- mapping from IP address to client name
- mapping from client name and filesystem to export options
- mapping from UID to list of GIDs, to work around NFS's limitation of 16 gids.
- mappings between local UID/GID and remote UID/GID for sites that do not have uniform uid assignment
- mapping from network identify to public key for crypto authentication.

The common code handles such things as:

- general cache lookup with correct locking
- supporting 'NEGATIVE' as well as positive entries

- allowing an EXPIRED time on cache items, and removing items after they expire, and are no longer in-use.
- making requests to user-space to fill in cache entries
- allowing user-space to directly set entries in the cache
- delaying RPC requests that depend on as-yet incomplete cache entries, and replaying those requests when the cache entry is complete.
- clean out old entries as they expire.

Creating a Cache

- A cache needs a datum to store. This is in the form of a structure definition that must contain a struct cache_head as an element, usually the first. It will also contain a key and some content. Each cache element is reference counted and contains expiry and update times for use in cache management.
- A cache needs a “cache_detail” structure that describes the cache. This stores the hash table, some parameters for cache management, and some operations detailing how to work with particular cache items.

The operations are:

struct cache_head *alloc(void) This simply allocates appropriate memory and returns a pointer to the cache_detail embedded within the structure

void cache_put(struct kref *) This is called when the last reference to an item is dropped. The pointer passed is to the ‘ref’ field in the cache_head. cache_put should release any references create by ‘cache_init’ and, if CACHE_VALID is set, any references created by cache_update. It should then release the memory allocated by ‘alloc’ .

int match(struct cache_head *orig, struct cache_head *new)
test if the keys in the two structures match. Return 1 if they do, 0 if they don’ t.

void init(struct cache_head *orig, struct cache_head *new)
Set the ‘key’ fields in ‘new’ from ‘orig’ . This may include taking references to shared objects.

void update(struct cache_head *orig, struct cache_head *new)
Set the ‘content’ fields in ‘new’ from ‘orig’ .

int cache_show(struct seq_file *m, struct cache_detail *cd, struct cache_head *h)
Optional. Used to provide a /proc file that lists the contents of a cache. This should show one item, usually on just one line.

int cache_request(struct cache_detail *cd, struct cache_head *h, char **bpp, int *blen)
Format a request to be send to user-space for an item to be instantiated. *bpp is a buffer of size *blen. bpp should be moved forward over the encoded message, and *blen should be reduced to show how much free space remains. Return 0 on success or <0 if not enough room or other problem.

int cache_parse(struct cache_detail *cd, char *buf, int len) A message from user space has arrived to fill out a cache entry. It is in 'buf' of length 'len'. cache_parse should parse this, find the item in the cache with sunrpc_cache_lookup_rcu, and update the item with sunrpc_cache_update.

- A cache needs to be registered using cache_register(). This includes it on a list of caches that will be regularly cleaned to discard old data.

Using a cache

To find a value in a cache, call sunrpc_cache_lookup_rcu passing a pointer to the cache_head in a sample item with the 'key' fields filled in. This will be passed to ->match to identify the target entry. If no entry is found, a new entry will be create, added to the cache, and marked as not containing valid data.

The item returned is typically passed to cache_check which will check if the data is valid, and may initiate an up-call to get fresh data. cache_check will return -ENOENT in the entry is negative or if an up call is needed but not possible, -EAGAIN if an upcall is pending, or 0 if the data is valid;

cache_check can be passed a "struct cache_req*" . This structure is typically embedded in the actual request and can be used to create a deferred copy of the request (struct cache_deferred_req). This is done when the found cache item is not uptodate, but the is reason to believe that userspace might provide information soon. When the cache item does become valid, the deferred copy of the request will be revisited (->revisit). It is expected that this method will reschedule the request for processing.

The value returned by sunrpc_cache_lookup_rcu can also be passed to sunrpc_cache_update to set the content for the item. A second item is passed which should hold the content. If the item found by _lookup has valid data, then it is discarded and a new item is created. This saves any user of an item from worrying about content changing while it is being inspected. If the item found by _lookup does not contain valid data, then the content is copied across and CACHE_VALID is set.

Populating a cache

Each cache has a name, and when the cache is registered, a directory with that name is created in /proc/net/rpc

This directory contains a file called 'channel' which is a channel for communicating between kernel and user for populating the cache. This directory may later contain other files of interacting with the cache.

The 'channel' works a bit like a datagram socket. Each 'write' is passed as a whole to the cache for parsing and interpretation. Each cache can treat the write requests differently, but it is expected that a message written will contain:

- a key
- an expiry time

- a content.

with the intention that an item in the cache with the give key should be create or updated to have the given content, and the expiry time should be set on that item.

Reading from a channel is a bit more interesting. When a cache lookup fails, or when it succeeds but finds an entry that may soon expire, a request is lodged for that cache item to be updated by user-space. These requests appear in the channel file.

Successive reads will return successive requests. If there are no more requests to return, read will return EOF, but a select or poll for read will block waiting for another request to be added.

Thus a user-space helper is likely to:

```
open the channel.  
  select for readable  
  read a request  
  write a response  
loop.
```

If it dies and needs to be restarted, any requests that have not been answered will still appear in the file and will be read by the new instance of the helper.

Each cache should define a “cache_parse” method which takes a message written from user-space and processes it. It should return an error (which propagates back to the write syscall) or 0.

Each cache should also define a “cache_request” method which takes a cache item and encodes a request into the buffer provided.

Note: If a cache has no active readers on the channel, and has had not active readers for more than 60 seconds, further requests will not be added to the channel but instead all lookups that do not find a valid entry will fail. This is partly for backward compatibility: The previous nfs exports table was deemed to be authoritative and a failed lookup meant a definite ‘no’ .

request/response format

While each cache is free to use its own format for requests and responses over channel, the following is recommended as appropriate and support routines are available to help: Each request or response record should be printable ASCII with precisely one newline character which should be at the end. Fields within the record should be separated by spaces, normally one. If spaces, newlines, or nul characters are needed in a field they much be quoted. two mechanisms are available:

- If a field begins ‘x’ then it must contain an even number of hex digits, and pairs of these digits provide the bytes in the field.
- otherwise a in the field must be followed by 3 octal digits which give the code for a byte. Other characters are treated as them selves. At the very least, space, newline, nul, and “ ” must be quoted in this way.

3.37.3 rpcsec_gss support for kernel RPC servers

This document gives references to the standards and protocols used to implement RPCGSS authentication in kernel RPC servers such as the NFS server and the NFS client's NFSv4.0 callback server. (But note that NFSv4.1 and higher don't require the client to act as a server for the purposes of authentication.)

RPCGSS is specified in a few IETF documents:

- RFC2203 v1: <http://tools.ietf.org/rfc/rfc2203.txt>
- RFC5403 v2: <http://tools.ietf.org/rfc/rfc5403.txt>

and there is a 3rd version being proposed:

- <http://tools.ietf.org/id/draft-williams-rpcsecgssv3.txt> (At draft n. 02 at the time of writing)

Background

The RPCGSS Authentication method describes a way to perform GSSAPI Authentication for NFS. Although GSSAPI is itself completely mechanism agnostic, in many cases only the KRB5 mechanism is supported by NFS implementations.

The Linux kernel, at the moment, supports only the KRB5 mechanism, and depends on GSSAPI extensions that are KRB5 specific.

GSSAPI is a complex library, and implementing it completely in kernel is unwarranted. However GSSAPI operations are fundamentally separable in 2 parts:

- initial context establishment
- integrity/privacy protection (signing and encrypting of individual packets)

The former is more complex and policy-independent, but less performance-sensitive. The latter is simpler and needs to be very fast.

Therefore, we perform per-packet integrity and privacy protection in the kernel, but leave the initial context establishment to userspace. We need upcalls to request userspace to perform context establishment.

NFS Server Legacy Upcall Mechanism

The classic upcall mechanism uses a custom text based upcall mechanism to talk to a custom daemon called `rpc.svcgssd` that is provided by the `nfs-utils` package.

This upcall mechanism has 2 limitations:

- A) It can handle tokens that are no bigger than 2KiB

In some Kerberos deployment GSSAPI tokens can be quite big, up and beyond 64KiB in size due to various authorization extensions attached to the Kerberos tickets, that needs to be sent through the GSS layer in order to perform context establishment.

- B) It does not properly handle creds where the user is member of more than a few thousand groups (the current hard limit in the kernel is 65K groups) due to limitation on the size of the buffer that can be sent back to the kernel (4KiB).

NFS Server New RPC Upcall Mechanism

The newer upcall mechanism uses RPC over a unix socket to a daemon called gss-proxy, implemented by a userspace program called Gssproxy.

The gss_proxy RPC protocol is currently documented [here](#).

This upcall mechanism uses the kernel rpc client and connects to the gssproxy userspace program over a regular unix socket. The gssproxy protocol does not suffer from the size limitations of the legacy protocol.

Negotiating Upcall Mechanisms

To provide backward compatibility, the kernel defaults to using the legacy mechanism. To switch to the new mechanism, gss-proxy must bind to `/var/run/gssproxy.sock` and then write “1” to `/proc/net/rpc/use-gss-proxy`. If gss-proxy dies, it must repeat both steps.

Once the upcall mechanism is chosen, it cannot be changed. To prevent locking into the legacy mechanisms, the above steps must be performed before starting `nfsd`. Whoever starts `nfsd` can guarantee this by reading from `/proc/net/rpc/use-gss-proxy` and checking that it contains a “1” -the read will block until gss-proxy has done its write to the file.

3.37.4 NFSv4.1 Server Implementation

Server support for minorversion 1 can be controlled using the `/proc/fs/nfsd/versions` control file. The string output returned by reading this file will contain either “+4.1” or “-4.1” correspondingly.

Currently, server support for minorversion 1 is enabled by default. It can be disabled at run time by writing the string “-4.1” to the `/proc/fs/nfsd/versions` control file. Note that to write this control file, the `nfsd` service must be taken down. You can use `rpc.nfsd` for this; see `rpc.nfsd(8)`.

(Warning: older servers will interpret “+4.1” and “-4.1” as “+4” and “-4” , respectively. Therefore, code meant to work on both new and old kernels must turn 4.1 on or off before turning support for version 4 on or off; `rpc.nfsd` does this correctly.)

The NFSv4 minorversion 1 (NFSv4.1) implementation in `nfsd` is based on RFC 5661.

From the many new features in NFSv4.1 the current implementation focuses on the mandatory-to-implement NFSv4.1 Sessions, providing “exactly once” semantics and better control and throttling of the resources allocated for each client.

The table below, taken from the NFSv4.1 document, lists the operations that are mandatory to implement (REQ), optional (OPT), and NFSv4.0 operations that are required not to implement (MNI) in minor version 1. The first column indicates the operations that are not supported yet by the linux server implementation.

The OPTIONAL features identified and their abbreviations are as follows:

- **pNFS** Parallel NFS

- **FDELG** File Delegations
- **DDELG** Directory Delegations

The following abbreviations indicate the linux server implementation status.

- **I** Implemented NFSv4.1 operations.
- **NS** Not Supported.
- **NS*** Unimplemented optional feature.

Operations

Implementation status	Operation	REQ,REC, OPT or NMI	Feature (REQ, REC or OPT)	Defin
	ACCESS	REQ		
I	BACKCHANNEL_CTL	REQ		
I	BIND_CONN_TO_SESSION	REQ		
	CLOSE	REQ		
	COMMIT	REQ		
	CREATE	REQ		
I	CREATE_SESSION	REQ		
NS*	DELEGPURGE	OPT	FDELG (REQ)	
	DELEGRETURN	OPT	FDELG, DDELG, pNFS (REQ)	
I	DESTROY_CLIENTID	REQ		
I	DESTROY_SESSION	REQ		
I	EXCHANGE_ID	REQ		
I	FREE_STATEID	REQ		
	GETATTR	REQ		
I	GETDEVICEINFO	OPT	pNFS (REQ)	
NS*	GETDEVICELIST	OPT	pNFS (OPT)	
	GETFH	REQ		
NS*	GET_DIR_DELEGATION	OPT	DDELG (REQ)	
I	LAYOUTCOMMIT	OPT	pNFS (REQ)	
I	LAYOUTGET	OPT	pNFS (REQ)	
I	LAYOUTRETURN	OPT	pNFS (REQ)	
	LINK	OPT		
	LOCK	REQ		
	LOCKT	REQ		
	LOCKU	REQ		
	LOOKUP	REQ		
	LOOKUPP	REQ		
	NVERIFY	REQ		
	OPEN	REQ		
NS*	OPENATTR	OPT		
	OPEN_CONFIRM	MNI		
	OPEN_DOWNGRADE	REQ		
	PUTFH	REQ		

Continued on next page

Table 1 - continued from previous page

Implementation status	Operation	REQ,REC, OPT or MNI	Feature (REQ, REC or OPT)	Defin
	PUTPUBFH		REQ	
	PUTROOTFH		REQ	
	READ		REQ	
	READDIR		REQ	
	READLINK		OPT	
	RECLAIM_COMPLETE		REQ	
	RELEASE_LOCKOWNER		MNI	
	REMOVE		REQ	
	RENAME		REQ	
	RENEW		MNI	
	RESTOREFH		REQ	
	SAVEFH		REQ	
	SECINFO		REQ	
I	SECINFO_NO_NAME		REC	pNFS files layout (REQ)
I	SEQUENCE		REQ	
	SETATTR		REQ	
	SETCLIENTID		MNI	
	SETCLIENTID_CONFIRM		MNI	
NS	SET_SSV		REQ	
I	TEST_STATEID		REQ	
	VERIFY		REQ	
NS*	WANT_DELEGATION		OPT	FDELG (OPT)
	WRITE		REQ	

Callback Operations

Implementa- tion status	Operation	REQ,REC, OPT or NMI	Feature (REQ, REC or OPT)	Defini- tion
	CB_GETATTR	OPT	FDELG (REQ)	Section 20.1
I	CB_LAYOUTRECALL	OPT	pNFS (REQ)	Section 20.3
NS*	CB_NOTIFY	OPT	DDELG (REQ)	Section 20.4
NS*	CB_NOTIFY_DEVICEID	OPT	pNFS (OPT)	Section 20.12
NS*	CB_NOTIFY_LOCK	OPT		Section 20.11
NS*	CB_PUSH_DELEG	OPT	FDELG (OPT)	Section 20.5
	CB_RECALL	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.2
NS*	CB_RECALL_ANY	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.6
NS	CB_RECALL_SLOT	REQ		Section 20.8
NS*	CB_RECALLABLE_OBJ_AVAIL	OPT	DDELG, pNFS (REQ)	Section 20.7
I	CB_SEQUENCE	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.9
NS*	CB_WANTS_CANCELLED	OPT	FDELG, DDELG, pNFS (REQ)	Section 20.10

Implementation notes:

SSV: The spec claims this is mandatory, but we don't actually know of any implementations, so we're ignoring it for now. The server returns NFS4ERR_ENCR_ALG_UNSUPP on EXCHANGE_ID, which should be future-proof.

GSS on the backchannel: Again, theoretically required but not widely implemented (in particular, the current Linux client doesn't request it). We return NFS4ERR_ENCR_ALG_UNSUPP on CREATE_SESSION.

DELEGPURGE: mandatory only for servers that support

CLAIM_DELEGATE_PREV and/or CLAIM_DELEG_PREV_FH (which allows clients to keep delegations that persist across client reboots). Thus we need not implement this for now.

EXCHANGE_ID: implementation ids are ignored

CREATE_SESSION: backchannel attributes are ignored

SEQUENCE: no support for dynamic slot table renegotiation (optional)

Nonstandard compound limitations: No support for a sessions fore channel RPC compound that requires both a ca_maxrequestsize request and a ca_maxresponsesize reply, so we may fail to live up to the promise we made in CREATE_SESSION fore channel negotiation.

See also http://wiki.linux-nfs.org/wiki/index.php/Server_4.0_and_4.1_issues.

3.37.5 Kernel NFS Server Statistics

Authors Greg Banks <gnb@sgi.com> - 26 Mar 2009

This document describes the format and semantics of the statistics which the kernel NFS server makes available to userspace. These statistics are available in several text form pseudo files, each of which is described separately below.

In most cases you don't need to know these formats, as the nfsstat(8) program from the nfs-utils distribution provides a helpful command-line interface for extracting and printing them.

All the files described here are formatted as a sequence of text lines, separated by newline 'n' characters. Lines beginning with a hash '#' character are comments intended for humans and should be ignored by parsing routines. All other lines contain a sequence of fields separated by whitespace.

/proc/fs/nfsd/pool_stats

This file is available in kernels from 2.6.30 onwards, if the /proc/fs/nfsd filesystem is mounted (it almost always should be).

The first line is a comment which describes the fields present in all the other lines. The other lines present the following data as a sequence of unsigned decimal numeric fields. One line is shown for each NFS thread pool.

All counters are 64 bits wide and wrap naturally. There is no way to zero these counters, instead applications should do their own rate conversion.

pool The id number of the NFS thread pool to which this line applies. This number does not change.

Thread pool ids are a contiguous set of small integers starting at zero. The maximum value depends on the thread pool mode, but currently cannot be larger than the number of CPUs in the system. Note that in the default case there will be a single thread pool which contains all the nfsd threads and all the CPUs in the system, and thus this file will have a single line with a pool id of "0" .

packets-arrived Counts how many NFS packets have arrived. More precisely, this is the number of times that the network stack has notified the sunrpc server layer that new data may be available on a transport (e.g. an NFS or UDP socket or an NFS/RDMA endpoint).

Depending on the NFS workload patterns and various network stack effects (such as Large Receive Offload) which can combine packets on the wire, this may be either more or less than the number of NFS calls received (which statistic is available elsewhere). However this is a more accurate and less workload-dependent measure of how much CPU load is being placed on the sunrpc server layer due to NFS network traffic.

sockets-enqueued Counts how many times an NFS transport is enqueued to wait for an nfsd thread to service it, i.e. no nfsd thread was considered available.

The circumstance this statistic tracks indicates that there was NFS network-facing work to be done but it couldn't be done immediately, thus introducing a small delay in servicing NFS calls. The ideal rate of change for this counter is zero; significantly non-zero values may indicate a performance limitation.

This can happen because there are too few nfsd threads in the thread pool for the NFS workload (the workload is thread-limited), in which case configuring more nfsd threads will probably improve the performance of the NFS workload.

threads-woken Counts how many times an idle nfsd thread is woken to try to receive some data from an NFS transport.

This statistic tracks the circumstance where incoming network-facing NFS work is being handled quickly, which is a good thing. The ideal rate of change for this counter will be close to but less than the rate of change of the packets-arrived counter.

threads-timedout Counts how many times an nfsd thread triggered an idle timeout, i.e. was not woken to handle any incoming network packets for some time.

This statistic counts a circumstance where there are more nfsd threads configured than can be used by the NFS workload. This is a clue that the number of nfsd threads can be reduced without affecting performance. Unfortunately, it's only a clue and not a strong indication, for a couple of reasons:

- Currently the rate at which the counter is incremented is quite slow; the idle timeout is 60 minutes. Unless the NFS workload remains constant for hours at a time, this counter is unlikely to be providing information that is still useful.
- It is usually a wise policy to provide some slack, i.e. configure a few more nfsds than are currently needed, to allow for future spikes in load.

Note that incoming packets on NFS transports will be dealt with in one of three ways. An nfsd thread can be woken (threads-woken counts this case), or the transport can be enqueued for later attention (sockets-enqueued counts this case), or the packet can be temporarily deferred because the transport is currently being used by an nfsd thread. This last case is not very interesting and is not explicitly counted, but can be inferred from the other counters thus:

`packets-deferred = packets-arrived - (sockets-enqueued + threads-woken)`

More

Descriptions of the other statistics file should go here.

3.38 The Linux NTFS filesystem driver

3.38.1 Overview

Linux-NTFS comes with a number of user-space programs known as ntfsprogs. These include mkntfs, a full-featured ntfs filesystem format utility, ntfsundelete used for recovering files that were unintentionally deleted from an NTFS volume and ntfsresize which is used to resize an NTFS partition. See the web site for more information.

To mount an NTFS 1.2/3.x (Windows NT4/2000/XP/2003) volume, use the file system type 'ntfs'. The driver currently supports read-only mode (with no fault-tolerance, encryption or journalling) and very limited, but safe, write support.

For fault tolerance and raid support (i.e. volume and stripe sets), you can use the kernel's Software RAID / MD driver. See section "Using Software RAID with NTFS" for details.

3.38.2 Web site

There is plenty of additional information on the linux-ntfs web site at <http://www.linux-ntfs.org/>

The web site has a lot of additional information, such as a comprehensive FAQ, documentation on the NTFS on-disk format, information on the Linux-NTFS userspace utilities, etc.

3.38.3 Features

- This is a complete rewrite of the NTFS driver that used to be in the 2.4 and earlier kernels. This new driver implements NTFS read support and is functionally equivalent to the old ntfs driver and it also implements limited write support. The biggest limitation at present is that files/directories cannot be created or deleted. See below for the list of write features that are so far supported. Another limitation is that writing to compressed files is not implemented at all. Also, neither read nor write access to encrypted files is so far implemented.
- The new driver has full support for sparse files on NTFS 3.x volumes which the old driver isn't happy with.
- The new driver supports execution of binaries due to mmap() now being supported.

- The new driver supports loopback mounting of files on NTFS which is used by some Linux distributions to enable the user to run Linux from an NTFS partition by creating a large file while in Windows and then loopback mounting the file while in Linux and creating a Linux filesystem on it that is used to install Linux on it.
- A comparison of the two drivers using:

```
time find . -type f -exec md5sum "{}" \;
```

run three times in sequence with each driver (after a reboot) on a 1.4GiB NTFS partition, showed the new driver to be 20% faster in total time elapsed (from 9:43 minutes on average down to 7:53). The time spent in user space was unchanged but the time spent in the kernel was decreased by a factor of 2.5 (from 85 CPU seconds down to 33).

- The driver does not support short file names in general. For backwards compatibility, we implement access to files using their short file names if they exist. The driver will not create short file names however, and a rename will discard any existing short file name.
- The new driver supports exporting of mounted NTFS volumes via NFS.
- The new driver supports async io (aio).
- The new driver supports fsync(2), fdatasync(2), and msync(2).
- The new driver supports readv(2) and writev(2).
- The new driver supports access time updates (including mtime and ctime).
- The new driver supports truncate(2) and open(2) with O_TRUNC. But at present only very limited support for highly fragmented files, i.e. ones which have their data attribute split across multiple extents, is included. Another limitation is that at present truncate(2) will never create sparse files, since to mark a file sparse we need to modify the directory entry for the file and we do not implement directory modifications yet.
- The new driver supports write(2) which can both overwrite existing data and extend the file size so that you can write beyond the existing data. Also, writing into sparse regions is supported and the holes are filled in with clusters. But at present only limited support for highly fragmented files, i.e. ones which have their data attribute split across multiple extents, is included. Another limitation is that write(2) will never create sparse files, since to mark a file sparse we need to modify the directory entry for the file and we do not implement directory modifications yet.

3.38.4 Supported mount options

In addition to the generic mount options described by the manual page for the mount command (man 8 mount, also see man 5 fstab), the NTFS driver supports the following mount options:

<code>iocharset=name</code>	Deprecated option. Still supported but please use <code>nls=name</code> in the future. See description for <code>nls=name</code> .
<code>nls=name</code>	Character set to use when returning file names. Unlike VFAT, NTFS suppresses names that contain unconvertible characters. Note that most character sets contain insufficient characters to represent all possible Unicode characters that can exist on NTFS. To be sure you are not missing any files, you are advised to use <code>nls=utf8</code> which is capable of representing all Unicode characters.
<code>utf8=<bool></code>	Option no longer supported. Currently mapped to <code>nls=utf8</code> but please use <code>nls=utf8</code> in the future and make sure <code>utf8</code> is compiled either as module or into the kernel. See description for <code>nls=name</code> .
<code>uid=</code>	
<code>gid=</code>	
<code>umask=</code>	Provide default owner, group, and access mode mask. These options work as documented in <code>mount(8)</code> . By default, the files/directories are owned by root and he/she has read and write permissions, as well as browse permission for directories. No one else has any access permissions. I.e. the mode on all files is by default <code>rw---</code> and for directories <code>rwx---</code> , a consequence of the default <code>fmask=0177</code> and <code>dmask=0077</code> . Using a <code>umask</code> of zero will grant all permissions to everyone, i.e. all files and directories will have mode <code>rw-rwxrwx</code> .
<code>fmask=</code>	
<code>dmask=</code>	Instead of specifying <code>umask</code> which applies both to files and directories, <code>fmask</code> applies only to files and <code>dmask</code> only to directories.
<code>sloppy=<BOOL></code>	If <code>sloppy</code> is specified, ignore unknown mount options. Otherwise the default behaviour is to abort mount if any unknown options are found.
<code>show_sys_files=<BOOL></code>	If <code>show_sys_files</code> is specified, show the system files in directory listings. Otherwise the default behaviour is to hide the system files. Note that even when <code>show_sys_files</code> is specified, “\$MFT” will not be visible due to bugs/mis-features

3.38.5 Known bugs and (mis-)features

- The link count on each directory inode entry is set to 1, due to Linux not supporting directory hard links. This may well confuse some user space applications, since the directory names will have the same inode numbers. This also speeds up `ntfs_read_inode()` immensely. And we haven't found any problems with this approach so far. If you find a problem with this, please let us know.

Please send bug reports/comments/feedback/abuse to the Linux-NTFS development list at sourceforge: linux-ntfs-dev@lists.sourceforge.net

3.38.6 Using NTFS volume and stripe sets

For support of volume and stripe sets, you can either use the kernel's Device-Mapper driver or the kernel's Software RAID / MD driver. The former is the recommended one to use for linear raid. But the latter is required for raid level 5. For striping and mirroring, either driver should work fine.

The Device-Mapper driver

You will need to create a table of the components of the volume/stripe set and how they fit together and load this into the kernel using the `dmsetup` utility (see `man 8 dmsetup`).

Linear volume sets, i.e. linear raid, has been tested and works fine. Even though untested, there is no reason why stripe sets, i.e. raid level 0, and mirrors, i.e. raid level 1 should not work, too. Stripes with parity, i.e. raid level 5, unfortunately cannot work yet because the current version of the Device-Mapper driver does not support raid level 5. You may be able to use the Software RAID / MD driver for raid level 5, see the next section for details.

To create the table describing your volume you will need to know each of its components and their sizes in sectors, i.e. multiples of 512-byte blocks.

For NT4 fault tolerant volumes you can obtain the sizes using `fdisk`. So for example if one of your partitions is `/dev/hda2` you would do:

```
$ fdisk -ul /dev/hda
Disk /dev/hda: 81.9 GB, 81964302336 bytes
255 heads, 63 sectors/track, 9964 cylinders, total 160086528 sectors
Units = sectors of 1 * 512 = 512 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1    *           63     4209029     2104483+  83  Linux
/dev/hda2             4209030    37768814    16779892+  86  NTFS
/dev/hda3          37768815    46170809     4200997+  83  Linux
```

And you would know that `/dev/hda2` has a size of `37768814 - 4209030 + 1 = 33559785` sectors.

For Win2k and later dynamic disks, you can for example use the `ldminfo` utility which is part of the Linux LDM tools (the latest version at the time of writing is `linux-ldm-0.0.8.tar.bz2`). You can download it from:

<http://www.linux-ntfs.org/>

Simply extract the downloaded archive (`tar xvjf linux-ldm-0.0.8.tar.bz2`), go into it (`cd linux-ldm-0.0.8`) and change to the test directory (`cd test`). You will find the precompiled (i386) `ldminfo` utility there. NOTE: You will not be able to compile this yourself easily so use the binary version!

Then you would use `ldminfo` in dump mode to obtain the necessary information:

```
$ ./ldminfo --dump /dev/hda
```

This would dump the LDM database found on `/dev/hda` which describes all of your dynamic disks and all the volumes on them. At the bottom you will see the `VOLUME DEFINITIONS` section which is all you really need. You may need to look further above to determine which of the disks in the volume definitions is which device in Linux. Hint: Run `ldminfo` on each of your dynamic disks and look at the Disk Id close to the top of the output for each (the `PRIVATE HEADER` section). You can then find these Disk Ids in the `VBLK DATABASE` section in the `<Disk>` components where you will get the LDM Name for the disk that is found in the `VOLUME DEFINITIONS` section.

Note you will also need to enable the LDM driver in the Linux kernel. If your distribution did not enable it, you will need to recompile the kernel with it enabled. This will create the LDM partitions on each device at boot time. You would then use those devices (for `/dev/hda` they would be `/dev/hda1, 2, 3`, etc) in the Device-Mapper table.

You can also bypass using the LDM driver by using the main device (e.g. `/dev/hda`) and then using the offsets of the LDM partitions into this device as the “Start sector of device” when creating the table. Once again `ldminfo` would give you the correct information to do this.

Assuming you know all your devices and their sizes things are easy.

For a linear raid the table would look like this (note all values are in 512-byte sectors):

#	Offset into →sector	Size of this	Raid type	Device	Start _u
# volume	device				of device
0	1028161	linear		/dev/hda1	0
	1028161	3903762	linear	/dev/hdb2	0
	4931923	2103211	linear	/dev/hdc1	0

For a striped volume, i.e. raid level 0, you will need to know the chunk size you used when creating the volume. Windows uses 64kiB as the default, so it will probably be this unless you changes the defaults when creating the array.

For a raid level 0 the table would look like this (note all values are in 512-byte sectors):

```
# Offset  Size      Raid   Number  Chunk  1st      Start  2nd
↳ Start
# in of the type of size Device in Device
↳ in
# volume volume stripes device
↳ device
0      2056320 striped 2       128    /dev/hda1 0      /dev/hdb1 0
```

If there are more than two devices, just add each of them to the end of the line.

Finally, for a mirrored volume, i.e. raid level 1, the table would look like this (note all values are in 512-byte sectors):

```
# ofs Size  Raid  Log  Number Region Should Number Source  Start Target
↳ Start
# in of the type type of log size sync? of Device in Device
↳ in
# vol volume params mirrors Device
↳ Device
0      2056320 mirror core 2 16    nosync 2 /dev/hda1 0 /dev/hdb1 0
```

If you are mirroring to multiple devices you can specify further targets at the end of the line.

Note the “Should sync?” parameter “nosync” means that the two mirrors are already in sync which will be the case on a clean shutdown of Windows. If the mirrors are not clean, you can specify the “sync” option instead of “nosync” and the Device-Mapper driver will then copy the entirety of the “Source Device” to the “Target Device” or if you specified multiple target devices to all of them.

Once you have your table, save it in a file somewhere (e.g. /etc/ntfsvolume1), and hand it over to dmsetup to work with, like so:

```
$ dmsetup create myvolume1 /etc/ntfsvolume1
```

You can obviously replace “myvolume1” with whatever name you like.

If it all worked, you will now have the device /dev/device-mapper/myvolume1 which you can then just use as an argument to the mount command as usual to mount the ntfs volume. For example:

```
$ mount -t ntfs -o ro /dev/device-mapper/myvolume1 /mnt/myvol1
```

(You need to create the directory /mnt/myvol1 first and of course you can use anything you like instead of /mnt/myvol1 as long as it is an existing directory.)

It is advisable to do the mount read-only to see if the volume has been setup correctly to avoid the possibility of causing damage to the data on the ntfs volume.

The Software RAID / MD driver

An alternative to using the Device-Mapper driver is to use the kernel's Software RAID / MD driver. For which you need to set up your `/etc/raidtab` appropriately (see `man 5 raidtab`).

Linear volume sets, i.e. linear raid, as well as stripe sets, i.e. raid level 0, have been tested and work fine (though see section “Limitations when using the MD driver with NTFS volumes” especially if you want to use linear raid). Even though untested, there is no reason why mirrors, i.e. raid level 1, and stripes with parity, i.e. raid level 5, should not work, too.

You have to use the “persistent-superblock 0” option for each raid-disk in the NTFS volume/stripe you are configuring in `/etc/raidtab` as the persistent superblock used by the MD driver would damage the NTFS volume.

Windows by default uses a stripe chunk size of 64k, so you probably want the “chunk-size 64k” option for each raid-disk, too.

For example, if you have a stripe set consisting of two partitions `/dev/hda5` and `/dev/hdb1` your `/etc/raidtab` would look like this:

```
raiddev /dev/md0
raid-level 0
nr-raid-disks 2
nr-spare-disks 0
persistent-superblock 0
chunk-size 64k
device /dev/hda5
raid-disk 0
device /dev/hdb1
raid-disk 1
```

For linear raid, just change the `raid-level` above to “`raid-level linear`”, for mirrors, change it to “`raid-level 1`”, and for stripe sets with parity, change it to “`raid-level 5`”.

Note for stripe sets with parity you will also need to tell the MD driver which parity algorithm to use by specifying the option “`parity-algorithm which`”, where you need to replace “`which`” with the name of the algorithm to use (see `man 5 raidtab` for available algorithms) and you will have to try the different available algorithms until you find one that works. Make sure you are working read-only when playing with this as you may damage your data otherwise. If you find which algorithm works please let us know (email the linux-ntfs developers list linux-ntfs-dev@lists.sourceforge.net or drop in on IRC in channel `#ntfs` on the `irc.freenode.net` network) so we can update this documentation.

Once the `raidtab` is setup, run for example `raid0run -a` to start all devices or `raid0run /dev/md0` to start a particular md device, in this case `/dev/md0`.

Then just use the `mount` command as usual to mount the ntfs volume using for example:

```
mount -t ntfs -o ro /dev/md0 /mnt/myntfsvolume
```

It is advisable to do the `mount` read-only to see if the md volume has been setup correctly to avoid the possibility of causing damage to the data on the ntfs volume.

Limitations when using the Software RAID / MD driver

Using the md driver will not work properly if any of your NTFS partitions have an odd number of sectors. This is especially important for linear raid as all data after the first partition with an odd number of sectors will be offset by one or more sectors so if you mount such a partition with write support you will cause massive damage to the data on the volume which will only become apparent when you try to use the volume again under Windows.

So when using linear raid, make sure that all your partitions have an even number of sectors BEFORE attempting to use it. You have been warned!

Even better is to simply use the Device-Mapper for linear raid and then you do not have this problem with odd numbers of sectors.

3.39 OCFS2 filesystem

OCFS2 is a general purpose extent based shared disk cluster file system with many similarities to ext3. It supports 64 bit inode numbers, and has automatically extending metadata groups which may also make it attractive for non-clustered use.

You'll want to install the ocfs2-tools package in order to at least get "mount.ocfs2" and "ocfs2_hb_ctl" .

Project web page: <http://ocfs2.wiki.kernel.org> Tools git tree: <https://github.com/markfasheh/ocfs2-tools> OCFS2 mailing lists: <http://oss.oracle.com/projects/ocfs2/mailman/>

All code copyright 2005 Oracle except when otherwise noted.

3.39.1 Credits

Lots of code taken from ext3 and other projects.

Authors in alphabetical order:

- Joel Becker <joel.becker@oracle.com>
- Zach Brown <zach.brown@oracle.com>
- Mark Fasheh <mfasheh@suse.com>
- Kurt Hackel <kurt.hackel@oracle.com>
- Tao Ma <tao.ma@oracle.com>
- Sunil Mushran <sunil.mushran@oracle.com>
- Manish Singh <manish.singh@oracle.com>
- Tiger Yang <tiger.yang@oracle.com>

3.39.2 Caveats

Features which OCFS2 does not support yet:

- Directory change notification (F_NOTIFY)
- Distributed Caching (F_SETLEASE/F_GETLEASE/break_lease)

3.39.3 Mount options

OCFS2 supports the following mount options:

(*) == default

barrier=1	This enables/disables barriers. barrier=0 disables it, barrier=1 enables it.
errorro(*)	Remount the filesystem read-only on an error.
errors=panic	Panic and halt the machine if an error occurs.
intr(*)	Allow signals to interrupt cluster operations.
nointr	Do not allow signals to interrupt cluster operations.
noatime	Do not update access time.
relatime(*)	Update atime if the previous atime is older than mtime or ctime
strictatime	Always update atime, but the minimum update interval is specified by atime_quantum.
atime=off	OCFS2 will not update atime unless this number of seconds has passed since the last update. Set to zero to always update atime. This option need work with strictatime.
data=ordered	All data are forced directly out to the main file system prior to its metadata being committed to the journal.
data=writeback	Data ordering is not preserved, data may be written into the main file system after its metadata has been committed to the journal.
prefered_slot=0(*)	During mount, try to use this filesystem slot first. If it is in use by another slot, the first empty one found will be chosen. Invalid values will be ignored.
commit=5sec(*)	Ocfs2 can be told to sync all its data and metadata every ‘nrsec’ seconds. The default value is 5 seconds. This means that if you lose your power, you will lose as much as the latest 5 seconds of work (your filesystem will not be damaged though, thanks to the journaling). This default value (or any low value) will hurt performance, but it’s good for data-safety. Setting it to 0 will have the same effect as leaving it at the default (5 seconds). Setting it to very large values will improve performance.
localalloc=8(*)	Allows custom localalloc size in MB. If the value is too large, the fs will silently revert it to the default.
localflocks	This disables cluster aware flock.
inode64	Indicates that Ocfs2 is allowed to create inodes at any location in the filesystem, including those which will result in inode numbers occupying more than 32 bits of significance.
user_xattr(*)	Enables Extended User Attributes.
nouser_xattr	Disables Extended User Attributes.
acl	Enables POSIX Access Control Lists support.
noacl(*)	Disables POSIX Access Control Lists support.
resvbehav(*)	Set how aggressive allocation reservations will be. Valid values are between 0 (reservations off) to 8 (maximum space for reservations).
dir_resvbehav(*)	By default, directory reservations will scale with file reservations - users should rarely need to change this value. If allocation reservations are turned off, this option will have no effect.
co- b- l- o- c- k- e- t- t- e- d	Disallow concurrent O_DIRECT writes, cluster inode lock will be taken to

3.40 OCFS2 file system - online file check

This document will describe OCFS2 online file check feature.

3.40.1 Introduction

OCFS2 is often used in high-availability systems. However, OCFS2 usually converts the filesystem to read-only when encounters an error. This may not be necessary, since turning the filesystem read-only would affect other running processes as well, decreasing availability. Then, a mount option (`errors=continue`) is introduced, which would return the `-EIO` errno to the calling process and terminate further processing so that the filesystem is not corrupted further. The filesystem is not converted to read-only, and the problematic file's inode number is reported in the kernel log. The user can try to check/fix this file via online filecheck feature.

3.40.2 Scope

This effort is to check/fix small issues which may hinder day-to-day operations of a cluster filesystem by turning the filesystem read-only. The scope of checking/fixing is at the file level, initially for regular files and eventually to all files (including system files) of the filesystem.

In case of directory to file links is incorrect, the directory inode is reported as erroneous.

This feature is not suited for extravagant checks which involve dependency of other components of the filesystem, such as but not limited to, checking if the bits for file blocks in the allocation has been set. In case of such an error, the offline fsck should/would be recommended.

Finally, such an operation/feature should not be automated lest the filesystem may end up with more damage than before the repair attempt. So, this has to be performed using user interaction and consent.

3.40.3 User interface

When there are errors in the OCFS2 filesystem, they are usually accompanied by the inode number which caused the error. This inode number would be the input to check/fix the file.

There is a `sysfs` directory for each OCFS2 file system mounting:

```
/sys/fs/ocfs2/<devname>/filecheck
```

Here, `<devname>` indicates the name of OCFS2 volume device which has been already mounted. The file above would accept inode numbers. This could be used to communicate with kernel space, tell which file(inode number) will be checked or fixed. Currently, three operations are supported, which includes checking inode, fixing inode and setting the size of result record history.

1. If you want to know what error exactly happened to `<inode>` before fixing, do:

```
# echo "<inode>" > /sys/fs/ocfs2/<devname>/filecheck/check
# cat /sys/fs/ocfs2/<devname>/filecheck/check
```

The output is like this:

INO	DONE	ERROR	GENERATION
39502		1	

<INO> lists the inode numbers.
 <DONE> indicates whether the operation has been finished.
 <ERROR> says what kind of errors was found. For the detailed error numbers, please refer to the file linux/fs/ocfs2/filecheck.h.

2. If you determine to fix this inode, do:

```
# echo "<inode>" > /sys/fs/ocfs2/<devname>/filecheck/fix
# cat /sys/fs/ocfs2/<devname>/filecheck/fix
```

The output is like this::

INO	DONE	ERROR	SUCCESS
39502		1	

This time, the <ERROR> column indicates whether this fix is successful or not.

3. The record cache is used to store the history of check/fix results. It's default size is 10, and can be adjust between the range of 10 ~ 100. You can adjust the size like this:

```
# echo "<size>" > /sys/fs/ocfs2/<devname>/filecheck/set
```

3.40.4 Fixing stuff

On receiving the inode, the filesystem would read the inode and the file meta-data. In case of errors, the filesystem would fix the errors and report the problems it fixed in the kernel log. As a precautionary measure, the inode must first be checked for errors before performing a final fix.

The inode and the result history will be maintained temporarily in a small linked list buffer which would contain the last (N) inodes fixed/checked, the detailed errors which were fixed/checked are printed in the kernel log.

3.41 Optimized MPEG Filesystem (OMFS)

3.41.1 Overview

OMFS is a filesystem created by SonicBlue for use in the ReplayTV DVR and Rio Karma MP3 player. The filesystem is extent-based, utilizing block sizes from 2k to 8k, with hash-based directories. This filesystem driver may be used to read and write disks from these devices.

Note, it is not recommended that this FS be used in place of a general filesystem for your own streaming media device. Native Linux filesystems will likely perform better.

More information is available at:

<http://linux-karma.sf.net/>

Various utilities, including mkomfs and omfsck, are included with omfsprogs, available at:

<http://bobcopeland.com/karma/>

Instructions are included in its README.

3.41.2 Options

OMFS supports the following mount-time options:

uid=n	make all files owned by specified user
gid=n	make all files owned by specified group
umask=xxx	set permission umask to xxx
fmask=xxx	set umask to xxx for files
dmask=xxx	set umask to xxx for directories

3.41.3 Disk format

OMFS discriminates between “sysblocks” and normal data blocks. The sysblock group consists of super block information, file metadata, directory structures, and extents. Each sysblock has a header containing CRCs of the entire sysblock, and may be mirrored in successive blocks on the disk. A sysblock may have a smaller size than a data block, but since they are both addressed by the same 64-bit block number, any remaining space in the smaller sysblock is unused.

Sysblock header information:

```
struct omfs_header {
    __be64 h_self;                /* FS block where this is located_
→*/
    __be32 h_body_size;          /* size of useful data after_
→header */
    __be16 h_crc;                /* crc-ccitt of body_size bytes */
    char h_fill1[2];
    u8 h_version;                /* version, always 1 */
    char h_type;                 /* OMFS_INODE_X */
    u8 h_magic;                  /* OMFS_IMAGIC */
    u8 h_check_xor;              /* XOR of header bytes before this_
→*/
    __be32 h_fill2;
};
```

Files and directories are both represented by omfs_inode:

```

struct omfs_inode {
    struct omfs_header i_head;        /* header */
    __be64 i_parent;                 /* parent containing this inode */
    __be64 i_sibling;                /* next inode in hash bucket */
    __be64 i_ctime;                   /* ctime, in milliseconds */
    char i_fill1[35];
    char i_type;                       /* OMFS_[DIR,FILE] */
    __be32 i_fill2;
    char i_fill3[64];
    char i_name[OMFS_NAMELEN];        /* filename */
    __be64 i_size;                     /* size of file, in bytes */
};

```

Directories in OMFS are implemented as a large hash table. Filenames are hashed then prepended into the bucket list beginning at `OMFS_DIR_START`. Lookup requires hashing the filename, then seeking across `i_sibling` pointers until a match is found on `i_name`. Empty buckets are represented by block pointers with all-1s (~0).

A file is an `omfs_inode` structure followed by an extent table beginning at `OMFS_EXTENT_START`:

```

struct omfs_extent_entry {
    __be64 e_cluster;                 /* start location of a set of
↳ blocks */
    __be64 e_blocks;                  /* number of blocks after e_
↳ cluster */
};

struct omfs_extent {
    __be64 e_next;                     /* next extent table location */
    __be32 e_extent_count;             /* total # extents in this table */
    __be32 e_fill;
    struct omfs_extent_entry e_entry;  /* start of extent entries
↳ */
};

```

Each extent holds the block offset followed by number of blocks allocated to the extent. The final extent in each table is a terminator with `e_cluster` being ~0 and `e_blocks` being ones' -complement of the total number of blocks in the table.

If this table overflows, a continuation inode is written and pointed to by `e_next`. These have a header but lack the rest of the inode structure.

3.42 ORANGEFS

OrangeFS is an LGPL userspace scale-out parallel storage system. It is ideal for large storage problems faced by HPC, BigData, Streaming Video, Genomics, Bioinformatics.

Orangefs, originally called PVFS, was first developed in 1993 by Walt Ligon and Eric Blumer as a parallel file system for Parallel Virtual Machine (PVM) as part of a NASA grant to study the I/O patterns of parallel programs.

Orangefs features include:

- Distributes file data among multiple file servers
- Supports simultaneous access by multiple clients
- Stores file data and metadata on servers using local file system and access methods
- Userspace implementation is easy to install and maintain
- Direct MPI support
- Stateless

3.42.1 Mailing List Archives

http://lists.orangeefs.org/pipermail/devel_lists.orangeefs.org/

3.42.2 Mailing List Submissions

devel@lists.orangeefs.org

3.42.3 Documentation

<http://www.orangeefs.org/documentation/>

3.42.4 Running ORANGEFS On a Single Server

OrangeFS is usually run in large installations with multiple servers and clients, but a complete filesystem can be run on a single machine for development and testing.

On Fedora, install `orangeefs` and `orangeefs-server`:

```
dnf -y install orangeefs orangeefs-server
```

There is an example server configuration file in `/etc/orangeefs/orangeefs.conf`. Change `localhost` to your hostname if necessary.

To generate a filesystem to run `xfstests` against, see below.

There is an example client configuration file in `/etc/pvfs2tab`. It is a single line. Uncomment it and change the hostname if necessary. This controls clients which use `libpvfs2`. This does not control the `pvfs2-client-core`.

Create the filesystem:

```
pvfs2-server -f /etc/orangeefs/orangeefs.conf
```

Start the server:

```
systemctl start orangeefs-server
```

Test the server:

```
pvfs2-ping -m /pvfsmnt
```

Start the client. The module must be compiled in or loaded before this point:

```
systemctl start orangefs-client
```

Mount the filesystem:

```
mount -t pvfs2 tcp://localhost:3334/orangefs /pvfsmnt
```

3.42.5 Userspace Filesystem Source

<http://www.orangefs.org/download>

Orangefs versions prior to 2.9.3 would not be compatible with the upstream version of the kernel client.

3.42.6 Building ORANGEFS on a Single Server

Where OrangeFS cannot be installed from distribution packages, it may be built from source.

You can omit `-prefix` if you don't care that things are sprinkled around in `/usr/local`. As of version 2.9.6, OrangeFS uses Berkeley DB by default, we will probably be changing the default to LMDB soon.

```
./configure --prefix=/opt/ofs --with-db-backend=lmdb --disable-usrint  
make  
make install
```

Create an `orangefs` config file by running `pvfs2-genconfig` and specifying a target config file. `Pvfs2-genconfig` will prompt you through. Generally it works fine to take the defaults, but you should use your server's hostname, rather than "localhost" when it comes to that question:

```
/opt/ofs/bin/pvfs2-genconfig /etc/pvfs2.conf
```

Create an `/etc/pvfs2tab` file (localhost is fine):

```
echo tcp://localhost:3334/orangefs /pvfsmnt pvfs2 defaults,noauto 0 0 > \  
/etc/pvfs2tab
```

Create the mount point you specified in the tab file if needed:

```
mkdir /pvfsmnt
```

Bootstrap the server:

```
/opt/ofs/sbin/pvfs2-server -f /etc/pvfs2.conf
```

Start the server:

```
/opt/ofs/sbin/pvfs2-server /etc/pvfs2.conf
```

Now the server should be running. Pvfs2-ls is a simple test to verify that the server is running:

```
/opt/ofs/bin/pvfs2-ls /pvfsmnt
```

If stuff seems to be working, load the kernel module and turn on the client core:

```
/opt/ofs/sbin/pvfs2-client -p /opt/ofs/sbin/pvfs2-client-core
```

Mount your filesystem:

```
mount -t pvfs2 tcp://`hostname`:3334/orangefs /pvfsmnt
```

3.42.7 Running xfstests

It is useful to use a scratch filesystem with xfstests. This can be done with only one server.

Make a second copy of the FileSystem section in the server configuration file, which is `/etc/orangefs/orangefs.conf`. Change the Name to `scratch`. Change the ID to something other than the ID of the first FileSystem section (2 is usually a good choice).

Then there are two FileSystem sections: `orangefs` and `scratch`.

This change should be made before creating the filesystem.

```
pvfs2-server -f /etc/orangefs/orangefs.conf
```

To run xfstests, create `/etc/xfstests.conf`:

```
TEST_DIR=/orangefs
TEST_DEV=tcp://localhost:3334/orangefs
SCRATCH_MNT=/scratch
SCRATCH_DEV=tcp://localhost:3334/scratch
```

Then xfstests can be run:

```
./check -pvfs2
```

3.42.8 Options

The following mount options are accepted:

- acl** Allow the use of Access Control Lists on files and directories.
- intr** Some operations between the kernel client and the user space filesystem can be interruptible, such as changes in debug levels and the setting of tunable parameters.
- local_lock** Enable posix locking from the perspective of “this” kernel. The default file_operations lock action is to return ENOSYS. Posix

locking kicks in if the filesystem is mounted with `-o local_lock`. Distributed locking is being worked on for the future.

3.42.9 Debugging

If you want the debug (GOSSIP) statements in a particular source file (inode.c for example) go to syslog:

```
echo inode > /sys/kernel/debug/orangefs/kernel-debug
```

No debugging (the default):

```
echo none > /sys/kernel/debug/orangefs/kernel-debug
```

Debugging from several source files:

```
echo inode,dir > /sys/kernel/debug/orangefs/kernel-debug
```

All debugging:

```
echo all > /sys/kernel/debug/orangefs/kernel-debug
```

Get a list of all debugging keywords:

```
cat /sys/kernel/debug/orangefs/debug-help
```

3.42.10 Protocol between Kernel Module and Userspace

Orangefs is a user space filesystem and an associated kernel module. We'll just refer to the user space part of Orangefs as "userspace" from here on out. Orangefs descends from PVFS, and userspace code still uses PVFS for function and variable names. Userspace typedefs many of the important structures. Function and variable names in the kernel module have been transitioned to "orangefs", and The Linux Coding Style avoids typedefs, so kernel module structures that correspond to userspace structures are not typedefed.

The kernel module implements a pseudo device that userspace can read from and write to. Userspace can also manipulate the kernel module through the pseudo device with `ioctl`.

The Bufmap

At startup userspace allocates two page-size-aligned (`posix_memalign`) mlocked memory buffers, one is used for IO and one is used for `readdir` operations. The IO buffer is 41943040 bytes and the `readdir` buffer is 4194304 bytes. Each buffer contains logical chunks, or partitions, and a pointer to each buffer is added to its own `PVFS_dev_map_desc` structure which also describes its total size, as well as the size and number of the partitions.

A pointer to the IO buffer's `PVFS_dev_map_desc` structure is sent to a mapping routine in the kernel module with an `ioctl`. The structure is copied from user space

to kernel space with `copy_from_user` and is used to initialize the kernel module's "bufmap" (struct `orangefs_bufmap`), which then contains:

- `refcnt` - a reference counter
- `desc_size` - `PVFS2_BUFMAP_DEFAULT_DESC_SIZE` (4194304) - the IO buffer's partition size, which represents the filesystem's block size and is used for `s_blocksize` in super blocks.
- `desc_count` - `PVFS2_BUFMAP_DEFAULT_DESC_COUNT` (10) - the number of partitions in the IO buffer.
- `desc_shift` - `log2(desc_size)`, used for `s_blocksize_bits` in super blocks.
- `total_size` - the total size of the IO buffer.
- `page_count` - the number of 4096 byte pages in the IO buffer.
- `page_array` - a pointer to `page_count * (sizeof(struct page*))` bytes of `kcallocated` memory. This memory is used as an array of pointers to each of the pages in the IO buffer through a call to `get_user_pages`.
- `desc_array` - a pointer to `desc_count * (sizeof(struct orangefs_bufmap_desc))` bytes of `kcallocated` memory. This memory is further initialized:

`user_desc` is the kernel's copy of the IO buffer's `ORANGEFS_dev_map_desc` structure. `user_desc->ptr` points to the IO buffer.

```
pages_per_desc = bufmap->desc_size / PAGE_SIZE
offset = 0

bufmap->desc_array[0].page_array = &bufmap->page_array[offset]
bufmap->desc_array[0].array_count = pages_per_desc = 1024
bufmap->desc_array[0].uaddr = (user_desc->ptr) + (0 * 1024 * 4096)
offset += 1024

      .
      .
      .
bufmap->desc_array[9].page_array = &bufmap->page_array[offset]
bufmap->desc_array[9].array_count = pages_per_desc = 1024
bufmap->desc_array[9].uaddr = (user_desc->ptr) +
                          (9 * 1024 * 4096)
offset += 1024
```

- `buffer_index_array` - a `desc_count` sized array of ints, used to indicate which of the IO buffer's partitions are available to use.
- `buffer_index_lock` - a spinlock to protect `buffer_index_array` during update.
- `readdir_index_array` - a five (`ORANGEFS_READDIR_DEFAULT_DESC_COUNT`) element int array used to indicate which of the `readdir` buffer's partitions are available to use.
- `readdir_index_lock` - a spinlock to protect `readdir_index_array` during update.

Operations

The kernel module builds an “op” (struct `orangepfs_kernel_op_s`) when it needs to communicate with userspace. Part of the op contains the “upcall” which expresses the request to userspace. Part of the op eventually contains the “downcall” which expresses the results of the request.

The slab allocator is used to keep a cache of op structures handy.

At init time the kernel module defines and initializes a request list and an `in_progress` hash table to keep track of all the ops that are in flight at any given time.

Ops are stateful:

- **unknown**
 - op was just initialized
- **waiting**
 - op is on `request_list` (upward bound)
- **inprogr**
 - op is in progress (waiting for downcall)
- **serviced**
 - op has matching downcall; ok
- **purged**
 - op has to start a timer since client-core exited uncleanly before servicing op
- **given up**
 - submitter has given up waiting for it

When some arbitrary userspace program needs to perform a filesystem operation on Orangepfs (readdir, I/O, create, whatever) an op structure is initialized and tagged with a distinguishing ID number. The upcall part of the op is filled out, and the op is passed to the “`service_operation`” function.

`Service_operation` changes the op’s state to “waiting”, puts it on the request list, and signals the Orangepfs `file_operations.poll` function through a wait queue. Userspace is polling the pseudo-device and thus becomes aware of the upcall request that needs to be read.

When the Orangepfs `file_operations.read` function is triggered, the request list is searched for an op that seems ready-to-process. The op is removed from the request list. The tag from the op and the filled-out upcall struct are `copy_to_user`’ed back to userspace.

If any of these (and some additional protocol) `copy_to_users` fail, the op’s state is set to “waiting” and the op is added back to the request list. Otherwise, the op’s state is changed to “in progress”, and the op is hashed on its tag and put onto the end of a list in the `in_progress` hash table at the index the tag hashed to.

When userspace has assembled the response to the upcall, it writes the response, which includes the distinguishing tag, back to the pseudo device in a series of `io_vecs`. This triggers the `Orangefs file_operations.write_iter` function to find the `op` with the associated tag and remove it from the `in_progress` hash table. As long as the `op`'s state is not "canceled" or "given up", its state is set to "serviced". The `file_operations.write_iter` function returns to the waiting `vfs`, and back to `service_operation` through `wait_for_matching_downcall`.

Service operation returns to its caller with the `op`'s downcall part (the response to the upcall) filled out.

The "client-core" is the bridge between the kernel module and userspace. The client-core is a daemon. The client-core has an associated watchdog daemon. If the client-core is ever signaled to die, the watchdog daemon restarts the client-core. Even though the client-core is restarted "right away", there is a period of time during such an event that the client-core is dead. A dead client-core can't be triggered by the `Orangefs file_operations.poll` function. Ops that pass through `service_operation` during a "dead spell" can timeout on the wait queue and one attempt is made to recycle them. Obviously, if the client-core stays dead too long, the arbitrary userspace processes trying to use `Orangefs` will be negatively affected. Waiting ops that can't be serviced will be removed from the request list and have their states set to "given up". In-progress ops that can't be serviced will be removed from the `in_progress` hash table and have their states set to "given up".

Readdir and I/O ops are atypical with respect to their payloads.

- readdir ops use the smaller of the two pre-allocated pre-partitioned memory buffers. The readdir buffer is only available to userspace. The kernel module obtains an index to a free partition before launching a readdir op. Userspace deposits the results into the indexed partition and then writes them to back to the `pvfs` device.
- io (read and write) ops use the larger of the two pre-allocated pre-partitioned memory buffers. The IO buffer is accessible from both userspace and the kernel module. The kernel module obtains an index to a free partition before launching an io op. The kernel module deposits write data into the indexed partition, to be consumed directly by userspace. Userspace deposits the results of read requests into the indexed partition, to be consumed directly by the kernel module.

Responses to kernel requests are all packaged in `pvfs2_downcall_t` structs. Besides a few other members, `pvfs2_downcall_t` contains a union of structs, each of which is associated with a particular response type.

The several members outside of the union are:

`int32_t type`

- type of operation.

`int32_t status`

- return code for the operation.

`int64_t trailer_size`

- 0 unless readdir operation.

char *trailer_buf

- initialized to NULL, used during readdir operations.

The appropriate member inside the union is filled out for any particular response.

PVFS2_VFS_OP_FILE_IO fill a `pvfs2_io_response_t`

PVFS2_VFS_OP_LOOKUP fill a `PVFS_object_kref`

PVFS2_VFS_OP_CREATE fill a `PVFS_object_kref`

PVFS2_VFS_OP_SYMLINK fill a `PVFS_object_kref`

PVFS2_VFS_OP_GETATTR fill in a `PVFS_sys_attr_s` (tons of stuff the kernel doesn't need) fill in a string with the link target when the object is a symlink.

PVFS2_VFS_OP_MKDIR fill a `PVFS_object_kref`

PVFS2_VFS_OP_STATFS fill a `pvfs2_statfs_response_t` with useless info <g>. It is hard for us to know, in a timely fashion, these statistics about our distributed network filesystem.

PVFS2_VFS_OP_FS_MOUNT fill a `pvfs2_fs_mount_response_t` which is just like a `PVFS_object_kref` except its members are in a different order and “__pad1” is replaced with “id” .

PVFS2_VFS_OP_GETXATTR fill a `pvfs2_getxattr_response_t`

PVFS2_VFS_OP_LISTXATTR fill a `pvfs2_listxattr_response_t`

PVFS2_VFS_OP_PARAM fill a `pvfs2_param_response_t`

PVFS2_VFS_OP_PERF_COUNT fill a `pvfs2_perf_count_response_t`

PVFS2_VFS_OP_FSKEY file a `pvfs2_fs_key_response_t`

PVFS2_VFS_OP_READDIR jamb everything needed to represent a `pvfs2_readdir_response_t` into the readdir buffer descriptor specified in the upcall.

Userspace uses `writew()` on `/dev/pvfs2-req` to pass responses to the requests made by the kernel side.

A `buffer_list` containing:

- a pointer to the prepared response to the request from the kernel (`struct pvfs2_downcall_t`).
- and also, in the case of a readdir request, a pointer to a buffer containing descriptors for the objects in the target directory.

...is sent to the function (`PINT_dev_write_list`) which performs the `writew`.

`PINT_dev_write_list` has a local `iovec` array: `struct iovec io_array[10]`;

The first four elements of `io_array` are initialized like this for all responses:

```
io_array[0].iov_base = address of local variable "proto_ver" (int32_t)
io_array[0].iov_len = sizeof(int32_t)

io_array[1].iov_base = address of global variable "pdev_magic" (int32_t)
```

(continues on next page)

(continued from previous page)

```
io_array[1].iov_len = sizeof(int32_t)

io_array[2].iov_base = address of parameter "tag" (PVFS_id_gen_t)
io_array[2].iov_len = sizeof(int64_t)

io_array[3].iov_base = address of out_downcall member (pvfs2_downcall_t)
                      of global variable vfs_request (vfs_request_t)
io_array[3].iov_len = sizeof(pvfs2_downcall_t)
```

Readdir responses initialize the fifth element `io_array` like this:

```
io_array[4].iov_base = contents of member trailer_buf (char *)
                      from out_downcall member of global variable
                      vfs_request
io_array[4].iov_len = contents of member trailer_size (PVFS_size)
                      from out_downcall member of global variable
                      vfs_request
```

Orangefs exploits the dcache in order to avoid sending redundant requests to userspace. We keep object inode attributes up-to-date with `orangefs_inode_getattr`. `Orangefs_inode_getattr` uses two arguments to help it decide whether or not to update an inode: “new” and “bypass”. Orangefs keeps private data in an object’s inode that includes a short timeout value, `getattr_time`, which allows any iteration of `orangefs_inode_getattr` to know how long it has been since the inode was updated. When the object is not new (`new == 0`) and the bypass flag is not set (`bypass == 0`) `orangefs_inode_getattr` returns without updating the inode if `getattr_time` has not timed out. `getattr_time` is updated each time the inode is updated.

Creation of a new object (file, dir, sym-link) includes the evaluation of its pathname, resulting in a negative directory entry for the object. A new inode is allocated and associated with the dentry, turning it from a negative dentry into a “productive full member of society”. Orangefs obtains the new inode from Linux with `new_inode()` and associates the inode with the dentry by sending the pair back to Linux with `d_instantiate()`.

The evaluation of a pathname for an object resolves to its corresponding dentry. If there is no corresponding dentry, one is created for it in the dcache. Whenever a dentry is modified or verified Orangefs stores a short timeout value in the dentry’s `d_time`, and the dentry will be trusted for that amount of time. Orangefs is a network filesystem, and objects can potentially change out-of-band with any particular Orangefs kernel module instance, so trusting a dentry is risky. The alternative to trusting dentries is to always obtain the needed information from userspace - at least a trip to the client-core, maybe to the servers. Obtaining information from a dentry is cheap, obtaining it from userspace is relatively expensive, hence the motivation to use the dentry when possible.

The timeout values `d_time` and `getattr_time` are jiffy based, and the code is designed to avoid the jiffy-wrap problem:

```
"In general, if the clock may have wrapped around more than once, there
is no way to tell how much time has elapsed. However, if the times t1
and t2 are known to be fairly close, we can reliably compute the
```

(continues on next page)

(continued from previous page)

difference in a way that takes into account the possibility that the clock may have wrapped between times."

from course notes by instructor Andy Wang

Written by: Neil Brown Please see MAINTAINERS file for where to send questions.

3.43 Overlay Filesystem

This document describes a prototype for a new approach to providing overlay-filesystem functionality in Linux (sometimes referred to as union-filesystems). An overlay-filesystem tries to present a filesystem which is the result over overlaying one filesystem on top of the other.

3.43.1 Overlay objects

The overlay filesystem approach is ‘hybrid’ , because the objects that appear in the filesystem do not always appear to belong to that filesystem. In many cases, an object accessed in the union will be indistinguishable from accessing the corresponding object from the original filesystem. This is most obvious from the ‘st_dev’ field returned by stat(2).

While directories will report an st_dev from the overlay-filesystem, non-directory objects may report an st_dev from the lower filesystem or upper filesystem that is providing the object. Similarly st_ino will only be unique when combined with st_dev, and both of these can change over the lifetime of a non-directory object. Many applications and tools ignore these values and will not be affected.

In the special case of all overlay layers on the same underlying filesystem, all objects will report an st_dev from the overlay filesystem and st_ino from the underlying filesystem. This will make the overlay mount more compliant with filesystem scanners and overlay objects will be distinguishable from the corresponding objects in the original filesystem.

On 64bit systems, even if all overlay layers are not on the same underlying filesystem, the same compliant behavior could be achieved with the “xino” feature. The “xino” feature composes a unique object identifier from the real object st_ino and an underlying fsid index.

If all underlying filesystems support NFS file handles and export file handles with 32bit inode number encoding (e.g. ext4), overlay filesystem will use the high inode number bits for fsid. Even when the underlying filesystem uses 64bit inode numbers, users can still enable the “xino” feature with the “-o xino=on” overlay mount option. That is useful for the case of underlying filesystems like xfs and tmpfs, which use 64bit inode numbers, but are very unlikely to use the high inode number bits. In case the underlying inode number does overflow into the high xino bits, overlay filesystem will fall back to the non xino behavior for that inode.

The following table summarizes what can be expected in different overlay configurations.

Inode properties

Configuration	Persistent st_ino		Uniform st_dev		st_ino == d_ino		d_ino == i_ino [*]	
	dir	!dir	dir	!dir	dir	!dir	dir	!dir
All layers on same fs	Y	Y	Y	Y	Y	Y	Y	Y
Layers not on same fs, xino=off	N	Y	Y	N	N	Y	N	Y
xino=on/auto	Y	Y	Y	Y	Y	Y	Y	Y
xino=on/auto, ino overflow	N	Y	Y	N	N	Y	N	Y

[*] nfsd v3 readdirplus verifies `d_ino == i_ino`. `i_ino` is exposed via several `/proc` files, such as `/proc/locks` and `/proc/self/fdinfo/<fd>` of an inotify file descriptor.

3.43.2 Upper and Lower

An overlay filesystem combines two filesystems - an ‘upper’ filesystem and a ‘lower’ filesystem. When a name exists in both filesystems, the object in the ‘upper’ filesystem is visible while the object in the ‘lower’ filesystem is either hidden or, in the case of directories, merged with the ‘upper’ object.

It would be more correct to refer to an upper and lower ‘directory tree’ rather than ‘filesystem’ as it is quite possible for both directory trees to be in the same filesystem and there is no requirement that the root of a filesystem be given for either upper or lower.

The lower filesystem can be any filesystem supported by Linux and does not need to be writable. The lower filesystem can even be another overlayfs. The upper filesystem will normally be writable and if it is it must support the creation of trusted.* extended attributes, and must provide valid `d_type` in readdir responses, so NFS is not suitable.

A read-only overlay of two read-only filesystems may use any filesystem type.

3.43.3 Directories

Overlaying mainly involves directories. If a given name appears in both upper and lower filesystems and refers to a non-directory in either, then the lower object is hidden - the name refers only to the upper object.

Where both upper and lower objects are directories, a merged directory is formed.

At mount time, the two directories given as mount options “lowerdir” and “upperdir” are combined into a merged directory:

```
mount -t overlay overlay -olowerdir=/lower,upperdir=/upper,workdir=/work
/merged
```

The “workdir” needs to be an empty directory on the same filesystem as upperdir.

Then whenever a lookup is requested in such a merged directory, the lookup is performed in each actual directory and the combined result is cached in the dentry belonging to the overlay filesystem. If both actual lookups find directories, both are stored and a merged directory is created, otherwise only one is stored: the upper if it exists, else the lower.

Only the lists of names from directories are merged. Other content such as meta-data and extended attributes are reported for the upper directory only. These attributes of the lower directory are hidden.

3.43.4 whiteouts and opaque directories

In order to support `rm` and `rmdir` without changing the lower filesystem, an overlay filesystem needs to record in the upper filesystem that files have been removed. This is done using whiteouts and opaque directories (non-directories are always opaque).

A whiteout is created as a character device with 0/0 device number. When a whiteout is found in the upper level of a merged directory, any matching name in the lower level is ignored, and the whiteout itself is also hidden.

A directory is made opaque by setting the `xattr` “`trusted.overlay.opaque`” to “`y`”. Where the upper filesystem contains an opaque directory, any directory in the lower filesystem with the same name is ignored.

3.43.5 readdir

When a ‘`readdir`’ request is made on a merged directory, the upper and lower directories are each read and the name lists merged in the obvious way (upper is read first, then lower - entries that already exist are not re-added). This merged name list is cached in the ‘`struct file`’ and so remains as long as the file is kept open. If the directory is opened and read by two processes at the same time, they will each have separate caches. A `seekdir` to the start of the directory (offset 0) followed by a `readdir` will cause the cache to be discarded and rebuilt.

This means that changes to the merged directory do not appear while a directory is being read. This is unlikely to be noticed by many programs.

seek offsets are assigned sequentially when the directories are read. Thus if

- read part of a directory
- remember an offset, and close the directory
- re-open the directory some time later
- seek to the remembered offset

there may be little correlation between the old and new locations in the list of filenames, particularly if anything has changed in the directory.

`Readdir` on directories that are not merged is simply handled by the underlying directory (upper or lower).

3.43.6 renaming directories

When renaming a directory that is on the lower layer or merged (i.e. the directory was not created on the upper layer to start with) overlayfs can handle it in two different ways:

1. return EXDEV error: this error is returned by `rename(2)` when trying to move a file or directory across filesystem boundaries. Hence applications are usually prepared to handle this error (`mv(1)` for example recursively copies the directory tree). This is the default behavior.
2. If the “`redirect_dir`” feature is enabled, then the directory will be copied up (but not the contents). Then the “`trusted.overlay.redirect`” extended attribute is set to the path of the original location from the root of the overlay. Finally the directory is moved to the new location.

There are several ways to tune the “`redirect_dir`” feature.

Kernel config options:

- **OVERLAY_FS_REDIRECT_DIR:** If this is enabled, then `redirect_dir` is turned on by default.
- **OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW:** If this is enabled, then redirects are always followed by default. Enabling this results in a less secure configuration. Enable this option only when worried about backward compatibility with kernels that have the `redirect_dir` feature and follow redirects even if turned off.

Module options (can also be changed through `/sys/module/overlay/parameters/`):

- “**redirect_dir=BOOL**” : See `OVERLAY_FS_REDIRECT_DIR` kernel config option above.
- “**redirect_always_follow=BOOL**” : See `OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW` kernel config option above.
- “**redirect_max=NUM**” : The maximum number of bytes in an absolute redirect (default is 256).

Mount options:

- “**redirect_dir=on**” : Redirects are enabled.
- “**redirect_dir=follow**” : Redirects are not created, but followed.
- “**redirect_dir=off**” : Redirects are not created and only followed if “`redirect_always_follow`” feature is enabled in the kernel/module config.
- “**redirect_dir=nofollow**” : Redirects are not created and not followed (equivalent to “`redirect_dir=off`” if “`redirect_always_follow`” feature is not enabled).

When the NFS export feature is enabled, every copied up directory is indexed by the file handle of the lower inode and a file handle of the upper directory is stored in a “`trusted.overlay.upper`” extended attribute on the index entry. On lookup of a merged directory, if the upper directory does not match the file handle stores in the index, that is an indication that multiple upper directories may be redirected

to the same lower directory. In that case, lookup returns an error and warns about a possible inconsistency.

Because lower layer redirects cannot be verified with the index, enabling NFS export support on an overlay filesystem with no upper layer requires turning off redirect follow (e.g. “`redirect_dir=nofollow`”).

3.43.7 Non-directories

Objects that are not directories (files, symlinks, device-special files etc.) are presented either from the upper or lower filesystem as appropriate. When a file in the lower filesystem is accessed in a way that requires write-access, such as opening for write access, changing some metadata etc., the file is first copied from the lower filesystem to the upper filesystem (`copy_up`). Note that creating a hard-link also requires `copy_up`, though of course creation of a symlink does not.

The `copy_up` may turn out to be unnecessary, for example if the file is opened for read-write but the data is not modified.

The `copy_up` process first makes sure that the containing directory exists in the upper filesystem - creating it and any parents as necessary. It then creates the object with the same metadata (owner, mode, mtime, symlink-target etc.) and then if the object is a file, the data is copied from the lower to the upper filesystem. Finally any extended attributes are copied up.

Once the `copy_up` is complete, the overlay filesystem simply provides direct access to the newly created file in the upper filesystem - future operations on the file are barely noticed by the overlay filesystem (though an operation on the name of the file such as rename or unlink will of course be noticed and handled).

3.43.8 Permission model

Permission checking in the overlay filesystem follows these principles:

- 1) permission check SHOULD return the same result before and after copy up
- 2) task creating the overlay mount MUST NOT gain additional privileges
- 3) non-mounting task MAY gain additional privileges through the overlay, compared to direct access on underlying lower or upper filesystems

This is achieved by performing two permission checks on each access

- a) check if current task is allowed access based on local DAC (owner, group, mode and posix acl), as well as MAC checks
- b) check if mounting task would be allowed real operation on lower or upper layer based on underlying filesystem permissions, again including MAC checks

Check (a) ensures consistency (1) since owner, group, mode and posix acls are copied up. On the other hand it can result in server enforced permissions (used by NFS, for example) being ignored (3).

Check (b) ensures that no task gains permissions to underlying layers that the mounting task does not have (2). This also means that it is possible to create setups where the consistency rule (1) does not hold; normally, however, the mounting task will have sufficient privileges to perform all operations.

Another way to demonstrate this model is drawing parallels between

```
mount -t overlay overlay -olowerdir=/lower,upperdir=/upper,···/merged
```

and

```
cp -a /lower /upper mount -bind /upper /merged
```

The resulting access permissions should be the same. The difference is in the time of copy (on-demand vs. up-front).

3.43.9 Multiple lower layers

Multiple lower layers can now be given using the the colon (“:”) as a separator character between the directory names. For example:

```
mount -t overlay overlay -olowerdir=/lower1:/lower2:/lower3 /merged
```

As the example shows, “upperdir=” and “workdir=” may be omitted. In that case the overlay will be read-only.

The specified lower directories will be stacked beginning from the rightmost one and going left. In the above example lower1 will be the top, lower2 the middle and lower3 the bottom layer.

3.43.10 Metadata only copy up

When metadata only copy up feature is enabled, overlayfs will only copy up metadata (as opposed to whole file), when a metadata specific operation like chown/chmod is performed. Full file will be copied up later when file is opened for WRITE operation.

In other words, this is delayed data copy up operation and data is copied up when there is a need to actually modify data.

There are multiple ways to enable/disable this feature. A config option CONFIG_OVERLAY_FS_METACOPY can be set/unset to enable/disable this feature by default. Or one can enable/disable it at module load time with module parameter metacopy=on/off. Lastly, there is also a per mount option metacopy=on/off to enable/disable this feature per mount.

Do not use metacopy=on with untrusted upper/lower directories. Otherwise it is possible that an attacker can create a handcrafted file with appropriate REDIRECT and METACOPY xattrs, and gain access to file on lower pointed by REDIRECT. This should not be possible on local system as setting “trusted.” xattrs will require CAP_SYS_ADMIN. But it should be possible for untrusted layers like from a pen drive.

Note: redirect_dir={off|nofollow|follow[*]} and nfs_export=on mount options conflict with metacopy=on, and will result in an error.

[*] `redirect_dir=follow` only conflicts with `metacopy=on` if `upperdir=...` is given.

3.43.11 Sharing and copying layers

Lower layers may be shared among several overlay mounts and that is indeed a very common practice. An overlay mount may use the same lower layer path as another overlay mount and it may use a lower layer path that is beneath or above the path of another overlay lower layer path.

Using an upper layer path and/or a `workdir` path that are already used by another overlay mount is not allowed and may fail with `EBUSY`. Using partially overlapping paths is not allowed and may fail with `EBUSY`. If files are accessed from two overlays mounts which share or overlap the upper layer and/or `workdir` path the behavior of the overlay is undefined, though it will not result in a crash or deadlock.

Mounting an overlay using an upper layer path, where the upper layer path was previously used by another mounted overlay in combination with a different lower layer path, is allowed, unless the “inodes index” feature or “metadata only copy up” feature is enabled.

With the “inodes index” feature, on the first time mount, an NFS file handle of the lower layer root directory, along with the UUID of the lower filesystem, are encoded and stored in the “trusted.overlay.origin” extended attribute on the upper layer root directory. On subsequent mount attempts, the lower root directory file handle and lower filesystem UUID are compared to the stored origin in upper root directory. On failure to verify the lower root origin, mount will fail with `ESTALE`. An overlays mount with “inodes index” enabled will fail with `EOPNOTSUPP` if the lower filesystem does not support NFS export, lower filesystem does not have a valid UUID or if the upper filesystem does not support extended attributes.

For “metadata only copy up” feature there is no verification mechanism at mount time. So if same upper is mounted with different set of lower, mount probably will succeed but expect the unexpected later on. So don't do it.

It is quite a common practice to copy overlay layers to a different directory tree on the same or different underlying filesystem, and even to a different machine. With the “inodes index” feature, trying to mount the copied layers will fail the verification of the lower root file handle.

3.43.12 Non-standard behavior

Current version of overlays can act as a mostly POSIX compliant filesystem.

This is the list of cases that overlays doesn't currently handle:

- a) POSIX mandates updating `st_atime` for reads. This is currently not done in the case when the file resides on a lower layer.
- b) If a file residing on a lower layer is opened for read-only and then memory mapped with `MAP_SHARED`, then subsequent changes to the file are not reflected in the memory mapping.

The following options allow overlays to act more like a standards compliant filesystem:

1) “redirect_dir”

Enabled with the mount option or module option: “redirect_dir=on” or with the kernel config option CONFIG_OVERLAY_FS_REDIRECT_DIR=y.

If this feature is disabled, then rename(2) on a lower or merged directory will fail with EXDEV (“Invalid cross-device link”).

2) “inode index”

Enabled with the mount option or module option “index=on” or with the kernel config option CONFIG_OVERLAY_FS_INDEX=y.

If this feature is disabled and a file with multiple hard links is copied up, then this will “break” the link. Changes will not be propagated to other names referring to the same inode.

3) “xino”

Enabled with the mount option “xino=auto” or “xino=on” , with the module option “xino_auto=on” or with the kernel config option CONFIG_OVERLAY_FS_XINO_AUTO=y. Also implicitly enabled by using the same underlying filesystem for all layers making up the overlay.

If this feature is disabled or the underlying filesystem doesn’ t have enough free bits in the inode number, then overlayfs will not be able to guarantee that the values of st_ino and st_dev returned by stat(2) and the value of d_ino returned by readdir(3) will act like on a normal filesystem. E.g. the value of st_dev may be different for two objects in the same overlay filesystem and the value of st_ino for directory objects may not be persistent and could change even while the overlay filesystem is mounted, as summarized in the Inode properties table above.

3.43.13 Changes to underlying filesystems

Offline changes, when the overlay is not mounted, are allowed to either the upper or the lower trees.

Changes to the underlying filesystems while part of a mounted overlay filesystem are not allowed. If the underlying filesystem is changed, the behavior of the overlay is undefined, though it will not result in a crash or deadlock.

When the overlay NFS export feature is enabled, overlay filesystems behavior on offline changes of the underlying lower layer is different than the behavior when NFS export is disabled.

On every copy_up, an NFS file handle of the lower inode, along with the UUID of the lower filesystem, are encoded and stored in an extended attribute “trusted.overlay.origin” on the upper inode.

When the NFS export feature is enabled, a lookup of a merged directory, that found a lower directory at the lookup path or at the path pointed to by the “trusted.overlay.redirect” extended attribute, will verify that the found lower directory file handle and lower filesystem UUID match the origin file handle that was stored at copy_up time. If a found lower directory does not match the stored origin, that directory will not be merged with the upper directory.

3.43.14 NFS export

When the underlying filesystems supports NFS export and the “nfs_export” feature is enabled, an overlay filesystem may be exported to NFS.

With the “nfs_export” feature, on copy_up of any lower object, an index entry is created under the index directory. The index entry name is the hexadecimal representation of the copy up origin file handle. For a non-directory object, the index entry is a hard link to the upper inode. For a directory object, the index entry has an extended attribute “trusted.overlay.upper” with an encoded file handle of the upper directory inode.

When encoding a file handle from an overlay filesystem object, the following rules apply:

1. For a non-upper object, encode a lower file handle from lower inode
2. For an indexed object, encode a lower file handle from copy_up origin
3. For a pure-upper object and for an existing non-indexed upper object, encode an upper file handle from upper inode

The encoded overlay file handle includes:

- Header including path type information (e.g. lower/upper)
- UUID of the underlying filesystem
- Underlying filesystem encoding of underlying inode

This encoding format is identical to the encoding format file handles that are stored in extended attribute “trusted.overlay.origin” .

When decoding an overlay file handle, the following steps are followed:

1. Find underlying layer by UUID and path type information.
2. Decode the underlying filesystem file handle to underlying dentry.
3. For a lower file handle, lookup the handle in index directory by name.
4. If a whiteout is found in index, return ESTALE. This represents an overlay object that was deleted after its file handle was encoded.
5. For a non-directory, instantiate a disconnected overlay dentry from the decoded underlying dentry, the path type and index inode, if found.
6. For a directory, use the connected underlying decoded dentry, path type and index, to lookup a connected overlay dentry.

Decoding a non-directory file handle may return a disconnected dentry. copy_up of that disconnected dentry will create an upper index entry with no upper alias.

When overlay filesystem has multiple lower layers, a middle layer directory may have a “redirect” to lower directory. Because middle layer “redirects” are not indexed, a lower file handle that was encoded from the “redirect” origin directory, cannot be used to find the middle or upper layer directory. Similarly, a lower file handle that was encoded from a descendant of the “redirect” origin directory, cannot be used to reconstruct a connected overlay path. To mitigate the cases of directories that cannot be decoded from a lower file handle, these directories are copied up on encode and encoded as an upper file handle. On an overlay

filesystem with no upper layer this mitigation cannot be used NFS export in this setup requires turning off redirect follow (e.g. “`redirect_dir=nofollow`”).

The overlay filesystem does not support non-directory connectable file handles, so exporting with the ‘`subtree_check`’ `exportfs` configuration will cause failures to lookup files over NFS.

When the NFS export feature is enabled, all directory index entries are verified on mount time to check that upper file handles are not stale. This verification may cause significant overhead in some cases.

Note: the mount options `index=off,nfs_export=on` are conflicting and will result in an error.

3.43.15 Testsuite

There’s a testsuite originally developed by David Howells and currently maintained by Amir Goldstein at:

<https://github.com/amir73il/unionmount-testsuite.git>

Run as root:

```
# cd unionmount-testsuite # ./run -ov -verify
```

3.44 The /proc Filesystem

/proc/sys	Terrehon Bowden <terrehon@pacbell.net>, Bodo Bauer <bb@ricochet.net>	October 7 1999
2.4.x update	Jorge Nerin <comandante@zaralinux.com>	November 14 2000
move /proc/sys	Shen Feng <shen@cn.fujitsu.com>	April 1 2009
fixes/update part 1.1	Stefani Seibold <stefani@seibold.net>	June 9 2009

3.44.1 Preface

0.1 Introduction/Credits

This documentation is part of a soon (or so we hope) to be released book on the SuSE Linux distribution. As there is no complete documentation for the /proc file system and we’ve used many freely available sources to write these chapters, it seems only fair to give the work back to the Linux community. This work is based on the 2.2.* kernel version and the upcoming 2.4.*. I’m afraid it’s still far from complete, but we hope it will be useful. As far as we know, it is the first ‘all-in-one’ document about the /proc file system. It is focused on the Intel x86 hardware, so if you are looking for PPC, ARM, SPARC, AXP, etc., features, you probably won’t find what you are looking for. It also only covers IPv4 networking, not IPv6 nor

other protocols - sorry. But additions and patches are welcome and will be added to this document if you mail them to Bodo.

We' d like to thank Alan Cox, Rik van Riel, and Alexey Kuznetsov and a lot of other people for help compiling this documentation. We' d also like to extend a special thank you to Andi Kleen for documentation, which we relied on heavily to create this document, as well as the additional information he provided. Thanks to everybody else who contributed source or docs to the Linux kernel and helped create a great piece of software...

If you have any comments, corrections or additions, please don't hesitate to contact Bodo Bauer at bb@ricochet.net. We' ll be happy to add them to this document.

The latest version of this document is available online at <http://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html>

If the above direction does not works for you, you could try the kernel mailing list at linux-kernel@vger.kernel.org and/or try to reach me at comandante@zaralinux.com.

0.2 Legal Stuff

We don' t guarantee the correctness of this document, and if you come to us complaining about how you screwed up your system because of incorrect documentation, we won' t feel responsible...

3.44.2 Chapter 1: Collecting System Information

In This Chapter

- Investigating the properties of the pseudo file system /proc and its ability to provide information on the running Linux system
- Examining /proc' s structure
- Uncovering various information about the kernel and the processes running on the system

The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl).

First, we' ll take a look at the read-only parts of /proc. In Chapter 2, we show you how you can use /proc/sys to change settings.

1.1 Process-Specific Subdirectories

The directory `/proc` contains (among other things) one subdirectory for each process running on the system, which is named after the process ID (PID).

The link `self` points to the process reading the file system. Each process subdirectory has the entries listed in Table 1-1.

Note that an open a file descriptor to `/proc/<pid>` or to any of its contained files or subdirectories does not prevent `<pid>` being reused for some other process in the event that `<pid>` exits. Operations on open `/proc/<pid>` file descriptors corresponding to dead processes never act on any new process that the kernel may, through chance, have also assigned the process ID `<pid>`. Instead, operations on these FDs usually fail with `ESRCH`.

Table 2: Table 1-1: Process specific entries in `/proc`

File	Content
<code>clear_refs</code>	Clears page referenced bits shown in <code>smaps</code> output
<code>cmd-line</code>	Command line arguments
<code>cpu</code>	Current and last <code>cpu</code> in which it was executed (2.4)(<code>smp</code>)
<code>cwd</code>	Link to the current working directory
<code>environ</code>	Values of environment variables
<code>exe</code>	Link to the executable of this process
<code>fd</code>	Directory, which contains all file descriptors
<code>maps</code>	Memory maps to executables and library files (2.4)
<code>mem</code>	Memory held by this process
<code>root</code>	Link to the root directory of this process
<code>stat</code>	Process status
<code>statm</code>	Process memory status information
<code>status</code>	Process status in human readable form
<code>wchan</code>	Present with <code>CONFIG_KALLSYMS=y</code> : it shows the kernel function symbol the task is blocked in - or "0" if not blocked.
<code>pagemap</code>	Page table
<code>stack</code>	Report full stack trace, enable via <code>CONFIG_STACKTRACE</code>
<code>smaps</code>	An extension based on <code>maps</code> , showing the memory consumption of each mapping and flags associated with it
<code>smaps_ro</code>	Accumulated <code>smaps</code> stats for all mappings of the process. This can be derived from <code>smaps</code> , but is faster and more convenient
<code>numa_maps</code>	An extension based on <code>maps</code> , showing the memory locality and binding policy as well as <code>mem</code> usage (in pages) of each mapping.

For example, to get the status information of a process, all you have to do is read the file `/proc/PID/status`:

```
>cat /proc/self/status
Name:   cat
State:  R (running)
Tgid:   5452
Pid:    5452
```

(continues on next page)

(continued from previous page)

```

PPid: 743
TracerPid: 0 (2.4)
Uid: 501 501 501 501
Gid: 100 100 100 100
FDSize: 256
Groups: 100 14 16
VmPeak: 5004 kB
VmSize: 5004 kB
VmLck: 0 kB
VmHWM: 476 kB
VmRSS: 476 kB
RssAnon: 352 kB
RssFile: 120 kB
RssShmem: 4 kB
VmData: 156 kB
VmStk: 88 kB
VmExe: 68 kB
VmLib: 1412 kB
VmPTE: 20 kb
VmSwap: 0 kB
HugetlbPages: 0 kB
CoreDumping: 0
THP_enabled: 1
Threads: 1
SigQ: 0/28578
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 00000000fffffeff
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
CapAmb: 0000000000000000
NoNewPrivs: 0
Seccomp: 0
Speculation_Store_Bypass: thread vulnerable
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 1

```

This shows you nearly the same information you would get if you viewed it with the `ps` command. In fact, `ps` uses the `proc` file system to obtain its information. But you get a more detailed view of the process by reading the file `/proc/PID/status`. Its fields are described in table 1-2.

The `statm` file contains more detailed information about the process memory usage. Its seven fields are explained in Table 1-3. The `stat` file contains details information about the process itself. Its fields are explained in Table 1-4.

(for SMP CONFIG users)

For making accounting scalable, RSS related information are handled in an asynchronous manner and the value may not be very precise. To see a precise snapshot of a moment, you can see `/proc/<pid>/smaps` file and scan page table. It's slow but very precise.

Table 3: Table 1-2: Contents of the status files (as of 4.19)

	Field	Content
Name		filename of the executable
Umask		file mode creation mask
State		state (R is running, S is sleeping, D is sleeping in an uninterruptible wait, Z is zombie)
Tgid		thread group ID
Ngid		NUMA group ID (0 if none)
Pid		process id
PPid		process id of the parent process
TracerPid		PID of process tracing this process (0 if not)
Uid		Real, effective, saved set, and file system UIDs
Gid		Real, effective, saved set, and file system GIDs
FDSize		number of file descriptor slots currently allocated
Groups		supplementary group list
NSStgid		descendant namespace thread group ID hierarchy
NSPid		descendant namespace process ID hierarchy
NSPgId		descendant namespace process group ID hierarchy
NSSid		descendant namespace session ID hierarchy
VmPeak		peak virtual memory size
VmSize		total program size
VmLck		locked memory size
VmPin		pinned memory size
VmHWM		peak resident set size (“high water mark”)
VmRSS		size of memory portions. It contains the three following parts (VmRSS = RssAnon + RssFile + RssShmem)
RssAnon		size of resident anonymous memory
RssFile		size of resident file mappings
RssShmem		size of resident shmem memory (includes SysV shm, mapping of tmpfs and shmem)
VmData		size of private data segments
VmStk		size of stack segments
VmExe		size of text segment
VmLib		size of shared library code
VmPTE		size of page table entries
VmSwap		amount of swap used by anonymous private data (shmem swap usage is not included)
HugetlbPages		size of hugetlb memory portions
CoreDumping		process’ s memory is currently being dumped (killing the process may lead to core dump)
THP_enabled		process is allowed to use THP (returns 0 when PR_SET_THP_DISABLE is set)
Threads		number of threads
SigQ		number of signals queued/max. number for queue
SigPnd		bitmap of pending signals for the thread
ShdPnd		bitmap of shared pending signals for the process
SigBlk		bitmap of blocked signals
SigIgn		bitmap of ignored signals
SigCgt		bitmap of caught signals
CapInh		bitmap of inheritable capabilities
CapPrm		bitmap of permitted capabilities
CapEff		bitmap of effective capabilities
CapBnd		bitmap of capabilities bounding set

Continued on next page

Table 3 – continued from previous page

Field	Content
CapAmb	bitmap of ambient capabilities
NoNewPrivs	no_new_privs, like prctl(PR_GET_NO_NEW_PRIV, ...)
Seccomp	seccomp mode, like prctl(PR_GET_SECCOMP, ...)
Speculation_Store_Bypass	speculative store bypass mitigation status
Cpus_allowed	mask of CPUs on which this process may run
Cpus_allowed_list	Same as previous, but in “list format”
Mems_allowed	mask of memory nodes allowed to this process
Mems_allowed_list	Same as previous, but in “list format”
voluntary_ctxt_switches	number of voluntary context switches
nonvoluntary_ctxt_switches	number of non voluntary context switches

Table 4: Table 1-3: Contents of the statm files (as of 2.6.8-rc3)

Field	Content	
size	total program size (pages)	(same as VmSize in status)
resident	size of memory portions (pages)	(same as VmRSS in status)
shared	number of pages that are shared	(i.e. backed by a file, same as Rss-File+RssShmem in status)
trs	number of pages that are ‘code’	(not including libs; broken, includes data segment)
lrs	number of pages of library	(always 0 on 2.6)
drs	number of pages of data/stack	(including libs; broken, includes library text)
dt	number of dirty pages	(always 0 on 2.6)

Table 5: Table 1-4: Contents of the stat files (as of 2.6.30-rc7)

Field	Content
pid	process id
tcomm	filename of the executable
state	state (R is running, S is sleeping, D is sleeping in an uninterruptible wait, Z is zombie)
ppid	process id of the parent process
pgrp	pgrp of the process
sid	session id
tty_nr	tty the process uses
tty_pgrp	pgrp of the tty
flags	task flags
minflt	number of minor faults
cminflt	number of minor faults with child’ s
majflt	number of major faults
cmajflt	number of major faults with child’ s
utime	user mode jiffies

Continued on next page

Table 5 - continued from previous page

	Field	Content
stime		kernel mode jiffies
cutime		user mode jiffies with child' s
cstime		kernel mode jiffies with child' s
priority		priority level
nice		nice level
num_threads		number of threads
it_real_value		(obsolete, always 0)
start_time		time the process started after system boot
vsize		virtual memory size
rss		resident set memory size
rsslim		current limit in bytes on the rss
start_code		address above which program text can run
end_code		address below which program text can run
start_stack		address of the start of the main process stack
esp		current value of ESP
eip		current value of EIP
pending		bitmap of pending signals
blocked		bitmap of blocked signals
sigign		bitmap of ignored signals
sigcatch		bitmap of caught signals
0		(place holder, used to be the wchan address, use /proc/PID/wchan instead)
0		(place holder)
0		(place holder)
exit_signal		signal to send to parent thread on exit
task_cpu		which CPU the task is scheduled on
rt_priority		realtime priority
policy		scheduling policy (man sched_setscheduler)
blkio_ticks		time spent waiting for block IO
gtime		guest time of the task in jiffies
cgtime		guest time of the task children in jiffies
start_data		address above which program data+bss is placed
end_data		address below which program data+bss is placed
start_brk		address above which program heap can be expanded with brk()
arg_start		address above which program command line is placed
arg_end		address below which program command line is placed
env_start		address above which program environment is placed
env_end		address below which program environment is placed
exit_code		the thread' s exit_code in the form reported by the waitpid system call

The /proc/PID/maps file contains the currently mapped memory regions and their access permissions.

The format is:

address	perms	offset	dev	inode	pathname
08048000-08049000	r-xp	00000000	03:00	8312	/opt/test
08049000-0804a000	rw-p	00001000	03:00	8312	/opt/test

(continues on next page)

(continued from previous page)

0804a000-0806b000	rw-p	00000000	00:00	0		[heap]
a7cb1000-a7cb2000	---p	00000000	00:00	0		
a7cb2000-a7eb2000	rw-p	00000000	00:00	0		
a7eb2000-a7eb3000	---p	00000000	00:00	0		
a7eb3000-a7ed5000	rw-p	00000000	00:00	0		
a7ed5000-a8008000	r-xp	00000000	03:00	4222		/lib/libc.so.6
a8008000-a800a000	r--p	00133000	03:00	4222		/lib/libc.so.6
a800a000-a800b000	rw-p	00135000	03:00	4222		/lib/libc.so.6
a800b000-a800e000	rw-p	00000000	00:00	0		
a800e000-a8022000	r-xp	00000000	03:00	14462		/lib/libpthread.so.0
a8022000-a8023000	r--p	00013000	03:00	14462		/lib/libpthread.so.0
a8023000-a8024000	rw-p	00014000	03:00	14462		/lib/libpthread.so.0
a8024000-a8027000	rw-p	00000000	00:00	0		
a8027000-a8043000	r-xp	00000000	03:00	8317		/lib/ld-linux.so.2
a8043000-a8044000	r--p	0001b000	03:00	8317		/lib/ld-linux.so.2
a8044000-a8045000	rw-p	0001c000	03:00	8317		/lib/ld-linux.so.2
aff35000-aff4a000	rw-p	00000000	00:00	0		[stack]
ffffe000-ffffff00	r-xp	00000000	00:00	0		[vdso]

where “address” is the address space in the process that it occupies, “perms” is a set of permissions:

```
r = read
w = write
x = execute
s = shared
p = private (copy on write)
```

“offset” is the offset into the mapping, “dev” is the device (major:minor), and “inode” is the inode on that device. 0 indicates that no inode is associated with the memory region, as the case would be with BSS (uninitialized data). The “pathname” shows the name associated file for this mapping. If the mapping is not associated with a file:

[heap]	the heap of the program
[stack]	the stack of the main process
[vdso]	the “virtual dynamic shared object” , the kernel system call handler

or if empty, the mapping is anonymous.

The /proc/PID/smmaps is an extension based on maps, showing the memory consumption for each of the process’ s mappings. For each mapping (aka Virtual Memory Area, or VMA) there is a series of lines such as the following:

08048000-080bc000	r-xp	00000000	03:02	13130		/bin/bash
Size:		1084	kB			
KernelPageSize:		4	kB			
MMUPageSize:		4	kB			
Rss:		892	kB			
Pss:		374	kB			

(continues on next page)

(continued from previous page)

```
Shared_Clean:      892 kB
Shared_Dirty:      0 kB
Private_Clean:     0 kB
Private_Dirty:     0 kB
Referenced:        892 kB
Anonymous:         0 kB
LazyFree:          0 kB
AnonHugePages:    0 kB
ShmemPmdMapped:   0 kB
Shared_Hugetlb:   0 kB
Private_Hugetlb:  0 kB
Swap:              0 kB
SwapPss:           0 kB
KernelPageSize:   4 kB
MMUPageSize:      4 kB
Locked:           0 kB
THPEligible:      0
VmFlags: rd ex mr mw me dw
```

The first of these lines shows the same information as is displayed for the mapping in `/proc/PID/maps`. Following lines show the size of the mapping (size); the size of each page allocated when backing a VMA (`KernelPageSize`), which is usually the same as the size in the page table entries; the page size used by the MMU when backing a VMA (in most cases, the same as `KernelPageSize`); the amount of the mapping that is currently resident in RAM (RSS); the process' proportional share of this mapping (PSS); and the number of clean and dirty shared and private pages in the mapping.

The “proportional set size” (PSS) of a process is the count of pages it has in memory, where each page is divided by the number of processes sharing it. So if a process has 1000 pages all to itself, and 1000 shared with one other process, its PSS will be 1500.

Note that even a page which is part of a `MAP_SHARED` mapping, but has only a single pte mapped, i.e. is currently used by only one process, is accounted as private and not as shared.

“Referenced” indicates the amount of memory currently marked as referenced or accessed.

“Anonymous” shows the amount of memory that does not belong to any file. Even a mapping associated with a file may contain anonymous pages: when `MAP_PRIVATE` and a page is modified, the file page is replaced by a private anonymous copy.

“LazyFree” shows the amount of memory which is marked by `madvise(MADV_FREE)`. The memory isn't freed immediately with `madvise()`. It's freed in memory pressure if the memory is clean. Please note that the printed value might be lower than the real value due to optimizations used in the current implementation. If this is not desirable please file a bug report.

“AnonHugePages” shows the amount of memory backed by transparent hugepage.

“ShmemPmdMapped” shows the amount of shared (shmem/tmpfs) memory backed by huge pages.

“Shared_Hugetlb” and “Private_Hugetlb” show the amounts of memory backed by hugetlbfs page which is not counted in “RSS” or “PSS” field for historical reasons. And these are not included in {Shared,Private}_{Clean,Dirty} field.

“Swap” shows how much would-be-anonymous memory is also used, but out on swap.

For shmem mappings, “Swap” includes also the size of the mapped (and not replaced by copy-on-write) part of the underlying shmem object out on swap. “SwapPss” shows proportional swap share of this mapping. Unlike “Swap”, this does not take into account swapped out page of underlying shmem objects. “Locked” indicates whether the mapping is locked in memory or not. “THPEligible” indicates whether the mapping is eligible for allocating THP pages - 1 if true, 0 otherwise. It just shows the current status.

“VmFlags” field deserves a separate description. This member represents the kernel flags associated with the particular virtual memory area in two letter encoded manner. The codes are the following:

Note that there is no guarantee that every flag and associated mnemonic will be present in all further kernel releases. Things get changed, the flags may be vanished or the reverse - new added. Interpretation of their meaning might change in future as well. So each consumer of these flags has to follow each specific kernel version for the exact semantic.

This file is only present if the CONFIG_MMU kernel configuration option is enabled.

Note: reading /proc/PID/maps or /proc/PID/smmaps is inherently racy (consistent output can be achieved only in the single read call).

This typically manifests when doing partial reads of these files while the memory map is being modified. Despite the races, we do provide the following guarantees:

- 1) The mapped addresses never go backwards, which implies no two regions will ever overlap.
- 2) If there is something at a given vaddr during the entirety of the life of the smmaps/maps walk, there will be some output for it.

The /proc/PID/smmaps_rollup file includes the same fields as /proc/PID/smmaps, but their values are the sums of the corresponding values for all mappings of the process. Additionally, it contains these fields:

- Pss_Anon
- Pss_File
- Pss_Shmem

They represent the proportional shares of anonymous, file, and shmem pages, as described for smmaps above. These fields are omitted in smmaps since each mapping identifies the type (anon, file, or shmem) of all pages it contains. Thus all information in smmaps_rollup can be derived from smmaps, but at a significantly higher cost.

The `/proc/PID/clear_refs` is used to reset the `PG_Referenced` and `ACCESSED/YOUNG` bits on both physical and virtual pages associated with a process, and the `soft-dirty` bit on `pte` (see `Documentation/admin-guide/mm/soft-dirty.rst` for details). To clear the bits for all the pages associated with the process:

```
> echo 1 > /proc/PID/clear_refs
```

To clear the bits for the anonymous pages associated with the process:

```
> echo 2 > /proc/PID/clear_refs
```

To clear the bits for the file mapped pages associated with the process:

```
> echo 3 > /proc/PID/clear_refs
```

To clear the `soft-dirty` bit:

```
> echo 4 > /proc/PID/clear_refs
```

To reset the peak resident set size (“high water mark”) to the process’ s current value:

```
> echo 5 > /proc/PID/clear_refs
```

Any other value written to `/proc/PID/clear_refs` will have no effect.

The `/proc/pid/pagemap` gives the `PFN`, which can be used to find the page-flags using `/proc/kpageflags` and number of times a page is mapped using `/proc/kpagecount`. For detailed explanation, see `Documentation/admin-guide/mm/pagemap.rst`.

The `/proc/pid/numa_maps` is an extension based on `maps`, showing the memory locality and binding policy, as well as the memory usage (in pages) of each mapping. The output follows a general format where mapping details get summarized separated by blank spaces, one mapping per each file line:

```
address  policy  mapping details
00400000 default file=/usr/local/bin/app mapped=1 active=0 N3=1
↳kernelpagesize_kB=4
00600000 default file=/usr/local/bin/app anon=1 dirty=1 N3=1
↳kernelpagesize_kB=4
3206000000 default file=/lib64/ld-2.12.so mapped=26 mapmax=6 N0=24 N3=2
↳kernelpagesize_kB=4
320621f000 default file=/lib64/ld-2.12.so anon=1 dirty=1 N3=1
↳kernelpagesize_kB=4
3206220000 default file=/lib64/ld-2.12.so anon=1 dirty=1 N3=1
↳kernelpagesize_kB=4
3206221000 default anon=1 dirty=1 N3=1 kernelpagesize_kB=4
3206800000 default file=/lib64/libc-2.12.so mapped=59 mapmax=21 active=55
↳N0=41 N3=18 kernelpagesize_kB=4
320698b000 default file=/lib64/libc-2.12.so
3206b8a000 default file=/lib64/libc-2.12.so anon=2 dirty=2 N3=2
↳kernelpagesize_kB=4
3206b8e000 default file=/lib64/libc-2.12.so anon=1 dirty=1 N3=1
↳kernelpagesize_kB=4
```

(continues on next page)

(continued from previous page)

```

3206b8f000 default anon=3 dirty=3 active=1 N3=3 kernelpagesize_kB=4
7f4dc10a2000 default anon=3 dirty=3 N3=3 kernelpagesize_kB=4
7f4dc10b4000 default anon=2 dirty=2 active=1 N3=2 kernelpagesize_kB=4
7f4dc1200000 default file=/anon_hugepage\040(deleted) huge anon=1 dirty=1
↳N3=1 kernelpagesize_kB=2048
7fff335f0000 default stack anon=3 dirty=3 N3=3 kernelpagesize_kB=4
7fff3369d000 default mapped=1 mapmax=35 active=0 N3=1 kernelpagesize_kB=4

```

Where:

“address” is the starting address for the mapping;

“policy” reports the NUMA memory policy set for the mapping (see Documentation/admin-guide/mm/numa_memory_policy.rst);

“mapping details” summarizes mapping data such as mapping type, page usage counters, node locality page counters (N0 == node0, N1 == node1, ...) and the kernel page size, in KB, that is backing the mapping up.

1.2 Kernel data

Similar to the process entries, the kernel data files give information about the running kernel. The files used to obtain this information are contained in /proc and are listed in Table 1-5. Not all of these will be present in your system. It depends on the kernel configuration and the loaded modules, which files are there, and which are missing.

Table 6: Table 1-5: Kernel info in /proc

	File	Content
apm		Advanced power management info
buddyinfo		Kernel memory allocator information (see text) (2.5)
bus		Directory containing bus specific information
cmdline		Kernel command line
cpuinfo		Info about the CPU
devices		Available devices (block and character)
dma		Used DMS channels
filesystems		Supported filesystems
driver		Various drivers grouped here, currently rtc (2.4)
execdomains		Execdomains, related to security (2.4)
fb		Frame Buffer devices (2.4)
fs		File system parameters, currently nfs/exports (2.4)
ide		Directory containing info about the IDE subsystem
interrupts		Interrupt usage
iomem		Memory map (2.4)
ioports		I/O port usage
irq		Masks for irq to cpu affinity (2.4)(smp?)
isapnp		ISA PnP (Plug&Play) Info (2.4)
kcore		Kernel core image (can be ELF or A.OUT(deprecated in 2.4))
kmsg		Kernel messages
ksyms		Kernel symbol table

Continued on next page

Table 6 - continued from previous page

	File	Content
loadavg		Load average of last 1, 5 & 15 minutes
locks		Kernel locks
meminfo		Memory info
misc		Miscellaneous
modules		List of loaded modules
mounts		Mounted filesystems
net		Networking info (see text)
pagetypeinfo		Additional page allocator information (see text) (2.5)
partitions		Table of partitions known to the system
pci		Deprecated info of PCI bus (new way -> /proc/bus/pci/, decoupled by lspci (2.4)
rtc		Real time clock
scsi		SCSI info (see text)
slabinfo		Slab pool info
softirqs		softirq usage
stat		Overall statistics
swaps		Swap space utilization
sys		See chapter 2
sysvipc		Info of SysVIPC Resources (msg, sem, shm) (2.4)
tty		Info of tty drivers
uptime		Wall clock since boot, combined idle time of all cpus
version		Kernel version
video		bttv info of video resources (2.4)
vmallocinfo		Show vmallocated areas

You can, for example, check which interrupts are currently in use and what they are used for by looking in the file /proc/interrupts:

```
> cat /proc/interrupts
          CPU0
 0:   8728810      XT-PIC  timer
 1:     895       XT-PIC  keyboard
 2:      0        XT-PIC  cascade
 3:   531695      XT-PIC  aha152x
 4:  2014133      XT-PIC  serial
 5:   44401       XT-PIC  pcnet_cs
 8:      2        XT-PIC  rtc
11:      8        XT-PIC  i82365
12:  182918      XT-PIC  PS/2 Mouse
13:      1        XT-PIC  fpu
14:  1232265      XT-PIC  ide0
15:      7        XT-PIC  ide1
NMI:      0
```

In 2.4.* a couple of lines were added to this file LOC & ERR (this time is the output of a SMP machine):

```
> cat /proc/interrupts
          CPU0      CPU1
 0:   1243498    1214548  IO-APIC-edge  timer
```

(continues on next page)

(continued from previous page)

1:	8949	8958	IO-APIC-edge	keyboard
2:	0	0	XT-PIC	cascade
5:	11286	10161	IO-APIC-edge	soundblaster
8:	1	0	IO-APIC-edge	rtc
9:	27422	27407	IO-APIC-edge	3c503
12:	113645	113873	IO-APIC-edge	PS/2 Mouse
13:	0	0	XT-PIC	fpu
14:	22491	24012	IO-APIC-edge	ide0
15:	2183	2415	IO-APIC-edge	ide1
17:	30564	30414	IO-APIC-level	eth0
18:	177	164	IO-APIC-level	bttv
NMI:	2457961	2457959		
LOC:	2457882	2457881		
ERR:	2155			

NMI is incremented in this case because every timer interrupt generates a NMI (Non Maskable Interrupt) which is used by the NMI Watchdog to detect lockups.

LOC is the local interrupt counter of the internal APIC of every CPU.

ERR is incremented in the case of errors in the IO-APIC bus (the bus that connects the CPUs in a SMP system. This means that an error has been detected, the IO-APIC automatically retry the transmission, so it should not be a big problem, but you should read the SMP-FAQ.

In 2.6.2* `/proc/interrupts` was expanded again. This time the goal was for `/proc/interrupts` to display every IRQ vector in use by the system, not just those considered ‘most important’. The new vectors are:

THR interrupt raised when a machine check threshold counter (typically counting ECC corrected errors of memory or cache) exceeds a configurable threshold. Only available on some systems.

TRM a thermal event interrupt occurs when a temperature threshold has been exceeded for the CPU. This interrupt may also be generated when the temperature drops back to normal.

SPU a spurious interrupt is some interrupt that was raised then lowered by some IO device before it could be fully processed by the APIC. Hence the APIC sees the interrupt but does not know what device it came from. For this case the APIC will generate the interrupt with a IRQ vector of 0xff. This might also be generated by chipset bugs.

RES, CAL, TLB] rescheduling, call and TLB flush interrupts are sent from one CPU to another per the needs of the OS. Typically, their statistics are used by kernel developers and interested users to determine the occurrence of interrupts of the given type.

The above IRQ vectors are displayed only when relevant. For example, the threshold vector does not exist on x86_64 platforms. Others are suppressed when the system is a uniprocessor. As of this writing, only i386 and x86_64 platforms support the new IRQ vector displays.

Of some interest is the introduction of the `/proc/irq` directory to 2.4. It could be used to set IRQ to CPU affinity, this means that you can “hook” an IRQ to only one

CPU, or to exclude a CPU of handling IRQs. The contents of the `irq` subdir is one subdir for each IRQ, and two files; `default_smp_affinity` and `prof_cpu_mask`.

For example:

```
> ls /proc/irq/  
0 10 12 14 16 18 2 4 6 8 prof_cpu_mask  
1 11 13 15 17 19 3 5 7 9 default_smp_affinity  
> ls /proc/irq/0/  
smp_affinity
```

`smp_affinity` is a bitmask, in which you can specify which CPUs can handle the IRQ, you can set it by doing:

```
> echo 1 > /proc/irq/10/smp_affinity
```

This means that only the first CPU will handle the IRQ, but you can also echo 5 which means that only the first and third CPU can handle the IRQ.

The contents of each `smp_affinity` file is the same by default:

```
> cat /proc/irq/0/smp_affinity  
ffffffff
```

There is an alternate interface, `smp_affinity_list` which allows specifying a cpu range instead of a bitmask:

```
> cat /proc/irq/0/smp_affinity_list  
1024-1031
```

The `default_smp_affinity` mask applies to all non-active IRQs, which are the IRQs which have not yet been allocated/activated, and hence which lack a `/proc/irq/[0-9]*` directory.

The `node` file on an SMP system shows the node to which the device using the IRQ reports itself as being attached. This hardware locality information does not include information about any possible driver locality preference.

`prof_cpu_mask` specifies which CPUs are to be profiled by the system wide profiler. Default value is `ffffffff` (all cpus if there are only 32 of them).

The way IRQs are routed is handled by the IO-APIC, and it's Round Robin between all the CPUs which are allowed to handle it. As usual the kernel has more info than you and does a better job than you, so the defaults are the best choice for almost everyone. [Note this applies only to those IO-APIC's that support "Round Robin" interrupt distribution.]

There are three more important subdirectories in `/proc`: `net`, `scsi`, and `sys`. The general rule is that the contents, or even the existence of these directories, depend on your kernel configuration. If SCSI is not enabled, the directory `scsi` may not exist. The same is true with the `net`, which is there only when networking support is present in the running kernel.

The `slabinfo` file gives information about memory usage at the slab level. Linux uses slab pools for memory management above page level in version 2.2. Commonly used objects have their own slab pool (such as network buffers, directory cache, and so on).

```
> cat /proc/buddyinfo
```

```
Node 0, zone DMA      0      4      5      4      4      3 ...
Node 0, zone Normal  1      0      0      1     101     8 ...
Node 0, zone HighMem 2      0      0      1      1      0 ...
```

External fragmentation is a problem under some workloads, and buddyinfo is a useful tool for helping diagnose these problems. Buddyinfo will give you a clue as to how big an area you can safely allocate, or why a previous allocation failed.

Each column represents the number of pages of a certain order which are available. In this case, there are 0 chunks of $2^0 \times \text{PAGE_SIZE}$ available in ZONE_DMA, 4 chunks of $2^1 \times \text{PAGE_SIZE}$ in ZONE_DMA, 101 chunks of $2^4 \times \text{PAGE_SIZE}$ available in ZONE_NORMAL, etc...

More information relevant to external fragmentation can be found in pagetypeinfo:

```
> cat /proc/pagetypeinfo
```

```
Page block order: 9
```

```
Pages per block: 512
```

```
Free pages count per migrate type at order 0 1 2 3
→ 4 5 6 7 8 9 10
Node 0, zone DMA, type Unmovable 0 0 0 1
→ 1 1 1 1 1 1 0
Node 0, zone DMA, type Reclaimable 0 0 0 0
→ 0 0 0 0 0 0 0
Node 0, zone DMA, type Movable 1 1 2 1
→ 2 1 1 0 1 0 2
Node 0, zone DMA, type Reserve 0 0 0 0
→ 0 0 0 0 0 1 0
Node 0, zone DMA, type Isolate 0 0 0 0
→ 0 0 0 0 0 0 0
Node 0, zone DMA32, type Unmovable 103 54 77 1
→ 1 1 11 8 7 1 9
Node 0, zone DMA32, type Reclaimable 0 0 2 1
→ 0 0 0 0 1 0 0
Node 0, zone DMA32, type Movable 169 152 113 91
→ 77 54 39 13 6 1 452
Node 0, zone DMA32, type Reserve 1 2 2 2
→ 2 0 1 1 1 1 0
Node 0, zone DMA32, type Isolate 0 0 0 0
→ 0 0 0 0 0 0 0

Number of blocks type Unmovable Reclaimable Movable Reserve
→ Isolate
Node 0, zone DMA 2 0 5 1
→ 0
Node 0, zone DMA32 41 6 967 2
→ 0
```

Fragmentation avoidance in the kernel works by grouping pages of different migrate types into the same contiguous regions of memory called page blocks. A page block is typically the size of the default hugepage size e.g. 2MB on X86-64. By keeping pages grouped based on their ability to move, the kernel can reclaim pages within a page block to satisfy a high-order allocation.

The `pagetypinfo` begins with information on the size of a page block. It then gives the same type of information as `buddyinfo` except broken down by migrate-type and finishes with details on how many page blocks of each type exist.

If `min_free_kbytes` has been tuned correctly (recommendations made by `hugeadm` from `libhugetlbfs` <https://github.com/libhugetlbfs/libhugetlbfs/>), one can make an estimate of the likely number of huge pages that can be allocated at a given point in time. All the “Movable” blocks should be allocatable unless memory has been `mlock()` d. Some of the Reclaimable blocks should also be allocatable although a lot of filesystem metadata may have to be reclaimed to achieve this.

meminfo

Provides information about distribution and utilization of memory. This varies by architecture and compile options. The following is from a 16GB PIII, which has `highmem` enabled. You may not have all of these fields.

```
> cat /proc/meminfo
MemTotal:      16344972 kB
MemFree:       13634064 kB
MemAvailable: 14836172 kB
Buffers:       3656 kB
Cached:        1195708 kB
SwapCached:    0 kB
Active:        891636 kB
Inactive:      1077224 kB
HighTotal:    15597528 kB
HighFree:     13629632 kB
LowTotal:     747444 kB
LowFree:      4432 kB
SwapTotal:    0 kB
SwapFree:     0 kB
Dirty:        968 kB
Writeback:    0 kB
AnonPages:    861800 kB
Mapped:       280372 kB
Shmem:        644 kB
KReclaimable: 168048 kB
Slab:         284364 kB
SReclaimable: 159856 kB
SUnreclaim:  124508 kB
PageTables:   24448 kB
NFS_Unstable: 0 kB
Bounce:       0 kB
WritebackTmp: 0 kB
CommitLimit: 7669796 kB
Committed_AS: 100056 kB
VmallocTotal: 112216 kB
VmallocUsed:  428 kB
VmallocChunk: 111088 kB
Percpu:       62080 kB
HardwareCorrupted: 0 kB
AnonHugePages: 49152 kB
```

(continues on next page)

(continued from previous page)

ShmemHugePages :	0 kB
ShmemPmdMapped :	0 kB

MemTotal Total usable ram (i.e. physical ram minus a few reserved bits and the kernel binary code)

MemFree The sum of LowFree+HighFree

MemAvailable An estimate of how much memory is available for starting new applications, without swapping. Calculated from MemFree, SReclaimable, the size of the file LRU lists, and the low watermarks in each zone. The estimate takes into account that the system needs some page cache to function well, and that not all reclaimable slab will be reclaimable, due to items being in use. The impact of those factors will vary from system to system.

Buffers Relatively temporary storage for raw disk blocks shouldn't get tremendously large (20MB or so)

Cached in-memory cache for files read from the disk (the pagecache). Doesn't include SwapCached

SwapCached Memory that once was swapped out, is swapped back in but still also is in the swapfile (if memory is needed it doesn't need to be swapped out AGAIN because it is already in the swapfile. This saves I/O)

Active Memory that has been used more recently and usually not reclaimed unless absolutely necessary.

Inactive Memory which has been less recently used. It is more eligible to be reclaimed for other purposes

HighTotal, HighFree Highmem is all memory above ~860MB of physical memory Highmem areas are for use by userspace programs, or for the pagecache. The kernel must use tricks to access this memory, making it slower to access than lowmem.

LowTotal, LowFree Lowmem is memory which can be used for everything that highmem can be used for, but it is also available for the kernel's use for its own data structures. Among many other things, it is where everything from the Slab is allocated. Bad things happen when you're out of lowmem.

SwapTotal total amount of swap space available

SwapFree Memory which has been evicted from RAM, and is temporarily on the disk

Dirty Memory which is waiting to get written back to the disk

Writeback Memory which is actively being written back to the disk

AnonPages Non-file backed pages mapped into userspace page tables

HardwareCorrupted The amount of RAM/memory in KB, the kernel identifies as corrupted.

AnonHugePages Non-file backed huge pages mapped into userspace page tables

Mapped files which have been mmaped, such as libraries

Shmem Total memory used by shared memory (shmem) and tmpfs

ShmemHugePages Memory used by shared memory (shmem) and tmpfs allocated with huge pages

ShmemPmdMapped Shared memory mapped into userspace with huge pages

KReclaimable Kernel allocations that the kernel will attempt to reclaim under memory pressure. Includes SReclaimable (below), and other direct allocations with a shrinker.

Slab in-kernel data structures cache

SReclaimable Part of Slab, that might be reclaimed, such as caches

SUnreclaim Part of Slab, that cannot be reclaimed on memory pressure

PageTables amount of memory dedicated to the lowest level of page tables.

NFS_Unstable Always zero. Previous counted pages which had been written to the server, but has not been committed to stable storage.

Bounce Memory used for block device “bounce buffers”

WritebackTmp Memory used by FUSE for temporary writeback buffers

CommitLimit Based on the overcommit ratio (‘vm.overcommit_ratio’), this is the total amount of memory currently available to be allocated on the system. This limit is only adhered to if strict overcommit accounting is enabled (mode 2 in ‘vm.overcommit_memory’).

The CommitLimit is calculated with the following formula:

$$\text{CommitLimit} = ([\text{total RAM pages}] - [\text{total huge TLB pages}]) * \text{overcommit_ratio} / 100 + [\text{total swap pages}]$$

For example, on a system with 1G of physical RAM and 7G of swap with a vm.overcommit_ratio of 30 it would yield a CommitLimit of 7.3G.

For more details, see the memory overcommit documentation in vm/overcommit-accounting.

Committed_AS The amount of memory presently allocated on the system. The committed memory is a sum of all of the memory which has been allocated by processes, even if it has not been “used” by them as of yet. A process which malloc()’ s 1G of memory, but only touches 300M of it will show up as using 1G. This 1G is memory which has been “committed” to by the VM and can be used at any time by the allocating application. With strict overcommit enabled on the system (mode 2 in ‘vm.overcommit_memory’), allocations which would exceed the CommitLimit (detailed above) will not be permitted. This is useful if one needs to guarantee that processes will not fail due to lack of memory once that memory has been successfully allocated.

VmallocTotal total size of vmalloc memory area

VmallocUsed amount of vmalloc area which is used

VmallocChunk largest contiguous block of vmalloc area which is free

Percpu Memory allocated to the percpu allocator used to back percpu allocations. This stat excludes the cost of metadata.

vmallocinfo

Provides information about vmallocated/vmapped areas. One line per area, containing the virtual address range of the area, size in bytes, caller information of the creator, and optional information depending on the kind of area :

pages=nr	number of pages
phys=addr	if a physical address was specified
ioremap	I/O mapping (ioremap() and friends)
vmalloc	vmalloc() area
vmap	vmap()ed pages
user	VM_USERMAP area
vpages	buffer for pages pointers was vmallocated (huge area)
N<node>=n	(Only on NUMA kernels) Number of pages allocated on memory node <node>

```
> cat /proc/vmallocinfo
0xfffffc2000000000-0xfffffc20000201000 2101248 alloc_large_system_
↳hash+0x204 ...
/0x2c0 pages=512 vmalloc N0=128 N1=128 N2=128 N3=128
0xfffffc20000201000-0xfffffc20000302000 1052672 alloc_large_system_
↳hash+0x204 ...
/0x2c0 pages=256 vmalloc N0=64 N1=64 N2=64 N3=64
0xfffffc20000302000-0xfffffc20000304000      8192 acpi_tb_verify_table+0x21/
↳0x4f...
phys=7fee8000 ioremap
0xfffffc20000304000-0xfffffc20000307000      12288 acpi_tb_verify_table+0x21/
↳0x4f...
phys=7fee7000 ioremap
0xfffffc2000031d000-0xfffffc2000031f000      8192 init_vdso_vars+0x112/0x210
0xfffffc2000031f000-0xfffffc2000032b000     49152 cramfs_uncompress_init+0x2e .
↳...
/0x80 pages=11 vmalloc N0=3 N1=3 N2=2 N3=3
0xfffffc2000033a000-0xfffffc2000033d000      12288 sys_swapon+0x640/0xac0      .
↳...
pages=2 vmalloc N1=2
0xfffffc20000347000-0xfffffc2000034c000      20480 xt_alloc_table_info+0xfe ...
/0x130 [x_tables] pages=4 vmalloc N0=4
0xfffffffffa0000000-0xfffffffffa000f000     61440 sys_init_module+0xc27/0x1d00_
↳...
pages=14 vmalloc N2=14
0xfffffffffa000f000-0xfffffffffa0014000     20480 sys_init_module+0xc27/0x1d00_
↳...
pages=4 vmalloc N1=4
0xfffffffffa0014000-0xfffffffffa0017000     12288 sys_init_module+0xc27/0x1d00_
↳...
pages=2 vmalloc N1=2
0xfffffffffa0017000-0xfffffffffa0022000     45056 sys_init_module+0xc27/0x1d00_
↳...
pages=10 vmalloc N0=10
```

softirqs

Provides counts of softirq handlers serviced since boot time, for each cpu.

```

> cat /proc/softirqs
          CPU0          CPU1          CPU2          CPU3
HI:             0             0             0             0
TIMER:        27166        27120        27097        27034
NET_TX:         0             0             0             17
NET_RX:         42             0             0             39
BLOCK:          0             0            107            1121
TASKLET:        0             0             0             290
SCHED:        27035        26983        26971        26746
HRTIMER:        0             0             0             0
RCU:           1678         1769         2178         2250

```

1.3 IDE devices in /proc/ide

The subdirectory /proc/ide contains information about all IDE devices of which the kernel is aware. There is one subdirectory for each IDE controller, the file drivers and a link for each IDE device, pointing to the device directory in the controller specific subtree.

The file drivers contains general information about the drivers used for the IDE devices:

```

> cat /proc/ide/drivers
ide-cdrom version 4.53
ide-disk version 1.08

```

More detailed information can be found in the controller specific subdirectories. These are named ide0, ide1 and so on. Each of these directories contains the files shown in table 1-6.

Table 7: Table 1-6: IDE controller info in /proc/ide/ide?

File	Content
channel	IDE channel (0 or 1)
config	Configuration (only for PCI/IDE bridge)
mate	Mate name
model	Type/Chipset of IDE controller

Each device connected to a controller has a separate subdirectory in the controllers directory. The files listed in table 1-7 are contained in these directories.

Table 8: Table 1-7: IDE device information

File	Content
cache	The cache
capacity	Capacity of the medium (in 512Byte blocks)
driver	driver and version
geometry	physical and logical geometry
identify	device identify block
media	media type
model	device identifier
settings	device setup
smart_thresholds	IDE disk management thresholds
smart_values	IDE disk management values

The most interesting file is `settings`. This file contains a nice overview of the drive parameters:

```
# cat /proc/ide/ide0/hda/settings
name          value          min            max            r
↔mode
-----
↔-
bios_cyl      526            0              65535         rw
bios_head    255            0              255           rw
bios_sect     63             0              63            rw
breada_readahead 4              0              127           rw
bswap        0              0              1             r
file_readahead 72            0              2097151      rw
io_32bit     0              0              3             rw
keepsettings 0              0              1             rw
max_kb_per_request 122          1              127          rw
multcount    0              0              8            rw
nicel        1              0              1            rw
nowerr       0              0              1            rw
pio_mode     write-only     0              255          w
slow         0              0              1            rw
unmaskirq   0              0              1            rw
using_dma    0              0              1            rw
```

1.4 Networking info in /proc/net

The subdirectory `/proc/net` follows the usual pattern. Table 1-8 shows the additional values you get for IP version 6 if you configure the kernel to support this. Table 1-9 lists the files and their meaning.

Table 9: Table 1-8: IPv6 info in /proc/net

File	Content
udp6	UDP sockets (IPv6)
tcp6	TCP sockets (IPv6)
raw6	Raw device statistics (IPv6)
igmp6	IP multicast addresses, which this host joined (IPv6)
if_inet6	List of IPv6 interface addresses
ipv6_route	Kernel routing table for IPv6
rt6_stats	Global IPv6 routing tables statistics
sockstat6	Socket statistics (IPv6)
snmp6	Snmp data (IPv6)

Table 10: Table 1-9: Network info in /proc/net

File	Content
arp	Kernel ARP table
dev	network devices with statistics
dev_mcast	the Layer2 multicast groups a device is listening too (interface index, label, number of references, number of bound addresses).
dev_stat	network device status
ip_fwchain	Firewall chain linkage
ip_fwnames	Firewall chain names
ip_masq	Directory containing the masquerading tables
ip_masquerade	Major masquerading table
netstat	Network statistics
raw	raw device statistics
route	Kernel routing table
rpc	Directory containing rpc info
rt_cache	Routing cache
snmp	SNMP data
sock-stat	Socket statistics
tcp	TCP sockets
udp	UDP sockets
unix	UNIX domain sockets
wireless	Wireless interface data (Wavelan etc)
igmp	IP multicast addresses, which this host joined
psched	Global packet scheduler parameters.
netlink	List of PF_NETLINK sockets
ip_mr_vifs	List of multicast virtual interfaces
ip_mr_cache	List of multicast routing cache

You can use this information to see which network devices are available in your system and how much traffic was routed over those devices:

```
> cat /proc/net/dev
Inter-|Receive
face |bytes  packets errs drop fifo frame compressed multicast|...
```

(continues on next page)

(continued from previous page)

lo:	908188	5596	0	0	0	0	0	0	[...]
ppp0:	15475140	20721	410	0	0	410	0	0	[...]
eth0:	614530	7085	0	0	0	0	0	1	[...]
...] Transmit									
...]	bytes	packets	errs	drop	fifo	colls	carrier	compressed	
...]	908188	5596	0	0	0	0	0	0	
...]	1375103	17405	0	0	0	0	0	0	
...]	1703981	5535	0	0	0	3	0	0	

In addition, each Channel Bond interface has its own directory. For example, the `bond0` device will have a directory called `/proc/net/bond0/`. It will contain information that is specific to that bond, such as the current slaves of the bond, the link status of the slaves, and how many times the slaves link has failed.

1.5 SCSI info

If you have a SCSI host adapter in your system, you'll find a subdirectory named after the driver for this adapter in `/proc/scsi`. You'll also see a list of all recognized SCSI devices in `/proc/scsi`:

```
>cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: IBM          Model: DGHS09U          Rev: 03E0
  Type:   Direct-Access          ANSI SCSI revision: 03
Host: scsi0 Channel: 00 Id: 06 Lun: 00
  Vendor: PIONEER      Model: CD-ROM DR-U06S   Rev: 1.04
  Type:   CD-ROM          ANSI SCSI revision: 02
```

The directory named after the driver has one file for each adapter found in the system. These files contain information about the controller, including the used IRQ and the IO address range. The amount of information shown is dependent on the adapter you use. The example shows the output for an Adaptec AHA-2940 SCSI adapter:

```
> cat /proc/scsi/aic7xxx/0
Adaptec AIC7xxx driver version: 5.1.19/3.2.4
Compile Options:
  TCQ Enabled By Default : Disabled
  AIC7XXX_PROC_STATS     : Disabled
  AIC7XXX_RESET_DELAY    : 5
Adapter Configuration:
  SCSI Adapter: Adaptec AHA-294X Ultra SCSI host adapter
                Ultra Wide Controller
  PCI MMAPed I/O Base: 0xeb001000
  Adapter SEEPROM Config: SEEPROM found and used.
  Adaptec SCSI BIOS: Enabled
                    IRQ: 10
                    SCBs: Active 0, Max Active 2,
                          Allocated 15, HW 16, Page 255
  Interrupts: 160328
```

(continues on next page)

(continued from previous page)

```

    BIOS Control Word: 0x18b6
    Adapter Control Word: 0x005b
    Extended Translation: Enabled
Disconnect Enable Flags: 0xffff
    Ultra Enable Flags: 0x0001
    Tag Queue Enable Flags: 0x0000
Ordered Queue Tag Flags: 0x0000
Default Tag Queue Depth: 8
    Tagged Queue By Device array for aic7xxx host instance 0:
    {255,255,255,255,255,255,255,255,255,255,255,255,255,255,255}
    Actual queue depth per device for aic7xxx host instance 0:
    {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
Statistics:
(scsi0:0:0:0)
    Device using Wide/Sync transfers at 40.0 MByte/sec, offset 8
    Transinfo settings: current(12/8/1/0), goal(12/8/1/0), user(12/15/1/0)
    Total transfers 160151 (74577 reads and 85574 writes)
(scsi0:0:6:0)
    Device using Narrow/Sync transfers at 5.0 MByte/sec, offset 15
    Transinfo settings: current(50/15/0/0), goal(50/15/0/0), user(50/15/0/0)
    Total transfers 0 (0 reads and 0 writes)

```

1.6 Parallel port info in /proc/parport

The directory /proc/parport contains information about the parallel ports of your system. It has one subdirectory for each port, named after the port number (0,1,2, ...).

These directories contain the four files shown in Table 1-10.

Table 11: Table 1-10: Files in /proc/parport

File	Content
auto-probe	Any IEEE-1284 device ID information that has been acquired.
de-vices	list of the device drivers using that port. A + will appear by the name of the device currently using the port (it might not appear against any).
hard-ware	Parallel port's base address, IRQ line and DMA channel.
irq	IRQ that parport is using for that port. This is in a separate file to allow you to alter it by writing a new value in (IRQ number or none).

1.7 TTY info in /proc/tty

Information about the available and actually used tty's can be found in the directory /proc/tty. You'll find entries for drivers and line disciplines in this directory, as shown in Table 1-11.

Table 12: Table 1-11: Files in /proc/tty

File	Content
drivers	list of drivers and their usage
ldiscs	registered line disciplines
driver/serial	usage statistic and status of single tty lines

To see which tty' s are currently in use, you can simply look into the file /proc/tty/drivers:

```
> cat /proc/tty/drivers
pty_slave      /dev/pts      136  0-255 pty:slave
pty_master     /dev/ptm      128  0-255 pty:master
pty_slave      /dev/typ      3    0-255 pty:slave
pty_master     /dev/pty      2    0-255 pty:master
serial         /dev/cua      5    64-67 serial:callout
serial         /dev/ttyS     4    64-67 serial
/dev/tty0      /dev/tty0     4    0    system:vtmaster
/dev/ptmx      /dev/ptmx     5    2    system
/dev/console   /dev/console  5    1    system:console
/dev/tty       /dev/tty      5    0    system:/dev/tty
unknown       /dev/tty      4    1-63 console
```

1.8 Miscellaneous kernel statistics in /proc/stat

Various pieces of information about kernel activity are available in the /proc/stat file. All of the numbers reported in this file are aggregates since the system first booted. For a quick look, simply cat the file:

```
> cat /proc/stat
cpu 2255 34 2290 22625563 6290 127 456 0 0 0
cpu0 1132 34 1441 11311718 3675 127 438 0 0 0
cpu1 1123 0 849 11313845 2614 0 18 0 0 0
intr 114930548 113199788 3 0 5 263 0 4 [... lots more numbers ...]
ctxt 1990473
btime 1062191376
processes 2915
procs_running 1
procs_blocked 0
softirq 183433 0 21755 12 39 1137 231 21459 2263
```

The very first “cpu” line aggregates the numbers in all of the other “cpuN” lines. These numbers identify the amount of time the CPU has spent performing different kinds of work. Time units are in USER_HZ (typically hundredths of a second). The meanings of the columns are as follows, from left to right:

- user: normal processes executing in user mode
- nice: niced processes executing in user mode
- system: processes executing in kernel mode
- idle: twiddling thumbs
- iowait: In a word, iowait stands for waiting for I/O to complete. But there are several problems:

1. Cpu will not wait for I/O to complete, iowait is the time that a task is waiting for I/O to complete. When cpu goes into idle state for outstanding task io, another task will be scheduled on this CPU.
2. In a multi-core CPU, the task waiting for I/O to complete is not running on any CPU, so the iowait of each CPU is difficult to calculate.
3. The value of iowait field in `/proc/stat` will decrease in certain conditions.

So, the iowait is not reliable by reading from `/proc/stat`.

- `irq`: servicing interrupts
- `softirq`: servicing softirqs
- `steal`: involuntary wait
- `guest`: running a normal guest
- `guest_nice`: running a niced guest

The “`intr`” line gives counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

The “`ctxt`” line gives the total number of context switches across all CPUs.

The “`btime`” line gives the time at which the system booted, in seconds since the Unix epoch.

The “`processes`” line gives the number of processes and threads created, which includes (but is not limited to) those created by calls to the `fork()` and `clone()` system calls.

The “`procs_running`” line gives the total number of threads that are running or ready to run (i.e., the total number of runnable threads).

The “`procs_blocked`” line gives the number of processes currently blocked, waiting for I/O to complete.

The “`softirq`” line gives counts of softirqs serviced since boot time, for each of the possible system softirqs. The first column is the total of all softirqs serviced; each subsequent column is the total for that particular softirq.

1.9 Ext4 file system parameters

Information about mounted ext4 file systems can be found in `/proc/fs/ext4`. Each mounted filesystem will have a directory in `/proc/fs/ext4` based on its device name (i.e., `/proc/fs/ext4/hdc` or `/proc/fs/ext4/dm-0`). The files in each per-device directory are shown in Table 1-12, below.

Table 13: Table 1-12: Files in
`/proc/fs/ext4/<devname>`

File	Content
<code>mb_groups</code>	details of multiblock allocator buddy cache of free blocks

2.0 /proc/consoles

Shows registered system console lines.

To see which character device lines are currently used for the system console `/dev/console`, you may simply look into the file `/proc/consoles`:

```
> cat /proc/consoles
tty0          -WU (ECp)      4:7
ttyS0        -W- (Ep)      4:64
```

The columns are:

device	name of the device
operations	<ul style="list-style-type: none"> • R = can do read operations • W = can do write operations • U = can do unblank
flags	<ul style="list-style-type: none"> • E = it is enabled • C = it is preferred console • B = it is primary boot console • p = it is used for printk buffer • b = it is not a TTY but a Braille device • a = it is safe to use when cpu is offline
major:minor	major and minor number of the device separated by a colon

Summary

The `/proc` file system serves information about the running system. It not only allows access to process data but also allows you to request the kernel status by reading files in the hierarchy.

The directory structure of `/proc` reflects the types of information and makes it easy, if not obvious, where to look for specific data.

3.44.3 Chapter 2: Modifying System Parameters

In This Chapter

- Modifying kernel parameters by writing into files found in `/proc/sys`
- Exploring the files which modify certain parameters
- Review of the `/proc/sys` file tree

A very interesting part of `/proc` is the directory `/proc/sys`. This is not only a source of information, it also allows you to change parameters within the kernel. Be very careful when attempting this. You can optimize your system, but you can also cause it to crash. Never alter kernel parameters on a production system. Set up a development machine and test to make sure that everything works the way you want it to. You may have no alternative but to reboot the machine once an error has been made.

To change a value, simply echo the new value into the file. An example is given below in the section on the file system data. You need to be root to do this. You can create your own boot script to perform this every time your system boots.

The files in `/proc/sys` can be used to fine tune and monitor miscellaneous and general things in the operation of the Linux kernel. Since some of the files can inadvertently disrupt your system, it is advisable to read both documentation and source before actually making adjustments. In any case, be very careful when writing to any of these files. The entries in `/proc` may change slightly between the 2.1.* and the 2.2 kernel, so if there is any doubt review the kernel documentation in the directory `/usr/src/linux/Documentation`. This chapter is heavily based on the documentation included in the pre 2.2 kernels, and became part of it in version 2.2.1 of the Linux kernel.

Please see: `Documentation/admin-guide/sysctl/` directory for descriptions of these entries.

Summary

Certain aspects of kernel behavior can be modified at runtime, without the need to recompile the kernel, or even to reboot the system. The files in the `/proc/sys` tree can not only be read, but also modified. You can use the `echo` command to write value into these files, thereby changing the default settings of the kernel.

3.44.4 Chapter 3: Per-process Parameters

3.1 `/proc/<pid>/oom_adj` & `/proc/<pid>/oom_score_adj`- Adjust the oom-killer score

These file can be used to adjust the badness heuristic used to select which process gets killed in out of memory conditions.

The badness heuristic assigns a value to each candidate task ranging from 0 (never kill) to 1000 (always kill) to determine which process is targeted. The units are roughly a proportion along that range of allowed memory the process may allocate from based on an estimation of its current memory and swap use. For example, if a task is using all allowed memory, its badness score will be 1000. If it is using half of its allowed memory, its score will be 500.

There is an additional factor included in the badness score: the current memory and swap usage is discounted by 3% for root processes.

The amount of “allowed” memory depends on the context in which the oom killer was called. If it is due to the memory assigned to the allocating task’s `cpuset` being exhausted, the allowed memory represents the set of mems assigned to that

cpuset. If it is due to a mempolicy's node(s) being exhausted, the allowed memory represents the set of mempolicy nodes. If it is due to a memory limit (or swap limit) being reached, the allowed memory is that configured limit. Finally, if it is due to the entire system being out of memory, the allowed memory represents all allocatable resources.

The value of `/proc/<pid>/oom_score_adj` is added to the badness score before it is used to determine which task to kill. Acceptable values range from -1000 (`OOM_SCORE_ADJ_MIN`) to +1000 (`OOM_SCORE_ADJ_MAX`). This allows userspace to polarize the preference for oom killing either by always preferring a certain task or completely disabling it. The lowest possible value, -1000, is equivalent to disabling oom killing entirely for that task since it will always report a badness score of 0.

Consequently, it is very simple for userspace to define the amount of memory to consider for each task. Setting a `/proc/<pid>/oom_score_adj` value of +500, for example, is roughly equivalent to allowing the remainder of tasks sharing the same system, cpuset, mempolicy, or memory controller resources to use at least 50% more memory. A value of -500, on the other hand, would be roughly equivalent to discounting 50% of the task's allowed memory from being considered as scoring against the task.

For backwards compatibility with previous kernels, `/proc/<pid>/oom_adj` may also be used to tune the badness score. Its acceptable values range from -16 (`OOM_ADJUST_MIN`) to +15 (`OOM_ADJUST_MAX`) and a special value of -17 (`OOM_DISABLE`) to disable oom killing entirely for that task. Its value is scaled linearly with `/proc/<pid>/oom_score_adj`.

The value of `/proc/<pid>/oom_score_adj` may be reduced no lower than the last value set by a `CAP_SYS_RESOURCE` process. To reduce the value any lower requires `CAP_SYS_RESOURCE`.

Caveat: when a parent task is selected, the oom killer will sacrifice any first generation children with separate address spaces instead, if possible. This avoids servers and important system daemons from being killed and loses the minimal amount of work.

3.2 `/proc/<pid>/oom_score` - Display current oom-killer score

This file can be used to check the current score used by the oom-killer is for any given `<pid>`. Use it together with `/proc/<pid>/oom_score_adj` to tune which process should be killed in an out-of-memory situation.

3.3 `/proc/<pid>/io` - Display the IO accounting fields

This file contains IO statistics for each running process

Example

```
test:/tmp # dd if=/dev/zero of=/tmp/test.dat &
[1] 3828

test:/tmp # cat /proc/3828/io
rchar: 323934931
wchar: 323929600
syscr: 632687
syscw: 632675
read_bytes: 0
write_bytes: 323932160
cancelled_write_bytes: 0
```

Description

rchar

I/O counter: chars read The number of bytes which this task has caused to be read from storage. This is simply the sum of bytes which this process passed to read() and pread(). It includes things like tty IO and it is unaffected by whether or not actual physical disk IO was required (the read might have been satisfied from pagecache)

wchar

I/O counter: chars written The number of bytes which this task has caused, or shall cause to be written to disk. Similar caveats apply here as with rchar.

syscr

I/O counter: read syscalls Attempt to count the number of read I/O operations, i.e. syscalls like read() and pread().

syscw

I/O counter: write syscalls Attempt to count the number of write I/O operations, i.e. syscalls like write() and pwrite().

read_bytes

I/O counter: bytes read Attempt to count the number of bytes which this process really did cause to be fetched from the storage layer. Done at the `submit_bio()` level, so it is accurate for block-backed filesystems. <please add status regarding NFS and CIFS at a later time>

write_bytes

I/O counter: bytes written Attempt to count the number of bytes which this process caused to be sent to the storage layer. This is done at page-dirtying time.

cancelled_write_bytes

The big inaccuracy here is truncate. If a process writes 1MB to a file and then deletes the file, it will in fact perform no writeout. But it will have been accounted as having caused 1MB of write. In other words: The number of bytes which this process caused to not happen, by truncating pagecache. A task can cause “negative” IO too. If this task truncates some dirty pagecache, some IO which another task has been accounted for (in its `write_bytes`) will not be happening. We `_could_` just subtract that from the truncating task’s `write_bytes`, but there is information loss in doing that.

Note: At its current implementation state, this is a bit racy on 32-bit machines: if process A reads process B’s `/proc/pid/io` while process B is updating one of those 64-bit counters, process A could see an intermediate result.

More information about this can be found within the `taskstats` documentation in `Documentation/accounting`.

3.4 /proc/<pid>/coredump_filter - Core dump filtering settings

When a process is dumped, all anonymous memory is written to a core file as long as the size of the core file isn’t limited. But sometimes we don’t want to dump some memory segments, for example, huge shared memory or DAX. Conversely, sometimes we want to save file-backed memory segments into a core file, not only the individual files.

`/proc/<pid>/coredump_filter` allows you to customize which memory segments will be dumped when the `<pid>` process is dumped. `coredump_filter` is a bitmask of memory types. If a bit of the bitmask is set, memory segments of the corresponding memory type are dumped, otherwise they are not dumped.

The following 9 memory types are supported:

- (bit 0) anonymous private memory
- (bit 1) anonymous shared memory
- (bit 2) file-backed private memory

- (bit 3) file-backed shared memory
- (bit 4) ELF header pages in file-backed private memory areas (it is effective only if the bit 2 is cleared)
- (bit 5) hugetlb private memory
- (bit 6) hugetlb shared memory
- (bit 7) DAX private memory
- (bit 8) DAX shared memory

Note that MMIO pages such as frame buffer are never dumped and vDSO pages are always dumped regardless of the bitmask status.

Note that bits 0-4 don't affect hugetlb or DAX memory. hugetlb memory is only affected by bit 5-6, and DAX is only affected by bits 7-8.

The default value of `coredump_filter` is 0x33; this means all anonymous memory segments, ELF header pages and hugetlb private memory are dumped.

If you don't want to dump all shared memory segments attached to pid 1234, write 0x31 to the process's proc file:

```
$ echo 0x31 > /proc/1234/coredump_filter
```

When a new process is created, the process inherits the bitmask status from its parent. It is useful to set up `coredump_filter` before the program runs. For example:

```
$ echo 0x7 > /proc/self/coredump_filter
$ ./some_program
```

3.5 /proc/<pid>/mountinfo - Information about mounts

This file contains lines of the form:

```
36 35 98:0 /mnt1 /mnt2 rw,noatime master:1 - ext3 /dev/root rw,
↪errors=continue
(1)(2)(3) (4) (5) (6) (7) (8) (9) (10) (11)

(1) mount ID: unique identifier of the mount (may be reused after umount)
(2) parent ID: ID of parent (or of self for the top of the mount tree)
(3) major:minor: value of st_dev for files on filesystem
(4) root: root of the mount within the filesystem
(5) mount point: mount point relative to the process's root
(6) mount options: per mount options
(7) optional fields: zero or more fields of the form "tag[:value]"
(8) separator: marks the end of the optional fields
(9) filesystem type: name of filesystem of the form "type[.subtype]"
(10) mount source: filesystem specific information or "none"
(11) super options: per super block options
```

Parsers should ignore all unrecognised optional fields. Currently the possible optional fields are:

shared:X	mount is shared in peer group X
master:X	mount is slave to peer group X
propagate_from:X	mount is slave and receives propagation from peer group X ¹
unbindable	mount is unbindable

For more information on mount propagation see:

Documentation/filesystems/sharedsubtree.rst

3.6 /proc/<pid>/comm & /proc/<pid>/task/<tid>/comm

These files provide a method to access a tasks comm value. It also allows for a task to set its own or one of its thread siblings comm value. The comm value is limited in size compared to the cmdline value, so writing anything longer than the kernel's TASK_COMM_LEN (currently 16 chars) will result in a truncated comm value.

3.7 /proc/<pid>/task/<tid>/children - Information about task children

This file provides a fast way to retrieve first level children pids of a task pointed by <pid>/<tid> pair. The format is a space separated stream of pids.

Note the “first level” here - if a child has own children they will not be listed here, one needs to read /proc/<children-pid>/task/<tid>/children to obtain the descendants.

Since this interface is intended to be fast and cheap it doesn't guarantee to provide precise results and some children might be skipped, especially if they've exited right after we printed their pids, so one need to either stop or freeze processes being inspected if precise results are needed.

3.8 /proc/<pid>/fdinfo/<fd> - Information about opened file

This file provides information associated with an opened file. The regular files have at least three fields - 'pos', 'flags' and mnt_id. The 'pos' represents the current offset of the opened file in decimal form [see lseek(2) for details], 'flags' denotes the octal O_xxx mask the file has been created with [see open(2) for details] and 'mnt_id' represents mount ID of the file system containing the opened file [see 3.5 /proc/<pid>/mountinfo for details].

A typical output is:

```
pos:    0
flags:  0100002
mnt_id: 19
```

All locks associated with a file descriptor are shown in its fdinfo too:

¹ X is the closest dominant peer group under the process's root. If X is the immediate master of the mount, or if there's no dominant peer group under the same root, then only the “master:X” field is present and not the “propagate_from:X” field.

```
lock:      1: FLOCK  ADVISORY  WRITE 359 00:13:11691 0 EOF
```

The files such as `eventfd`, `fsnotify`, `signalfd`, `epoll` among the regular `pos/flags` pair provide additional information particular to the objects they represent.

Eventfd files

```
pos:      0
flags:    04002
mnt_id:   9
eventfd-count: 5a
```

where `'eventfd-count'` is hex value of a counter.

Signalfd files

```
pos:      0
flags:    04002
mnt_id:   9
sigmask:  00000000000000200
```

where `'sigmask'` is hex value of the signal mask associated with a file.

Epoll files

```
pos:      0
flags:    02
mnt_id:   9
tfd:      5 events:      1d data: ffffffff pos:0 ino:61af sdev:7
```

where `'tfd'` is a target file descriptor number in decimal form, `'events'` is events mask being watched and the `'data'` is data associated with a target [see `epoll(7)` for more details].

The `'pos'` is current offset of the target file in decimal form [see `lseek(2)`], `'ino'` and `'sdev'` are inode and device numbers where target file resides, all in hex format.

Fsnotify files

For inotify files the format is the following:

```
pos:      0
flags:    02000000
inotify wd:3 ino:9e7e sdev:800013 mask:800afce ignored_mask:0 fhandle-
↳ bytes:8 fhandle-type:1 f_handle:7e9e0000640d1b6d
```

where `'wd'` is a watch descriptor in decimal form, ie a target file descriptor number, `'ino'` and `'sdev'` are inode and device where the target file resides and the `'mask'` is the mask of events, all in hex form [see `inotify(7)` for more details].

If the kernel was built with `exportfs` support, the path to the target file is encoded as a file handle. The file handle is provided by three fields `'fhandle-bytes'`, `'fhandle-type'` and `'f_handle'`, all in hex format.

If the kernel is built without `exportfs` support the file handle won't be printed out.

If there is no inotify mark attached yet the `'inotify'` line will be omitted.

For fanotify files the format is:

```
pos:      0
flags:    02
mnt_id:   9
fanotify flags:10 event-flags:0
fanotify mnt_id:12 mflags:40 mask:38 ignored_mask:40000003
fanotify ino:4f969 sdev:800013 mflags:0 mask:3b ignored_mask:40000000
↪fhandle-bytes:8 fhandle-type:1 f_handle:69f90400c275b5b4
```

where fanotify `'flags'` and `'event-flags'` are values used in `fanotify_init` call, `'mnt_id'` is the mount point identifier, `'mflags'` is the value of flags associated with mark which are tracked separately from events mask. `'ino'`, `'sdev'` are target inode and device, `'mask'` is the events mask and `'ignored_mask'` is the mask of events which are to be ignored. All in hex format. Incorporation of `'mflags'`, `'mask'` and `'ignored_mask'` does provide information about flags and mask used in `fanotify_mark` call [see `fsnotify` manpage for details].

While the first three lines are mandatory and always printed, the rest is optional and may be omitted if no marks created yet.

Timerfd files

```
pos:      0
flags:    02
mnt_id:   9
clockid:  0
ticks:    0
settime flags: 01
it_value: (0, 49406829)
it_interval: (1, 0)
```

where `'clockid'` is the clock type and `'ticks'` is the number of the timer expirations that have occurred [see `timerfd_create(2)` for details]. `'settime flags'` are flags in octal form been used to setup the timer [see `timerfd_settime(2)` for details]. `'it_value'` is remaining time until the timer expiration. `'it_interval'` is the interval for the timer. Note the timer might be set up with `TIMER_ABSTIME` option which will be shown in `'settime flags'`, but `'it_value'` still exhibits timer's remaining time.

3.9 /proc/<pid>/map_files - Information about memory mapped files

This directory contains symbolic links which represent memory mapped files the process is maintaining. Example output:

```
| lr----- 1 root root 64 Jan 27 11:24 333c600000-333c620000 -> /usr/  
↳ lib64/ld-2.18.so  
| lr----- 1 root root 64 Jan 27 11:24 333c81f000-333c820000 -> /usr/  
↳ lib64/ld-2.18.so  
| lr----- 1 root root 64 Jan 27 11:24 333c820000-333c821000 -> /usr/  
↳ lib64/ld-2.18.so  
| ...  
| lr----- 1 root root 64 Jan 27 11:24 35d0421000-35d0422000 -> /usr/  
↳ lib64/libselinux.so.1  
| lr----- 1 root root 64 Jan 27 11:24 400000-41a000 -> /usr/bin/ls
```

The name of a link represents the virtual memory bounds of a mapping, i.e. `vm_area_struct::vm_start-vm_area_struct::vm_end`.

The main purpose of the `map_files` is to retrieve a set of memory mapped files in a fast way instead of parsing `/proc/<pid>/maps` or `/proc/<pid>/smaps`, both of which contain many more records. At the same time one can open(2) mappings from the listings of two processes and comparing their inode numbers to figure out which anonymous memory areas are actually shared.

3.10 /proc/<pid>/timerslack_ns - Task timerslack value

This file provides the value of the task's timerslack value in nanoseconds. This value specifies a amount of time that normal timers may be deferred in order to coalesce timers and avoid unnecessary wakeups.

This allows a task's interactivity vs power consumption trade off to be adjusted.

Writing 0 to the file will set the tasks timerslack to the default value.

Valid values are from 0 - `ULLONG_MAX`

An application setting the value must have `PTRACE_MODE_ATTACH_FSCREDS` level permissions on the task specified to change its `timerslack_ns` value.

3.11 /proc/<pid>/patch_state - Livepatch patch operation state

When `CONFIG_LIVEPATCH` is enabled, this file displays the value of the patch state for the task.

A value of `'-1'` indicates that no patch is in transition.

A value of `'0'` indicates that a patch is in transition and the task is unpatched. If the patch is being enabled, then the task hasn't been patched yet. If the patch is being disabled, then the task has already been unpatched.

A value of `'1'` indicates that a patch is in transition and the task is patched. If the patch is being enabled, then the task has already been patched. If the patch is being disabled, then the task hasn't been unpatched yet.

3.12 /proc/<pid>/arch_status - task architecture specific status

When CONFIG_PROC_PID_ARCH_STATUS is enabled, this file displays the architecture specific status of the task.

Example

```
$ cat /proc/6753/arch_status
AVX512_elapsed_ms:      8
```

Description

x86 specific entries:

AVX512_elapsed_ms:

If AVX512 is supported on the machine, this entry shows the milliseconds elapsed since the last time AVX512 usage was recorded. The recording happens on a best effort basis when a task is scheduled out. This means that the value depends on two factors:

- 1) The time which the task spent on the CPU without being scheduled out. With CPU isolation and a single runnable task this can take several seconds.
- 2) The time since the task was scheduled out last. Depending on the reason for being scheduled out (time slice exhausted, syscall ...) this can be arbitrary long time.

As a consequence the value cannot be considered precise and authoritative information. The application which uses this information has to be aware of the overall scenario on the system in order to determine whether a task is a real AVX512 user or not. Precise information can be obtained with performance counters.

A special value of '-1' indicates that no AVX512 usage was recorded, thus the task is unlikely an AVX512 user, but depends on the workload and the scheduling scenario, it also could be a false negative mentioned above.

Configuring procfs

4.1 Mount options

The following mount options are supported:

hidepid=	Set /proc/<pid>/ access mode.
gid=	Set the group authorized to learn processes information.
subset=	Show only the specified subset of procfs.

hidepid=off or hidepid=0 means classic mode - everybody may access all /proc/<pid>/ directories (default).

hidepid=noaccess or hidepid=1 means users may not access any /proc/<pid>/ directories but their own. Sensitive files like cmdline, sched*, status are now protected against other users. This makes it impossible to learn whether any user runs specific program (given the program doesn't reveal itself by its behaviour). As an additional bonus, as /proc/<pid>/cmdline is unaccessible for other users, poorly written programs passing sensitive information via program arguments are now protected against local eavesdroppers.

hidepid=invisible or hidepid=2 means hidepid=1 plus all /proc/<pid>/ will be fully invisible to other users. It doesn't mean that it hides a fact whether a process with a specific pid value exists (it can be learned by other means, e.g. by "kill -0 \$PID"), but it hides process' uid and gid, which may be learned by stat()' ing /proc/<pid>/ otherwise. It greatly complicates an intruder' s task of gathering information about running processes, whether some daemon runs with elevated privileges, whether other user runs some sensitive program, whether other users run any program at all, etc.

hidepid=ptraceable or hidepid=4 means that procfs should only contain /proc/<pid>/ directories that the caller can ptrace.

gid= defines a group authorized to learn processes information otherwise prohibited by hidepid=. If you use some daemon like identd which needs to learn information about processes information, just add identd to this group.

subset=pid hides all top level files and directories in the procfs that are not related to tasks.

5 Filesystem behavior

Originally, before the advent of pid namespace, procfs was a global file system. It means that there was only one procfs instance in the system.

When pid namespace was added, a separate procfs instance was mounted in each pid namespace. So, procfs mount options are global among all mountpoints within the same namespace.

```
# grep ^proc /proc/mounts
```

```
proc /proc proc rw,relatime,hidepid=2 0 0
```

```
# strace -e mount mount -o hidepid=1 -t proc proc /tmp/proc mount( "proc" ,  
"/tmp/proc" , "proc" , 0, "hidepid=1" ) = 0 +++ exited with 0 +++
```

```
# grep ^proc /proc/mounts proc /proc proc rw,relatime,hidepid=2 0 0 proc  
/tmp/proc proc rw,relatime,hidepid=2 0 0
```

and only after remounting procfs mount options will change at all mountpoints.

```
# mount -o remount,hidepid=1 -t proc proc /tmp/proc
```

```
# grep ^proc /proc/mounts proc /proc proc rw,relatime,hidepid=1 0 0 proc  
/tmp/proc proc rw,relatime,hidepid=1 0 0
```

This behavior is different from the behavior of other filesystems.

The new procfs behavior is more like other filesystems. Each procfs mount creates a new procfs instance. Mount options affect own procfs instance. It means that it became possible to have several procfs instances displaying tasks with different filtering options in one pid namespace.

```
# mount -o hidepid=invisible -t proc proc /proc # mount -o hide-  
pid=noaccess -t proc proc /tmp/proc # grep ^proc /proc/mounts  
proc /proc proc rw,relatime,hidepid=invisible 0 0 proc /tmp/proc proc  
rw,relatime,hidepid=noaccess 0 0
```

3.45 The QNX6 Filesystem

The qnx6fs is used by newer QNX operating system versions. (e.g. Neutrino) It got introduced in QNX 6.4.0 and is used default since 6.4.1.

3.45.1 Option

mmi_fs Mount filesystem as used for example by Audi MMI 3G system

3.45.2 Specification

qnx6fs shares many properties with traditional Unix filesystems. It has the concepts of blocks, inodes and directories.

On QNX it is possible to create little endian and big endian qnx6 filesystems. This feature makes it possible to create and use a different endianness fs for the target (QNX is used on quite a range of embedded systems) platform running on a different endianness.

The Linux driver handles endianness transparently. (LE and BE)

Blocks

The space in the device or file is split up into blocks. These are a fixed size of 512, 1024, 2048 or 4096, which is decided when the filesystem is created.

Blockpointers are 32bit, so the maximum space that can be addressed is $2^{32} * 4096$ bytes or 16TB

The superblocks

The superblock contains all global information about the filesystem. Each qnx6fs got two superblocks, each one having a 64bit serial number. That serial number is used to identify the “active” superblock. In write mode with reach new snapshot (after each synchronous write), the serial of the new master superblock is increased (old superblock serial + 1)

So basically the snapshot functionality is realized by an atomic final update of the serial number. Before updating that serial, all modifications are done by copying

all modified blocks during that specific write request (or period) and building up a new (stable) filesystem structure under the inactive superblock.

Each superblock holds a set of root inodes for the different filesystem parts. (Inode, Bitmap and Longfilenames) Each of these root nodes holds information like total size of the stored data and the addressing levels in that specific tree. If the level value is 0, up to 16 direct blocks can be addressed by each node.

Level 1 adds an additional indirect addressing level where each indirect addressing block holds up to $\text{blocksize} / 4$ bytes pointers to data blocks. Level 2 adds an additional indirect addressing block level (so, already up to $16 * 256 * 256 = 1048576$ blocks that can be addressed by such a tree).

Unused block pointers are always set to ~ 0 - regardless of root node, indirect addressing blocks or inodes.

Data leaves are always on the lowest level. So no data is stored on upper tree levels.

The first Superblock is located at 0x2000. (0x2000 is the bootblock size) The Audi MMI 3G first superblock directly starts at byte 0.

Second superblock position can either be calculated from the superblock information (total number of filesystem blocks) or by taking the highest device address, zeroing the last 3 bytes and then subtracting 0x1000 from that address.

0x1000 is the size reserved for each superblock - regardless of the blocksize of the filesystem.

Inodes

Each object in the filesystem is represented by an inode. (index node) The inode structure contains pointers to the filesystem blocks which contain the data held in the object and all of the metadata about an object except its longname. (filenames longer than 27 characters) The metadata about an object includes the permissions, owner, group, flags, size, number of blocks used, access time, change time and modification time.

Object mode field is POSIX format. (which makes things easier)

There are also pointers to the first 16 blocks, if the object data can be addressed with 16 direct blocks.

For more than 16 blocks an indirect addressing in form of another tree is used. (scheme is the same as the one used for the superblock root nodes)

The filesize is stored 64bit. Inode counting starts with 1. (while long filename inodes start with 0)

Directories

A directory is a filesystem object and has an inode just like a file. It is a specially formatted file containing records which associate each name with an inode number.

‘.’ inode number points to the directory inode

‘..’ inode number points to the parent directory inode

Each filename record additionally got a filename length field.

One special case are long filenames or subdirectory names.

These got set a filename length field of 0xff in the corresponding directory record plus the longfile inode number also stored in that record.

With that longfilename inode number, the longfilename tree can be walked starting with the superblock longfilename root node pointers.

Special files

Symbolic links are also filesystem objects with inodes. They got a specific bit in the inode mode field identifying them as symbolic link.

The directory entry file inode pointer points to the target file inode.

Hard links got an inode, a directory entry, but a specific mode bit set, no block pointers and the directory file record pointing to the target file inode.

Character and block special devices do not exist in QNX as those files are handled by the QNX kernel/drivers and created in /dev independent of the underlying filesystem.

Long filenames

Long filenames are stored in a separate addressing tree. The starting point is the longfilename root node in the active superblock.

Each data block (tree leaves) holds one long filename. That filename is limited to 510 bytes. The first two starting bytes are used as length field for the actual filename.

If that structure shall fit for all allowed blocksizes, it is clear why there is a limit of 510 bytes for the actual filename stored.

Bitmap

The qnx6fs filesystem allocation bitmap is stored in a tree under bitmap root node in the superblock and each bit in the bitmap represents one filesystem block.

The first block is block 0, which starts 0x1000 after superblock start. So for a normal qnx6fs 0x3000 (bootblock + superblock) is the physical address at which block 0 is located.

Bits at the end of the last bitmap block are set to 1, if the device is smaller than addressing space in the bitmap.

Bitmap system area

The bitmap itself is divided into three parts.

First the system area, that is split into two halves.

Then userspace.

The requirement for a static, fixed preallocated system area comes from how qnx6fs deals with writes.

Each superblock got it's own half of the system area. So superblock #1 always uses blocks from the lower half while superblock #2 just writes to blocks represented by the upper half bitmap system area bits.

Bitmap blocks, Inode blocks and indirect addressing blocks for those two tree structures are treated as system blocks.

The rational behind that is that a write request can work on a new snapshot (system area of the inactive - resp. lower serial numbered superblock) while at the same time there is still a complete stable filesystem structure in the other half of the system area.

When finished with writing (a sync write is completed, the maximum sync leap time or a filesystem sync is requested), serial of the previously inactive superblock atomically is increased and the fs switches over to that - then stable declared - superblock.

For all data outside the system area, blocks are just copied while writing.

3.46 Ramfs, rootfs and initramfs

October 17, 2005

3.46.1 Rob Landley <rob@landley.net>

What is ramfs?

Ramfs is a very simple filesystem that exports Linux' s disk caching mechanisms (the page cache and dentry cache) as a dynamically resizable RAM-based filesystem.

Normally all files are cached in memory by Linux. Pages of data read from backing store (usually the block device the filesystem is mounted on) are kept around in case it' s needed again, but marked as clean (freeable) in case the Virtual Memory system needs the memory for something else. Similarly, data written to files is marked clean as soon as it has been written to backing store, but kept around for caching purposes until the VM reallocates the memory. A similar mechanism (the dentry cache) greatly speeds up access to directories.

With ramfs, there is no backing store. Files written into ramfs allocate dentries and page cache as usual, but there' s nowhere to write them to. This means the pages are never marked clean, so they can' t be freed by the VM when it' s looking to recycle memory.

The amount of code required to implement ramfs is tiny, because all the work is done by the existing Linux caching infrastructure. Basically, you' re mounting the disk cache as a filesystem. Because of this, ramfs is not an optional component removable via menuconfig, since there would be negligible space savings.

ramfs and ramdisk:

The older “ram disk” mechanism created a synthetic block device out of an area of RAM and used it as backing store for a filesystem. This block device was of fixed size, so the filesystem mounted on it was of fixed size. Using a ram disk also required unnecessarily copying memory from the fake block device into the page cache (and copying changes back out), as well as creating and destroying dentries. Plus it needed a filesystem driver (such as ext2) to format and interpret this data.

Compared to ramfs, this wastes memory (and memory bus bandwidth), creates unnecessary work for the CPU, and pollutes the CPU caches. (There are tricks to avoid this copying by playing with the page tables, but they' re unpleasantly complicated and turn out to be about as expensive as the copying anyway.) More to the point, all the work ramfs is doing has to happen *anyway*, since all file access goes through the page and dentry caches. The RAM disk is simply unnecessary; ramfs is internally much simpler.

Another reason ramdisks are semi-obsolete is that the introduction of loopback devices offered a more flexible and convenient way to create synthetic block devices, now from files instead of from chunks of memory. See `losetup (8)` for details.

ramfs and tmpfs:

One downside of ramfs is you can keep writing data into it until you fill up all memory, and the VM can't free it because the VM thinks that files should get written to backing store (rather than swap space), but ramfs hasn't got any backing store. Because of this, only root (or a trusted user) should be allowed write access to a ramfs mount.

A ramfs derivative called tmpfs was created to add size limits, and the ability to write the data to swap space. Normal users can be allowed write access to tmpfs mounts. See [Documentation/filesystems/tmpfs.rst](#) for more information.

What is rootfs?

Rootfs is a special instance of ramfs (or tmpfs, if that's enabled), which is always present in 2.6 systems. You can't unmount rootfs for approximately the same reason you can't kill the init process; rather than having special code to check for and handle an empty list, it's smaller and simpler for the kernel to just make sure certain lists can't become empty.

Most systems just mount another filesystem over rootfs and ignore it. The amount of space an empty instance of ramfs takes up is tiny.

If `CONFIG_TMPFS` is enabled, rootfs will use tmpfs instead of ramfs by default. To force ramfs, add `"rootfstype=ramfs"` to the kernel command line.

What is initramfs?

All 2.6 Linux kernels contain a gzipped "cpio" format archive, which is extracted into rootfs when the kernel boots up. After extracting, the kernel checks to see if rootfs contains a file "init", and if so it executes it as PID 1. If found, this init process is responsible for bringing the system the rest of the way up, including locating and mounting the real root device (if any). If rootfs does not contain an init program after the embedded cpio archive is extracted into it, the kernel will fall through to the older code to locate and mount a root partition, then exec some variant of `/sbin/init` out of that.

All this differs from the old `initrd` in several ways:

- The old `initrd` was always a separate file, while the `initramfs` archive is linked into the linux kernel image. (The directory `linux-*/usr` is devoted to generating this archive during the build.)
- The old `initrd` file was a gzipped filesystem image (in some file format, such as `ext2`, that needed a driver built into the kernel), while the new `initramfs` archive is a gzipped cpio archive (like `tar` only simpler, see `cpio(1)` and [Documentation/driver-api/early-userspace/buffer-format.rst](#)). The kernel's cpio extraction code is not only extremely small, it's also `__init` text and data that can be discarded during the boot process.
- The program run by the old `initrd` (which was called `/initrd`, not `/init`) did some setup and then returned to the kernel, while the `init` program from `initramfs` is not expected to return to the kernel. (If `/init` needs to hand off control it

can overmount / with a new root device and exec another init program. See the `switch_root` utility, below.)

- When switching another root device, `initrd` would `pivot_root` and then unmount the ramdisk. But `initramfs` is `rootfs`: you can neither `pivot_root` `rootfs`, nor unmount it. Instead delete everything out of `rootfs` to free up the space (find `-xdev / -exec rm '{ } ;'`), overmount `rootfs` with the new root (`cd /newmount; mount -move . /; chroot .`), attach `stdin/stdout/stderr` to the new `/dev/console`, and exec the new `init`.

Since this is a remarkably persnickety process (and involves deleting commands before you can run them), the `klibc` package introduced a helper program (`utils/run_init.c`) to do all this for you. Most other packages (such as `busybox`) have named this command “`switch_root`” .

Populating `initramfs`:

The 2.6 kernel build process always creates a gzipped `cpio` format `initramfs` archive and links it into the resulting kernel binary. By default, this archive is empty (consuming 134 bytes on x86).

The config option `CONFIG_INITRAMFS_SOURCE` (in General Setup in `menuconfig`, and living in `usr/Kconfig`) can be used to specify a source for the `initramfs` archive, which will automatically be incorporated into the resulting binary. This option can point to an existing gzipped `cpio` archive, a directory containing files to be archived, or a text file specification such as the following example:

```
dir /dev 755 0 0
nod /dev/console 644 0 0 c 5 1
nod /dev/loop0 644 0 0 b 7 0
dir /bin 755 1000 1000
slink /bin/sh busybox 777 0 0
file /bin/busybox initramfs/busybox 755 0 0
dir /proc 755 0 0
dir /sys 755 0 0
dir /mnt 755 0 0
file /init initramfs/init.sh 755 0 0
```

Run “`usr/gen_init_cpio`” (after the kernel build) to get a usage message documenting the above file format.

One advantage of the configuration file is that root access is not required to set permissions or create device nodes in the new archive. (Note that those two example “file” entries expect to find files named “`init.sh`” and “`busybox`” in a directory called “`initramfs`” , under the `linux-2.6.*` directory. See [Documentation/driver-api/early-userspace/early_userspace_support.rst](#) for more details.)

The kernel does not depend on external `cpio` tools. If you specify a directory instead of a configuration file, the kernel’s build infrastructure creates a configuration file from that directory (`usr/Makefile` calls `usr/gen_initramfs_list.sh`), and proceeds to package up that directory using the config file (by feeding it to `usr/gen_init_cpio`, which is created from `usr/gen_init_cpio.c`). The kernel’s build-time `cpio` creation code is entirely self-contained, and the kernel’s boot-time extractor is also (obviously) self-contained.

The one thing you might need external cpio utilities installed for is creating or extracting your own preprepared cpio files to feed to the kernel build (instead of a config file or directory).

The following command line can extract a cpio image (either by the above script or by the kernel build) back into its component files:

```
cpio -i -d -H newc -F initramfs_data.cpio --no-absolute-filenames
```

The following shell script can create a prebuilt cpio archive you can use in place of the above config file:

```
#!/bin/sh

# Copyright 2006 Rob Landley <rob@landley.net> and TimeSys Corporation.
# Licensed under GPL version 2

if [ $# -ne 2 ]
then
    echo "usage: mkinitramfs directory imagename.cpio.gz"
    exit 1
fi

if [ -d "$1" ]
then
    echo "creating $2 from $1"
    (cd "$1"; find . | cpio -o -H newc | gzip) > "$2"
else
    echo "First argument must be a directory"
    exit 1
fi
```

Note: The cpio man page contains some bad advice that will break your initramfs archive if you follow it. It says “A typical way to generate the list of filenames is with the find command; you should give find the -depth option to minimize problems with permissions on directories that are unwritable or not searchable.” Don’ t do this when creating initramfs.cpio.gz images, it won’ t work. The Linux kernel cpio extractor won’ t create files in a directory that doesn’ t exist, so the directory entries must go before the files that go in those directories. The above script gets them in the right order.

External initramfs images:

If the kernel has initrd support enabled, an external cpio.gz archive can also be passed into a 2.6 kernel in place of an initrd. In this case, the kernel will autodetect the type (initramfs, not initrd) and extract the external cpio archive into rootfs before trying to run /init.

This has the memory efficiency advantages of initramfs (no ramdisk block device) but the separate packaging of initrd (which is nice if you have non-GPL code you’ d like to run from initramfs, without conflating it with the GPL licensed Linux kernel binary).

It can also be used to supplement the kernel's built-in initramfs image. The files in the external archive will overwrite any conflicting files in the built-in initramfs archive. Some distributors also prefer to customize a single kernel image with task-specific initramfs images, without recompiling.

Contents of initramfs:

An initramfs archive is a complete self-contained root filesystem for Linux. If you don't already understand what shared libraries, devices, and paths you need to get a minimal root filesystem up and running, here are some references:

- <http://www.tldp.org/HOWTO/Bootdisk-HOWTO/>
- <http://www.tldp.org/HOWTO/From-PowerUp-To-Bash-Prompt-HOWTO.html>
- <http://www.linuxfromscratch.org/lfs/view/stable/>

The “klibc” package (<http://www.kernel.org/pub/linux/libs/klibc>) is designed to be a tiny C library to statically link early userspace code against, along with some related utilities. It is BSD licensed.

I use uClibc (<http://www.uclibc.org>) and busybox (<http://www.busybox.net>) myself. These are LGPL and GPL, respectively. (A self-contained initramfs package is planned for the busybox 1.3 release.)

In theory you could use glibc, but that's not well suited for small embedded uses like this. (A “hello world” program statically linked against glibc is over 400k. With uClibc it's 7k. Also note that glibc dlopens libnss to do name lookups, even when otherwise statically linked.)

A good first step is to get initramfs to run a statically linked “hello world” program as init, and test it under an emulator like qemu (www.qemu.org) or User Mode Linux, like so:

```
cat > hello.c << EOF
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    sleep(999999999);
}
EOF
gcc -static hello.c -o init
echo init | cpio -o -H newc | gzip > test.cpio.gz
# Testing external initramfs using the initrd loading mechanism.
qemu -kernel /boot/vmlinuz -initrd test.cpio.gz /dev/zero
```

When debugging a normal root filesystem, it's nice to be able to boot with “init=/bin/sh”. The initramfs equivalent is “rdinit=/bin/sh”, and it's just as useful.

Why cpio rather than tar?

This decision was made back in December, 2001. The discussion started here:

<http://www.uwsg.iu.edu/hypertext/linux/kernel/0112.2/1538.html>

And spawned a second thread (specifically on tar vs cpio), starting here:

<http://www.uwsg.iu.edu/hypertext/linux/kernel/0112.2/1587.html>

The quick and dirty summary version (which is no substitute for reading the above threads) is:

- 1) cpio is a standard. It's decades old (from the AT&T days), and already widely used on Linux (inside RPM, Red Hat's device driver disks). Here's a Linux Journal article about it from 1996:

<http://www.linuxjournal.com/article/1213>

It's not as popular as tar because the traditional cpio command line tools require `_truly_hideous_` command line arguments. But that says nothing either way about the archive format, and there are alternative tools, such as:

<http://freecode.com/projects/afio>

- 2) The cpio archive format chosen by the kernel is simpler and cleaner (and thus easier to create and parse) than any of the (literally dozens of) various tar archive formats. The complete initramfs archive format is explained in `buffer-format.txt`, created in `usr/gen_init_cpio.c`, and extracted in `init/initramfs.c`. All three together come to less than 26k total of human-readable text.
- 3) The GNU project standardizing on tar is approximately as relevant as Windows standardizing on zip. Linux is not part of either, and is free to make its own technical decisions.
- 4) Since this is a kernel internal format, it could easily have been something brand new. The kernel provides its own tools to create and extract this format anyway. Using an existing standard was preferable, but not essential.
- 5) Al Viro made the decision (quote: "tar is ugly as hell and not going to be supported on the kernel side"):

<http://www.uwsg.iu.edu/hypertext/linux/kernel/0112.2/1540.html>

explained his reasoning:

- <http://www.uwsg.iu.edu/hypertext/linux/kernel/0112.2/1550.html>
- <http://www.uwsg.iu.edu/hypertext/linux/kernel/0112.2/1638.html>

and, most importantly, designed and implemented the initramfs code.

Future directions:

Today (2.6.16), `initramfs` is always compiled in, but not always used. The kernel falls back to legacy boot code that is reached only if `initramfs` does not contain an `/init` program. The fallback is legacy code, there to ensure a smooth transition and allowing early boot functionality to gradually move to “early userspace” (I.E. `initramfs`).

The move to early userspace is necessary because finding and mounting the real root device is complex. Root partitions can span multiple devices (raid or separate journal). They can be out on the network (requiring `dhcp`, setting a specific MAC address, logging into a server, etc). They can live on removable media, with dynamically allocated major/minor numbers and persistent naming issues requiring a full `udev` implementation to sort out. They can be compressed, encrypted, copy-on-write, loopback mounted, strangely partitioned, and so on.

This kind of complexity (which inevitably includes policy) is rightly handled in userspace. Both `klibc` and `busybox/uClibc` are working on simple `initramfs` packages to drop into a kernel build.

The `klibc` package has now been accepted into Andrew Morton’s 2.6.17-mm tree. The kernel’s current early boot code (partition detection, etc) will probably be migrated into a default `initramfs`, automatically created and used by the kernel build.

3.47 relay interface (formerly relays)

The relay interface provides a means for kernel applications to efficiently log and transfer large quantities of data from the kernel to userspace via user-defined ‘relay channels’.

A ‘relay channel’ is a kernel->user data relay mechanism implemented as a set of per-cpu kernel buffers (‘channel buffers’), each represented as a regular file (‘relay file’) in user space. Kernel clients write into the channel buffers using efficient write functions; these automatically log into the current cpu’s channel buffer. User space applications `mmap()` or `read()` from the relay files and retrieve the data as it becomes available. The relay files themselves are files created in a host filesystem, e.g. `debugfs`, and are associated with the channel buffers using the API described below.

The format of the data logged into the channel buffers is completely up to the kernel client; the relay interface does however provide hooks which allow kernel clients to impose some structure on the buffer data. The relay interface doesn’t implement any form of data filtering - this also is left to the kernel client. The purpose is to keep things as simple as possible.

This document provides an overview of the relay interface API. The details of the function parameters are documented along with the functions in the relay interface code - please see that for details.

3.47.1 Semantics

Each relay channel has one buffer per CPU, each buffer has one or more sub-buffers. Messages are written to the first sub-buffer until it is too full to contain a new message, in which case it is written to the next (if available). Messages are never split across sub-buffers. At this point, userspace can be notified so it empties the first sub-buffer, while the kernel continues writing to the next.

When notified that a sub-buffer is full, the kernel knows how many bytes of it are padding i.e. unused space occurring because a complete message couldn't fit into a sub-buffer. Userspace can use this knowledge to copy only valid data.

After copying it, userspace can notify the kernel that a sub-buffer has been consumed.

A relay channel can operate in a mode where it will overwrite data not yet collected by userspace, and not wait for it to be consumed.

The relay channel itself does not provide for communication of such data between userspace and kernel, allowing the kernel side to remain simple and not impose a single interface on userspace. It does provide a set of examples and a separate helper though, described below.

The `read()` interface both removes padding and internally consumes the read sub-buffers; thus in cases where `read(2)` is being used to drain the channel buffers, special-purpose communication between kernel and user isn't necessary for basic operation.

One of the major goals of the relay interface is to provide a low overhead mechanism for conveying kernel data to userspace. While the `read()` interface is easy to use, it's not as efficient as the `mmap()` approach; the example code attempts to make the tradeoff between the two approaches as small as possible.

3.47.2 klog and relay-apps example code

The relay interface itself is ready to use, but to make things easier, a couple simple utility functions and a set of examples are provided.

The relay-apps example tarball, available on the relay sourceforge site, contains a set of self-contained examples, each consisting of a pair of `.c` files containing boilerplate code for each of the user and kernel sides of a relay application. When combined these two sets of boilerplate code provide glue to easily stream data to disk, without having to bother with mundane housekeeping chores.

The 'klog debugging functions' patch (`klog.patch` in the relay-apps tarball) provides a couple of high-level logging functions to the kernel which allow writing formatted text or raw data to a channel, regardless of whether a channel to write into exists or not, or even whether the relay interface is compiled into the kernel or not. These functions allow you to put unconditional 'trace' statements anywhere in the kernel or kernel modules; only when there is a 'klog handler' registered will data actually be logged (see the klog and kleak examples for details).

It is of course possible to use the relay interface from scratch, i.e. without using any of the relay-apps example code or klog, but you'll have to implement communication between userspace and kernel, allowing both to convey the state of buffers

(full, empty, amount of padding). The read() interface both removes padding and internally consumes the read sub-buffers; thus in cases where read(2) is being used to drain the channel buffers, special-purpose communication between kernel and user isn't necessary for basic operation. Things such as buffer-full conditions would still need to be communicated via some channel though.

klog and the relay-apps examples can be found in the relay-apps tarball on <http://relayfs.sourceforge.net>

3.47.3 The relay interface user space API

The relay interface implements basic file operations for user space access to relay channel buffer data. Here are the file operations that are available and some comments regarding their behavior:

open()	enables user to open an <code>_existing_</code> channel buffer.
mmap()	results in channel buffer being mapped into the caller's memory space. Note that you can't do a partial mmap - you must map the entire file, which is <code>NRBUF * SUBBUFSIZE</code> .
read()	read the contents of a channel buffer. The bytes read are 'consumed' by the reader, i.e. they won't be available again to subsequent reads. If the channel is being used in no-overwrite mode (the default), it can be read at any time even if there's an active kernel writer. If the channel is being used in overwrite mode and there are active channel writers, results may be unpredictable - users should make sure that all logging to the channel has ended before using read() with overwrite mode. Sub-buffer padding is automatically removed and will not be seen by the reader.
sendfile()	transfer data from a channel buffer to an output file descriptor. Sub-buffer padding is automatically removed and will not be seen by the reader.
poll()	POLLIN/POLLRDNORM/POLLERR supported. User applications are notified when sub-buffer boundaries are crossed.
close()	decrements the channel buffer's refcount. When the refcount reaches 0, i.e. when no process or kernel client has the buffer open, the channel buffer is freed.

In order for a user application to make use of relay files, the host filesystem must be mounted. For example:

```
mount -t debugfs debugfs /sys/kernel/debug
```

Note: the host filesystem doesn't need to be mounted for kernel clients to create or use channels - it only needs to be mounted when user space applications need access to the buffer data.

3.47.4 The relay interface kernel API

Here's a summary of the API the relay interface provides to in-kernel clients:

TBD(curr. line MT:/API/) channel management functions:

```
relay_open(base_filename, parent, subbuf_size, n_subbufs,
           callbacks, private_data)
relay_close(chan)
relay_flush(chan)
relay_reset(chan)
```

channel management typically called on instigation of userspace:

```
relay_subbufs_consumed(chan, cpu, subbufs_consumed)
```

write functions:

```
relay_write(chan, data, length)
__relay_write(chan, data, length)
relay_reserve(chan, length)
```

callbacks:

```
subbuf_start(buf, subbuf, prev_subbuf, prev_padding)
buf_mapped(buf, filp)
buf_unmapped(buf, filp)
create_buf_file(filename, parent, mode, buf, is_global)
remove_buf_file(dentry)
```

helper functions:

```
relay_buf_full(buf)
subbuf_start_reserve(buf, length)
```

Creating a channel

`relay_open()` is used to create a channel, along with its per-cpu channel buffers. Each channel buffer will have an associated file created for it in the host filesystem, which can be mapped or read from in user space. The files are named `base-name0...basenameN-1` where `N` is the number of online cpus, and by default will be created in the root of the filesystem (if the parent param is `NULL`). If you want a directory structure to contain your relay files, you should create it using the host filesystem's directory creation function, e.g. `debugfs_create_dir()`, and pass the parent directory to `relay_open()`. Users are responsible for cleaning up any directory structure they create, when the channel is closed - again the host filesystem's directory removal functions should be used for that, e.g. `debugfs_remove()`.

In order for a channel to be created and the host filesystem's files associated with its channel buffers, the user must provide definitions for two callback functions, `create_buf_file()` and `remove_buf_file()`. `create_buf_file()` is called once for each per-cpu buffer from `relay_open()` and allows the user to create the file which will be used to represent the corresponding channel buffer. The callback should return the dentry of the file created to represent the channel buffer. `remove_buf_file()`

must also be defined; it's responsible for deleting the file(s) created in `create_buf_file()` and is called during `relay_close()`.

Here are some typical definitions for these callbacks, in this case using `debugfs`:

```

/*
 * create_buf_file() callback.  Creates relay file in debugfs.
 */
static struct dentry *create_buf_file_handler(const char *filename,
                                             struct dentry *parent,
                                             umode_t mode,
                                             struct rchan_buf *buf,
                                             int *is_global)
{
    return debugfs_create_file(filename, mode, parent, buf,
                              &relay_file_operations);
}

/*
 * remove_buf_file() callback.  Removes relay file from debugfs.
 */
static int remove_buf_file_handler(struct dentry *dentry)
{
    debugfs_remove(dentry);

    return 0;
}

/*
 * relay interface callbacks
 */
static struct rchan_callbacks relay_callbacks =
{
    .create_buf_file = create_buf_file_handler,
    .remove_buf_file = remove_buf_file_handler,
};

```

And an example `relay_open()` invocation using them:

```

chan = relay_open("cpu", NULL, SUBBUF_SIZE, N_SUBBUFS, &relay_callbacks,
↳NULL);

```

If the `create_buf_file()` callback fails, or isn't defined, channel creation and thus `relay_open()` will fail.

The total size of each per-cpu buffer is calculated by multiplying the number of sub-buffers by the sub-buffer size passed into `relay_open()`. The idea behind sub-buffers is that they're basically an extension of double-buffering to N buffers, and they also allow applications to easily implement random-access-on-buffer-boundary schemes, which can be important for some high-volume applications. The number and size of sub-buffers is completely dependent on the application and even for the same application, different conditions will warrant different values for these parameters at different times. Typically, the right values to use are best decided after some experimentation; in general, though, it's safe to assume that having only 1 sub-buffer is a bad idea - you're guaranteed to either overwrite data or lose events depending on the channel mode being used.

The `create_buf_file()` implementation can also be defined in such a way as to allow the creation of a single ‘global’ buffer instead of the default per-cpu set. This can be useful for applications interested mainly in seeing the relative ordering of system-wide events without the need to bother with saving explicit timestamps for the purpose of merging/sorting per-cpu files in a postprocessing step.

To have `relay_open()` create a global buffer, the `create_buf_file()` implementation should set the value of the `is_global` outparam to a non-zero value in addition to creating the file that will be used to represent the single buffer. In the case of a global buffer, `create_buf_file()` and `remove_buf_file()` will be called only once. The normal channel-writing functions, e.g. `relay_write()`, can still be used - writes from any cpu will transparently end up in the global buffer - but since it is a global buffer, callers should make sure they use the proper locking for such a buffer, either by wrapping writes in a spinlock, or by copying a write function from `relay.h` and creating a local version that internally does the proper locking.

The `private_data` passed into `relay_open()` allows clients to associate user-defined data with a channel, and is immediately available (including in `create_buf_file()`) via `chan->private_data` or `buf->chan->private_data`.

Buffer-only channels

These channels have no files associated and can be created with `relay_open(NULL, NULL, ...)`. Such channels are useful in scenarios such as when doing early tracing in the kernel, before the VFS is up. In these cases, one may open a buffer-only channel and then call `relay_late_setup_files()` when the kernel is ready to handle files, to expose the buffered data to the userspace.

Channel ‘modes’

relay channels can be used in either of two modes - ‘overwrite’ or ‘no-overwrite’. The mode is entirely determined by the implementation of the `subbuf_start()` callback, as described below. The default if no `subbuf_start()` callback is defined is ‘no-overwrite’ mode. If the default mode suits your needs, and you plan to use the `read()` interface to retrieve channel data, you can ignore the details of this section, as it pertains mainly to `mmap()` implementations.

In ‘overwrite’ mode, also known as ‘flight recorder’ mode, writes continuously cycle around the buffer and will never fail, but will unconditionally overwrite old data regardless of whether it’s actually been consumed. In no-overwrite mode, writes will fail, i.e. data will be lost, if the number of unconsumed sub-buffers equals the total number of sub-buffers in the channel. It should be clear that if there is no consumer or if the consumer can’t consume sub-buffers fast enough, data will be lost in either case; the only difference is whether data is lost from the beginning or the end of a buffer.

As explained above, a relay channel is made of up one or more per-cpu channel buffers, each implemented as a circular buffer subdivided into one or more sub-buffers. Messages are written into the current sub-buffer of the channel’s current per-cpu buffer via the write functions described below. Whenever a message can’t fit into the current sub-buffer, because there’s no room left for it, the client is notified via the `subbuf_start()` callback that a switch to a new sub-buffer is about to

occur. The client uses this callback to 1) initialize the next sub-buffer if appropriate 2) finalize the previous sub-buffer if appropriate and 3) return a boolean value indicating whether or not to actually move on to the next sub-buffer.

To implement ‘no-overwrite’ mode, the userspace client would provide an implementation of the `subbuf_start()` callback something like the following:

```
static int subbuf_start(struct rchan_buf *buf,
                       void *subbuf,
                       void *prev_subbuf,
                       unsigned int prev_padding)
{
    if (prev_subbuf)
        *((unsigned *)prev_subbuf) = prev_padding;

    if (relay_buf_full(buf))
        return 0;

    subbuf_start_reserve(buf, sizeof(unsigned int));

    return 1;
}
```

If the current buffer is full, i.e. all sub-buffers remain unconsumed, the callback returns 0 to indicate that the buffer switch should not occur yet, i.e. until the consumer has had a chance to read the current set of ready sub-buffers. For the `relay_buf_full()` function to make sense, the consumer is responsible for notifying the relay interface when sub-buffers have been consumed via `relay_subbufs_consumed()`. Any subsequent attempts to write into the buffer will again invoke the `subbuf_start()` callback with the same parameters; only when the consumer has consumed one or more of the ready sub-buffers will `relay_buf_full()` return 0, in which case the buffer switch can continue.

The implementation of the `subbuf_start()` callback for ‘overwrite’ mode would be very similar:

```
static int subbuf_start(struct rchan_buf *buf,
                       void *subbuf,
                       void *prev_subbuf,
                       size_t prev_padding)
{
    if (prev_subbuf)
        *((unsigned *)prev_subbuf) = prev_padding;

    subbuf_start_reserve(buf, sizeof(unsigned int));

    return 1;
}
```

In this case, the `relay_buf_full()` check is meaningless and the callback always returns 1, causing the buffer switch to occur unconditionally. It’s also meaningless for the client to use the `relay_subbufs_consumed()` function in this mode, as it’s never consulted.

The default `subbuf_start()` implementation, used if the client doesn’t define any callbacks, or doesn’t define the `subbuf_start()` callback, implements the simplest

possible ‘no-overwrite’ mode, i.e. it does nothing but return 0.

Header information can be reserved at the beginning of each sub-buffer by calling the `subbuf_start_reserve()` helper function from within the `subbuf_start()` callback. This reserved area can be used to store whatever information the client wants. In the example above, `room` is reserved in each sub-buffer to store the padding count for that sub-buffer. This is filled in for the previous sub-buffer in the `subbuf_start()` implementation; the padding value for the previous sub-buffer is passed into the `subbuf_start()` callback along with a pointer to the previous sub-buffer, since the padding value isn’t known until a sub-buffer is filled. The `subbuf_start()` callback is also called for the first sub-buffer when the channel is opened, to give the client a chance to reserve space in it. In this case the previous sub-buffer pointer passed into the callback will be `NULL`, so the client should check the value of the `prev_subbuf` pointer before writing into the previous sub-buffer.

Writing to a channel

Kernel clients write data into the current `cpu`’s channel buffer using `relay_write()` or `__relay_write()`. `relay_write()` is the main logging function - it uses `local_irqsave()` to protect the buffer and should be used if you might be logging from interrupt context. If you know you’ll never be logging from interrupt context, you can use `__relay_write()`, which only disables preemption. These functions don’t return a value, so you can’t determine whether or not they failed - the assumption is that you wouldn’t want to check a return value in the fast logging path anyway, and that they’ll always succeed unless the buffer is full and no-overwrite mode is being used, in which case you can detect a failed write in the `subbuf_start()` callback by calling the `relay_buf_full()` helper function.

`relay_reserve()` is used to reserve a slot in a channel buffer which can be written to later. This would typically be used in applications that need to write directly into a channel buffer without having to stage data in a temporary buffer beforehand. Because the actual write may not happen immediately after the slot is reserved, applications using `relay_reserve()` can keep a count of the number of bytes actually written, either in space reserved in the sub-buffers themselves or as a separate array. See the ‘reserve’ example in the `relay-apps` tarball at <http://relayfs.sourceforge.net> for an example of how this can be done. Because the write is under control of the client and is separated from the reserve, `relay_reserve()` doesn’t protect the buffer at all - it’s up to the client to provide the appropriate synchronization when using `relay_reserve()`.

Closing a channel

The client calls `relay_close()` when it’s finished using the channel. The channel and its associated buffers are destroyed when there are no longer any references to any of the channel buffers. `relay_flush()` forces a sub-buffer switch on all the channel buffers, and can be used to finalize and process the last sub-buffers before the channel is closed.

Misc

Some applications may want to keep a channel around and re-use it rather than open and close a new channel for each use. `relay_reset()` can be used for this purpose - it resets a channel to its initial state without reallocating channel buffer memory or destroying existing mappings. It should however only be called when it's safe to do so, i.e. when the channel isn't currently being written to.

Finally, there are a couple of utility callbacks that can be used for different purposes. `buf_mapped()` is called whenever a channel buffer is mmapped from user space and `buf_unmapped()` is called when it's unmapped. The client can use this notification to trigger actions within the kernel application, such as enabling/disabling logging to the channel.

3.47.5 Resources

For news, example code, mailing list, etc. see the relay interface homepage:

<http://relayfs.sourceforge.net>

3.47.6 Credits

The ideas and specs for the relay interface came about as a result of discussions on tracing involving the following:

Michel Dagenais <michel.dagenais@polymtl.ca> Richard Moore <richardj_moore@uk.ibm.com> Bob Wisniewski <bob@watson.ibm.com> Karim Yaghmour <karim@opersys.com> Tom Zanussi <zanussi@us.ibm.com>

Also thanks to Hubertus Franke for a lot of useful suggestions and bug reports.

3.48 ROMFS - ROM File System

This is a quite dumb, read only filesystem, mainly for initial RAM disks of installation disks. It has grown up by the need of having modules linked at boot time. Using this filesystem, you get a very similar feature, and even the possibility of a small kernel, with a file system which doesn't take up useful memory from the router functions in the basement of your office.

For comparison, both the older minix and xiafs (the latter is now defunct) filesystems, compiled as module need more than 20000 bytes, while romfs is less than a page, about 4000 bytes (assuming i586 code). Under the same conditions, the msdos filesystem would need about 30K (and does not support device nodes or symlinks), while the nfs module with nfsroot is about 57K. Furthermore, as a bit unfair comparison, an actual rescue disk used up 3202 blocks with ext2, while with romfs, it needed 3079 blocks.

To create such a file system, you'll need a user program named `genromfs`. It is available on <http://romfs.sourceforge.net/>

As the name suggests, romfs could be also used (space-efficiently) on various read-only media, like (E)EPROM disks if someone will have the motivation.. :)

However, the main purpose of romfs is to have a very small kernel, which has only this filesystem linked in, and then can load any module later, with the current module utilities. It can also be used to run some program to decide if you need SCSI devices, and even IDE or floppy drives can be loaded later if you use the "initrd" -initial RAM disk-feature of the kernel. This would not be really news flash, but with romfs, you can even spare off your ext2 or minix or maybe even affs filesystem until you really know that you need it.

For example, a distribution boot disk can contain only the cd disk drivers (and possibly the SCSI drivers), and the ISO 9660 filesystem module. The kernel can be small enough, since it doesn't have other filesystems, like the quite large ext2fs module, which can then be loaded off the CD at a later stage of the installation. Another use would be for a recovery disk, when you are reinstalling a workstation from the network, and you will have all the tools/modules available from a nearby server, so you don't want to carry two disks for this purpose, just because it won't fit into ext2.

romfs operates on block devices as you can expect, and the underlying structure is very simple. Every accessible structure begins on 16 byte boundaries for fast access. The minimum space a file will take is 32 bytes (this is an empty file, with a less than 16 character name). The maximum overhead for any non-empty file is the header, and the 16 byte padding for the name and the contents, also $16+14+15 = 45$ bytes. This is quite rare however, since most file names are longer than 3 bytes, and shorter than 15 bytes.

The layout of the filesystem is the following:

offset	content	
0	+---+---+---+---+ - r o m \ +---+---+---+---+	The ASCII representation of those bytes
4	1 f s - / +---+---+---+---+	(i.e. "-romlfs-")
8	full size +---+---+---+---+	The number of accessible bytes in this fs.
12	checksum +---+---+---+---+	The checksum of the FIRST 512 BYTES.
16	volume name : : +---+---+---+---+	The zero terminated name of the volume, padded to 16 byte boundary.
xx	file : headers :	

Every multi byte value (32 bit words, I'll use the longwords term from now on) must be in big endian order.

The first eight bytes identify the filesystem, even for the casual inspector. After that, in the 3rd longword, it contains the number of bytes accessible from the start of this filesystem. The 4th longword is the checksum of the first 512 bytes (or the number of bytes accessible, whichever is smaller). The applied algorithm is the same as in the AFFS filesystem, namely a simple sum of the longwords (assuming bigendian quantities again). For details, please consult the source. This algorithm was chosen because although it's not quite reliable, it does not require any tables, and it is very simple.

The following bytes are now part of the file system; each file header must begin on a 16 byte boundary:

offset	content	
0	+---+---+---+---+ next filehdr X	The offset of the next file header (zero if no more files)
4	+---+---+---+---+ spec.info	Info for directories/hard links/devices
8	+---+---+---+---+ size	The size of this file in bytes
12	+---+---+---+---+ checksum	Covering the meta data, including the file name, and padding
16	+---+---+---+---+ file name	The zero terminated name of the file, padded to 16 byte boundary
	: :	
xx	+---+---+---+---+ file data	
	: :	

Since the file headers begin always at a 16 byte boundary, the lowest 4 bits would be always zero in the next filehdr pointer. These four bits are used for the mode information. Bits 0..2 specify the type of the file; while bit 4 shows if the file is executable or not. The permissions are assumed to be world readable, if this bit is not set, and world executable if it is; except the character and block devices, they are never accessible for other than owner. The owner of every file is user and group 0, this should never be a problem for the intended use. The mapping of the 8 possible values to file types is the following:

0	hard link	link destination [file header]
1	directory	first file' s header
2	regular file	unused, must be zero [MBZ]
3	symbolic link	unused, MBZ (file data is the link content)
4	block device	16/16 bits major/minor number
5	char device	• ” -
6	socket	unused, MBZ
7	fifo	unused, MBZ

Note that hard links are specifically marked in this filesystem, but they will behave as you can expect (i.e. share the inode number). Note also that it is your responsibility to not create hard link loops, and creating all the . and .. links for directories. This is normally done correctly by the genromfs program. Please refrain from using the executable bits for special purposes on the socket and fifo special files, they may have other uses in the future. Additionally, please remember that only regular files, and symlinks are supposed to have a nonzero size field; they contain the number of bytes available directly after the (padded) file name.

Another thing to note is that romfs works on file headers and data aligned to 16 byte boundaries, but most hardware devices and the block device drivers are unable to cope with smaller than block-sized data. To overcome this limitation, the whole size of the file system must be padded to an 1024 byte boundary.

If you have any problems or suggestions concerning this file system, please contact me. However, think twice before wanting me to add features and code, because the primary and most important advantage of this file system is the small code. On the other hand, don't be alarmed, I'm not getting that much romfs related mail. Now I can understand why Avery wrote poems in the ARCnet docs to get some more feedback. :)

romfs has also a mailing list, and to date, it hasn't received any traffic, so you are welcome to join it to discuss your ideas. :)

It's run by ezmlm, so you can subscribe to it by sending a message to romfs-subscribe@shadow.banki.hu, the content is irrelevant.

Pending issues:

- Permissions and owner information are pretty essential features of a Un*x like system, but romfs does not provide the full possibilities. I have never found this limiting, but others might.
- The file system is read only, so it can be very small, but in case one would want to write `_anything_` to a file system, he still needs a writable file system, thus negating the size advantages. Possible solutions: implement write access as a compile-time option, or a new, similarly small writable filesystem for RAM disks.
- Since the files are only required to have alignment on a 16 byte boundary, it is currently possibly suboptimal to read or execute files from the filesystem. It might be resolved by reordering file data to have most of it (i.e. except the start and the end) laying at "natural" boundaries, thus it would be possible to directly map a big portion of the file contents to the mm subsystem.
- Compression might be an useful feature, but memory is quite a limiting factor in my eyes.
- Where it is used?
- Does it work on other architectures than intel and motorola?

Have fun,

Janos Farkas <chexum@shadow.banki.hu>

3.49 SPU Filesystem

3.49.1 spufs

Name

spufs - the SPU file system

Description

The SPU file system is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs).

The file system provides a name space similar to posix shared memory or message queues. Users that have write permissions on the file system can use `spu_create(2)` to establish SPU contexts in the `spufs` root.

Every SPU context is represented by a directory containing a predefined set of files. These files can be used for manipulating the state of the logical SPU. Users can change permissions on those files, but not actually add or remove files.

Mount Options

uid=<uid> set the user owning the mount point, the default is 0 (root).

gid=<gid> set the group owning the mount point, the default is 0 (root).

Files

The files in `spufs` mostly follow the standard behavior for regular system calls like `read(2)` or `write(2)`, but often support only a subset of the operations supported on regular file systems. This list details the supported operations and the deviations from the behaviour in the respective man pages.

All files that support the `read(2)` operation also support `readv(2)` and all files that support the `write(2)` operation also support `writew(2)`. All files support the `access(2)` and `stat(2)` family of operations, but only the `st_mode`, `st_nlink`, `st_uid` and `st_gid` fields of `struct stat` contain reliable information.

All files support the `chmod(2)/fchmod(2)` and `chown(2)/fchown(2)` operations, but will not be able to grant permissions that contradict the possible operations, e.g. read access on the `wbox` file.

The current set of files is:

/mem the contents of the local storage memory of the SPU. This can be accessed like a regular shared memory file and contains both code and data in the address space of the SPU. The possible operations on an open `mem` file are:

read(2), pread(2), write(2), pwrite(2), lseek(2)

These operate as documented, with the exception that `seek(2)`, `write(2)` and `pwrite(2)` are not supported beyond the end of the file. The file size is the size

of the local storage of the SPU, which normally is 256 kilobytes.

mmap(2) Mapping mem into the process address space gives access to the SPU local storage within the process address space. Only MAP_SHARED mappings are allowed.

/mbox

The first SPU to CPU communication mailbox. This file is read-only and can be read in units of 32 bits. The file can only be used in non-blocking mode and it even poll() will not block on it. The possible operations on an open mbox file are:

read(2) If a count smaller than four is requested, read returns -1 and sets errno to EINVAL. If there is no data available in the mail box, the return value is set to -1 and errno becomes EAGAIN. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

/ibox

The second SPU to CPU communication mailbox. This file is similar to the first mailbox file, but can be read in blocking I/O mode, and the poll family of system calls can be used to wait for it. The possible operations on an open ibox file are:

read(2) If a count smaller than four is requested, read returns -1 and sets errno to EINVAL. If there is no data available in the mail box and the file descriptor has been opened with O_NONBLOCK, the return value is set to -1 and errno becomes EAGAIN.

If there is no data available in the mail box and the file descriptor has been opened without O_NONBLOCK, the call will block until the SPU writes to its interrupt mailbox channel. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

poll(2) Poll on the ibox file returns (POLLIN | POLLRDNORM) whenever data is available for reading.

/wbox

The CPU to SPU communication mailbox. It is write-only and can be written in units of 32 bits. If the mailbox is full, write() will block

and poll can be used to wait for it becoming empty again. The possible operations on an open mbox file are: write(2) If a count smaller than four is requested, write returns -1 and sets errno to EINVAL. If there is no space available in the mail box and the file descriptor has been opened with O_NONBLOCK, the return value is set to -1 and errno becomes EAGAIN.

If there is no space available in the mail box and the file descriptor has been opened without O_NONBLOCK, the call will block until the SPU reads from its PPE mailbox channel. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

poll(2) Poll on the mbox file returns (POLL-OUT | POLLWRNORM) whenever space is available for writing.

/mbox_stat, /ibox_stat, /wbox_stat Read-only files that contain the length of the current queue, i.e. how many words can be read from mbox or mbox or how many words can be written to wbox without blocking. The files can be read only in 4-byte units and return a big-endian binary integer number. The possible operations on an open *box_stat file are:

read(2) If a count smaller than four is requested, read returns -1 and sets errno to EINVAL. Otherwise, a four byte value is placed in the data buffer, containing the number of elements that can be read from (for mbox_stat and mbox_stat) or written to (for wbox_stat) the respective mail box without blocking or resulting in EAGAIN.

/npc, /decr, /decr_status, /spu_tag_mask, /event_mask, /srr0

Internal registers of the SPU. The representation is an ASCII string with the numeric value of the next instruction to be executed. These can be used in read/write mode for debugging, but normal operation of programs should not rely on them because access to any of them except npc requires an SPU context save and is therefore very inefficient.

The contents of these files are:

npc	Next Program Counter
decr	SPU Decrementer
decr_status	Decrementer Status
spu_tag_mask	MFC tag mask for SPU DMA
event_mask	Event mask for SPU interrupts
srr0	Interrupt Return address register

The possible operations on an open npc, decr, decr_status, spu_tag_mask, event_mask or srr0 file are:

read(2) When the count supplied to the read call is shorter than the required length for the pointer value plus a newline character, subsequent reads from the same file descriptor will result in completing the string, regardless of changes to the register by a running SPU task. When a complete string has been read, all subsequent read operations will return zero bytes and a new file descriptor needs to be opened to read the value again.

write(2) A write operation on the file results in setting the register to the value given in the string. The string is parsed from the beginning to the first non-numeric character or the end of the buffer. Subsequent writes to the same file descriptor overwrite the previous setting.

/fpcr

This file gives access to the Floating Point Status and Control Register as a four byte long file. The operations on the fpcr file are:

read(2) If a count smaller than four is requested, read returns -1 and sets errno to EINVAL. Otherwise, a four byte value is placed in the data buffer, containing the current value of the fpcr register.

write(2) If a count smaller than four is requested, write returns -1 and sets errno to EINVAL. Otherwise, a four byte value is copied from the data buffer, updating the value of the fpcr register.

/signal1, /signal2 The two signal notification channels of an SPU. These are read-write files that operate on a 32 bit word. Writing to one of these files triggers an interrupt on the SPU. The value

written to the signal files can be read from the SPU through a channel read or from host user space through the file. After the value has been read by the SPU, it is reset to zero. The possible operations on an open `signal1` or `signal2` file are:

read(2) If a count smaller than four is requested, read returns -1 and sets `errno` to `EINVAL`. Otherwise, a four byte value is placed in the data buffer, containing the current value of the specified signal notification register.

write(2) If a count smaller than four is requested, write returns -1 and sets `errno` to `EINVAL`. Otherwise, a four byte value is copied from the data buffer, updating the value of the specified signal notification register. The signal notification register will either be replaced with the input data or will be updated to the bitwise OR of the old value and the input data, depending on the contents of the `signal1_type`, or `signal2_type` respectively, file.

/signal1_type, /signal2_type These two files change the behavior of the `signal1` and `signal2` notification files. They contain a numerical ASCII string which is read as either "1" or "0". In mode 0 (overwrite), the hardware replaces the contents of the signal channel with the data that is written to it. In mode 1 (logical OR), the hardware accumulates the bits that are subsequently written to it. The possible operations on an open `signal1_type` or `signal2_type` file are:

read(2) When the count supplied to the read call is shorter than the required length for the digit plus a newline character, subsequent reads from the same file descriptor will result in completing the string. When a complete string has been read, all subsequent read operations will return zero bytes and a new file descriptor needs to be opened to read the value again.

write(2) A write operation on the file results in setting the register to the value given in the string. The string is parsed from the beginning to the first non-numeric character or the end of the buffer. Subsequent writes to the same file descriptor over-

write the previous setting.

Examples

```
/etc/fstab entry none /spu spufs gid=spu 0 0
```

Authors

Arnd Bergmann <arndb@de.ibm.com>, Mark Nutter
<mnutter@us.ibm.com>, Ulrich Weigand
<Ulrich.Weigand@de.ibm.com>

See Also

capabilities(7), close(2), spu_create(2), spu_run(2), spufs(7)

3.49.2 spu_create

Name

spu_create - create a new spu context

Synopsis

```
#include <sys/types.h>
#include <sys/spu.h>

int spu_create(const char *pathname, int flags, mode_t mode);
```

Description

The `spu_create` system call is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs). It creates a new logical context for an SPU in `pathname` and returns a handle to associated with it. `pathname` must point to a non-existing directory in the mount point of the SPU file system (`spufs`). When `spu_create` is successful, a directory gets created on `pathname` and it is populated with files.

The returned file handle can only be passed to `spu_run(2)` or closed, other operations are not defined on it. When it is closed, all associated directory entries in `spufs` are removed. When the last file handle pointing either inside of the context directory or to this file descriptor is closed, the logical SPU context is destroyed.

The parameter `flags` can be zero or any bitwise or'ed combination of the following constants:

SPU_RAWIO Allow mapping of some of the hardware registers of the SPU into user space. This flag requires the CAP_SYS_RAWIO capability, see capabilities(7).

The mode parameter specifies the permissions used for creating the new directory in spufs. mode is modified with the user's umask(2) value and then used for both the directory and the files contained in it. The file permissions mask out some more bits of mode because they typically support only read or write access. See stat(2) for a full list of the possible mode values.

Return Value

spu_create returns a new file descriptor. It may return -1 to indicate an error condition and set errno to one of the error codes listed below.

Errors

EACCES The current user does not have write access on the spufs mount point.

EEXIST An SPU context already exists at the given path name.

EFAULT **pathname is not a valid string pointer in the current address space.**

EINVAL pathname is not a directory in the spufs mount point.

ELOOP Too many symlinks were found while resolving pathname.

EMFILE The process has reached its maximum open file limit.

ENAMETOOLONG pathname was too long.

ENFILE The system has reached the global open file limit.

ENOENT Part of pathname could not be resolved.

ENOMEM The kernel could not allocate all resources required.

ENOSPC **There are not enough SPU resources available to create a new context or the user specific limit for the number of SPU contexts has been reached.**

ENOSYS **the functionality is not provided by the current system, because either the hardware does not provide SPUs or the spufs module is not loaded.**

ENOTDIR A part of pathname is not a directory.

Notes

`spu_create` is meant to be used from libraries that implement a more abstract interface to SPUs, not to be used from regular applications. See <http://www.bsc.es/projects/deepcomputing/linuxoncell/> for the recommended libraries.

Files

`pathname` must point to a location beneath the mount point of `spufs`. By convention, it gets mounted in `/spu`.

Conforming to

This call is Linux specific and only implemented by the ppc64 architecture. Programs using this system call are not portable.

Bugs

The code does not yet fully implement all features lined out here.

Author

Arnd Bergmann <arndb@de.ibm.com>

See Also

`capabilities(7)`, `close(2)`, `spu_run(2)`, `spufs(7)`

3.49.3 `spu_run`

Name

`spu_run` - execute an spu context

Synopsis

```
#include <sys/spu.h>

int spu_run(int fd, unsigned int *npc, unsigned int *event);
```

Description

The `spu_run` system call is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs). It uses the `fd` that was returned from `spu_create(2)` to address a specific SPU context. When the context gets scheduled to a physical SPU, it starts execution at the instruction pointer passed in `npc`.

Execution of SPU code happens synchronously, meaning that `spu_run` does not return while the SPU is still running. If there is a need to execute SPU code in parallel with other code on either the main CPU or other SPUs, you need to create a new thread of execution first, e.g. using the `pthread_create(3)` call.

When `spu_run` returns, the current value of the SPU instruction pointer is written back to `npc`, so you can call `spu_run` again without updating the pointers.

event can be a NULL pointer or point to an extended status code that gets filled when `spu_run` returns. It can be one of the following constants:

SPE_EVENT_DMA_ALIGNMENT A DMA alignment error

SPE_EVENT_SPE_DATA_SEGMENT A DMA segmentation error

SPE_EVENT_SPE_DATA_STORAGE A DMA storage error

If NULL is passed as the event argument, these errors will result in a signal delivered to the calling process.

Return Value

`spu_run` returns the value of the `spu_status` register or -1 to indicate an error and set `errno` to one of the error codes listed below. The `spu_status` register value contains a bit mask of status codes and optionally a 14 bit code returned from the stop-and-signal instruction on the SPU. The bit masks for the status codes are:

0x02 SPU was stopped by stop-and-signal.

0x04 SPU was stopped by halt.

0x08 SPU is waiting for a channel.

0x10 SPU is in single-step mode.

0x20 SPU has tried to execute an invalid instruction.

0x40 SPU has tried to access an invalid channel.

0x3fff0000 The bits masked with this value contain the code returned from stop-and-signal.

There are always one or more of the lower eight bits set or an error code is returned from `spu_run`.

Errors

EAGAIN or EWOULDBLOCK fd is in non-blocking mode and spu_run would block.

EBADF fd is not a valid file descriptor.

EFAULT npc is not a valid pointer or status is neither NULL nor a valid pointer.

EINTR A signal occurred while spu_run was in progress. The npc value has been updated to the new program counter value if necessary.

EINVAL fd is not a file descriptor returned from spu_create(2).

ENOMEM Insufficient memory was available to handle a page fault result-ing from an MFC direct memory access.

ENOSYS the functionality is not provided by the current system, because either the hardware does not provide SPUs or the spufs module is not loaded.

Notes

spu_run is meant to be used from libraries that implement a more abstract interface to SPUs, not to be used from regular applications. See <http://www.bsc.es/projects/deepcomputing/linuxoncell/> for the recommended libraries.

Conforming to

This call is Linux specific and only implemented by the ppc64 architecture. Programs using this system call are not portable.

Bugs

The code does not yet fully implement all features lined out here.

Author

Arnd Bergmann <arndb@de.ibm.com>

See Also

capabilities(7), close(2), spu_create(2), spufs(7)

3.50 Squashfs 4.0 Filesystem

Squashfs is a compressed read-only filesystem for Linux.

It uses zlib, lz4, lzo, or xz compression to compress files, inodes and directories. Inodes in the system are very small and all blocks are packed to minimise data overhead. Block sizes greater than 4K are supported up to a maximum of 1Mbytes (default block size 128K).

Squashfs is intended for general read-only filesystem use, for archival use (i.e. in cases where a .tar.gz file may be used), and in constrained block device/memory systems (e.g. embedded systems) where low overhead is needed.

Mailing list: squashfs-devel@lists.sourceforge.net Web site: www.squashfs.org

3.50.1 1. Filesystem Features

Squashfs filesystem features versus Cramfs:

Max filesystem size	2 ⁶⁴	256 MiB
Max file size	~ 2 TiB	16 MiB
Max files	unlimited	unlimited
Max directories	unlimited	unlimited
Max entries per directory	unlimited	unlimited
Max block size	1 MiB	4 KiB
Metadata compression	yes	no
Directory indexes	yes	no
Sparse file support	yes	no
Tail-end packing (fragments)	yes	no
Exportable (NFS etc.)	yes	no
Hard link support	yes	no
“.” and “..” in readdir	yes	no
Real inode numbers	yes	no
32-bit uids/gids	yes	no
File creation time	yes	no
Xattr support	yes	no
ACL support	no	no

Squashfs compresses data, inodes and directories. In addition, inode and directory data are highly compacted, and packed on byte boundaries. Each compressed inode is on average 8 bytes in length (the exact length varies on file type, i.e. regular file, directory, symbolic link, and block/char device inodes have different sizes).

3.50.2 2. Using Squashfs

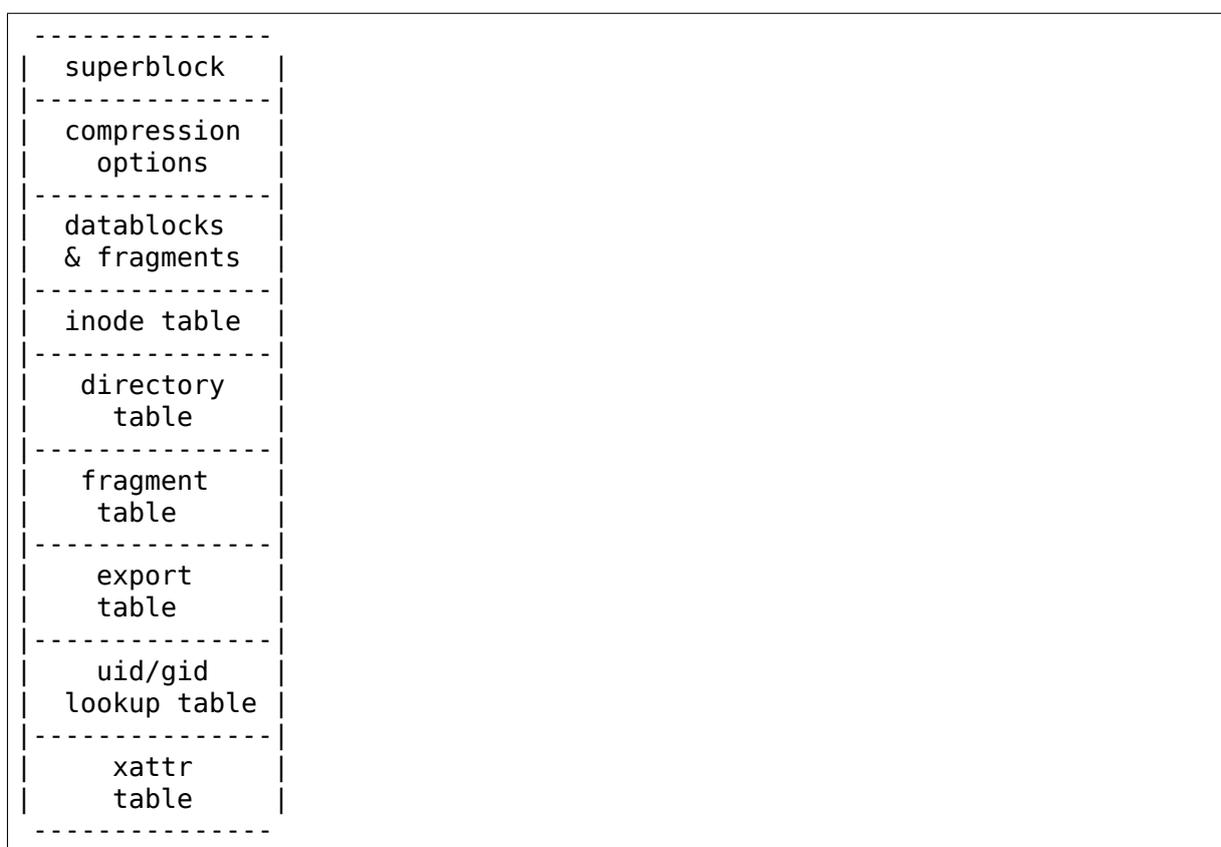
As squashfs is a read-only filesystem, the mksquashfs program must be used to create populated squashfs filesystems. This and other squashfs utilities can be obtained from <http://www.squashfs.org>. Usage instructions can be obtained from this site also.

The squashfs-tools development tree is now located on kernel.org

`git://git.kernel.org/pub/scm/fs/squashfs/squashfs-tools.git`

3.50.3 3. Squashfs Filesystem Design

A squashfs filesystem consists of a maximum of nine parts, packed together on a byte alignment:



Compressed data blocks are written to the filesystem as files are read from the source directory, and checked for duplicates. Once all file data has been written the completed inode, directory, fragment, export, uid/gid lookup and xattr tables are written.

3.50.4 3.1 Compression options

Compressors can optionally support compression specific options (e.g. dictionary size). If non-default compression options have been used, then these are stored here.

3.50.5 3.2 Inodes

Metadata (inodes and directories) are compressed in 8Kbyte blocks. Each compressed block is prefixed by a two byte length, the top bit is set if the block is uncompressed. A block will be uncompressed if the `-noI` option is set, or if the compressed block was larger than the uncompressed block.

Inodes are packed into the metadata blocks, and are not aligned to block boundaries, therefore inodes overlap compressed blocks. Inodes are identified by a 48-bit number which encodes the location of the compressed metadata block containing the inode, and the byte offset into that block where the inode is placed (`<block, offset>`).

To maximise compression there are different inodes for each file type (regular file, directory, device, etc.), the inode contents and length varying with the type.

To further maximise compression, two types of regular file inode and directory inode are defined: inodes optimised for frequently occurring regular files and directories, and extended types where extra information has to be stored.

3.50.6 3.3 Directories

Like inodes, directories are packed into compressed metadata blocks, stored in a directory table. Directories are accessed using the start address of the metablock containing the directory and the offset into the decompressed block (`<block, offset>`).

Directories are organised in a slightly complex way, and are not simply a list of file names. The organisation takes advantage of the fact that (in most cases) the inodes of the files will be in the same compressed metadata block, and therefore, can share the start block. Directories are therefore organised in a two level list, a directory header containing the shared start block value, and a sequence of directory entries, each of which share the shared start block. A new directory header is written once/if the inode start block changes. The directory header/directory entry list is repeated as many times as necessary.

Directories are sorted, and can contain a directory index to speed up file lookup. Directory indexes store one entry per metablock, each entry storing the index/filename mapping to the first directory header in each metadata block. Directories are sorted in alphabetical order, and at lookup the index is scanned linearly looking for the first filename alphabetically larger than the filename being looked up. At this point the location of the metadata block the filename is in has been found. The general idea of the index is to ensure only one metadata block needs to be decompressed to do a lookup irrespective of the length of the directory. This scheme has the advantage that it doesn't require extra memory overhead and doesn't require much extra storage on disk.

3.50.7 3.4 File data

Regular files consist of a sequence of contiguous compressed blocks, and/or a compressed fragment block (tail-end packed block). The compressed size of each datablock is stored in a block list contained within the file inode.

To speed up access to datablocks when reading ‘large’ files (256 Mbytes or larger), the code implements an index cache that caches the mapping from block index to datablock location on disk.

The index cache allows Squashfs to handle large files (up to 1.75 TiB) while retaining a simple and space-efficient block list on disk. The cache is split into slots, caching up to eight 224 GiB files (128 KiB blocks). Larger files use multiple slots, with 1.75 TiB files using all 8 slots. The index cache is designed to be memory efficient, and by default uses 16 KiB.

3.50.8 3.5 Fragment lookup table

Regular files can contain a fragment index which is mapped to a fragment location on disk and compressed size using a fragment lookup table. This fragment lookup table is itself stored compressed into metadata blocks. A second index table is used to locate these. This second index table for speed of access (and because it is small) is read at mount time and cached in memory.

3.50.9 3.6 Uid/gid lookup table

For space efficiency regular files store uid and gid indexes, which are converted to 32-bit uids/gids using an id look up table. This table is stored compressed into metadata blocks. A second index table is used to locate these. This second index table for speed of access (and because it is small) is read at mount time and cached in memory.

3.50.10 3.7 Export table

To enable Squashfs filesystems to be exportable (via NFS etc.) filesystems can optionally (disabled with the `-no-exports` Mksquashfs option) contain an inode number to inode disk location lookup table. This is required to enable Squashfs to map inode numbers passed in filehandles to the inode location on disk, which is necessary when the export code reinstatiates expired/flushed inodes.

This table is stored compressed into metadata blocks. A second index table is used to locate these. This second index table for speed of access (and because it is small) is read at mount time and cached in memory.

3.50.11 3.8 Xattr table

The xattr table contains extended attributes for each inode. The xattrs for each inode are stored in a list, each list entry containing a type, name and value field. The type field encodes the xattr prefix (“user.” , “trusted.” etc) and it also encodes how the name/value fields should be interpreted. Currently the type indicates whether the value is stored inline (in which case the value field contains the xattr value), or if it is stored out of line (in which case the value field stores a reference to where the actual value is stored). This allows large values to be stored out of line improving scanning and lookup performance and it also allows values to be de-duplicated, the value being stored once, and all other occurrences holding an out of line reference to that value.

The xattr lists are packed into compressed 8K metadata blocks. To reduce overhead in inodes, rather than storing the on-disk location of the xattr list inside each inode, a 32-bit xattr id is stored. This xattr id is mapped into the location of the xattr list using a second xattr id lookup table.

3.50.12 4. TODOs and Outstanding Issues

3.50.13 4.1 TODO list

Implement ACL support.

3.50.14 4.2 Squashfs Internal Cache

Blocks in Squashfs are compressed. To avoid repeatedly decompressing recently accessed data Squashfs uses two small metadata and fragment caches.

The cache is not used for file datablocks, these are decompressed and cached in the page-cache in the normal way. The cache is used to temporarily cache fragment and metadata blocks which have been read as a result of a metadata (i.e. inode or directory) or fragment access. Because metadata and fragments are packed together into blocks (to gain greater compression) the read of a particular piece of metadata or fragment will retrieve other metadata/fragments which have been packed with it, these because of locality-of-reference may be read in the near future. Temporarily caching them ensures they are available for near future access without requiring an additional read and decompress.

In the future this internal cache may be replaced with an implementation which uses the kernel page cache. Because the page cache operates on page sized units this may introduce additional complexity in terms of locking and associated race conditions.

3.51 sysfs - `_The_` filesystem for exporting kernel objects

Patrick Mochel <mochel@osdl.org>

Mike Murphy <mamurph@cs.clemson.edu>

Revised 16 August 2011

Original 10 January 2003

3.51.1 What it is:

sysfs is a ram-based filesystem initially based on ramfs. It provides a means to export kernel data structures, their attributes, and the linkages between them to userspace.

sysfs is tied inherently to the kobject infrastructure. Please read Documentation/core-api/kobject.rst for more information concerning the kobject interface.

3.51.2 Using sysfs

sysfs is always compiled in if CONFIG_SYSFS is defined. You can access it by doing:

```
mount -t sysfs sysfs /sys
```

3.51.3 Directory Creation

For every kobject that is registered with the system, a directory is created for it in sysfs. That directory is created as a subdirectory of the kobject's parent, expressing internal object hierarchies to userspace. Top-level directories in sysfs represent the common ancestors of object hierarchies; i.e. the subsystems the objects belong to.

Sysfs internally stores a pointer to the kobject that implements a directory in the kernfs_node object associated with the directory. In the past this kobject pointer has been used by sysfs to do reference counting directly on the kobject whenever the file is opened or closed. With the current sysfs implementation the kobject reference count is only modified directly by the function sysfs_schedule_callback().

3.51.4 Attributes

Attributes can be exported for kobjects in the form of regular files in the filesystem. Sysfs forwards file I/O operations to methods defined for the attributes, providing a means to read and write kernel attributes.

Attributes should be ASCII text files, preferably with only one value per file. It is noted that it may not be efficient to contain only one value per file, so it is socially acceptable to express an array of values of the same type.

Mixing types, expressing multiple lines of data, and doing fancy formatting of data is heavily frowned upon. Doing these things may get you publicly humiliated and your code rewritten without notice.

An attribute definition is simply:

```
struct attribute {
    char                * name;
    struct module       *owner;
    umode_t             mode;
};

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

A bare attribute contains no means to read or write the value of the attribute. Subsystems are encouraged to define their own attribute structure and wrapper functions for adding and removing attributes for a specific object type.

For example, the driver model defines struct device_attribute like:

```
struct device_attribute {
    struct attribute    attr;
    ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                    const char *buf, size_t count);
};

int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

It also defines this helper for defining device attributes:

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _
↳store)
```

For example, declaring:

```
static DEVICE_ATTR(foo, S_IWUSR | S_IRUGO, show_foo, store_foo);
```

is equivalent to doing:

```
static struct device_attribute dev_attr_foo = {
    .attr = {
        .name = "foo",
        .mode = S_IWUSR | S_IRUGO,
    },
    .show = show_foo,
    .store = store_foo,
};
```

Note as stated in `include/linux/kernel.h` “OTHER_WRITABLE? Generally considered a bad idea.” so trying to set a sysfs file writable for everyone will fail reverting to RO mode for “Others” .

For the common cases `sysfs.h` provides convenience macros to make defining attributes easier as well as making code more concise and readable. The above case could be shortened to:

```
static struct device_attribute dev_attr_foo = __ATTR_RW(foo);
```

the list of helpers available to define your wrapper function is:

__ATTR_RO(name): assumes default `name_show` and mode 0444

__ATTR_WO(name): assumes a `name_store` only and is restricted to mode 0200 that is root write access only.

__ATTR_RO_MODE(name, mode): for more restrictive RO access currently only use case is the EFI System Resource Table (see `drivers/firmware/efi/esrt.c`)

__ATTR_RW(name): assumes default `name_show`, `name_store` and setting mode to 0644.

__ATTR_NULL: which sets the name to NULL and is used as end of list indicator (see: `kernel/workqueue.c`)

3.51.5 Subsystem-Specific Callbacks

When a subsystem defines a new attribute type, it must implement a set of sysfs operations for forwarding read and write calls to the show and store methods of the attribute owners:

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *, struct attribute *, char *);
    ssize_t (*store)(struct kobject *, struct attribute *, const char *
↳, size_t);
};
```

[Subsystems should have already defined a `struct kobj_type` as a descriptor for this type, which is where the `sysfs_ops` pointer is stored. See the `kobject` documentation for more information.]

When a file is read or written, sysfs calls the appropriate method for the type. The method then translates the generic `struct kobject` and `struct attribute` pointers to the appropriate pointer types, and calls the associated methods.

To illustrate:

```

#define to_dev(obj) container_of(obj, struct device, kobj)
#define to_dev_attr(_attr) container_of(_attr, struct device_attribute,
↳attr)

static ssize_t dev_attr_show(struct kobject *kobj, struct attribute *attr,
                             char *buf)
{
    struct device_attribute *dev_attr = to_dev_attr(attr);
    struct device *dev = to_dev(kobj);
    ssize_t ret = -EIO;

    if (dev_attr->show)
        ret = dev_attr->show(dev, dev_attr, buf);
    if (ret >= (ssize_t)PAGE_SIZE) {
        printk("dev_attr_show: %pS returned bad count\n",
               dev_attr->show);
    }
    return ret;
}

```

3.51.6 Reading/Writing Attribute Data

To read or write attributes, `show()` or `store()` methods must be specified when declaring the attribute. The method types should be as simple as those defined for device attributes:

```

ssize_t (*show)(struct device *dev, struct device_attribute *attr, char
↳*buf);
ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                 const char *buf, size_t count);

```

IOW, they should take only an object, an attribute, and a buffer as parameters.

sysfs allocates a buffer of size (`PAGE_SIZE`) and passes it to the method. Sysfs will call the method exactly once for each read or write. This forces the following behavior on the method implementations:

- On `read(2)`, the `show()` method should fill the entire buffer. Recall that an attribute should only be exporting one value, or an array of similar values, so this shouldn't be that expensive.

This allows userspace to do partial reads and forward seeks arbitrarily over the entire file at will. If userspace seeks back to zero or does a `pread(2)` with an offset of '0' the `show()` method will be called again, rearmed, to fill the buffer.

- On `write(2)`, sysfs expects the entire buffer to be passed during the first write. Sysfs then passes the entire buffer to the `store()` method. A terminating null is added after the data on stores. This makes functions like `sysfs_streq()` safe to use.

When writing sysfs files, userspace processes should first read the entire file, modify the values it wishes to change, then write the entire buffer back.

Attribute method implementations should operate on an identical buffer when reading and writing values.

Other notes:

- Writing causes the `show()` method to be rearmed regardless of current file position.
- The buffer will always be `PAGE_SIZE` bytes in length. On i386, this is 4096.
- `show()` methods should return the number of bytes printed into the buffer. This is the return value of `scnprintf()`.
- `show()` must not use `sprintf()` when formatting the value to be returned to user space. If you can guarantee that an overflow will never happen you can use `sprintf()` otherwise you must use `scnprintf()`.
- `store()` should return the number of bytes used from the buffer. If the entire buffer has been used, just return the count argument.
- `show()` or `store()` can always return errors. If a bad value comes through, be sure to return an error.
- The object passed to the methods will be pinned in memory via sysfs referencing counting its embedded object. However, the physical entity (e.g. device) the object represents may not be present. Be sure to have a way to check this, if necessary.

A very simple (and naive) implementation of a device attribute is:

```
static ssize_t show_name(struct device *dev, struct device_attribute *attr,
                        char *buf)
{
    return scnprintf(buf, PAGE_SIZE, "%s\n", dev->name);
}

static ssize_t store_name(struct device *dev, struct device_attribute
↳*attr,
                        const char *buf, size_t count)
{
    sprintf(dev->name, sizeof(dev->name), "%.*s",
            (int)min(count, sizeof(dev->name) - 1), buf);
    return count;
}

static DEVICE_ATTR(name, S_IRUGO, show_name, store_name);
```

(Note that the real implementation doesn't allow userspace to set the name for a device.)

3.51.7 Top Level Directory Layout

The sysfs directory arrangement exposes the relationship of kernel data structures.

The top level sysfs directory looks like:

```
block/
bus/
class/
```

(continues on next page)

(continued from previous page)

```
dev/
devices/
firmware/
net/
fs/
```

devices/ contains a filesystem representation of the device tree. It maps directly to the internal kernel device tree, which is a hierarchy of struct device.

bus/ contains flat directory layout of the various bus types in the kernel. Each bus' s directory contains two subdirectories:

```
devices/
drivers/
```

devices/ contains symlinks for each device discovered in the system that point to the device' s directory under root/.

drivers/ contains a directory for each device driver that is loaded for devices on that particular bus (this assumes that drivers do not span multiple bus types).

fs/ contains a directory for some filesystems. Currently each filesystem wanting to export attributes must create its own hierarchy below fs/ (see ./fuse.txt for an example).

dev/ contains two directories char/ and block/. Inside these two directories there are symlinks named <major>:<minor>. These symlinks point to the sysfs directory for the given device. /sys/dev provides a quick way to lookup the sysfs interface for a device from the result of a stat(2) operation.

More information can driver-model specific features can be found in Documentation/driver-api/driver-model/.

TODO: Finish this section.

3.51.8 Current Interfaces

The following interface layers currently exist in sysfs:

devices (include/linux/device.h)

Structure:

```
struct device_attribute {
    struct attribute      attr;
    ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                    const char *buf, size_t count);
};
```

Declaring:

```
DEVICE_ATTR(_name, _mode, _show, _store);
```

Creation/Removal:

```
int device_create_file(struct device *dev, const struct device_attribute *  
↪attr);  
void device_remove_file(struct device *dev, const struct device_attribute_  
↪* attr);
```

bus drivers (include/linux/device.h)

Structure:

```
struct bus_attribute {  
    struct attribute          attr;  
    ssize_t (*show)(struct bus_type *, char * buf);  
    ssize_t (*store)(struct bus_type *, const char * buf, size_t_  
↪count);  
};
```

Declaring:

```
static BUS_ATTR_RW(name);  
static BUS_ATTR_RO(name);  
static BUS_ATTR_WO(name);
```

Creation/Removal:

```
int bus_create_file(struct bus_type *, struct bus_attribute *);  
void bus_remove_file(struct bus_type *, struct bus_attribute *);
```

device drivers (include/linux/device.h)

Structure:

```
struct driver_attribute {  
    struct attribute          attr;  
    ssize_t (*show)(struct device_driver *, char * buf);  
    ssize_t (*store)(struct device_driver *, const char * buf,  
                    size_t count);  
};
```

Declaring:

```
DRIVER_ATTR_RO(_name)  
DRIVER_ATTR_RW(_name)
```

Creation/Removal:

```
int driver_create_file(struct device_driver *, const struct driver_  
↪attribute *);  
void driver_remove_file(struct device_driver *, const struct driver_  
↪attribute *);
```

3.51.9 Documentation

The sysfs directory structure and the attributes in each directory define an ABI between the kernel and user space. As for any ABI, it is important that this ABI is stable and properly documented. All new sysfs attributes must be documented in Documentation/ABI. See also Documentation/ABI/README for more information.

3.52 SystemV Filesystem

It implements all of

- Xenix FS,
- SystemV/386 FS,
- Coherent FS.

To install:

- Answer the ‘System V and Coherent filesystem support’ question with ‘y’ when configuring the kernel.
- To mount a disk or a partition, use:

```
mount [-r] -t sysv device mountpoint
```

The file system type names:

```
-t sysv
-t xenix
-t coherent
```

may be used interchangeably, but the last two will eventually disappear.

Bugs in the present implementation:

- Coherent FS:
 - The “free list interleave” n:m is currently ignored.
 - Only file systems with no filesystem name and no pack name are recognized. (See Coherent “man mkfs” for a description of these features.)
- SystemV Release 2 FS:

The superblock is only searched in the blocks 9, 15, 18, which corresponds to the beginning of track 1 on floppy disks. No support for this FS on hard disk yet.

These filesystems are rather similar. Here is a comparison with Minix FS:

- Linux fdisk reports on partitions
 - Minix FS 0x81 Linux/Minix
 - Xenix FS ??
 - SystemV FS ??
 - Coherent FS 0x08 AIX bootable

- Size of a block or zone (data allocation unit on disk)
 - Minix FS 1024
 - Xenix FS 1024 (also 512 ??)
 - SystemV FS 1024 (also 512 and 2048)
 - Coherent FS 512
- General layout: all have one boot block, one super block and separate areas for inodes and for directories/data. On SystemV Release 2 FS (e.g. Microport) the first track is reserved and all the block numbers (including the super block) are offset by one track.
- Byte ordering of “short” (16 bit entities) on disk:
 - Minix FS little endian 0 1
 - Xenix FS little endian 0 1
 - SystemV FS little endian 0 1
 - Coherent FS little endian 0 1

Of course, this affects only the file system, not the data of files on it!

- Byte ordering of “long” (32 bit entities) on disk:
 - Minix FS little endian 0 1 2 3
 - Xenix FS little endian 0 1 2 3
 - SystemV FS little endian 0 1 2 3
 - Coherent FS PDP-11 2 3 0 1

Of course, this affects only the file system, not the data of files on it!

- Inode on disk: “short” , 0 means non-existent, the root dir ino is:

Minix FS	1
Xenix FS, SystemV FS, Coherent FS	2

- Maximum number of hard links to a file:

Minix FS	250
Xenix FS	??
SystemV FS	??
Coherent FS	>=10000

- Free inode management:
 - **Minix FS** a bitmap
 - **Xenix FS, SystemV FS, Coherent FS** There is a cache of a certain number of free inodes in the super-block. When it is exhausted, new free inodes are found using a linear search.
- Free block management:

- **Minix FS** a bitmap
 - **Xenix FS, SystemV FS, Coherent FS** Free blocks are organized in a “free list” . Maybe a misleading term, since it is not true that every free block contains a pointer to the next free block. Rather, the free blocks are organized in chunks of limited size, and every now and then a free block contains pointers to the free blocks pertaining to the next chunk; the first of these contains pointers and so on. The list terminates with a “block number” 0 on Xenix FS and SystemV FS, with a block zeroed out on Coherent FS.
- Super-block location:

Minix FS	block 1 = bytes 1024..2047
Xenix FS	block 1 = bytes 1024..2047
SystemV FS	bytes 512..1023
Coherent FS	block 1 = bytes 512..1023

- Super-block layout:

- Minix FS:

```
unsigned short s_ninodes;
unsigned short s_nzones;
unsigned short s_imap_blocks;
unsigned short s_zmap_blocks;
unsigned short s_firstdatazone;
unsigned short s_log_zone_size;
unsigned long s_max_size;
unsigned short s_magic;
```

- Xenix FS, SystemV FS, Coherent FS:

```
unsigned short s_firstdatazone;
unsigned long s_nzones;
unsigned short s_fzone_count;
unsigned long s_fzones[NICFREE];
unsigned short s_finode_count;
unsigned short s_finodes[NICINOD];
char s_flock;
char s_ilock;
char s_modified;
char s_rdonly;
unsigned long s_time;
short s_dinfo[4]; -- SystemV FS only
unsigned long s_free_zones;
unsigned short s_free_inodes;
short s_dinfo[4]; -- Xenix FS only
unsigned short s_interleave_m,s_interleave_n; -- Coherent FS only
char s_fname[6];
char s_fpack[6];
```

then they differ considerably:

Xenix FS:

```
char      s_clean;
char      s_fill[371];
long      s_magic;
long      s_type;
```

SystemV FS:

```
long      s_fill[12 or 14];
long      s_state;
long      s_magic;
long      s_type;
```

Coherent FS:

```
unsigned long s_unique;
```

Note that Coherent FS has no magic.

- Inode layout:

- Minix FS:

```
unsigned short i_mode;
unsigned short i_uid;
unsigned long i_size;
unsigned long i_time;
unsigned char i_gid;
unsigned char i_nlinks;
unsigned short i_zone[7+1+1];
```

- Xenix FS, SystemV FS, Coherent FS:

```
unsigned short i_mode;
unsigned short i_nlink;
unsigned short i_uid;
unsigned short i_gid;
unsigned long i_size;
unsigned char i_zone[3*(10+1+1+1)];
unsigned long i_atime;
unsigned long i_mtime;
unsigned long i_ctime;
```

- Regular file data blocks are organized as

- Minix FS:

- * 7 direct blocks
 - * 1 indirect block (pointers to blocks)
 - * 1 double-indirect block (pointer to pointers to blocks)

- Xenix FS, SystemV FS, Coherent FS:

- * 10 direct blocks
 - * 1 indirect block (pointers to blocks)
 - * 1 double-indirect block (pointer to pointers to blocks)

* 1 triple-indirect block (pointer to pointers to pointers to blocks)

Minix FS	32	32
Xenix FS	64	16
SystemV FS	64	16
Coherent FS	64	8

- Directory entry on disk

- Minix FS:

```
unsigned short inode;
char name[14/30];
```

- Xenix FS, SystemV FS, Coherent FS:

```
unsigned short inode;
char name[14];
```

Minix FS	16/32	64/32
Xenix FS	16	64
SystemV FS	16	64
Coherent FS	16	32

- How to implement symbolic links such that the host fsck doesn't scream:

- Minix FS normal
 - Xenix FS kludge: as regular files with chmod 1000
 - SystemV FS ??
 - Coherent FS kludge: as regular files with chmod 1000

Notation: We often speak of a “block” but mean a zone (the allocation unit) and not the disk driver's notion of “block” .

3.53 Tmpfs

Tmpfs is a file system which keeps all files in virtual memory.

Everything in tmpfs is temporary in the sense that no files will be created on your hard drive. If you unmount a tmpfs instance, everything stored therein is lost.

tmpfs puts everything into the kernel internal caches and grows and shrinks to accommodate the files it contains and is able to swap unneeded pages out to swap space. It has maximum size limits which can be adjusted on the fly via ‘mount -o remount ...’

If you compare it to ramfs (which was the template to create tmpfs) you gain swapping and limit checking. Another similar thing is the RAM disk (/dev/ram*), which

simulates a fixed size hard disk in physical RAM, where you have to create an ordinary filesystem on top. Ramdisks cannot swap and you do not have the possibility to resize them.

Since tmpfs lives completely in the page cache and on swap, all tmpfs pages will be shown as “Shmem” in /proc/meminfo and “Shared” in free(1). Notice that these counters also include shared memory (shmem, see ipcs(1)). The most reliable way to get the count is using df(1) and du(1).

tmpfs has the following uses:

- 1) There is always a kernel internal mount which you will not see at all. This is used for shared anonymous mappings and SYSV shared memory.

This mount does not depend on CONFIG_TMPFS. If CONFIG_TMPFS is not set, the user visible part of tmpfs is not build. But the internal mechanisms are always present.

- 2) glibc 2.2 and above expects tmpfs to be mounted at /dev/shm for POSIX shared memory (shm_open, shm_unlink). Adding the following line to /etc/fstab should take care of this:

tmpfs	/dev/shm	tmpfs	defaults	0	0
-------	----------	-------	----------	---	---

Remember to create the directory that you intend to mount tmpfs on if necessary.

This mount is not needed for SYSV shared memory. The internal mount is used for that. (In the 2.3 kernel versions it was necessary to mount the predecessor of tmpfs (shm fs) to use SYSV shared memory)

- 3) Some people (including me) find it very convenient to mount it e.g. on /tmp and /var/tmp and have a big swap partition. And now loop mounts of tmpfs files do work, so mkinitrd shipped by most distributions should succeed with a tmpfs /tmp.
- 4) And probably a lot more I do not know about :-)

tmpfs has three mount options for sizing:

size	The limit of allocated bytes for this tmpfs instance. The default is half of your physical RAM without swap. If you oversize your tmpfs instances the machine will deadlock since the OOM handler will not be able to free that memory.
nr_blocks	The same as size, but in blocks of PAGE_SIZE.
nr_inodes	The maximum number of inodes for this instance. The default is half of the number of your physical RAM pages, or (on a machine with highmem) the number of lowmem RAM pages, whichever is the lower.

These parameters accept a suffix k, m or g for kilo, mega and giga and can be changed on remount. The size parameter also accepts a suffix % to limit this tmpfs instance to that percentage of your physical RAM: the default, when neither size nor nr_blocks is specified, is size=50%

If nr_blocks=0 (or size=0), blocks will not be limited in that instance; if nr_inodes=0, inodes will not be limited. It is generally unwise to mount with such

options, since it allows any user with write access to use up all the memory on the machine; but enhances the scalability of that instance in a system with many cpus making intensive use of it.

tmpfs has a mount option to set the NUMA memory allocation policy for all files in that instance (if CONFIG_NUMA is enabled) - which can be adjusted on the fly via ‘mount -o remount ...’

mpol=default	use the process allocation policy (see <code>set_mempolicy(2)</code>)
mpol=prefer:Node	prefers to allocate memory from the given Node
mpol=bind:NodeList	allocates memory only from nodes in NodeList
mpol=interleave	prefers to allocate from each node in turn
mpol=interleave:NodeList	allocates from each node of NodeList in turn
mpol=local	prefers to allocate memory from the local node

NodeList format is a comma-separated list of decimal numbers and ranges, a range being two hyphen-separated decimal numbers, the smallest and largest node numbers in the range. For example, `mpol=bind:0-3,5,7,9-15`

A memory policy with a valid NodeList will be saved, as specified, for use at file creation time. When a task allocates a file in the file system, the mount option memory policy will be applied with a NodeList, if any, modified by the calling task’s cgroup constraints [See Documentation/admin-guide/cgroup-v1/cpusets.rst] and any optional flags, listed below. If the resulting NodeLists is the empty set, the effective memory policy for the file will revert to “default” policy.

NUMA memory allocation policies have optional flags that can be used in conjunction with their modes. These optional flags can be specified when tmpfs is mounted by appending them to the mode before the NodeList. See Documentation/admin-guide/mm/numa_memory_policy.rst for a list of all available memory allocation policy mode flags and their effect on memory policy.

<code>=static</code>	is equivalent to	<code>MPOL_F_STATIC_NODES</code>
<code>=relative</code>	is equivalent to	<code>MPOL_F_RELATIVE_NODES</code>

For example, `mpol=bind=static:NodeList`, is the equivalent of an allocation policy of `MPOL_BIND | MPOL_F_STATIC_NODES`.

Note that trying to mount a tmpfs with an mpol option will fail if the running kernel does not support NUMA; and will fail if its nodelist specifies a node which is not online. If your system relies on that tmpfs being mounted, but from time to time runs a kernel built without NUMA capability (perhaps a safe recovery kernel), or with fewer nodes online, then it is advisable to omit the mpol option from automatic mount options. It can be added later, when the tmpfs is already mounted on MountPoint, by ‘mount -o remount,mpol=Policy:NodeList MountPoint’ .

To specify the initial root directory you can use the following mount options:

mode	The permissions as an octal number
uid	The user id
gid	The group id

These options do not have any effect on remount. You can change these parameters with `chmod(1)`, `chown(1)` and `chgrp(1)` on a mounted filesystem.

So `'mount -t tmpfs -o size=10G,nr_inodes=10k,mode=700 tmpfs /mytmpfs'` will give you tmpfs instance on /mytmpfs which can allocate 10GB RAM/SWAP in 10240 inodes and it is only accessible by root.

Author Christoph Rohland <cr@sap.com>, 1.12.01

Updated Hugh Dickins, 4 June 2007

Updated KOSAKI Motohiro, 16 Mar 2010

3.54 UBI File System

3.54.1 Introduction

UBIFS file-system stands for UBI File System. UBI stands for “Unsorted Block Images” . UBIFS is a flash file system, which means it is designed to work with flash devices. It is important to understand, that UBIFS is completely different to any traditional file-system in Linux, like Ext2, XFS, JFS, etc. UBIFS represents a separate class of file-systems which work with MTD devices, not block devices. The other Linux file-system of this class is JFFS2.

To make it more clear, here is a small comparison of MTD devices and block devices.

- 1 MTD devices represent flash devices and they consist of eraseblocks of** rather large size, typically about 128KiB. Block devices consist of small blocks, typically 512 bytes.
- 2 MTD devices support 3 main operations - read from some offset within an** eraseblock, write to some offset within an eraseblock, and erase a whole eraseblock. Block devices support 2 main operations - read a whole block and write a whole block.
- 3 The whole eraseblock has to be erased before it becomes possible to** re-write its contents. Blocks may be just re-written.
- 4 Eraseblocks become worn out after some number of erase cycles -** typically 100K-1G for SLC NAND and NOR flashes, and 1K-10K for MLC NAND flashes. Blocks do not have the wear-out property.
- 5 Eraseblocks may become bad (only on NAND flashes) and software should** deal with this. Blocks on hard drives typically do not become bad, because hardware has mechanisms to substitute bad blocks, at least in modern LBA disks.

It should be quite obvious why UBIFS is very different to traditional file-systems.

UBIFS works on top of UBI. UBI is a separate software layer which may be found in `drivers/mtd/ubi`. UBI is basically a volume management and wear-leveling layer. It provides so called UBI volumes which is a higher level abstraction than a MTD device. The programming model of UBI devices is very similar to MTD devices - they still consist of large eraseblocks, they have read/write/erase operations, but

UBI devices are devoid of limitations like wear and bad blocks (items 4 and 5 in the above list).

In a sense, UBIFS is a next generation of JFFS2 file-system, but it is very different and incompatible to JFFS2. The following are the main differences.

- JFFS2 works on top of MTD devices, UBIFS depends on UBI and works on top of UBI volumes.
- JFFS2 does not have on-media index and has to build it while mounting, which requires full media scan. UBIFS maintains the FS indexing information on the flash media and does not require full media scan, so it mounts many times faster than JFFS2.
- JFFS2 is a write-through file-system, while UBIFS supports write-back, which makes UBIFS much faster on writes.

Similarly to JFFS2, UBIFS supports on-the-flight compression which makes it possible to fit quite a lot of data to the flash.

Similarly to JFFS2, UBIFS is tolerant of unclean reboots and power-cuts. It does not need stuff like `fsck.ext2`. UBIFS automatically replays its journal and recovers from crashes, ensuring that the on-flash data structures are consistent.

UBIFS scales logarithmically (most of the data structures it uses are trees), so the mount time and memory consumption do not linearly depend on the flash size, like in case of JFFS2. This is because UBIFS maintains the FS index on the flash media. However, UBIFS depends on UBI, which scales linearly. So overall UBI/UBIFS stack scales linearly. Nevertheless, UBI/UBIFS scales considerably better than JFFS2.

The authors of UBIFS believe, that it is possible to develop UBI2 which would scale logarithmically as well. UBI2 would support the same API as UBI, but it would be binary incompatible to UBI. So UBIFS would not need to be changed to use UBI2

3.54.2 Mount options

(*) == default.

bulk_read	read more in one go to take advantage of flash media that read faster sequentially
no_bulk_read (*)	do not bulk-read
no_chk_data	skip checking of CRCs on data nodes in order to improve read performance. Use this option only if the flash media is highly reliable. The effect of this option is that corruption of the contents of a file can go unnoticed.
chk_data	do not skip checking CRCs on data nodes
compr=none	override default compressor and set it to "none"
compr=lzo	override default compressor and set it to "lzo"
compr=zlib	override default compressor and set it to "zlib"
auth_key	specify the key used for authenticating the filesystem. Passing this option makes authentication mandatory. The passed key must be present in the kernel keyring and must be of type 'logon'
auth_hash	The hash algorithm used for authentication. Used for both hashing and for creating HMACs. Typical values include "sha256" or "sha512"

3.54.3 Quick usage instructions

The UBI volume to mount is specified using "ubiX_Y" or "ubiX:NAME" syntax, where "X" is UBI device number, "Y" is UBI volume number, and "NAME" is UBI volume name.

Mount volume 0 on UBI device 0 to /mnt/ubifs:

```
$ mount -t ubifs ubi0_0 /mnt/ubifs
```

Mount "rootfs" volume of UBI device 0 to /mnt/ubifs ("rootfs" is volume name):

```
$ mount -t ubifs ubi0:rootfs /mnt/ubifs
```

The following is an example of the kernel boot arguments to attach mtd0 to UBI and mount volume "rootfs" : ubi.mtd=0 root=ubi0:rootfs rootfstype=ubifs

3.54.4 References

UBIFS documentation and FAQ/HOWTO at the MTD web site:

- <http://www.linux-mtd.infradead.org/doc/ubifs.html>
- <http://www.linux-mtd.infradead.org/faq/ubifs.html>

3.55 Introduction

UBIFS utilizes the fscrypt framework to provide confidentiality for file contents and file names. This prevents attacks where an attacker is able to read contents of the filesystem on a single point in time. A classic example is a lost smartphone where the attacker is unable to read personal data stored on the device without the filesystem decryption key.

At the current state, UBIFS encryption however does not prevent attacks where the attacker is able to modify the filesystem contents and the user uses the device afterwards. In such a scenario an attacker can modify filesystem contents arbitrarily without the user noticing. One example is to modify a binary to perform a malicious action when executed [DMC-CBC-ATTACK]. Since most of the filesystem metadata of UBIFS is stored in plain, this makes it fairly easy to swap files and replace their contents.

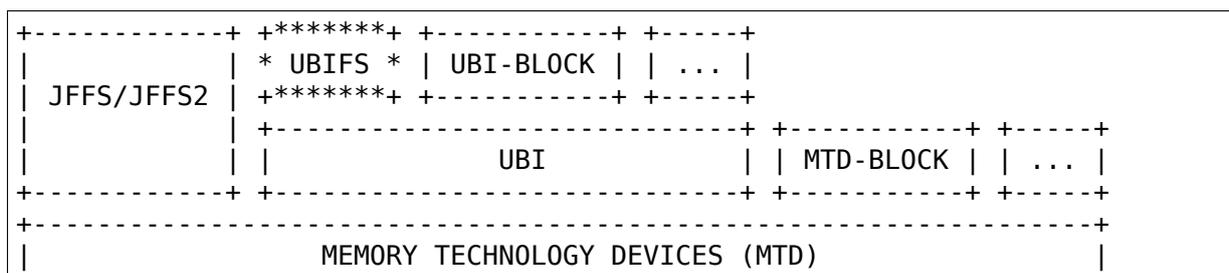
Other full disk encryption systems like dm-crypt cover all filesystem metadata, which makes such kinds of attacks more complicated, but not impossible. Especially, if the attacker is given access to the device multiple points in time. For dm-crypt and other filesystems that build upon the Linux block IO layer, the dm-integrity or dm-verity subsystems [DM-INTEGRITY, DM-VERITY] can be used to get full data authentication at the block layer. These can also be combined with dm-crypt [CRYPTSETUP2].

This document describes an approach to get file contents and full metadata authentication for UBIFS. Since UBIFS uses fscrypt for file contents and file name encryption, the authentication system could be tied into fscrypt such that existing features like key derivation can be utilized. It should however also be possible to use UBIFS authentication without using encryption.

3.55.1 MTD, UBI & UBIFS

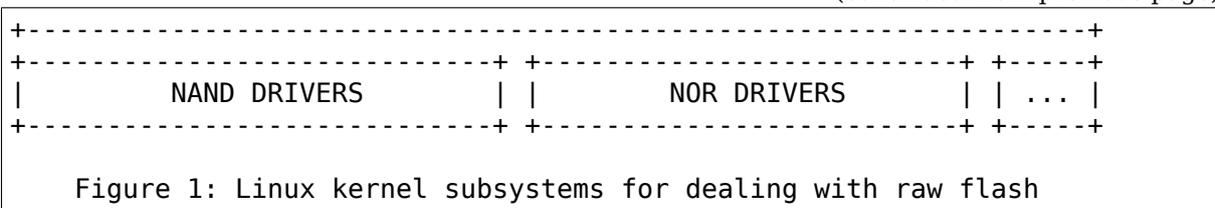
On Linux, the MTD (Memory Technology Devices) subsystem provides a uniform interface to access raw flash devices. One of the more prominent subsystems that work on top of MTD is UBI (Unsorted Block Images). It provides volume management for flash devices and is thus somewhat similar to LVM for block devices. In addition, it deals with flash-specific wear-leveling and transparent I/O error handling. UBI offers logical erase blocks (LEBs) to the layers on top of it and maps them transparently to physical erase blocks (PEBs) on the flash.

UBIFS is a filesystem for raw flash which operates on top of UBI. Thus, wear leveling and some flash specifics are left to UBI, while UBIFS focuses on scalability, performance and recoverability.



(continues on next page)

(continued from previous page)



Internally, UBIFS maintains multiple data structures which are persisted on the flash:

- Index: an on-flash B+ tree where the leaf nodes contain filesystem data
- Journal: an additional data structure to collect FS changes before updating the on-flash index and reduce flash wear.
- Tree Node Cache (TNC): an in-memory B+ tree that reflects the current FS state to avoid frequent flash reads. It is basically the in-memory representation of the index, but contains additional attributes.
- LEB property tree (LPT): an on-flash B+ tree for free space accounting per UBI LEB.

In the remainder of this section we will cover the on-flash UBIFS data structures in more detail. The TNC is of less importance here since it is never persisted onto the flash directly. More details on UBIFS can also be found in [UBIFS-WP].

UBIFS Index & Tree Node Cache

Basic on-flash UBIFS entities are called nodes. UBIFS knows different types of nodes. Eg. data nodes (`struct ubifs_data_node`) which store chunks of file contents or inode nodes (`struct ubifs_ino_node`) which represent VFS inodes. Almost all types of nodes share a common header (`ubifs_ch`) containing basic information like node type, node length, a sequence number, etc. (see `fs/ubifs/ubifs-media.h` in kernel source). Exceptions are entries of the LPT and some less important node types like padding nodes which are used to pad unusable content at the end of LEBs.

To avoid re-writing the whole B+ tree on every single change, it is implemented as wandering tree, where only the changed nodes are re-written and previous versions of them are obsoleted without erasing them right away. As a result, the index is not stored in a single place on the flash, but wanders around and there are obsolete parts on the flash as long as the LEB containing them is not reused by UBIFS. To find the most recent version of the index, UBIFS stores a special node called master node into UBI LEB 1 which always points to the most recent root node of the UBIFS index. For recoverability, the master node is additionally duplicated to LEB 2. Mounting UBIFS is thus a simple read of LEB 1 and 2 to get the current master node and from there get the location of the most recent on-flash index.

The TNC is the in-memory representation of the on-flash index. It contains some additional runtime attributes per node which are not persisted. One of these is a dirty-flag which marks nodes that have to be persisted the next time the index is written onto the flash. The TNC acts as a write-back cache and all modifications of the on-flash index are done through the TNC. Like other caches, the TNC does not

have to mirror the full index into memory, but reads parts of it from flash whenever needed. A commit is the UBIFS operation of updating the on-flash filesystem structures like the index. On every commit, the TNC nodes marked as dirty are written to the flash to update the persisted index.

Journal

To avoid wearing out the flash, the index is only persisted (committed) when certain conditions are met (eg. `fsync(2)`). The journal is used to record any changes (in form of inode nodes, data nodes etc.) between commits of the index. During mount, the journal is read from the flash and replayed onto the TNC (which will be created on-demand from the on-flash index).

UBIFS reserves a bunch of LEBs just for the journal called log area. The amount of log area LEBs is configured on filesystem creation (using `mkfs.ubifs`) and stored in the superblock node. The log area contains only two types of nodes: reference nodes and commit start nodes. A commit start node is written whenever an index commit is performed. Reference nodes are written on every journal update. Each reference node points to the position of other nodes (inode nodes, data nodes etc.) on the flash that are part of this journal entry. These nodes are called buds and describe the actual filesystem changes including their data.

The log area is maintained as a ring. Whenever the journal is almost full, a commit is initiated. This also writes a commit start node so that during mount, UBIFS will seek for the most recent commit start node and just replay every reference node after that. Every reference node before the commit start node will be ignored as they are already part of the on-flash index.

When writing a journal entry, UBIFS first ensures that enough space is available to write the reference node and buds part of this entry. Then, the reference node is written and afterwards the buds describing the file changes. On replay, UBIFS will record every reference node and inspect the location of the referenced LEBs to discover the buds. If these are corrupt or missing, UBIFS will attempt to recover them by re-reading the LEB. This is however only done for the last referenced LEB of the journal. Only this can become corrupt because of a power cut. If the recovery fails, UBIFS will not mount. An error for every other LEB will directly cause UBIFS to fail the mount operation.

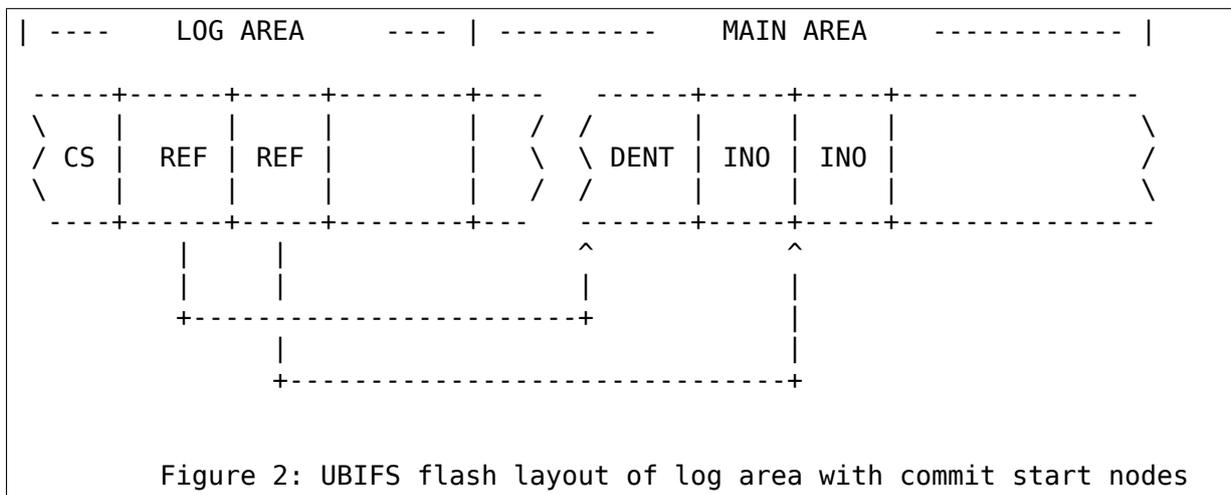


Figure 2: UBIFS flash layout of log area with commit start nodes

(continues on next page)

(continued from previous page)

(CS) and reference nodes (REF) pointing to main area containing their buds
--

LEB Property Tree/Table

The LEB property tree is used to store per-LEB information. This includes the LEB type and amount of free and dirty (old, obsolete content) space¹ on the LEB. The type is important, because UBIFS never mixes index nodes with data nodes on a single LEB and thus each LEB has a specific purpose. This again is useful for free space calculations. See [UBIFS-WP] for more details.

The LEB property tree again is a B+ tree, but it is much smaller than the index. Due to its smaller size it is always written as one chunk on every commit. Thus, saving the LPT is an atomic operation.

3.56 UBIFS Authentication

This chapter introduces UBIFS authentication which enables UBIFS to verify the authenticity and integrity of metadata and file contents stored on flash.

3.56.1 Threat Model

UBIFS authentication enables detection of offline data modification. While it does not prevent it, it enables (trusted) code to check the integrity and authenticity of on-flash file contents and filesystem metadata. This covers attacks where file contents are swapped.

UBIFS authentication will not protect against rollback of full flash contents. Ie. an attacker can still dump the flash and restore it at a later time without detection. It will also not protect against partial rollback of individual index commits. That means that an attacker is able to partially undo changes. This is possible because UBIFS does not immediately overwrites obsolete versions of the index tree or the journal, but instead marks them as obsolete and garbage collection erases them at a later time. An attacker can use this by erasing parts of the current tree and restoring old versions that are still on the flash and have not yet been erased. This is possible, because every commit will always write a new version of the index root node and the master node without overwriting the previous version. This is further helped by the wear-leveling operations of UBI which copies contents from one physical eraseblock to another and does not atomically erase the first eraseblock.

UBIFS authentication does not cover attacks where an attacker is able to execute code on the device after the authentication key was provided. Additional measures like secure boot and trusted boot have to be taken to ensure that only trusted code is executed on a device.

¹ Since LEBs can only be appended and never overwritten, there is a difference between free space ie. the remaining space left on the LEB to be written to without erasing it and previously written content that is obsolete but can't be overwritten without erasing the full LEB.

(continued from previous page)

V ...

Since the hash also includes the reference nodes an attacker cannot reorder or skip any journal heads for replay. An attacker can only remove bud nodes or reference nodes from the end of the journal, effectively rewinding the filesystem at maximum back to the last commit.

The location of the log area is stored in the master node. Since the master node is authenticated with a HMAC as described above, it is not possible to tamper with that without detection. The size of the log area is specified when the filesystem is created using `mkfs.ubifs` and stored in the superblock node. To avoid tampering with this and other values stored there, a HMAC is added to the superblock struct. The superblock node is stored in LEB 0 and is only modified on feature flag or similar changes, but never on file changes.

LPT Authentication

The location of the LPT root node on the flash is stored in the UBIFS master node. Since the LPT is written and read atomically on every commit, there is no need to authenticate individual nodes of the tree. It suffices to protect the integrity of the full LPT by a simple hash stored in the master node. Since the master node itself is authenticated, the LPTs authenticity can be verified by verifying the authenticity of the master node and comparing the LPT hash stored there with the hash computed from the read on-flash LPT.

3.56.3 Key Management

For simplicity, UBIFS authentication uses a single key to compute the HMACs of superblock, master, commit start and reference nodes. This key has to be available on creation of the filesystem (`mkfs.ubifs`) to authenticate the superblock node. Further, it has to be available on mount of the filesystem to verify authenticated nodes and generate new HMACs for changes.

UBIFS authentication is intended to operate side-by-side with UBIFS encryption (`fsencrypt`) to provide confidentiality and authenticity. Since UBIFS encryption has a different approach of encryption policies per directory, there can be multiple `fsencrypt` master keys and there might be folders without encryption. UBIFS authentication on the other hand has an all-or-nothing approach in the sense that it either authenticates everything of the filesystem or nothing. Because of this and because UBIFS authentication should also be usable without encryption, it does not share the same master key with `fsencrypt`, but manages a dedicated authentication key.

The API for providing the authentication key has yet to be defined, but the key can eg. be provided by userspace through a keyring similar to the way it is currently done in `fsencrypt`. It should however be noted that the current `fsencrypt` approach has shown its flaws and the userspace API will eventually change [FSCRYPT-POLICY2].

Nevertheless, it will be possible for a user to provide a single passphrase or key in userspace that covers UBIFS authentication and encryption. This can be solved by

the corresponding userspace tools which derive a second key for authentication in addition to the derived fscrypt master key used for encryption.

To be able to check if the proper key is available on mount, the UBIFS superblock node will additionally store a hash of the authentication key. This approach is similar to the approach proposed for fscrypt encryption policy v2 [FSCRYPT-POLICY2].

3.57 Future Extensions

In certain cases where a vendor wants to provide an authenticated filesystem image to customers, it should be possible to do so without sharing the secret UBIFS authentication key. Instead, in addition to each HMAC a digital signature could be stored where the vendor shares the public key alongside the filesystem image. In case this filesystem has to be modified afterwards, UBIFS can exchange all digital signatures with HMACs on first mount similar to the way the IMA/EVM subsystem deals with such situations. The HMAC key will then have to be provided beforehand in the normal way.

3.58 References

[CRYPTSETUP2] <http://www.saout.de/pipermail/dm-crypt/2017-November/005745.html>

[DMC-CBC-ATTACK] <http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/>

[DM-INTEGRITY] <https://www.kernel.org/doc/Documentation/device-mapper/dm-integrity.rst>

[DM-VERITY] <https://www.kernel.org/doc/Documentation/device-mapper/verity.rst>

[FSCRYPT-POLICY2] <https://www.spinics.net/lists/linux-ext4/msg58710.html>

[UBIFS-WP] http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf

3.59 UDF file system

If you encounter problems with reading UDF discs using this driver, please report them according to MAINTAINERS file.

Write support requires a block driver which supports writing. Currently dvd+rw drives and media support true random sector writes, and so a udf filesystem on such devices can be directly mounted read/write. CD-RW media however, does not support this. Instead the media can be formatted for packet mode using the utility cdrwtool, then the pktcdvd driver can be bound to the underlying cd device to provide the required buffering and read-modify-write cycles to allow the filesystem random sector writes while providing the hardware with only full packet writes. While not required for dvd+rw media, use of the pktcdvd driver often enhances performance due to very poor read-modify-write support supplied internally by drive firmware.

The following mount options are supported:

gid=	Set the default group.
umask=	Set the default umask.
mode=	Set the default file permissions.
dmode=	Set the default directory permissions.
uid=	Set the default user.
bs=	Set the block size.
unhide	Show otherwise hidden files.
undelete	Show deleted files in lists.
adinicb	Embed data in the inode (default)
noadinicb	Don't embed data in the inode
shortad	Use short ad's
longad	Use long ad's (default)
nostrict	Unset strict conformance
iocharset=	Set the NLS character set

The uid= and gid= options need a bit more explaining. They will accept a decimal numeric value and all inodes on that mount will then appear as belonging to that uid and gid. Mount options also accept the string "forget". The forget option causes all IDs to be written to disk as -1 which is a way of UDF standard to indicate that IDs are not supported for these files .

For typical desktop use of removable media, you should set the ID to that of the interactively logged on user, and also specify the forget option. This way the interactive user will always see the files on the disk as belonging to him.

The remaining are for debugging and disaster recovery:

novrs	Skip volume sequence recognition
-------	----------------------------------

The following expect a offset from 0.

session=	Set the CDROM session (default= last session)
anchor=	Override standard anchor location. (default= 256)
lastblock=	Set the last block of the filesystem/

For the latest version and toolset see: <https://github.com/pali/udftools>

Documentation on UDF and ECMA 167 is available FREE from:

- <http://www.osta.org/>
- <http://www.ecma-international.org/>

3.60 virtiofs: virtio-fs host<->guest shared file system

- Copyright (C) 2019 Red Hat, Inc.

3.60.1 Introduction

The virtiofs file system for Linux implements a driver for the paravirtualized VIR-TIO “virtio-fs” device for guest<->host file system sharing. It allows a guest to mount a directory that has been exported on the host.

Guests often require access to files residing on the host or remote systems. Use cases include making files available to new guests during installation, booting from a root file system located on the host, persistent storage for stateless or ephemeral guests, and sharing a directory between guests.

Although it is possible to use existing network file systems for some of these tasks, they require configuration steps that are hard to automate and they expose the storage network to the guest. The virtio-fs device was designed to solve these problems by providing file system access without networking.

Furthermore the virtio-fs device takes advantage of the co-location of the guest and host to increase performance and provide semantics that are not possible with network file systems.

3.60.2 Usage

Mount file system with tag myfs on /mnt:

```
guest# mount -t virtiofs myfs /mnt
```

Please see <https://virtio-fs.gitlab.io/> for details on how to configure QEMU and the virtiofsd daemon.

Mount options

virtiofs supports general VFS mount options, for example, remount, ro, rw, context, etc. It also supports FUSE mount options.

atime behavior

The atime-related mount options, for example, noatime, strictatime, are ignored. The atime behavior for virtiofs is the same as the underlying filesystem of the directory that has been exported on the host.

3.60.3 Internals

Since the virtio-fs device uses the FUSE protocol for file system requests, the virtiofs file system for Linux is integrated closely with the FUSE file system client. The guest acts as the FUSE client while the host acts as the FUSE server. The `/dev/fuse` interface between the kernel and userspace is replaced with the virtio-fs device interface.

FUSE requests are placed into a virtqueue and processed by the host. The response portion of the buffer is filled in by the host and the guest handles the request completion.

Mapping `/dev/fuse` to virtqueues requires solving differences in semantics between `/dev/fuse` and virtqueues. Each time the `/dev/fuse` device is read, the FUSE client may choose which request to transfer, making it possible to prioritize certain requests over others. Virtqueues have queue semantics and it is not possible to change the order of requests that have been enqueued. This is especially important if the virtqueue becomes full since it is then impossible to add high priority requests. In order to address this difference, the virtio-fs device uses a “hiprio” virtqueue specifically for requests that have priority over normal requests.

3.61 VFAT

3.61.1 USING VFAT

To use the vfat filesystem, use the filesystem type ‘vfat’ . i.e.:

```
mount -t vfat /dev/fd0 /mnt
```

No special partition formatter is required, ‘mkdosfs’ will work fine if you want to format from within Linux.

3.61.2 VFAT MOUNT OPTIONS

uid=### Set the owner of all files on this filesystem. The default is the uid of current process.

gid=### Set the group of all files on this filesystem. The default is the gid of current process.

umask=### The permission mask (for files and directories, see `umask(1)`). The default is the umask of current process.

dmask=### The permission mask for the directory. The default is the umask of current process.

fmask=### The permission mask for files. The default is the umask of current process.

allow_utime=### This option controls the permission check of mtime/atime.

-20: If current process is in group of file’s group ID, you can change timestamp.

-2: Other users can change timestamp.

The default is set from `dmask` option. If the directory is writable, `utime(2)` is also allowed. i.e. `~dmask & 022`.

Normally `utime(2)` checks current process is owner of the file, or it has `CAP_FOWNER` capability. But FAT filesystem doesn't have uid/gid on disk, so normal check is too unflexible. With this option you can relax it.

codepage=### Sets the codepage number for converting to shortname characters on FAT filesystem. By default, `FAT_DEFAULT_CODEPAGE` setting is used.

iocharset=<name> Character set to use for converting between the encoding is used for user visible filename and 16 bit Unicode characters. Long filenames are stored on disk in Unicode format, but Unix for the most part doesn't know how to deal with Unicode. By default, `FAT_DEFAULT_IOCHARSET` setting is used.

There is also an option of doing UTF-8 translations with the `utf8` option.

Note: `iocharset=utf8` is not recommended. If unsure, you should consider the `utf8` option instead.

utf8=<bool> UTF-8 is the filesystem safe version of Unicode that is used by the console. It can be enabled or disabled for the filesystem with this option. If `'uni_xlate'` gets set, UTF-8 gets disabled. By default, `FAT_DEFAULT_UTF8` setting is used.

uni_xlate=<bool> Translate unhandled Unicode characters to special escaped sequences. This would let you backup and restore filenames that are created with any Unicode characters. Until Linux supports Unicode for real, this gives you an alternative. Without this option, a `'?`' is used when no translation is possible. The escape character is `'\'` because it is otherwise illegal on the `vfat` filesystem. The escape sequence that gets used is `'\'` and the four digits of hexadecimal unicode.

nonumtail=<bool> When creating 8.3 aliases, normally the alias will end in `'~1'` or tilde followed by some number. If this option is set, then if the filename is `"longfilename.txt"` and `"longfile.txt"` does not currently exist in the directory, `longfile.txt` will be the short alias instead of `longfi~1.txt`.

usefree Use the "free clusters" value stored on `FSINFO`. It will be used to determine number of free clusters without scanning disk. But it's not used by default, because recent Windows don't update it correctly in some case. If you are sure the "free clusters" on `FSINFO` is correct, by this option you can avoid scanning disk.

quiet Stops printing certain warning messages.

check=s|r|n Case sensitivity checking setting.

s: strict, case sensitive

r: relaxed, case insensitive

n: normal, default setting, currently case insensitive

nocase This was deprecated for vfat. Use `shortname=win95` instead.

shortname=lower|win95|winnt|mixed Shortname display/create setting.

lower: convert to lowercase for display, emulate the Windows 95 rule for create.

win95: emulate the Windows 95 rule for display/create.

winnt: emulate the Windows NT rule for display/create.

mixed: emulate the Windows NT rule for display, emulate the Windows 95 rule for create.

Default setting is mixed.

tz=UTC Interpret timestamps as UTC rather than local time. This option disables the conversion of timestamps between local time (as used by Windows on FAT) and UTC (which Linux uses internally). This is particularly useful when mounting devices (like digital cameras) that are set to UTC in order to avoid the pitfalls of local time.

time_offset=minutes Set offset for conversion of timestamps from local time used by FAT to UTC. I.e. `<minutes>` minutes will be subtracted from each timestamp to convert it to UTC used internally by Linux. This is useful when time zone set in `sys_tz` is not the time zone used by the filesystem. Note that this option still does not provide correct time stamps in all cases in presence of DST - time stamps in a different DST setting will be off by one hour.

showexec If set, the execute permission bits of the file will be allowed only if the extension part of the name is `.EXE`, `.COM`, or `.BAT`. Not set by default.

debug Can be set, but unused by the current implementation.

sys_immutable If set, `ATTR_SYS` attribute on FAT is handled as `IMMUTABLE` flag on Linux. Not set by default.

flush If set, the filesystem will try to flush to disk more early than normal. Not set by default.

rodir FAT has the `ATTR_RO` (read-only) attribute. On Windows, the `ATTR_RO` of the directory will just be ignored, and is used only by applications as a flag (e.g. it's set for the customized folder).

If you want to use `ATTR_RO` as read-only flag even for the directory, set this option.

errors=panic|continue|remount-ro specify FAT behavior on critical errors: panic, continue without doing anything or remount the partition in read-only mode (default behavior).

discard If set, issues `discard/TRIM` commands to the block device when blocks are freed. This is useful for SSD devices and sparse/thinly-provisioned LUNs.

nfs=stale_rw|nostale_ro Enable this only if you want to export the FAT filesystem over NFS.

stale_rw: This option maintains an index (cache) of directory inodes by `i_logstart` which is used by the nfs-related code to improve look-

ups. Full file operations (read/write) over NFS is supported but with cache eviction at NFS server, this could result in ESTALE issues.

nostale_ro: This option bases the inode number and filehandle on the on-disk location of a file in the MS-DOS directory entry. This ensures that ESTALE will not be returned after a file is evicted from the inode cache. However, it means that operations such as rename, create and unlink could cause filehandles that previously pointed at one file to point at a different file, potentially causing data corruption. For this reason, this option also mounts the filesystem readonly.

To maintain backward compatibility, ' -o nfs ' is also accepted, defaulting to "stale_rw" .

dos1xfloppy <bool>: 0,1,yes,no,true,false If set, use a fallback default BIOS Parameter Block configuration, determined by backing device size. These static parameters match defaults assumed by DOS 1.x for 160 kiB, 180 kiB, 320 kiB, and 360 kiB floppies and floppy images.

3.61.3 LIMITATION

The fallocated region of file is discarded at umount/evict time when using fallocate with FALLOC_FL_KEEP_SIZE. So, User should assume that fallocated region can be discarded at last close if there is memory pressure resulting in eviction of the inode from the memory. As a result, for any dependency on the fallocated region, user should make sure to recheck fallocate after reopening the file.

3.61.4 TODO

Need to get rid of the raw scanning stuff. Instead, always use a get next directory entry approach. The only thing left that uses raw scanning is the directory renaming code.

3.61.5 POSSIBLE PROBLEMS

- vfat_valid_longname does not properly checked reserved names.
- When a volume name is the same as a directory name in the root directory of the filesystem, the directory name sometimes shows up as an empty file.
- autoconv option does not work correctly.

3.61.6 TEST SUITE

If you plan to make any modifications to the vfat filesystem, please get the test suite that comes with the vfat distribution at

http://web.archive.org/web/*/http://bmr.c.berkeley.edu/people/chaffee/vfat.html

This tests quite a few parts of the vfat filesystem and additional tests for new features or untested features would be appreciated.

3.61.7 NOTES ON THE STRUCTURE OF THE VFAT FILESYSTEM

This documentation was provided by Galen C. Hunt gchunt@cs.rochester.edu and lightly annotated by Gordon Chaffee.

This document presents a very rough, technical overview of my knowledge of the extended FAT file system used in Windows NT 3.5 and Windows 95. I don't guarantee that any of the following is correct, but it appears to be so.

The extended FAT file system is almost identical to the FAT file system used in DOS versions up to and including 6.223410239847 :-). The significant change has been the addition of long file names. These names support up to 255 characters including spaces and lower case characters as opposed to the traditional 8.3 short names.

Here is the description of the traditional FAT entry in the current Windows 95 filesystem:

```
struct directory { // Short 8.3 names
    unsigned char name[8];           // file name
    unsigned char ext[3];            // file extension
    unsigned char attr;              // attribute byte
    unsigned char lcase;            // Case for base and extension
    unsigned char ctime_ms;         // Creation time, milliseconds
    unsigned char ctime[2];         // Creation time
    unsigned char cdate[2];         // Creation date
    unsigned char adate[2];         // Last access date
    unsigned char reserved[2];      // reserved values (ignored)
    unsigned char time[2];          // time stamp
    unsigned char date[2];          // date stamp
    unsigned char start[2];         // starting cluster number
    unsigned char size[4];          // size of the file
};
```

The lcase field specifies if the base and/or the extension of an 8.3 name should be capitalized. This field does not seem to be used by Windows 95 but it is used by Windows NT. The case of filenames is not completely compatible from Windows NT to Windows 95. It is not completely compatible in the reverse direction, however. Filenames that fit in the 8.3 namespace and are written on Windows NT to be lowercase will show up as uppercase on Windows 95.

Note: Note that the start and size values are actually little endian integer values. The descriptions of the fields in this structure are public knowledge and can be found elsewhere.

With the extended FAT system, Microsoft has inserted extra directory entries for any files with extended names. (Any name which legally fits within the old 8.3 encoding scheme does not have extra entries.) I call these extra entries slots. Basically, a slot is a specially formatted directory entry which holds up to 13 characters of a file's extended name. Think of slots as additional labeling for the directory entry of the file to which they correspond. Microsoft prefers to refer to the 8.3 entry for a file as its alias and the extended slot directory entries as the file name.

The C structure for a slot directory entry follows:

```
struct slot { // Up to 13 characters of a long name
    unsigned char id;           // sequence number for slot
    unsigned char name0_4[10];  // first 5 characters in name
    unsigned char attr;        // attribute byte
    unsigned char reserved;     // always 0
    unsigned char alias_checksum; // checksum for 8.3 alias
    unsigned char name5_10[12]; // 6 more characters in name
    unsigned char start[2];     // starting cluster number
    unsigned char name11_12[4]; // last 2 characters in name
};
```

If the layout of the slots looks a little odd, it's only because of Microsoft's efforts to maintain compatibility with old software. The slots must be disguised to prevent old software from panicking. To this end, a number of measures are taken:

- 1) The attribute byte for a slot directory entry is always set to 0x0f. This corresponds to an old directory entry with attributes of "hidden", "system", "read-only", and "volume label". Most old software will ignore any directory entries with the "volume label" bit set. Real volume label entries don't have the other three bits set.
- 2) The starting cluster is always set to 0, an impossible value for a DOS file.

Because the extended FAT system is backward compatible, it is possible for old software to modify directory entries. Measures must be taken to ensure the validity of slots. An extended FAT system can verify that a slot does in fact belong to an 8.3 directory entry by the following:

- 1) Positioning. Slots for a file always immediately proceed their corresponding 8.3 directory entry. In addition, each slot has an id which marks its order in the extended file name. Here is a very abbreviated view of an 8.3 directory entry and its corresponding long name slots for the file "My Big File.Extension which is long" :

```
<proceeding files...>
<slot #3, id = 0x43, characters = "h is long">
<slot #2, id = 0x02, characters = "xtension whic">
<slot #1, id = 0x01, characters = "My Big File.E">
<directory entry, name = "MYBIGFIL.EXT">
```

Note: Note that the slots are stored from last to first. Slots are numbered from 1 to N. The Nth slot is or'd with 0x40 to mark it as the last one.

- 2) Checksum. Each slot has an `alias_checksum` value. The checksum is calculated from the 8.3 name using the following algorithm:

```
for (sum = i = 0; i < 11; i++) {
    sum = (((sum&1)<<7)|((sum&0xfe)>>1)) + name[i]
}
```

- 3) If there is free space in the final slot, a Unicode NULL (`0x0000`) is stored after the final character. After that, all unused characters in the final slot are set to Unicode `0xFFFF`.

Finally, note that the extended name is stored in Unicode. Each Unicode character takes either two or four bytes, UTF-16LE encoded.

3.62 XFS Delayed Logging Design

3.62.1 Introduction to Re-logging in XFS

XFS logging is a combination of logical and physical logging. Some objects, such as inodes and dquots, are logged in logical format where the details logged are made up of the changes to in-core structures rather than on-disk structures. Other objects - typically buffers - have their physical changes logged. The reason for these differences is to reduce the amount of log space required for objects that are frequently logged. Some parts of inodes are more frequently logged than others, and inodes are typically more frequently logged than any other object (except maybe the superblock buffer) so keeping the amount of metadata logged low is of prime importance.

The reason that this is such a concern is that XFS allows multiple separate modifications to a single object to be carried in the log at any given time. This allows the log to avoid needing to flush each change to disk before recording a new change to the object. XFS does this via a method called “re-logging”. Conceptually, this is quite simple - all it requires is that any new change to the object is recorded with a new copy of all the existing changes in the new transaction that is written to the log.

That is, if we have a sequence of changes A through to F, and the object was written to disk after change D, we would see in the log the following series of transactions, their contents and the log sequence number (LSN) of the transaction:

Transaction	Contents	LSN
A	A	X
B	A+B	X+n
C	A+B+C	X+n+m
D	A+B+C+D	X+n+m+o
<object written to disk>		
E	E	Y (> X+n+m+o)
F	E+F	Y+p

In other words, each time an object is relogged, the new transaction contains the aggregation of all the previous changes currently held only in the log.

This relogging technique also allows objects to be moved forward in the log so that an object being relogged does not prevent the tail of the log from ever moving

forward. This can be seen in the table above by the changing (increasing) LSN of each subsequent transaction - the LSN is effectively a direct encoding of the location in the log of the transaction.

This relogging is also used to implement long-running, multiple-commit transactions. These transactions are known as rolling transactions, and require a special log reservation known as a permanent transaction reservation. A typical example of a rolling transaction is the removal of extents from an inode which can only be done at a rate of two extents per transaction because of reservation size limitations. Hence a rolling extent removal transaction keeps relogging the inode and btree buffers as they get modified in each removal operation. This keeps them moving forward in the log as the operation progresses, ensuring that current operation never gets blocked by itself if the log wraps around.

Hence it can be seen that the relogging operation is fundamental to the correct working of the XFS journalling subsystem. From the above description, most people should be able to see why the XFS metadata operations writes so much to the log - repeated operations to the same objects write the same changes to the log over and over again. Worse is the fact that objects tend to get dirtier as they get relogged, so each subsequent transaction is writing more metadata into the log.

Another feature of the XFS transaction subsystem is that most transactions are asynchronous. That is, they don't commit to disk until either a log buffer is filled (a log buffer can hold multiple transactions) or a synchronous operation forces the log buffers holding the transactions to disk. This means that XFS is doing aggregation of transactions in memory - batching them, if you like - to minimise the impact of the log IO on transaction throughput.

The limitation on asynchronous transaction throughput is the number and size of log buffers made available by the log manager. By default there are 8 log buffers available and the size of each is 32kB - the size can be increased up to 256kB by use of a mount option.

Effectively, this gives us the maximum bound of outstanding metadata changes that can be made to the filesystem at any point in time - if all the log buffers are full and under IO, then no more transactions can be committed until the current batch completes. It is now common for a single current CPU core to be able to issue enough transactions to keep the log buffers full and under IO permanently. Hence the XFS journalling subsystem can be considered to be IO bound.

3.62.2 Delayed Logging: Concepts

The key thing to note about the asynchronous logging combined with the relogging technique XFS uses is that we can be relogging changed objects multiple times before they are committed to disk in the log buffers. If we return to the previous relogging example, it is entirely possible that transactions A through D are committed to disk in the same log buffer.

That is, a single log buffer may contain multiple copies of the same object, but only one of those copies needs to be there - the last one "D", as it contains all the changes from the previous changes. In other words, we have one necessary copy in the log buffer, and three stale copies that are simply wasting space. When we are doing repeated operations on the same set of objects, these "stale objects" can be over 90% of the space used in the log buffers. It is clear that reducing

the number of stale objects written to the log would greatly reduce the amount of metadata we write to the log, and this is the fundamental goal of delayed logging.

From a conceptual point of view, XFS is already doing relogging in memory (where memory == log buffer), only it is doing it extremely inefficiently. It is using logical to physical formatting to do the relogging because there is no infrastructure to keep track of logical changes in memory prior to physically formatting the changes in a transaction to the log buffer. Hence we cannot avoid accumulating stale objects in the log buffers.

Delayed logging is the name we've given to keeping and tracking transactional changes to objects in memory outside the log buffer infrastructure. Because of the relogging concept fundamental to the XFS journalling subsystem, this is actually relatively easy to do - all the changes to logged items are already tracked in the current infrastructure. The big problem is how to accumulate them and get them to the log in a consistent, recoverable manner. Describing the problems and how they have been solved is the focus of this document.

One of the key changes that delayed logging makes to the operation of the journalling subsystem is that it disassociates the amount of outstanding metadata changes from the size and number of log buffers available. In other words, instead of there only being a maximum of 2MB of transaction changes not written to the log at any point in time, there may be a much greater amount being accumulated in memory. Hence the potential for loss of metadata on a crash is much greater than for the existing logging mechanism.

It should be noted that this does not change the guarantee that log recovery will result in a consistent filesystem. What it does mean is that as far as the recovered filesystem is concerned, there may be many thousands of transactions that simply did not occur as a result of the crash. This makes it even more important that applications that care about their data use `fsync()` where they need to ensure application level data integrity is maintained.

It should be noted that delayed logging is not an innovative new concept that warrants rigorous proofs to determine whether it is correct or not. The method of accumulating changes in memory for some period before writing them to the log is used effectively in many filesystems including `ext3` and `ext4`. Hence no time is spent in this document trying to convince the reader that the concept is sound. Instead it is simply considered a "solved problem" and as such implementing it in XFS is purely an exercise in software engineering.

The fundamental requirements for delayed logging in XFS are simple:

1. Reduce the amount of metadata written to the log by at least an order of magnitude.
2. Supply sufficient statistics to validate Requirement #1.
3. Supply sufficient new tracing infrastructure to be able to debug problems with the new code.
4. No on-disk format change (metadata or log format).
5. Enable and disable with a mount option.
6. No performance regressions for synchronous transaction workloads.

3.62.3 Delayed Logging: Design

Storing Changes

The problem with accumulating changes at a logical level (i.e. just using the existing log item dirty region tracking) is that when it comes to writing the changes to the log buffers, we need to ensure that the object we are formatting is not changing while we do this. This requires locking the object to prevent concurrent modification. Hence flushing the logical changes to the log would require us to lock every object, format them, and then unlock them again.

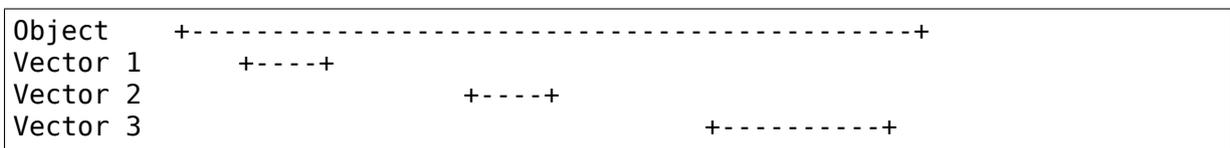
This introduces lots of scope for deadlocks with transactions that are already running. For example, a transaction has object A locked and modified, but needs the delayed logging tracking lock to commit the transaction. However, the flushing thread has the delayed logging tracking lock already held, and is trying to get the lock on object A to flush it to the log buffer. This appears to be an unsolvable deadlock condition, and it was solving this problem that was the barrier to implementing delayed logging for so long.

The solution is relatively simple - it just took a long time to recognise it. Put simply, the current logging code formats the changes to each item into an vector array that points to the changed regions in the item. The log write code simply copies the memory these vectors point to into the log buffer during transaction commit while the item is locked in the transaction. Instead of using the log buffer as the destination of the formatting code, we can use an allocated memory buffer big enough to fit the formatted vector.

If we then copy the vector into the memory buffer and rewrite the vector to point to the memory buffer rather than the object itself, we now have a copy of the changes in a format that is compatible with the log buffer writing code. that does not require us to lock the item to access. This formatting and rewriting can all be done while the object is locked during transaction commit, resulting in a vector that is transactionally consistent and can be accessed without needing to lock the owning item.

Hence we avoid the need to lock items when we need to flush outstanding asynchronous transactions to the log. The differences between the existing formatting method and the delayed logging formatting can be seen in the diagram below.

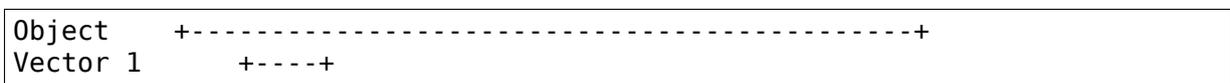
Current format log vector:



After formatting:



Delayed logging vector:



(continues on next page)

(continued from previous page)

Vector 2	+-----+	
Vector 3		+-----+

After formatting:

Memory Buffer	+V1-V2-----V3-----+
Vector 1	+-----+
Vector 2	+-----+
Vector 3	+-----+

The memory buffer and associated vector need to be passed as a single object, but still need to be associated with the parent object so if the object is relogged we can replace the current memory buffer with a new memory buffer that contains the latest changes.

The reason for keeping the vector around after we've formatted the memory buffer is to support splitting vectors across log buffer boundaries correctly. If we don't keep the vector around, we do not know where the region boundaries are in the item, so we'd need a new encapsulation method for regions in the log buffer writing (i.e. double encapsulation). This would be an on-disk format change and as such is not desirable. It also means we'd have to write the log region headers in the formatting stage, which is problematic as there is per region state that needs to be placed into the headers during the log write.

Hence we need to keep the vector, but by attaching the memory buffer to it and rewriting the vector addresses to point at the memory buffer we end up with a self-describing object that can be passed to the log buffer write code to be handled in exactly the same manner as the existing log vectors are handled. Hence we avoid needing a new on-disk format to handle items that have been relogged in memory.

Tracking Changes

Now that we can record transactional changes in memory in a form that allows them to be used without limitations, we need to be able to track and accumulate them so that they can be written to the log at some later point in time. The log item is the natural place to store this vector and buffer, and also makes sense to be the object that is used to track committed objects as it will always exist once the object has been included in a transaction.

The log item is already used to track the log items that have been written to the log but not yet written to disk. Such log items are considered "active" and as such are stored in the Active Item List (AIL) which is a LSN-ordered double linked list. Items are inserted into this list during log buffer IO completion, after which they are unpinned and can be written to disk. An object that is in the AIL can be relogged, which causes the object to be pinned again and then moved forward in the AIL when the log buffer IO completes for that transaction.

Essentially, this shows that an item that is in the AIL can still be modified and relogged, so any tracking must be separate to the AIL infrastructure. As such, we cannot reuse the AIL list pointers for tracking committed items, nor can we store state in any field that is protected by the AIL lock. Hence the committed item tracking needs its own locks, lists and state fields in the log item.

Similar to the AIL, tracking of committed items is done through a new list called the Committed Item List (CIL). The list tracks log items that have been committed and have formatted memory buffers attached to them. It tracks objects in transaction commit order, so when an object is relogged it is removed from its place in the list and re-inserted at the tail. This is entirely arbitrary and done to make it easy for debugging - the last items in the list are the ones that are most recently modified. Ordering of the CIL is not necessary for transactional integrity (as discussed in the next section) so the ordering is done for convenience/sanity of the developers.

Delayed Logging: Checkpoints

When we have a log synchronisation event, commonly known as a “log force” , all the items in the CIL must be written into the log via the log buffers. We need to write these items in the order that they exist in the CIL, and they need to be written as an atomic transaction. The need for all the objects to be written as an atomic transaction comes from the requirements of relogging and log replay - all the changes in all the objects in a given transaction must either be completely replayed during log recovery, or not replayed at all. If a transaction is not replayed because it is not complete in the log, then no later transactions should be replayed, either.

To fulfill this requirement, we need to write the entire CIL in a single log transaction. Fortunately, the XFS log code has no fixed limit on the size of a transaction, nor does the log replay code. The only fundamental limit is that the transaction cannot be larger than just under half the size of the log. The reason for this limit is that to find the head and tail of the log, there must be at least one complete transaction in the log at any given time. If a transaction is larger than half the log, then there is the possibility that a crash during the write of a such a transaction could partially overwrite the only complete previous transaction in the log. This will result in a recovery failure and an inconsistent filesystem and hence we must enforce the maximum size of a checkpoint to be slightly less than a half the log.

Apart from this size requirement, a checkpoint transaction looks no different to any other transaction - it contains a transaction header, a series of formatted log items and a commit record at the tail. From a recovery perspective, the checkpoint transaction is also no different - just a lot bigger with a lot more items in it. The worst case effect of this is that we might need to tune the recovery transaction object hash size.

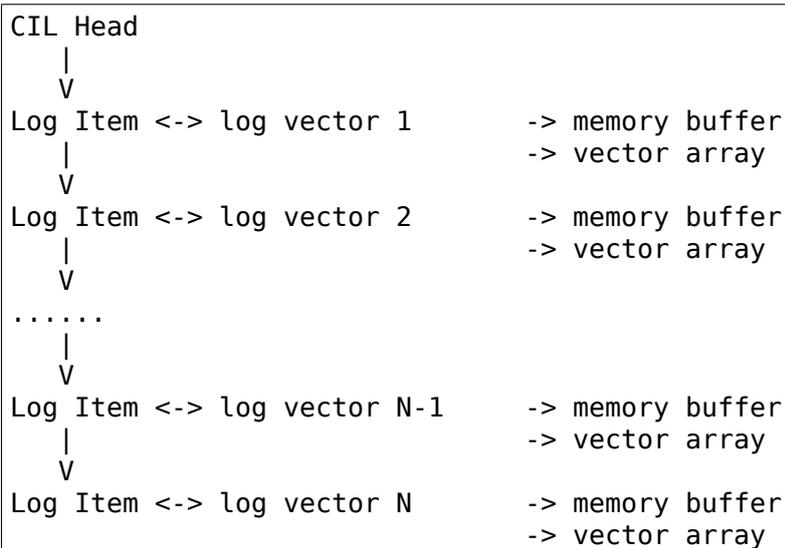
Because the checkpoint is just another transaction and all the changes to log items are stored as log vectors, we can use the existing log buffer writing code to write the changes into the log. To do this efficiently, we need to minimise the time we hold the CIL locked while writing the checkpoint transaction. The current log write code enables us to do this easily with the way it separates the writing of the transaction contents (the log vectors) from the transaction commit record, but tracking this requires us to have a per-checkpoint context that travels through the log write process through to checkpoint completion.

Hence a checkpoint has a context that tracks the state of the current checkpoint from initiation to checkpoint completion. A new context is initiated at the same time a checkpoint transaction is started. That is, when we remove all the current items from the CIL during a checkpoint operation, we move all those changes into

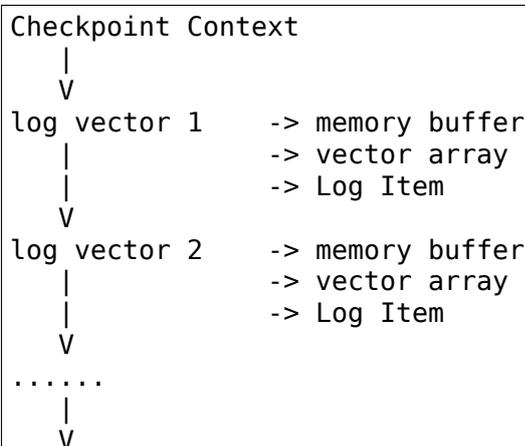
the current checkpoint context. We then initialise a new context and attach that to the CIL for aggregation of new transactions.

This allows us to unlock the CIL immediately after transfer of all the committed items and effectively allow new transactions to be issued while we are formatting the checkpoint into the log. It also allows concurrent checkpoints to be written into the log buffers in the case of log force heavy workloads, just like the existing transaction commit code does. This, however, requires that we strictly order the commit records in the log so that checkpoint sequence order is maintained during log replay.

To ensure that we can be writing an item into a checkpoint transaction at the same time another transaction modifies the item and inserts the log item into the new CIL, then checkpoint transaction commit code cannot use log items to store the list of log vectors that need to be written into the transaction. Hence log vectors need to be able to be chained together to allow them to be detached from the log items. That is, when the CIL is flushed the memory buffer and log vector attached to each log item needs to be attached to the checkpoint context so that the log item can be released. In diagrammatic form, the CIL would look like this before the flush:



And after the flush the CIL head is empty, and the checkpoint context log vector list would look like:



(continues on next page)

(continued from previous page)

```
log vector N-1  -> memory buffer
  |             -> vector array
  |             -> Log Item
  v
log vector N    -> memory buffer
  |             -> vector array
  |             -> Log Item
```

Once this transfer is done, the CIL can be unlocked and new transactions can start, while the checkpoint flush code works over the log vector chain to commit the checkpoint.

Once the checkpoint is written into the log buffers, the checkpoint context is attached to the log buffer that the commit record was written to along with a completion callback. Log IO completion will call that callback, which can then run transaction committed processing for the log items (i.e. insert into AIL and unpin) in the log vector chain and then free the log vector chain and checkpoint context.

Discussion Point: I am uncertain as to whether the log item is the most efficient way to track vectors, even though it seems like the natural way to do it. The fact that we walk the log items (in the CIL) just to chain the log vectors and break the link between the log item and the log vector means that we take a cache line hit for the log item list modification, then another for the log vector chaining. If we track by the log vectors, then we only need to break the link between the log item and the log vector, which means we should dirty only the log item cachelines. Normally I wouldn't be concerned about one vs two dirty cachelines except for the fact I've seen upwards of 80,000 log vectors in one checkpoint transaction. I'd guess this is a "measure and compare" situation that can be done after a working and reviewed implementation is in the dev tree...

Delayed Logging: Checkpoint Sequencing

One of the key aspects of the XFS transaction subsystem is that it tags committed transactions with the log sequence number of the transaction commit. This allows transactions to be issued asynchronously even though there may be future operations that cannot be completed until that transaction is fully committed to the log. In the rare case that a dependent operation occurs (e.g. re-using a freed metadata extent for a data extent), a special, optimised log force can be issued to force the dependent transaction to disk immediately.

To do this, transactions need to record the LSN of the commit record of the transaction. This LSN comes directly from the log buffer the transaction is written into. While this works just fine for the existing transaction mechanism, it does not work for delayed logging because transactions are not written directly into the log buffers. Hence some other method of sequencing transactions is required.

As discussed in the checkpoint section, delayed logging uses per-checkpoint contexts, and as such it is simple to assign a sequence number to each checkpoint. Because the switching of checkpoint contexts must be done atomically, it is simple to ensure that each new context has a monotonically increasing sequence number assigned to it without the need for an external atomic counter - we can just take the current context sequence number and add one to it for the new context.

Then, instead of assigning a log buffer LSN to the transaction commit LSN during the commit, we can assign the current checkpoint sequence. This allows operations that track transactions that have not yet completed know what checkpoint sequence needs to be committed before they can continue. As a result, the code that forces the log to a specific LSN now needs to ensure that the log forces to a specific checkpoint.

To ensure that we can do this, we need to track all the checkpoint contexts that are currently committing to the log. When we flush a checkpoint, the context gets added to a “committing” list which can be searched. When a checkpoint commit completes, it is removed from the committing list. Because the checkpoint context records the LSN of the commit record for the checkpoint, we can also wait on the log buffer that contains the commit record, thereby using the existing log force mechanisms to execute synchronous forces.

It should be noted that the synchronous forces may need to be extended with mitigation algorithms similar to the current log buffer code to allow aggregation of multiple synchronous transactions if there are already synchronous transactions being flushed. Investigation of the performance of the current design is needed before making any decisions here.

The main concern with log forces is to ensure that all the previous checkpoints are also committed to disk before the one we need to wait for. Therefore we need to check that all the prior contexts in the committing list are also complete before waiting on the one we need to complete. We do this synchronisation in the log force code so that we don't need to wait anywhere else for such serialisation - it only matters when we do a log force.

The only remaining complexity is that a log force now also has to handle the case where the forcing sequence number is the same as the current context. That is, we need to flush the CIL and potentially wait for it to complete. This is a simple addition to the existing log forcing code to check the sequence numbers and push if required. Indeed, placing the current sequence checkpoint flush in the log force code enables the current mechanism for issuing synchronous transactions to remain untouched (i.e. commit an asynchronous transaction, then force the log at the LSN of that transaction) and so the higher level code behaves the same regardless of whether delayed logging is being used or not.

Delayed Logging: Checkpoint Log Space Accounting

The big issue for a checkpoint transaction is the log space reservation for the transaction. We don't know how big a checkpoint transaction is going to be ahead of time, nor how many log buffers it will take to write out, nor the number of split log vector regions are going to be used. We can track the amount of log space required as we add items to the commit item list, but we still need to reserve the space in the log for the checkpoint.

A typical transaction reserves enough space in the log for the worst case space usage of the transaction. The reservation accounts for log record headers, transaction and region headers, headers for split regions, buffer tail padding, etc. as well as the actual space for all the changed metadata in the transaction. While some of this is fixed overhead, much of it is dependent on the size of the transaction and the number of regions being logged (the number of log vectors in the

transaction).

An example of the differences would be logging directory changes versus logging inode changes. If you modify lots of inode cores (e.g. `chmod -R g+w *`), then there are lots of transactions that only contain an inode core and an inode log format structure. That is, two vectors totaling roughly 150 bytes. If we modify 10,000 inodes, we have about 1.5MB of metadata to write in 20,000 vectors. Each vector is 12 bytes, so the total to be logged is approximately 1.75MB. In comparison, if we are logging full directory buffers, they are typically 4KB each, so we in 1.5MB of directory buffers we'd have roughly 400 buffers and a buffer format structure for each buffer - roughly 800 vectors or 1.51MB total space. From this, it should be obvious that a static log space reservation is not particularly flexible and is difficult to select the "optimal value" for all workloads.

Further, if we are going to use a static reservation, which bit of the entire reservation does it cover? We account for space used by the transaction reservation by tracking the space currently used by the object in the CIL and then calculating the increase or decrease in space used as the object is relogged. This allows for a checkpoint reservation to only have to account for log buffer metadata used such as log header records.

However, even using a static reservation for just the log metadata is problematic. Typically log record headers use at least 16KB of log space per 1MB of log space consumed (512 bytes per 32k) and the reservation needs to be large enough to handle arbitrary sized checkpoint transactions. This reservation needs to be made before the checkpoint is started, and we need to be able to reserve the space without sleeping. For a 8MB checkpoint, we need a reservation of around 150KB, which is a non-trivial amount of space.

A static reservation needs to manipulate the log grant counters - we can take a permanent reservation on the space, but we still need to make sure we refresh the write reservation (the actual space available to the transaction) after every checkpoint transaction completion. Unfortunately, if this space is not available when required, then the regrant code will sleep waiting for it.

The problem with this is that it can lead to deadlocks as we may need to commit checkpoints to be able to free up log space (refer back to the description of rolling transactions for an example of this). Hence we must always have space available in the log if we are to use static reservations, and that is very difficult and complex to arrange. It is possible to do, but there is a simpler way.

The simpler way of doing this is tracking the entire log space used by the items in the CIL and using this to dynamically calculate the amount of log space required by the log metadata. If this log metadata space changes as a result of a transaction commit inserting a new memory buffer into the CIL, then the difference in space required is removed from the transaction that causes the change. Transactions at this level will always have enough space available in their reservation for this as they have already reserved the maximal amount of log metadata space they require, and such a delta reservation will always be less than or equal to the maximal amount in the reservation.

Hence we can grow the checkpoint transaction reservation dynamically as items are added to the CIL and avoid the need for reserving and regrating log space up front. This avoids deadlocks and removes a blocking point from the checkpoint flush code.

As mentioned early, transactions can't grow to more than half the size of the log. Hence as part of the reservation growing, we need to also check the size of the reservation against the maximum allowed transaction size. If we reach the maximum threshold, we need to push the CIL to the log. This is effectively a "background flush" and is done on demand. This is identical to a CIL push triggered by a log force, only that there is no waiting for the checkpoint commit to complete. This background push is checked and executed by transaction commit code.

If the transaction subsystem goes idle while we still have items in the CIL, they will be flushed by the periodic log force issued by the `xfssyncd`. This log force will push the CIL to disk, and if the transaction subsystem stays idle, allow the idle log to be covered (effectively marked clean) in exactly the same manner that is done for the existing logging method. A discussion point is whether this log force needs to be done more frequently than the current rate which is once every 30s.

Delayed Logging: Log Item Pinning

Currently log items are pinned during transaction commit while the items are still locked. This happens just after the items are formatted, though it could be done any time before the items are unlocked. The result of this mechanism is that items get pinned once for every transaction that is committed to the log buffers. Hence items that are relogged in the log buffers will have a pin count for every outstanding transaction they were dirtied in. When each of these transactions is completed, they will unpin the item once. As a result, the item only becomes unpinned when all the transactions complete and there are no pending transactions. Thus the pinning and unpinning of a log item is symmetric as there is a 1:1 relationship with transaction commit and log item completion.

For delayed logging, however, we have an asymmetric transaction commit to completion relationship. Every time an object is relogged in the CIL it goes through the commit process without a corresponding completion being registered. That is, we now have a many-to-one relationship between transaction commit and log item completion. The result of this is that pinning and unpinning of the log items becomes unbalanced if we retain the "pin on transaction commit, unpin on transaction completion" model.

To keep pin/unpin symmetry, the algorithm needs to change to a "pin on insertion into the CIL, unpin on checkpoint completion". In other words, the pinning and unpinning becomes symmetric around a checkpoint context. We have to pin the object the first time it is inserted into the CIL - if it is already in the CIL during a transaction commit, then we do not pin it again. Because there can be multiple outstanding checkpoint contexts, we can still see elevated pin counts, but as each checkpoint completes the pin count will retain the correct value according to it's context.

Just to make matters more slightly more complex, this checkpoint level context for the pin count means that the pinning of an item must take place under the CIL commit/flush lock. If we pin the object outside this lock, we cannot guarantee which context the pin count is associated with. This is because of the fact pinning the item is dependent on whether the item is present in the current CIL or not. If we don't pin the CIL first before we check and pin the object, we have a race with CIL being flushed between the check and the pin (or not pinning, as the case may

be). Hence we must hold the CIL flush/commit lock to guarantee that we pin the items correctly.

Delayed Logging: Concurrent Scalability

A fundamental requirement for the CIL is that accesses through transaction commits must scale to many concurrent commits. The current transaction commit code does not break down even when there are transactions coming from 2048 processors at once. The current transaction code does not go any faster than if there was only one CPU using it, but it does not slow down either.

As a result, the delayed logging transaction commit code needs to be designed for concurrency from the ground up. It is obvious that there are serialisation points in the design - the three important ones are:

1. Locking out new transaction commits while flushing the CIL
2. Adding items to the CIL and updating item space accounting
3. Checkpoint commit ordering

Looking at the transaction commit and CIL flushing interactions, it is clear that we have a many-to-one interaction here. That is, the only restriction on the number of concurrent transactions that can be trying to commit at once is the amount of space available in the log for their reservations. The practical limit here is in the order of several hundred concurrent transactions for a 128MB log, which means that it is generally one per CPU in a machine.

The amount of time a transaction commit needs to hold out a flush is a relatively long period of time - the pinning of log items needs to be done while we are holding out a CIL flush, so at the moment that means it is held across the formatting of the objects into memory buffers (i.e. while `memcpy()`s are in progress). Ultimately a two pass algorithm where the formatting is done separately to the pinning of objects could be used to reduce the hold time of the transaction commit side.

Because of the number of potential transaction commit side holders, the lock really needs to be a sleeping lock - if the CIL flush takes the lock, we do not want every other CPU in the machine spinning on the CIL lock. Given that flushing the CIL could involve walking a list of tens of thousands of log items, it will get held for a significant time and so spin contention is a significant concern. Preventing lots of CPUs spinning doing nothing is the main reason for choosing a sleeping lock even though nothing in either the transaction commit or CIL flush side sleeps with the lock held.

It should also be noted that CIL flushing is also a relatively rare operation compared to transaction commit for asynchronous transaction workloads - only time will tell if using a read-write semaphore for exclusion will limit transaction commit concurrency due to cache line bouncing of the lock on the read side.

The second serialisation point is on the transaction commit side where items are inserted into the CIL. Because transactions can enter this code concurrently, the CIL needs to be protected separately from the above commit/flush exclusion. It also needs to be an exclusive lock but it is only held for a very short time and so a spin lock is appropriate here. It is possible that this lock will become a contention

point, but given the short hold time once per transaction I think that contention is unlikely.

The final serialisation point is the checkpoint commit record ordering code that is run as part of the checkpoint commit and log force sequencing. The code path that triggers a CIL flush (i.e. whatever triggers the log force) will enter an ordering loop after writing all the log vectors into the log buffers but before writing the commit record. This loop walks the list of committing checkpoints and needs to block waiting for checkpoints to complete their commit record write. As a result it needs a lock and a wait variable. Log force sequencing also requires the same lock, list walk, and blocking mechanism to ensure completion of checkpoints.

These two sequencing operations can use the mechanism even though the events they are waiting for are different. The checkpoint commit record sequencing needs to wait until checkpoint contexts contain a commit LSN (obtained through completion of a commit record write) while log force sequencing needs to wait until previous checkpoint contexts are removed from the committing list (i.e. they've completed). A simple wait variable and broadcast wakeups (thundering herds) has been used to implement these two serialisation queues. They use the same lock as the CIL, too. If we see too much contention on the CIL lock, or too many context switches as a result of the broadcast wakeups these operations can be put under a new spinlock and given separate wait lists to reduce lock contention and the number of processes woken by the wrong event.

Lifecycle Changes

The existing log item life cycle is as follows:

```

1. Transaction allocate
2. Transaction reserve
3. Lock item
4. Join item to transaction
   If not already attached,
       Allocate log item
       Attach log item to owner item
   Attach log item to transaction
5. Modify item
   Record modifications in log item
6. Transaction commit
   Pin item in memory
   Format item into log buffer
   Write commit LSN into transaction
   Unlock item
   Attach transaction to log buffer

<log buffer IO dispatched>
<log buffer IO completes>

7. Transaction completion
   Mark log item committed
   Insert log item into AIL
       Write commit LSN into log item
   Unpin log item
8. AIL traversal

```

(continues on next page)

(continued from previous page)

```
    Lock item
    Mark log item clean
    Flush item to disk

<item IO completion>

9. Log item removed from AIL
    Moves log tail
    Item unlocked
```

Essentially, steps 1-6 operate independently from step 7, which is also independent of steps 8-9. An item can be locked in steps 1-6 or steps 8-9 at the same time step 7 is occurring, but only steps 1-6 or 8-9 can occur at the same time. If the log item is in the AIL or between steps 6 and 7 and steps 1-6 are re-entered, then the item is relogged. Only when steps 8-9 are entered and completed is the object considered clean.

With delayed logging, there are new steps inserted into the life cycle:

```
1. Transaction allocate
2. Transaction reserve
3. Lock item
4. Join item to transaction
    If not already attached,
        Allocate log item
        Attach log item to owner item
    Attach log item to transaction
5. Modify item
    Record modifications in log item
6. Transaction commit
    Pin item in memory if not pinned in CIL
    Format item into log vector + buffer
    Attach log vector and buffer to log item
    Insert log item into CIL
    Write CIL context sequence into transaction
    Unlock item

<next log force>

7. CIL push
    lock CIL flush
    Chain log vectors and buffers together
    Remove items from CIL
    unlock CIL flush
    write log vectors into log
    sequence commit records
    attach checkpoint context to log buffer

<log buffer IO dispatched>
<log buffer IO completes>

8. Checkpoint completion
    Mark log item committed
    Insert item into AIL
        Write commit LSN into log item
```

(continues on next page)

(continued from previous page)

```

    Unpin log item
9. AIL traversal
    Lock item
    Mark log item clean
    Flush item to disk
<item IO completion>
10. Log item removed from AIL
    Moves log tail
    Item unlocked

```

From this, it can be seen that the only life cycle differences between the two logging methods are in the middle of the life cycle - they still have the same beginning and end and execution constraints. The only differences are in the committing of the log items to the log itself and the completion processing. Hence delayed logging should not introduce any constraints on log item behaviour, allocation or freeing that don't already exist.

As a result of this zero-impact “insertion” of delayed logging infrastructure and the design of the internal structures to avoid on disk format changes, we can basically switch between delayed logging and the existing mechanism with a mount option. Fundamentally, there is no reason why the log manager would not be able to swap methods automatically and transparently depending on load characteristics, but this should not be necessary if delayed logging works as designed.

3.63 XFS Self Describing Metadata

3.63.1 Introduction

The largest scalability problem facing XFS is not one of algorithmic scalability, but of verification of the filesystem structure. Scalability of the structures and indexes on disk and the algorithms for iterating them are adequate for supporting PB scale filesystems with billions of inodes, however it is this very scalability that causes the verification problem.

Almost all metadata on XFS is dynamically allocated. The only fixed location metadata is the allocation group headers (SB, AGF, AGFL and AGI), while all other metadata structures need to be discovered by walking the filesystem structure in different ways. While this is already done by userspace tools for validating and repairing the structure, there are limits to what they can verify, and this in turn limits the supportable size of an XFS filesystem.

For example, it is entirely possible to manually use `xfs_db` and a bit of scripting to analyse the structure of a 100TB filesystem when trying to determine the root cause of a corruption problem, but it is still mainly a manual task of verifying that things like single bit errors or misplaced writes weren't the ultimate cause of a corruption event. It may take a few hours to a few days to perform such forensic analysis, so for at this scale root cause analysis is entirely possible.

However, if we scale the filesystem up to 1PB, we now have 10x as much metadata to analyse and so that analysis blows out towards weeks/months of forensic work. Most of the analysis work is slow and tedious, so as the amount of analysis goes up,

the more likely that the cause will be lost in the noise. Hence the primary concern for supporting PB scale filesystems is minimising the time and effort required for basic forensic analysis of the filesystem structure.

3.63.2 Self Describing Metadata

One of the problems with the current metadata format is that apart from the magic number in the metadata block, we have no other way of identifying what it is supposed to be. We can't even identify if it is the right place. Put simply, you can't look at a single metadata block in isolation and say "yes, it is supposed to be there and the contents are valid" .

Hence most of the time spent on forensic analysis is spent doing basic verification of metadata values, looking for values that are in range (and hence not detected by automated verification checks) but are not correct. Finding and understanding how things like cross linked block lists (e.g. sibling pointers in a btree end up with loops in them) are the key to understanding what went wrong, but it is impossible to tell what order the blocks were linked into each other or written to disk after the fact.

Hence we need to record more information into the metadata to allow us to quickly determine if the metadata is intact and can be ignored for the purpose of analysis. We can't protect against every possible type of error, but we can ensure that common types of errors are easily detectable. Hence the concept of self describing metadata.

The first, fundamental requirement of self describing metadata is that the metadata object contains some form of unique identifier in a well known location. This allows us to identify the expected contents of the block and hence parse and verify the metadata object. IF we can't independently identify the type of metadata in the object, then the metadata doesn't describe itself very well at all!

Luckily, almost all XFS metadata has magic numbers embedded already - only the AGFL, remote symlinks and remote attribute blocks do not contain identifying magic numbers. Hence we can change the on-disk format of all these objects to add more identifying information and detect this simply by changing the magic numbers in the metadata objects. That is, if it has the current magic number, the metadata isn't self identifying. If it contains a new magic number, it is self identifying and we can do much more expansive automated verification of the metadata object at runtime, during forensic analysis or repair.

As a primary concern, self describing metadata needs some form of overall integrity checking. We cannot trust the metadata if we cannot verify that it has not been changed as a result of external influences. Hence we need some form of integrity check, and this is done by adding CRC32c validation to the metadata block. If we can verify the block contains the metadata it was intended to contain, a large amount of the manual verification work can be skipped.

CRC32c was selected as metadata cannot be more than 64k in length in XFS and hence a 32 bit CRC is more than sufficient to detect multi-bit errors in metadata blocks. CRC32c is also now hardware accelerated on common CPUs so it is fast. So while CRC32c is not the strongest of possible integrity checks that could be used, it is more than sufficient for our needs and has relatively little overhead. Adding support for larger integrity fields and/or algorithms does really provide

any extra value over CRC32c, but it does add a lot of complexity and so there is no provision for changing the integrity checking mechanism.

Self describing metadata needs to contain enough information so that the metadata block can be verified as being in the correct place without needing to look at any other metadata. This means it needs to contain location information. Just adding a block number to the metadata is not sufficient to protect against misdirected writes - a write might be misdirected to the wrong LUN and so be written to the “correct block” of the wrong filesystem. Hence location information must contain a filesystem identifier as well as a block number.

Another key information point in forensic analysis is knowing who the metadata block belongs to. We already know the type, the location, that it is valid and/or corrupted, and how long ago that it was last modified. Knowing the owner of the block is important as it allows us to find other related metadata to determine the scope of the corruption. For example, if we have a extent btree object, we don't know what inode it belongs to and hence have to walk the entire filesystem to find the owner of the block. Worse, the corruption could mean that no owner can be found (i.e. it's an orphan block), and so without an owner field in the metadata we have no idea of the scope of the corruption. If we have an owner field in the metadata object, we can immediately do top down validation to determine the scope of the problem.

Different types of metadata have different owner identifiers. For example, directory, attribute and extent tree blocks are all owned by an inode, while freespace btree blocks are owned by an allocation group. Hence the size and contents of the owner field are determined by the type of metadata object we are looking at. The owner information can also identify misplaced writes (e.g. freespace btree block written to the wrong AG).

Self describing metadata also needs to contain some indication of when it was written to the filesystem. One of the key information points when doing forensic analysis is how recently the block was modified. Correlation of set of corrupted metadata blocks based on modification times is important as it can indicate whether the corruptions are related, whether there's been multiple corruption events that lead to the eventual failure, and even whether there are corruptions present that the run-time verification is not detecting.

For example, we can determine whether a metadata object is supposed to be free space or still allocated if it is still referenced by its owner by looking at when the free space btree block that contains the block was last written compared to when the metadata object itself was last written. If the free space block is more recent than the object and the object's owner, then there is a very good chance that the block should have been removed from the owner.

To provide this “written timestamp”, each metadata block gets the Log Sequence Number (LSN) of the most recent transaction it was modified on written into it. This number will always increase over the life of the filesystem, and the only thing that resets it is running `xfs_repair` on the filesystem. Further, by use of the LSN we can tell if the corrupted metadata all belonged to the same log checkpoint and hence have some idea of how much modification occurred between the first and last instance of corrupt metadata on disk and, further, how much modification occurred between the corruption being written and when it was detected.

3.63.3 Runtime Validation

Validation of self-describing metadata takes place at runtime in two places:

- immediately after a successful read from disk
- immediately prior to write IO submission

The verification is completely stateless - it is done independently of the modification process, and seeks only to check that the metadata is what it says it is and that the metadata fields are within bounds and internally consistent. As such, we cannot catch all types of corruption that can occur within a block as there may be certain limitations that operational state enforces of the metadata, or there may be corruption of interblock relationships (e.g. corrupted sibling pointer lists). Hence we still need stateful checking in the main code body, but in general most of the per-field validation is handled by the verifiers.

For read verification, the caller needs to specify the expected type of metadata that it should see, and the IO completion process verifies that the metadata object matches what was expected. If the verification process fails, then it marks the object being read as EFSCORRUPTED. The caller needs to catch this error (same as for IO errors), and if it needs to take special action due to a verification error it can do so by catching the EFSCORRUPTED error value. If we need more discrimination of error type at higher levels, we can define new error numbers for different errors as necessary.

The first step in read verification is checking the magic number and determining whether CRC validating is necessary. If it is, the CRC32c is calculated and compared against the value stored in the object itself. Once this is validated, further checks are made against the location information, followed by extensive object specific metadata validation. If any of these checks fail, then the buffer is considered corrupt and the EFSCORRUPTED error is set appropriately.

Write verification is the opposite of the read verification - first the object is extensively verified and if it is OK we then update the LSN from the last modification made to the object, After this, we calculate the CRC and insert it into the object. Once this is done the write IO is allowed to continue. If any error occurs during this process, the buffer is again marked with a EFSCORRUPTED error for the higher layers to catch.

3.63.4 Structures

A typical on-disk structure needs to contain the following information:

```
struct xfs_ondisk_hdr {
    __be32  magic;           /* magic number */
    __be32  crc;           /* CRC, not logged */
    uuid_t  uuid;         /* filesystem identifier */
    __be64  owner;        /* parent object */
    __be64  blkno;        /* location on disk */
    __be64  lsn;          /* last modification in log, not
↳ logged */
};
```

Depending on the metadata, this information may be part of a header structure separate to the metadata contents, or may be distributed through an existing structure. The latter occurs with metadata that already contains some of this information, such as the superblock and AG headers.

Other metadata may have different formats for the information, but the same level of information is generally provided. For example:

- short btree blocks have a 32 bit owner (ag number) and a 32 bit block number for location. The two of these combined provide the same information as @owner and @blkno in eh above structure, but using 8 bytes less space on disk.
- directory/attribute node blocks have a 16 bit magic number, and the header that contains the magic number has other information in it as well. hence the additional metadata headers change the overall format of the metadata.

A typical buffer read verifier is structured as follows:

```
#define XFS_FOO_CRC_OFF          offsetof(struct xfs_ondisk_hdr, crc)

static void
xfs_foo_read_verify(
    struct xfs_buf      *bp)
{
    struct xfs_mount *mp = bp->b_mount;

    if ((xfs_sb_version_hascrc(&mp->m_sb) &&
        !xfs_verify_cksum(bp->b_addr, BBT0B(bp->b_length),
                        XFS_FOO_CRC_OFF)) ||
        !xfs_foo_verify(bp)) {
        XFS_CORRUPTION_ERROR(__func__, XFS_ERRLEVEL_LOW, mp, bp->b_
→addr);
        xfs_buf_ioerror(bp, EFSCORRUPTED);
    }
}
```

The code ensures that the CRC is only checked if the filesystem has CRCs enabled by checking the superblock of the feature bit, and then if the CRC verifies OK (or is not needed) it verifies the actual contents of the block.

The verifier function will take a couple of different forms, depending on whether the magic number can be used to determine the format of the block. In the case it can't, the code is structured as follows:

```
static bool
xfs_foo_verify(
    struct xfs_buf      *bp)
{
    struct xfs_mount *mp = bp->b_mount;
    struct xfs_ondisk_hdr *hdr = bp->b_addr;

    if (hdr->magic != cpu_to_be32(XFS_FOO_MAGIC))
        return false;

    if (!xfs_sb_version_hascrc(&mp->m_sb)) {
        if (!uuid_equal(&hdr->uuid, &mp->m_sb.sb_uuid))
```

(continues on next page)

(continued from previous page)

```

        return false;
    if (bp->b_bn != be64_to_cpu(hdr->blkno))
        return false;
    if (hdr->owner == 0)
        return false;
}

/* object specific verification checks here */

return true;
}

```

If there are different magic numbers for the different formats, the verifier will look like:

```

static bool
xfs_foo_verify(
    struct xfs_buf          *bp)
{
    struct xfs_mount      *mp = bp->b_mount;
    struct xfs_ondisk_hdr *hdr = bp->b_addr;

    if (hdr->magic == cpu_to_be32(XFS_FOO_CRC_MAGIC)) {
        if (!uuid_equal(&hdr->uuid, &mp->m_sb.sb_uuid))
            return false;
        if (bp->b_bn != be64_to_cpu(hdr->blkno))
            return false;
        if (hdr->owner == 0)
            return false;
    } else if (hdr->magic != cpu_to_be32(XFS_FOO_MAGIC))
        return false;

    /* object specific verification checks here */

    return true;
}

```

Write verifiers are very similar to the read verifiers, they just do things in the opposite order to the read verifiers. A typical write verifier:

```

static void
xfs_foo_write_verify(
    struct xfs_buf          *bp)
{
    struct xfs_mount      *mp = bp->b_mount;
    struct xfs_buf_log_item *bip = bp->b_fspriv;

    if (!xfs_foo_verify(bp)) {
        XFS_CORRUPTION_ERROR(__func__, XFS_ERRLEVEL_LOW, mp, bp->b_
→addr);
        xfs_buf_ioerror(bp, EFSCORRUPTED);
        return;
    }

    if (!xfs_sb_version_hascrc(&mp->m_sb))

```

(continues on next page)

(continued from previous page)

```

        return;

    if (bip) {
        struct xfs_ondisk_hdr      *hdr = bp->b_addr;
        hdr->lsn = cpu_to_be64(bip->bli_item.li_lsn);
    }
    xfs_update_cksum(bp->b_addr, BBT0B(bp->b_length), XFS_F00_CRC_OFF);
}

```

This will verify the internal structure of the metadata before we go any further, detecting corruptions that have occurred as the metadata has been modified in memory. If the metadata verifies OK, and CRCs are enabled, we then update the LSN field (when it was last modified) and calculate the CRC on the metadata. Once this is done, we can issue the IO.

3.63.5 Inodes and Dquots

Inodes and dquots are special snowflakes. They have per-object CRC and self-identifiers, but they are packed so that there are multiple objects per buffer. Hence we do not use per-buffer verifiers to do the work of per-object verification and CRC calculations. The per-buffer verifiers simply perform basic identification of the buffer - that they contain inodes or dquots, and that there are magic numbers in all the expected spots. All further CRC and verification checks are done when each inode is read from or written back to the buffer.

The structure of the verifiers and the identifiers checks is very similar to the buffer code described above. The only difference is where they are called. For example, inode read verification is done in `xfs_inode_from_disk()` when the inode is first read out of the buffer and the struct `xfs_inode` is instantiated. The inode is already extensively verified during writeback in `xfs_iflush_int`, so the only addition here is to add the LSN and CRC to the inode as it is copied back into the buffer.

XXX: inode unlinked list modification doesn't recalculate the inode CRC! None of the unlinked list modifications check or update CRCs, neither during unlink nor log recovery. So, it's gone unnoticed until now. This won't matter immediately - repair will probably complain about it - but it needs to be fixed.

3.64 ZoneFS - Zone filesystem for Zoned block devices

3.64.1 Introduction

zonefs is a very simple file system exposing each zone of a zoned block device as a file. Unlike a regular POSIX-compliant file system with native zoned block device support (e.g. `f2fs`), zonefs does not hide the sequential write constraint of zoned block devices to the user. Files representing sequential write zones of the device must be written sequentially starting from the end of the file (append only writes).

As such, zonefs is in essence closer to a raw block device access interface than to a full-featured POSIX file system. The goal of zonefs is to simplify the implemen-

tation of zoned block device support in applications by replacing raw block device file accesses with a richer file API, avoiding relying on direct block device file ioctls which may be more obscure to developers. One example of this approach is the implementation of LSM (log-structured merge) tree structures (such as used in RocksDB and LevelDB) on zoned block devices by allowing SSTables to be stored in a zone file similarly to a regular file system rather than as a range of sectors of the entire disk. The introduction of the higher level construct “one file is one zone” can help reducing the amount of changes needed in the application as well as introducing support for different application programming languages.

Zoned block devices

Zoned storage devices belong to a class of storage devices with an address space that is divided into zones. A zone is a group of consecutive LBAs and all zones are contiguous (there are no LBA gaps). Zones may have different types.

- Conventional zones: there are no access constraints to LBAs belonging to conventional zones. Any read or write access can be executed, similarly to a regular block device.
- Sequential zones: these zones accept random reads but must be written sequentially. Each sequential zone has a write pointer maintained by the device that keeps track of the mandatory start LBA position of the next write to the device. As a result of this write constraint, LBAs in a sequential zone cannot be overwritten. Sequential zones must first be erased using a special command (zone reset) before rewriting.

Zoned storage devices can be implemented using various recording and media technologies. The most common form of zoned storage today uses the SCSI Zoned Block Commands (ZBC) and Zoned ATA Commands (ZAC) interfaces on Shingled Magnetic Recording (SMR) HDDs.

Solid State Disks (SSD) storage devices can also implement a zoned interface to, for instance, reduce internal write amplification due to garbage collection. The NVMe Zoned NameSpace (ZNS) is a technical proposal of the NVMe standard committee aiming at adding a zoned storage interface to the NVMe protocol.

3.64.2 Zonefs Overview

Zonefs exposes the zones of a zoned block device as files. The files representing zones are grouped by zone type, which are themselves represented by sub-directories. This file structure is built entirely using zone information provided by the device and so does not require any complex on-disk metadata structure.

On-disk metadata

zonefs on-disk metadata is reduced to an immutable super block which persistently stores a magic number and optional feature flags and values. On mount, zonefs uses `blkdev_report_zones()` to obtain the device zone configuration and populates the mount point with a static file tree solely based on this information. File sizes come from the device zone type and write pointer position managed by the device itself.

The super block is always written on disk at sector 0. The first zone of the device storing the super block is never exposed as a zone file by zonefs. If the zone containing the super block is a sequential zone, the `mkzonefs` format tool always “finishes” the zone, that is, it transitions the zone to a full state to make it read-only, preventing any data write.

Zone type sub-directories

Files representing zones of the same type are grouped together under the same sub-directory automatically created on mount.

For conventional zones, the sub-directory “`cnv`” is used. This directory is however created if and only if the device has usable conventional zones. If the device only has a single conventional zone at sector 0, the zone will not be exposed as a file as it will be used to store the zonefs super block. For such devices, the “`cnv`” sub-directory will not be created.

For sequential write zones, the sub-directory “`seq`” is used.

These two directories are the only directories that exist in zonefs. Users cannot create other directories and cannot rename nor delete the “`cnv`” and “`seq`” sub-directories.

The size of the directories indicated by the `st_size` field of struct `stat`, obtained with the `stat()` or `fstat()` system calls, indicates the number of files existing under the directory.

Zone files

Zone files are named using the number of the zone they represent within the set of zones of a particular type. That is, both the “cnv” and “seq” directories contain files named “0” , “1” , “2” , …The file numbers also represent increasing zone start sector on the device.

All read and write operations to zone files are not allowed beyond the file maximum size, that is, beyond the zone size. Any access exceeding the zone size is failed with the -EFBIG error.

Creating, deleting, renaming or modifying any attribute of files and sub-directories is not allowed.

The number of blocks of a file as reported by stat() and fstat() indicates the size of the file zone, or in other words, the maximum file size.

Conventional zone files

The size of conventional zone files is fixed to the size of the zone they represent. Conventional zone files cannot be truncated.

These files can be randomly read and written using any type of I/O operation: buffered I/Os, direct I/Os, memory mapped I/Os (mmap), etc. There are no I/O constraint for these files beyond the file size limit mentioned above.

Sequential zone files

The size of sequential zone files grouped in the “seq” sub-directory represents the file’ s zone write pointer position relative to the zone start sector.

Sequential zone files can only be written sequentially, starting from the file end, that is, write operations can only be append writes. Zonefs makes no attempt at accepting random writes and will fail any write request that has a start offset not corresponding to the end of the file, or to the end of the last write issued and still in-flight (for asynchronous I/O operations).

Since dirty page writeback by the page cache does not guarantee a sequential write pattern, zonefs prevents buffered writes and writeable shared mappings on sequential files. Only direct I/O writes are accepted for these files. zonefs relies on the sequential delivery of write I/O requests to the device implemented by the block layer elevator. An elevator implementing the sequential write feature for zoned block device (ELEVATOR_F_ZBD_SEQ_WRITE elevator feature) must be used. This type of elevator (e.g. mq-deadline) is set by default for zoned block devices on device initialization.

There are no restrictions on the type of I/O used for read operations in sequential zone files. Buffered I/Os, direct I/Os and shared read mappings are all accepted.

Truncating sequential zone files is allowed only down to 0, in which case, the zone is reset to rewind the file zone write pointer position to the start of the zone, or up to the zone size, in which case the file’ s zone is transitioned to the FULL state (finish zone operation).

Format options

Several optional features of zonefs can be enabled at format time.

- Conventional zone aggregation: ranges of contiguous conventional zones can be aggregated into a single larger file instead of the default one file per zone.
- File ownership: The owner UID and GID of zone files is by default 0 (root) but can be changed to any valid UID/GID.
- File access permissions: the default 640 access permissions can be changed.

IO error handling

Zoned block devices may fail I/O requests for reasons similar to regular block devices, e.g. due to bad sectors. However, in addition to such known I/O failure pattern, the standards governing zoned block devices behavior define additional conditions that result in I/O errors.

- A zone may transition to the read-only condition (`BLK_ZONE_COND_READONLY`): While the data already written in the zone is still readable, the zone can no longer be written. No user action on the zone (zone management command or read/write access) can change the zone condition back to a normal read/write state. While the reasons for the device to transition a zone to read-only state are not defined by the standards, a typical cause for such transition would be a defective write head on an HDD (all zones under this head are changed to read-only).
- A zone may transition to the offline condition (`BLK_ZONE_COND_OFFLINE`): An offline zone cannot be read nor written. No user action can transition an offline zone back to an operational good state. Similarly to zone read-only transitions, the reasons for a drive to transition a zone to the offline condition are undefined. A typical cause would be a defective read-write head on an HDD causing all zones on the platter under the broken head to be inaccessible.
- Unaligned write errors: These errors result from the host issuing write requests with a start sector that does not correspond to a zone write pointer position when the write request is executed by the device. Even though zonefs enforces sequential file write for sequential zones, unaligned write errors may still happen in the case of a partial failure of a very large direct I/O operation split into multiple BIOs/requests or asynchronous I/O operations. If one of the write request within the set of sequential write requests issued to the device fails, all write requests queued after it will become unaligned and fail.
- Delayed write errors: similarly to regular block devices, if the device side write cache is enabled, write errors may occur in ranges of previously completed writes when the device write cache is flushed, e.g. on `fsync()`. Similarly to the previous immediate unaligned write error case, delayed write errors can propagate through a stream of cached sequential data for a zone causing all data to be dropped after the sector that caused the error.

All I/O errors detected by zonefs are notified to the user with an error code return for the system call that triggered or detected the error. The recovery actions taken

by zonefs in response to I/O errors depend on the I/O type (read vs write) and on the reason for the error (bad sector, unaligned writes or zone condition change).

- For read I/O errors, zonefs does not execute any particular recovery action, but only if the file zone is still in a good condition and there is no inconsistency between the file inode size and its zone write pointer position. If a problem is detected, I/O error recovery is executed (see below table).
- For write I/O errors, zonefs I/O error recovery is always executed.
- A zone condition change to read-only or offline also always triggers zonefs I/O error recovery.

Zonefs minimal I/O error recovery may change a file size and file access permissions.

- File size changes: Immediate or delayed write errors in a sequential zone file may cause the file inode size to be inconsistent with the amount of data successfully written in the file zone. For instance, the partial failure of a multi-BIO large write operation will cause the zone write pointer to advance partially, even though the entire write operation will be reported as failed to the user. In such case, the file inode size must be advanced to reflect the zone write pointer change and eventually allow the user to restart writing at the end of the file. A file size may also be reduced to reflect a delayed write error detected on fsync(): in this case, the amount of data effectively written in the zone may be less than originally indicated by the file inode size. After such I/O error, zonefs always fixes the file inode size to reflect the amount of data persistently stored in the file zone.
- Access permission changes: A zone condition change to read-only is indicated with a change in the file access permissions to render the file read-only. This disables changes to the file attributes and data modification. For offline zones, all permissions (read and write) to the file are disabled.

Further action taken by zonefs I/O error recovery can be controlled by the user with the “errors=xxx” mount option. The table below summarizes the result of zonefs I/O error processing depending on the mount option and on the zone conditions:

"errors=xxx" mount option	device zone condition	file size	Post error state access permissions			
			file read	file write	device read	zone write
remount-ro (default)	good	fixed	yes	no	yes	yes
	read-only	as is	yes	no	yes	no
	offline	0	no	no	no	no
zone-ro	good	fixed	yes	no	yes	yes
	read-only	as is	yes	no	yes	no
	offline	0	no	no	no	no
zone-offline	good	0	no	no	yes	yes
	read-only	0	no	no	yes	no
	offline	0	no	no	no	no

(continues on next page)

(continued from previous page)

	good	fixed	yes	yes	yes	yes
repair	read-only	as is	yes	no	yes	no
	offline	0	no	no	no	no

Further notes:

- The “errors=remount-ro” mount option is the default behavior of zonefs I/O error processing if no errors mount option is specified.
- With the “errors=remount-ro” mount option, the change of the file access permissions to read-only applies to all files. The file system is remounted read-only.
- Access permission and file size changes due to the device transitioning zones to the offline condition are permanent. Remounting or reformatting the device with mkfs.zonefs (mkzonefs) will not change back offline zone files to a good state.
- File access permission changes to read-only due to the device transitioning zones to the read-only condition are permanent. Remounting or reformatting the device will not re-enable file write access.
- File access permission changes implied by the remount-ro, zone-ro and zone-offline mount options are temporary for zones in a good condition. Unmounting and remounting the file system will restore the previous default (format time values) access rights to the files affected.
- The repair mount option triggers only the minimal set of I/O error recovery actions, that is, file size fixes for zones in a good condition. Zones indicated as being read-only or offline by the device still imply changes to the zone file access permissions as noted in the table above.

Mount options

zonefs define the “errors=<behavior>” mount option to allow the user to specify zonefs behavior in response to I/O errors, inode size inconsistencies or zone condition changes. The defined behaviors are as follow:

- remount-ro (default)
- zone-ro
- zone-offline
- repair

The run-time I/O error actions defined for each behavior are detailed in the previous section. Mount time I/O errors will cause the mount operation to fail. The handling of read-only zones also differs between mount-time and run-time. If a read-only zone is found at mount time, the zone is always treated in the same manner as offline zones, that is, all accesses are disabled and the zone file size set to 0. This is necessary as the write pointer of read-only zones is defined as invalid by the ZBC and ZAC standards, making it impossible to discover the amount of

data that has been written to the zone. In the case of a read-only zone discovered at run-time, as indicated in the previous section. the size of the zone file is left unchanged from its last updated value.

3.64.3 Zonefs User Space Tools

The mkzonefs tool is used to format zoned block devices for use with zonefs. This tool is available on Github at:

<https://github.com/damien-lemoal/zonefs-tools>

zonefs-tools also includes a test suite which can be run against any zoned block device, including null_blk block device created with zoned mode.

Examples

The following formats a 15TB host-managed SMR HDD with 256 MB zones with the conventional zones aggregation feature enabled:

```
# mkzonefs -o aggr_cnv /dev/sdX
# mount -t zonefs /dev/sdX /mnt
# ls -l /mnt/
total 0
dr-xr-xr-x 2 root root      1 Nov 25 13:23 cnv
dr-xr-xr-x 2 root root 55356 Nov 25 13:23 seq
```

The size of the zone files sub-directories indicate the number of files existing for each type of zones. In this example, there is only one conventional zone file (all conventional zones are aggregated under a single file):

```
# ls -l /mnt/cnv
total 137101312
-rw-r----- 1 root root 140391743488 Nov 25 13:23 0
```

This aggregated conventional zone file can be used as a regular file:

```
# mkfs.ext4 /mnt/cnv/0
# mount -o loop /mnt/cnv/0 /data
```

The “seq” sub-directory grouping files for sequential write zones has in this example 55356 zones:

```
# ls -lv /mnt/seq
total 14511243264
-rw-r----- 1 root root 0 Nov 25 13:23 0
-rw-r----- 1 root root 0 Nov 25 13:23 1
-rw-r----- 1 root root 0 Nov 25 13:23 2
...
-rw-r----- 1 root root 0 Nov 25 13:23 55354
-rw-r----- 1 root root 0 Nov 25 13:23 55355
```

For sequential write zone files, the file size changes as data is appended at the end of the file, similarly to any regular file system:

```
# dd if=/dev/zero of=/mnt/seq/0 bs=4096 count=1 conv=notrunc oflag=direct
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.00044121 s, 9.3 MB/s

# ls -l /mnt/seq/0
-rw-r----- 1 root root 4096 Nov 25 13:23 /mnt/seq/0
```

The written file can be truncated to the zone size, preventing any further write operation:

```
# truncate -s 268435456 /mnt/seq/0
# ls -l /mnt/seq/0
-rw-r----- 1 root root 268435456 Nov 25 13:49 /mnt/seq/0
```

Truncation to 0 size allows freeing the file zone storage space and restart appends to the file:

```
# truncate -s 0 /mnt/seq/0
# ls -l /mnt/seq/0
-rw-r----- 1 root root 0 Nov 25 13:49 /mnt/seq/0
```

Since files are statically mapped to zones on the disk, the number of blocks of a file as reported by `stat()` and `fstat()` indicates the size of the file zone:

```
# stat /mnt/seq/0
File: /mnt/seq/0
Size: 0          Blocks: 524288      IO Block: 4096   regular empty file
Device: 870h/2160d Inode: 50431       Links: 1
Access: (0640/-rw-r-----)  Uid: (   0/   root)  Gid: (   0/   root)
Access: 2019-11-25 13:23:57.048971997 +0900
Modify: 2019-11-25 13:52:25.553805765 +0900
Change: 2019-11-25 13:52:25.553805765 +0900
Birth: -
```

The number of blocks of the file (“Blocks”) in units of 512B blocks gives the maximum file size of $524288 * 512 \text{ B} = 256 \text{ MB}$, corresponding to the device zone size in this example. Of note is that the “IO block” field always indicates the minimum I/O size for writes and corresponds to the device physical sector size.