Linux Virtual File System

The linux VFS and FUSE - Filesystem in User Space

Andre Petris Esteve - andreesteve@gmail.com
Zhenlei Ji - zhenlei.ji@gmail.com

MC806 - Operational System Topics

October 20th, 2011

Agenda

1. Objectives
2. Overview
3. Core Elements
4. Operation example
5. Getting conFUSEed

Agenda

Objectives

What do we want?

- View the Linux's Virtual Filesystem as a series of object oriented entities (classes and objects)[1]

[1] Althought the linux kernel is written in C, it's possible to profit from some object oriented features through programming tricks. For further details see: OOC, Axel Schreiner

Objectives

What do we want?

- View the Linux's Virtual Filesystem as a series of object oriented entities (classes and objects)[1]
- Construct UML models to easy understanding

[1] Althought the linux kernel is written in C, it's possible to profit from some object oriented features through programming tricks. For further details see: OOC, Axel Schreiner

Objectives

What do we want?

- View the Linux's Virtual Filesystem as a series of object oriented entities (classes and objects)[1]
- Construct UML models to easy understanding
- Provide initial information so one can start developing a filesystem module for the Linux kernel

[1] Althought the linux kernel is written in C, it's possible to profit from some object oriented features through programming tricks. For further details see: OOC, Axel Schreiner

Warning

Please note!

- All information here is based extensively on linux kernel 3.1-rc8 source code[1]

[1] You can easily find something in the kernel source code using this tools: http://lxr.linux.no/linux

Warning

Please note!

- All information here is based extensively on linux kernel 3.1-rc8 source code[1]
- Some models are represented at a certain level of abstraction and may omit some implementation information

[1] You can easily find something in the kernel source code using this tools: http://lxr.linux.no/linux

Agenda

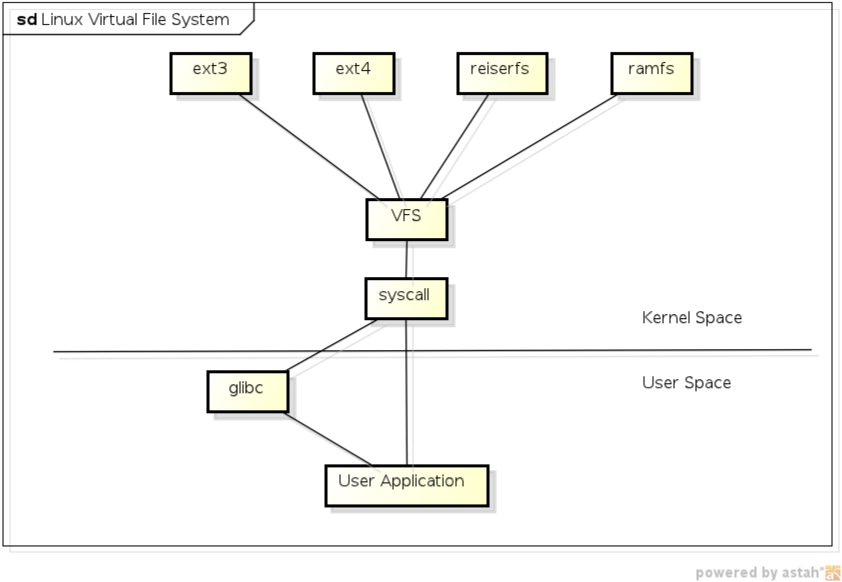
What's Linux's Virtual Filesystem

Definition

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. [1]

[1] Overview of the Linux Virtual File System, Richard Gooch, from Linux "documentation"

Linux's Virtual Filesystem Overview



Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist

[1] Short for "filesystem"

Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist
- Only point of access to fs calls

[1] Short for "filesystem"

Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist
- Only point of access to fs calls
- Implements common fs operations

[1] Short for "filesystem"

Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist
- Only point of access to fs calls
- Implements common fs operations
  - Common initialization operations

[1] Short for "filesystem"

Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist
- Only point of access to fs calls
- Implements common fs operations
  - Common initialization operations
  - Mounting (at a certain level) and managing mount points

[1] Short for "filesystem"

Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist
- Only point of access to fs calls
- Implements common fs operations
  - Common initialization operations
  - Mounting (at a certain level) and managing mount points
  - Path look-up

[1] Short for "filesystem"

Linux's Virtual Filesystem Overview

- Abstraction layer to allow different fs[1] to coexist
- Only point of access to fs calls
- Implements common fs operations
  - Common initialization operations
  - Mounting (at a certain level) and managing mount points
  - Path look-up
  - Caching

[1] Short for "filesystem"

How is a filesystem implemented?

With loadable kernel modules[1] (LKM), or just modules for short.

[1] For an extensive discussion about LKM, see: http://www.tldp.org/HOWTO/Module-HOWTO/

Linux's Virtual Filesystem Overview

How is a filesystem implemented?

With loadable kernel modules[1] (LKM), or just modules for short.

- It's possible to compile a LKM with the base kernel

[1] For an extensive discussion about LKM, see:
http://www.tldp.org/HOWTO/Module-HOWTO/

How is a filesystem implemented?

With loadable kernel modules[1] (LKM), or just modules for short.

- It's possible to compile a LKM with the base kernel
- Or just load the LKM during system usage

[1] For an extensive discussion about LKM, see: http://www.tldp.org/HOWTO/Module-HOWTO/

Agenda

Linux's Virtual Filesystem Core Elements

file_system_type Information about a specific fs type

Linux's Virtual Filesystem Core Elements

file_system_type Information about a specific fs type

vfsmount Mount point information

Linux's Virtual Filesystem Core Elements

file_system_type Information about a specific fs type

vfsmount Mount point information

super_block Represents a mounted filesystem

Linux's Virtual Filesystem Core Elements

file_system_type Information about a specific fs type

vfsmount Mount point information

super_block Represents a mounted filesystem

inode Information about a file (on disk, memory or network)

Linux's Virtual Filesystem Core Elements

file_system_type
: Information about a specific fs type

vfsmount
: Mount point information

super_block
: Represents a mounted filesystem

inode
: Information about a file (on disk, memory or network)

dentry
: A directory entry

Linux's Virtual Filesystem Core Elements

file_system_type Information about a specific fs type

vfsmount Mount point information

super_block Represents a mounted filesystem

inode Information about a file (on disk, memory or network)

dentry A directory entry

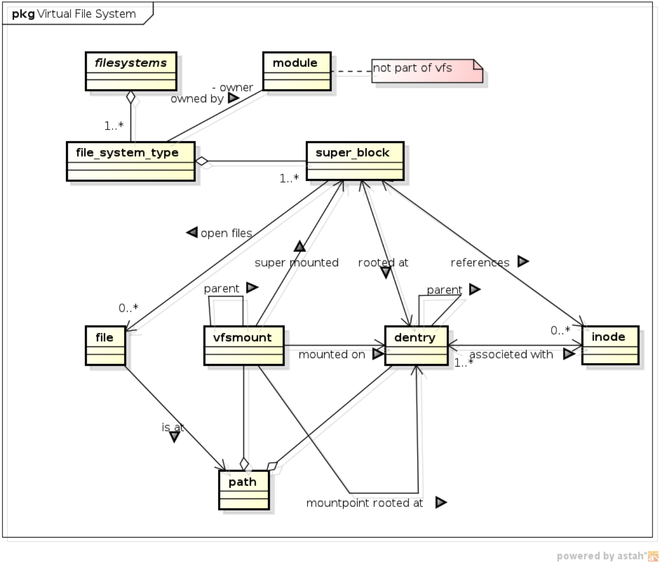
file A file abstraction - points to an inode

Linux's Virtual Filesystem Core Elements

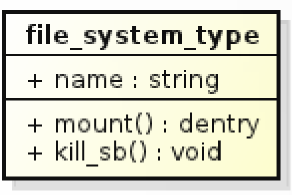
- **file_system_type** Information about a specific fs type
- **vfsmount** Mount point information
- **super_block** Represents a mounted filesystem
- **inode** Information about a file (on disk, memory or network)
- **dentry** A directory entry
- **file** A file abstraction - points to an inode

Note: Every element, except vfsmount, is defined at include/linux/fs.h.

Linux's Virtual Filesystem Core Elements



file_system_type



file_system_type

- Represents a filesystem type (e.g. ext3, nfs, fuse)

file_system_type

- Represents a filesystem type (e.g. ext3, nfs, fuse)
- fs/filesystems.c has a linked list of filesystem types

file_system_type

- Represents a filesystem type (e.g. ext3, nfs, fuse)
- fs/filesystems.c has a linked list of filesystem types
- Each filesystem type must have a unique name

file_system_type

- Represents a filesystem type (e.g. ext3, nfs, fuse)
- fs/filesystems.c has a linked list of filesystem types
- Each filesystem type must have a unique name
- Each filesystem type has a linked list of super blocks in use (i.e. mounted)

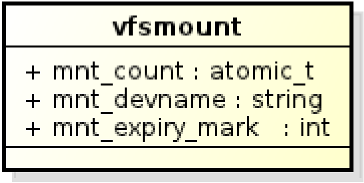
file_system_type

- Represents a filesystem type (e.g. ext3, nfs, fuse)
- fs/filesystems.c has a linked list of filesystem types
- Each filesystem type must have a unique name
- Each filesystem type has a linked list of super blocks in use (i.e. mounted)
- Each filesystem type is owned by a module (which implements it)

file_system_type

- Represents a filesystem type (e.g. ext3, nfs, fuse)
- fs/filesystems.c has a linked list of filesystem types
- Each filesystem type must have a unique name
- Each filesystem type has a linked list of super blocks in use (i.e. mounted)
- Each filesystem type is owned by a module (which implements it)
- Each filesystem type has a function to mount a (possibly new) instance of the filesystem

vfsmount


vfsmount
+ mnt_count : atomic_t
+ mnt_devname : string
+ mnt_expiry_mark : int

vfsmount

- Defined at include/linux/mount.h

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)
  - Use count (if 0 the fs could be unmounted if mnt_expiry_mark is set)

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)
  - Use count (if 0 the fs could be unmounted if mnt_expiry_mark is set)
- Refers to the parent mount point (the one its mounted on) and has a list of mounted children

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)
  - Use count (if 0 the fs could be unmounted if mnt_expiry_mark is set)
- Refers to the parent mount point (the one its mounted on) and has a list of mounted children
- Points to the parent (mount point) dentry root

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)
  - Use count (if 0 the fs could be unmounted if mnt_expiry_mark is set)
- Refers to the parent mount point (the one its mounted on) and has a list of mounted children
- Points to the parent (mount point) dentry root
- Has a dentry for its own root

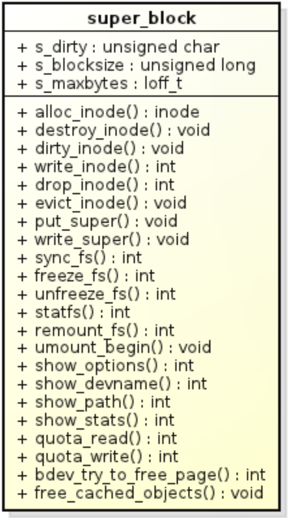
vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)
  - Use count (if 0 the fs could be unmounted if mnt_expiry_mark is set)
- Refers to the parent mount point (the one its mounted on) and has a list of mounted children
- Points to the parent (mount point) dentry root
- Has a dentry for its own root
- Has the super block of the mounted filesystem

vfsmount

- Defined at include/linux/mount.h
- Store information about a mount point
  - Device name (if any)
  - Use count (if 0 the fs could be unmounted if mnt_expiry_mark is set)
- Refers to the parent mount point (the one its mounted on) and has a list of mounted children
- Points to the parent (mount point) dentry root
- Has a dentry for its own root
- Has the super block of the mounted filesystem
- Not directly handled by a filesystem implementation

super_block


super_block
+ s_dirty : unsigned char
+ s_blocksize : unsigned long
+ s_maxbytes : loff_t
+ alloc_inode() : inode
+ destroy_inode() : void
+ dirty_inode() : void
+ write_inode() : int
+ drop_inode() : int
+ evict_inode() : void
+ put_super() : void
+ write_super() : void
+ sync_fs() : int
+ freeze_fs() : int
+ unfreeze_fs() : int
+ statfs() : int
+ remount_fs() : int
+ umount_begin() : void
+ show_options() : int
+ show_devname() : int
+ show_path() : int
+ show_stats() : int
+ quota_read() : int
+ quota_write() : int
+ bdev_try_to_free_page() : int
+ free_cached_objects() : void

super_block

- Represents a filesystem instance

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size
  - access time granularity, among others

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size
  - access time granularity, among others
- Refers to its filesystem type (and thus module owner)

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size
  - access time granularity, among others
- Refers to its filesystem type (and thus module owner)
- Points to its dentry root

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size
  - access time granularity, among others
- Refers to its filesystem type (and thus module owner)
- Points to its dentry root
- Has a dentry for its own root

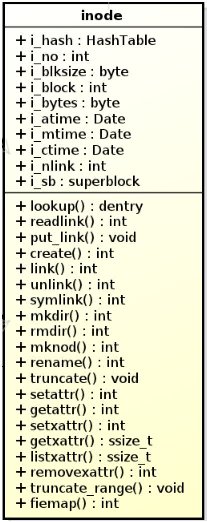
super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size
  - access time granularity, among others
- Refers to its filesystem type (and thus module owner)
- Points to its dentry root
- Has a dentry for its own root
- Has lists for open files and inodes in use

super_block

- Represents a filesystem instance
- When the filesystem is disk based, the super block usually is persisted on disk
- It's kept on memory, but there's a dirty flag so it can eventually be flush to disk (for disk based fs)
- Defines filesystem's properties
  - block size
  - maximum file size
  - access time granularity, among others
- Refers to its filesystem type (and thus module owner)
- Points to its dentry root
- Has a dentry for its own root
- Has lists for open files and inodes in use
- Has functions to handle quota operations and inode manipulation

inode


inode
+ i_hash : HashTable
+ i_no : int
+ i_blksize : byte
+ i_block : int
+ i_bytes : byte
+ i_atime : Date
+ i_mtime : Date
+ i_ctime : Date
+ i_nlink : int
+ i_sb : superblock
+ lookup() : dentry
+ readlink() : int
+ put_link() : void
+ create() : int
+ link() : int
+ unlink() : int
+ symlink() : int
+ mkdir() : int
+ rmdir() : int
+ mknod() : int
+ rename() : int
+ truncate() : void
+ setattr() : int
+ getattr() : int
+ setxattr() : int
+ getxattr() : ssize_t
+ listxattr() : ssize_t
+ removexattr() : int
+ truncate_range() : void
+ fiemap() : int

inode

- Each object in the filesytem is represented by an inode

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes
  - i_hash: Pointer for the hash list

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes
  - i_hash: Pointer for the hash list
  - i_ino: inode number

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes
  - i_hash: Pointer for the hash list
  - i_ino: inode number
  - i_blksize, i_block, i_bytes: respectively block size, number of block and block size of the last block

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes
  - i_hash: Pointer for the hash list
  - i_ino: inode number
  - i_blksize, i_block, i_bytes: respectively block size, number of block and block size of the last block
  - i_atime, i_mtime, i_ctime: respectively time of the last file access, write and change

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes
  - i_hash: Pointer for the hash list
  - i_ino: inode number
  - i_blksize, i_block, i_bytes: respectively block size, number of block and block size of the last block
  - i_atime, i_mtime, i_ctime: respectively time of the last file access, write and change
  - i_nlink: number of hard links

inode

- Each object in the filesytem is represented by an inode
- Each inode is identified by a unique inode number within the filesystem
- The inode is only instantiated in memory at the time the file is opened
- Defines inode's atributes
  - i_hash: Pointer for the hash list
  - i_ino: inode number
  - i_blksize, i_block, i_bytes: respectively block size, number of block and block size of the last block
  - i_atime, i_mtime, i_ctime: respectively time of the last file access, write and change
  - i_nlink: number of hard links
  - i_sb: Pointer to superblock object

inode

- Defines inode_operations

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)
  - link(old_dentry, dir, new_dentry)

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)
  - link(old_dentry, dir, new_dentry)
  - mkdir(dir, dentry, mode)

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)
  - link(old_dentry, dir, new_dentry)
  - mkdir(dir, dentry, mode)
  - rmdir(dir, dentry)

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)
  - link(old_dentry, dir, new_dentry)
  - mkdir(dir, dentry, mode)
  - rmdir(dir, dentry)
  - rename(old_dir, old_dentry, new_dir, new_dentry)

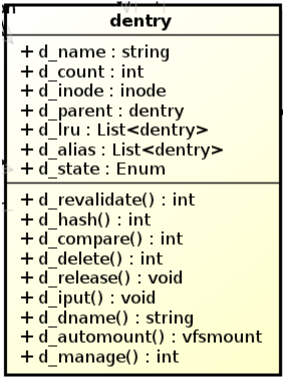
inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)
  - link(old_dentry, dir, new_dentry)
  - mkdir(dir, dentry, mode)
  - rmdir(dir, dentry)
  - rename(old_dir, old_dentry, new_dir, new_dentry)
  - permission(inode, mask, nameidada)

inode

- Defines inode_operations
  - create(dir, dentry, mode, nameidata)
  - lookup(dir, dentry, nameidata)
  - link(old_dentry, dir, new_dentry)
  - mkdir(dir, dentry, mode)
  - rmdir(dir, dentry)
  - rename(old_dir, old_dentry, new_dir, new_dentry)
  - permission(inode, mask, nameidada)
- A single inode can be pointed to by multiple dentries (hard links)

dentry


dentry
+ d_name : string
+ d_count : int
+ d_inode : inode
+ d_parent : dentry
+ d_lru : List<dentry>
+ d_alias : List<dentry>
+ d_state : Enum
+ d_revalidate() : int
+ d_hash() : int
+ d_compare() : int
+ d_delete() : int
+ d_release() : void
+ d_iput() : void
+ d_dname() : string
+ d_automount() : vfsmount
+ d_manage() : int

dentry

- The VFS considers each directory a file that contains a list of files and directories

dentry

- The VFS considers each directory a file that contains a list of files and directories
- Once a directory entry is read into memory, it is transformed by the VFS into a dentry object

dentry

- The VFS considers each directory a file that contains a list of files and directories
- Once a directory entry is read into memory, it is transformed by the VFS into a dentry object
  - Example: /tmp/tex
    tmp and tex are files, both represented by the inodes.

dentry

- The VFS considers each directory a file that contains a list of files and directories
- Once a directory entry is read into memory, it is transformed by the VFS into a dentry object
  - Example: /tmp/tex
    tmp and tex are files, both represented by the inodes.
- The concept of input directory (dentry)

dentry

- The VFS considers each directory a file that contains a list of files and directories
- Once a directory entry is read into memory, it is transformed by the VFS into a dentry object
  - Example: /tmp/tex
    tmp and tex are files, both represented by the inodes.
- The concept of input directory (dentry)
- Specific component of the path

dentry

- The VFS considers each directory a file that contains a list of files and directories
- Once a directory entry is read into memory, it is transformed by the VFS into a dentry object
  - Example: /tmp/tex
    tmp and tex are files, both represented by the inodes.
- The concept of input directory (dentry)
- Specific component of the path
- The VFS instantiates these objects "on the fly" when you make operations on directories

dentry

- The VFS considers each directory a file that contains a list of files and directories
- Once a directory entry is read into memory, it is transformed by the VFS into a dentry object
  - Example: /tmp/tex
    tmp and tex are files, both represented by the inodes.
- The concept of input directory (dentry)
- Specific component of the path
- The VFS instantiates these objects "on the fly" when you make operations on directories
- It's kept on memory

dentry

- Defines dentry's attributes

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries
  - d_alias: pointers for the list of dentries associated with the same inode

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries
  - d_alias: pointers for the list of dentries associated with the same inode
- Dentry States:

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries
  - d_alias: pointers for the list of dentries associated with the same inode
- Dentry States:
  - Free: no valid information and is not used

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries
  - d_alias: pointers for the list of dentries associated with the same inode
- Dentry States:
  - Free: no valid information and is not used
  - Unused: valid information and is not used, may be discarted if necessary

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries
  - d_alias: pointers for the list of dentries associated with the same inode
- Dentry States:
  - Free: no valid information and is not used
  - Unused: valid information and is not used, may be discarted if necessary
  - In use: valid information and is used, cannot be discarted

dentry

- Defines dentry's attributes
  - d_count: dentry object usage counter
  - d_inode: inode associated with filename
  - d_parent: dentry object of parent directory
  - d_name: filename
  - d_lru: pointer for the list of unused dentries
  - d_alias: pointers for the list of dentries associated with the same inode
- Dentry States:
  - Free: no valid information and is not used
  - Unused: valid information and is not used, may be discarted if necessary
  - In use: valid information and is used, cannot be discarted
  - Negative: the inode associated with the dentry does not exist or is invalid

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time

[1] Last recently used

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time
- A set of dentries in the in-use, unused, or negative states

[1] Last recently used

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time
- A set of dentries in the in-use, unused, or negative states
- A hash table to derive the dentry object associated with a given filename or directory quickly

[1] Last recently used

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time
- A set of dentries in the in-use, unused, or negative states
- A hash table to derive the dentry object associated with a given filename or directory quickly
- Stores dentry objects as follows

[1] Last recently used

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time
- A set of dentries in the in-use, unused, or negative states
- A hash table to derive the dentry object associated with a given filename or directory quickly
- Stores dentry objects as follows
  - All the "unused" dentries are included in a doubly linked LRU[1] list sorted by time of insertion

[1] Last recently used

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time
- A set of dentries in the in-use, unused, or negative states
- A hash table to derive the dentry object associated with a given filename or directory quickly
- Stores dentry objects as follows
  - All the "unused" dentries are included in a doubly linked LRU[1] list sorted by time of insertion
  - Each "in use" dentry object is inserted into a list specified by the i_dentry field of the corresponding inode object. The dentry object may become "negative" when the last hard link to the corresponding file is deleted.

[1] Last recently used

Dentry cache

- Reading a directory entry from disk and constructing the corresponding dentry object requires considerable time
- A set of dentries in the in-use, unused, or negative states
- A hash table to derive the dentry object associated with a given filename or directory quickly
- Stores dentry objects as follows
  - All the "unused" dentries are included in a doubly linked LRU[1] list sorted by time of insertion
  - Each "in use" dentry object is inserted into a list specified by the i_dentry field of the corresponding inode object. The dentry object may become "negative" when the last hard link to the corresponding file is deleted.
  - A hash table to quickly resolve the association between a given path and dentry object

[1] Last recently used

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.
  - Similar to a shortcut

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.
  - Similar to a shortcut
  - Contains a text string that is interpreted and followed by the OS as a path to another file or directory

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.
  - Similar to a shortcut
  - Contains a text string that is interpreted and followed by the OS as a path to another file or directory
  - If a symbolic link is deleted, its target remains unaffected

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.
  - Similar to a shortcut
  - Contains a text string that is interpreted and followed by the OS as a path to another file or directory
  - If a symbolic link is deleted, its target remains unaffected
  - If target is moved, renamed or deleted, the symbolic link wont't work

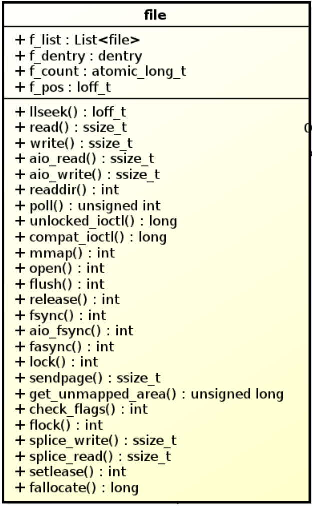
Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.
  - Similar to a shortcut
  - Contains a text string that is interpreted and followed by the OS as a path to another file or directory
  - If a symbolic link is deleted, its target remains unaffected
  - If target is moved, renamed or deleted, the symbolic link wont't work
  - Can create links between directories

Hard link vs Symbolic link

- Hard link is a directory entry that associates a name with a file on a filesystem
  - Multiple hard link to be created for the same file
  - Aliasing effect
  - If target is moved, renamed or deleted, any hard link continues to work
  - Cannot link directories
  - Can only refer to data that exists on the same filesystem
- Soft link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path.
  - Similar to a shortcut
  - Contains a text string that is interpreted and followed by the OS as a path to another file or directory
  - If a symbolic link is deleted, its target remains unaffected
  - If target is moved, renamed or deleted, the symbolic link wont't work
  - Can create links between directories
  - Can cross filesystem boundaries

file_object


file
+ f_list : List<file>
+ f_dentry : dentry
+ f_count : atomic_long_t
+ f_pos : loff_t
+ llseek() : loff_t
+ read() : ssize_t
+ write() : ssize_t
+ aio_read() : ssize_t
+ aio_write() : ssize_t
+ readdir() : int
+ poll() : unsigned int
+ unlocked_ioctl() : long
+ compat_ioctl() : long
+ mmap() : int
+ open() : int
+ flush() : int
+ release() : int
+ fsync() : int
+ aio_fsync() : int
+ fasync() : int
+ lock() : int
+ sendpage() : ssize_t
+ get_unmapped_area() : unsigned long
+ check_flags() : int
+ flock() : int
+ splice_write() : ssize_t
+ splice_read() : ssize_t
+ setlease() : int
+ fallocate() : long

file_object

- Describes how a process interacts with a file it has opened

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter
  - f_pos: Current file offset(file pointer)

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter
  - f_pos: Current file offset(file pointer)
- Defines file_operations

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter
  - f_pos: Current file offset(file pointer)
- Defines file_operations
  - llseek(file, offset, origin)

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter
  - f_pos: Current file offset(file pointer)
- Defines file_operations
  - llseek(file, offset, origin)
  - read(file, buf, count, offset)

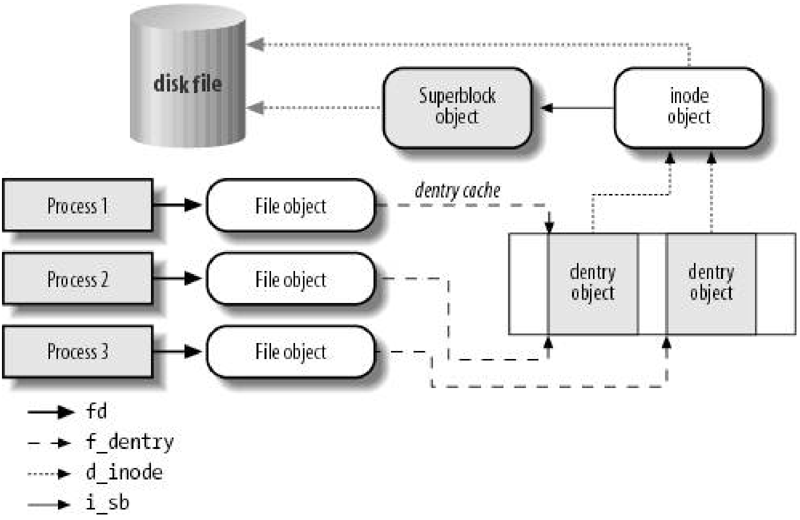
file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter
  - f_pos: Current file offset(file pointer)
- Defines file_operations
  - llseek(file, offset, origin)
  - read(file, buf, count, offset)
  - write(file, buf, count, offset)

file_object

- Describes how a process interacts with a file it has opened
- Created when the file is opened by a process
- Points to a dentry (which points to the inode)
- A dentry can be associated to many file objects
- Defines file's attributes
  - f_list: Pointers to generic file (super block file list)
  - f_dentry: Dentry object associated with the file
  - f_count: File object's reference counter
  - f_pos: Current file offset(file pointer)
- Defines file_operations
  - llseek(file, offset, origin)
  - read(file, buf, count, offset)
  - write(file, buf, count, offset)
  - open(inode, file)

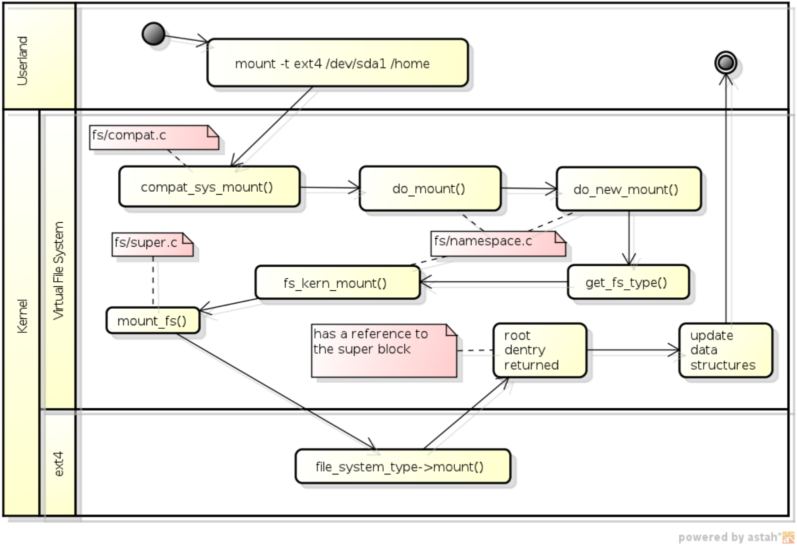
Interaction between process and the VFS



Understanding the Linux Kernel, 3rd Edition, Daniel P. Bovet, Marco Cesati

Agenda

Mount activity diagram



Agenda

What is FUSE?

Filesystem in User Space

- An open source framework for implementing filesystem in user land[1]

[1] http://fuse.sourceforge.net/

What is it good for?

- Higher abstraction - it's easier to write a fuse-based filesystem than a "native" linux filesystem

What is it good for?

- Higher abstraction - it's easier to write a fuse-based filesystem than a "native" linux filesystem
- No kernel recompilation or module installs

What is it good for?

- Higher abstraction - it's easier to write a fuse-based filesystem than a "native" linux filesystem
- No kernel recompilation or module installs
- FUSE is already compiled within the kernel in common distros (e.g. Ubuntu)

What is it good for?

- Higher abstraction - it's easier to write a fuse-based filesystem than a "native" linux filesystem
- No kernel recompilation or module installs
- FUSE is already compiled within the kernel in common distros (e.g. Ubuntu)
- Applications in user space have lots of ready-to-use libraries

What is it good for?

- Higher abstraction - it's easier to write a fuse-based filesystem than a "native" linux filesystem
- No kernel recompilation or module installs
- FUSE is already compiled within the kernel in common distros (e.g. Ubuntu)
- Applications in user space have lots of ready-to-use libraries
- Write your filesystem in any programming language

What is it good for?

- Higher abstraction - it's easier to write a fuse-based filesystem than a "native" linux filesystem
- No kernel recompilation or module installs
- FUSE is already compiled within the kernel in common distros (e.g. Ubuntu)
- Applications in user space have lots of ready-to-use libraries
- Write your filesystem in any programming language
- You won't crash the system :)

What is it NOT good for?

- Performance penalty (switches between user and kernel modes and higher indirection level)

What is it NOT good for?

- Performance penalty (switches between user and kernel modes and higher indirection level)
- If you need to override some kernel functionality (as the dentry cache, for instance)

What is fuse-based?

- Gmail filesystem[1]

[1] http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html
[2] http://fuse.sourceforge.net/sshfs.html
[3] http://en.wikipedia.org/wiki/WikipediaFS

What is fuse-based?

- Gmail filesystem[1]
- sshfs[2]

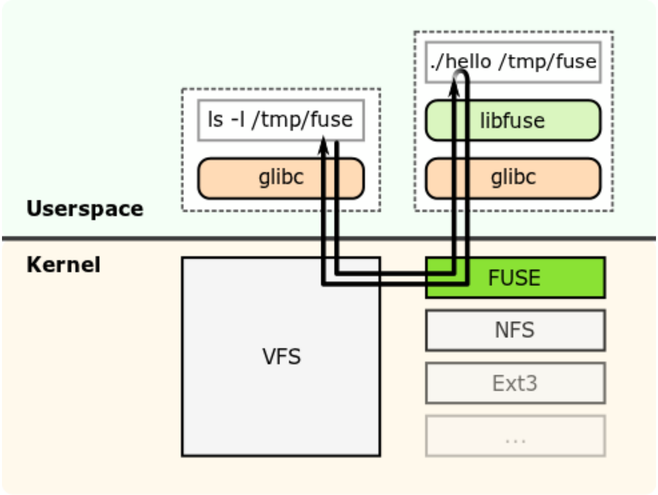
[1] http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html
[2] http://fuse.sourceforge.net/sshfs.html
[3] http://en.wikipedia.org/wiki/WikipediaFS

What is fuse-based?

- Gmail filesystem[1]
- sshfs[2]
- WikipediaFS[3]

[1] http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html
[2] http://fuse.sourceforge.net/sshfs.html
[3] http://en.wikipedia.org/wiki/WikipediaFS

FUSE basic workings



Source: http://fuse.sourceforge.net/

FUSE in deep

- FUSE is composed of two parts

FUSE in deep

- FUSE is composed of two parts
  - User space library - libfuse - provides to the filesystem application an API

FUSE in deep

- FUSE is composed of two parts
  - User space library - libfuse - provides to the filesystem application an API
  - Kernel surrogate filesystem implementation - fs/fuse

FUSE - User space library (libfuse)

- Provides an abstraction layer to the filesystem application

FUSE - User space library (libfuse)

- Provides an abstraction layer to the filesystem application
- Binds the userland application to the FUSE kernel module

FUSE - User space library (libfuse)

- Provides an abstraction layer to the filesystem application
- Binds the userland application to the FUSE kernel module
- Application has to provided implementation to FUSE operations

FUSE - User space library (libfuse)

- Provides an abstraction layer to the filesystem application
- Binds the userland application to the FUSE kernel module
- Application has to provided implementation to FUSE operations
- Communicates with the FUSE kernel module in behalf of the application

FUSE - User space library (libfuse)

- Provides an abstraction layer to the filesystem application
- Binds the userland application to the FUSE kernel module
- Application has to provided implementation to FUSE operations
- Communicates with the FUSE kernel module in behalf of the application
- Listen for FUSE kernel messages that should be forwarded to the application

FUSE - Kernel module (fusefs)

- Manages bound filesystem (but to the kernel there's just FUSE)

FUSE - Kernel module (fusefs)

- Manages bound filesystem (but to the kernel there's just FUSE)
- Selects the appropriate userland application to complete an operation, based on the mount point

FUSE - Kernel module (fusefs)

- Manages bound filesystem (but to the kernel there's just FUSE)
- Selects the appropriate userland application to complete an operation, based on the mount point
- Allows synchronous or multi-threaded operations (mount option)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse
  - libfuse issues a mount call passing /dev/fuse file descriptor (fd) as an option parameter

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse
  - libfuse issues a mount call passing /dev/fuse file descriptor (fd) as an option parameter
  - The VFS passes the mount call down to fusefs

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse
  - libfuse issues a mount call passing /dev/fuse file descriptor (fd) as an option parameter
  - The VFS passes the mount call down to fusefs
  - fusefs associates the mount point to the file from fd

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse
  - libfuse issues a mount call passing /dev/fuse file descriptor (fd) as an option parameter
  - The VFS passes the mount call down to fusefs
  - fusefs associates the mount point to the file from fd
  - libfuse reads the file and fusefs writes to the file

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse
  - libfuse issues a mount call passing /dev/fuse file descriptor (fd) as an option parameter
  - The VFS passes the mount call down to fusefs
  - fusefs associates the mount point to the file from fd
  - libfuse reads the file and fusefs writes to the file
  - libfuse forwards calls to the application through a UNIX socket

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

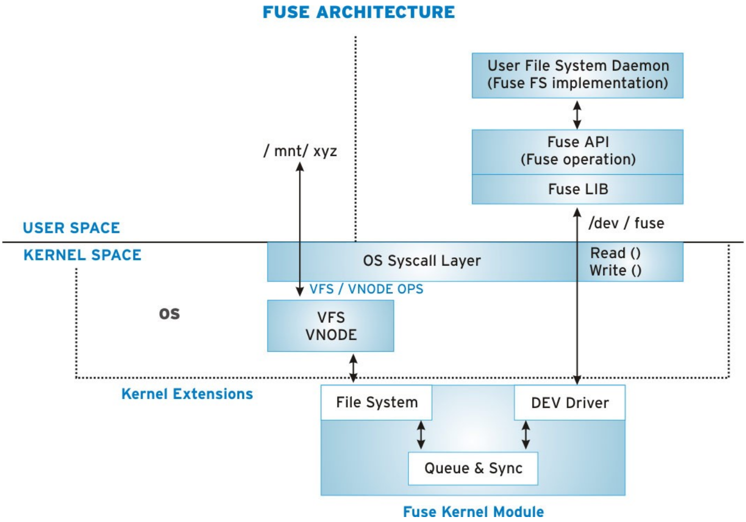
FUSE - How kernel and application communicates?

- fusefs registers a special character file: /dev/fuse
- The application wants to mount its filesystem implementation
- The application issues a libfuse call to start and...
  - libfuse forks
  - libfuse opens /dev/fuse
  - libfuse issues a mount call passing /dev/fuse file descriptor (fd) as an option parameter
  - The VFS passes the mount call down to fusefs
  - fusefs associates the mount point to the file from fd
  - libfuse reads the file and fusefs writes to the file
  - libfuse forwards calls to the application through a UNIX socket
- All above is done by libfuse and the user just need to implement some FUSE operations

---

Sources: FUSE Design Document, William Krier and Erick Liska, 2009)
FUSE Kernel Operations, Vikas Gera, 2006)

FUSE Architecture



Source: FUSE Kernel Operations, Vikas Gera, 2006

Questions?

Andre Petris Esteve - andreesteve@gmail.com
Zhenlei Ji - zhenlei.ji@gmail.com