VFS and filesystems in Linux

Krzysztof Lichota
lichota@mimuw.edu.pl

VFS layer

- Linux since long time has supported many various filesystems (both physical and virtual), now over 30
- The natural way of handling all filesystems was to create separate layer with common functions
- This layer is called Virtual File System (VFS) layer

Types of filesystems handled by VFS

- Physical (backed by physical device, like hard disk, flash or CD) – e.g. ext3, reiserfs, jfs, xfs, jffs, iso9660
- Network filesystems (including cluster filesystems) – e.g. nfs, coda, smbfs, gfs, ocfs
- Pseudofilesystems – provide various information based on some metainfo or allowing some actions – e.g. proc, sysfs, pipefs, futexfs, usbfs
- Special-purpose filesystems – tmpfs, ramfs, rootfs

VFS architecture

- Filesystems handling consists of:
  - generic operations handled by VFS
  - structures representing files and other filesystem objects (handled by VFS)
  - caches (handled by VFS)
  - structures specific to filesystem (defined by filesystem)
  - operations on structures specific to filesystem (defined by filesystem)

VFS and page cache

- VFS is integrated with page cache
- If allows to provide zero-copy semantics (file contents is served from page cache, no copying)
- It simplifies VFS in some cases – all operations (e.g. read) are done on pages
- It also simplifies FS implementation - filesystem can provide page-level operations and page-cache layer together with VFS will handle most work (like mmap and AIO)

VFS abstract structures

- Structures which represent generic (abstract) objects:
  - inode – represents inode on disk
  - dentry – represents directory entry
  - file – represents open file
  - super_block – represents filesystem instance
  - file_system_type – represents filesystem type handling
- Can contain filesystem-specific extra information

VFS caches

- inode cache – caches read inodes and performs delayed writeback of inodes
- dentry cache – caches directory entries to speed up path lookup (dentries are not written to disk)

Generic operations handled by VFS

- VFS code handles:
  - sys_open() and related functions - path lookup, crossing mounts, etc.
  - sys_read(), sys_write(), sys_mmap(), aio, and all other system-call interface functions
  - file descriptor operations (closing, duplicating, etc.)
  - mounting filesystems
- Most operations are done by doing generic work while delegating filesystem-specific functions to filesystem defined hooks (file/inode/super_block/dentry/...)

Example - opening file

- Function do_sys_open():
  - Get unused file descriptor
  - Perform path lookup using namei/ path_lookup functions (open_namei() and do_path_lookup())
    - get root
    - get directory entry - read using FS-specific function if not present in cache, calls iget in superblock if inode not present in cache
    - if directory – lookup name in directory
    - cross mount points, follow symlinks, ...
  - Call filesystem-specific open routine from file_operations (__dentry_open())

Example – filesystem mounting

- Code: do_mount()
  - Lookup directory to be mounted
  - Load filesystem handling module if not present (which registers its filesystem_type)
  - Get filesystem superblock using filesystem_type->get_sb() (vfs_kern_mount())
  - Add mount point to namespace

struct filesystem_type

- Represents filesystem type and is registered by module which handles such filesystem type
- Important fields (struct file_system_type):
  - get_sb() - mounts new instance of filesystem on given device and directory (e.g. ext3_get_sb()) and creates and fills in super_block
  - kill_sb() - unmounts filesystem instance
  - name – name of filesystem type (e.g. “ext3”)
  - fs_flags – various flags which tell how to handle this filesystem type (e.g. FS_REQUIRES_DEV)

struct filesystem_type (2)

- VFS provides convenience functions for filesystem_type operations:
  - get_sb_bdev() - fill superblock for block device-based filesystems
  - get_sb_nodev() - for non-physical filesystems
  - get_sb_single() - for filesystems with single instance
  - kill_block_super() - unmounts block-device based filesystem
  - kill_litter_super()/kill_anon_super() - unmounts non-physical filesystems

struct super_block

- Represents mounted filesystem instance
- Most important fields (struct super_block)
  - struct super_operations *s_op – operations on filesystem instance
  - struct file_system_type *s_type – FS type
  - struct export_operations *s_export_op – NFS export operations
  - s_flags – mounted filesystem instance flags
  - struct dentry *s_root – dentry of root

struct super_operations

- Keeps FS-specific operations performed by filesystem instance
- Important fields (struct super_operations)
  - read_inode – reads inode from disk to cache
  - dirty_inode – marks inode as dirty
  - write_inode – writes inode to disk
  - put_inode – inode is removed from cache
  - delete_inode – remove inode from disk (last hardlink was removed and inode is not used)

struct super_operations (2)

- Superblock important fields (filesystem wide):
  - put_super – filesystem is unmounted
  - write_super – superblock should be written to disk
  - sync_fs – write all state to disk
  - statfs – report statistics about filesystem (used and available space, etc.)

struct inode

- Keeps information about inode in memory
- Important fields (struct inode)
  - i_ino – unique inode number (inode identifier)
  - i_nlink – number of references to inode on disk (hardlinks) and by processes
  - struct inode_operations *i_op – filesystem-specific inode operations
  - struct file_operations *i_fop – default filesystem-specific file operations
  - struct super_block *i_sb – pointer to super_block of filesystem

struct inode (2)

- address_space *i_mapping – pagecache operations
- struct pipe_inode_info *i_pipe – inode information specific to pipes
- struct block_device *i_bdev – inode information specific to block device files
- struct cdev i_cdev – inode information specific to character device files
- i_generation – generation (version) of inode, used mainly for NFS exporting
- i_private – private data for filesystem kept with inode

struct inode_operations

- Important fields (struct inode_operations, e.g. ext3_dir_inode_operations, ext3_file_inode_operations, docs)
  - truncate – truncate file
  - create – create file in directory
  - mkdir – create directory in directory
  - lookup – lookup file name in directory inode
  - link – create hardlink
  - unlink – remove link to inode (and possibly delete if number of references drops to 0)
  - setattr – set file attributes

struct file

- Represents open file
- Most important fields (struct file)
  - f_dentry – dentry associated with file
  - f_vfsmnt – mount point of FS for file
  - struct file_operations *f_op – file operations
  - f_pos – file position
  - f_ra – readahead state
  - private_data – filesystem private data for file
  - struct address_space *f_mapping – address space object for file

struct file_operations

- Important fields (struct file_operations)
  - open – open file
  - read/write – standard synchronous read/write
  - aio_read/aio_write – asynchronous read/write
  - poll – descriptor status polling
  - mmap – memory mapping of file
  - flush – close file descriptor
  - release – close last reference to file
  - fsync/fasync – sync file contents with disk
  - readv/writev – scatter/gather operations

struct file_operations (2)

- sendfile – stream file contents to socket
- lock/setlease – file locking
- splice_read/splice_write – transfer file contents directly between descriptors (no userspace copy using splice() system call)

struct dentry

- Represents directory entry (file name in directory) which points to inode
- Dentries are only cache and not synced to disk
- Important fields (struct dentry)
  - struct inode *d_inode – inode associated with dentry
  - struct dentry *d_parent – dentry of parent directory
  - struct qstr d_name – file name in dentry
  - struct dentry_operations *d_op - operations

struct dentry_operations

- Most important fields (struct dentry_operations)
  - d_compare – comparison function for dentries (for example for case-insensitive comparison)
  - d_delete – all references to dentry have been removed, but it stays in cache
  - d_release – free dentry
  - d_dname – dynamic creation of file name in dentry
- dentry_operations are not so often overridden, for most filesystems it is not necessary

struct address_space (reminder from VM)

- address_space object represents inode which hosts data of some pages and defines how to read/write them on disk
- Page in page cache is represented as pointer to address_space object and index – i.e. offset in page units in file
- As reading/writing files is unified using page cache, filesystems define only address_space_operations (e.g. readpage, writepage) and they do not care if file is written using read/write or using mmap – this is handled by pagecache layer

struct address_space (2)

- Generic VFS functions use address_space object, so if filesystem uses generic functions, it gets much functionality for free (mmap, async IO), it just has to define address_space operations

Generic functions provided by VFS

- VFS provides also generic functions which can be used by filesystems (but they can implement them on their own)
- It is recommended to use VFS-provided functions as they are updated when VFS architecture changes and cause less code duplication (which avoids errors)
- Using generic functions you also get much functionality for free (like mmap, AIO)
- Some functions can be put directly in structure or can be called from filesystem-specific function after FS-specific job

Generic file functions provided by VFS

- Examples of generic file operations
  - generic_file_llseek() - performs llseek() operation as expected in standard systems, using struct file fields
  - generic_file_open() - opens the file
  - do_sync_read()/write() - standard, sync ops
  - generic_file_aio_read/write() - AIO
  - generic_file_mmap() - mmap handling
  - generic_file_splice_read()/write() - splice
- Example: struct ext3_file_operations

Generic inode operations provided by VFS

- Generic inode operations
  - generic_readlink() - generic symlink read
  - generic_setxattr()/getxattr/listxattr/removexattr – extended attributes routines
  - generic_fillattr() - fills inode attributes
  - generic_permission() - checks permissions for inode
  - generic_delete_inode() - deletes inode
  - generic_drop_inode() - generic drop inode

Generic address_space ops provided by VFS

- Generic address_space operations:
  - mpage_readpages(),mpage_readpage – reading page using FS-provided get block function
  - mpage_writepages/mpage_writepage – page write routine using get_block function
  - block_sync_page() - for sync_page
  - generic_block_bmap() - for bmap

Creating simple FS using generic functions

- So, to create the core of simple device-based filesystem, it is enough to provide:
  - get_block routine, used by generic address_space operations, used by generic file operations
  - inode read operation
- Remaining part can be done using generic functions, some default functions are transparently used by VFS is FS-specific function is not provided

Namespaces

- Linux VFS supports namespaces – view of filesystem structure can be different for processes
- Used to implement chroot(), but has more power
- It is possible to do tricks like bind mounts (mount single directory/file in 2 places at the same time)
- All dirty work is done by VFS

Special case - NFS exporting

- Standard VFS interface does not suffice to export filesystem using NFS
- Filesystem must provide struct export_operations in s_export_op of its super_block if it wants to be exported by NFS

Other VFS features

- Linux VFS is flexible enough to create, for example, stacked filesystems
- Linux VFS supports inotify/dnotify – notification of filesystem changes
- VFS supports extended attributes – extended attributes operations can be provided in file_operations or generic functions can be used, which use s_xattr handler defined in super_block

Bibliography

- http://www.cse.unsw.edu.au/~neilb/oss/lin
- http://www.science.unitn.it/~fiorella/guide
- http://www.ibm.com/developerworks/linux/
- http://lxr.linux.no/linux+v2.6.23.12/Docum
- http://linuxvfs.googlepages.com/linuxvfs.h
- http://lxr.linux.no/linux+v2.6.23.12/Docum