Operating Systems 2

Virtual File System

Arkadiusz Chrobot

Department of Computer Science

May 10, 2020

Outline

1. Introduction
2. Unix File System Model
3. Virtual File System Elements
4. Superblock Object
5. Inode Object
6. Dentry Object
7. File Object
8. Other Data Structures

Introduction

The slab allocator is not the only idea invented by employees of Sun Microsystem that has been also incorporated into the Linux kernel. One of the others is *Virtual File System* (VFS). It is an abstraction layer that mediates between the real file system and the rest of the kernel. The VFS enables Linux to support many different file systems. It is also interesting because its code is written in object-oriented style, although entirely with the use of the C language. The VFS provides a unified API for all file systems supported by Linux. It means that user-space software operates on a file using the same system calls, like: `open()`, `read()`, `write()`, `close()`, regardless of the file system of the storage device where this file is retained. The VFS "translates" those calls to operations specific to this file system. Summarizing, the Virtual File System creates a common model (an abstraction) of file system that represents features and operations of a real file system.

Unix File System Model

The VFS model is based on the four main elements of the original Unix file system: a *file*, a *directory*, an *inode* (also spelled as "*i-node*") and a *superblock*. Generally the file system is a data structure storing a hierarchically ordered information. In the Unix-like systems the file systems are *mounted* to the common directory tree in specified *mounting points*. They create a common *namespace* accessible to user-space processes[1]. When a user process accesses a file system it doesn't know on which physical medium this system is located, in contrast to the MS Windows system, where it has to specify the medium. Files are ordered sequence of bytes and also one of the two most important concepts of any Unix-like operating system (the other is a process). Each file has its unique name. The user processes can perform operations on files, like opening, reading, writing and closing.

[1] In most of the modern Unix-like systems each user-space process can have its own namespace. In Linux this option is available since the 2.4 kernel version.

Unix File System Model

The directories are files that store information (so-called metadata) about other files. Some of the directories, called *subdirectories* are nested in other directories. Sequences of subdirectories names form *paths*. Each element of the path (name of the directory or file) is called a *directory entry* or *dentry* for short. Unix handles directories the same way as regular files i.e. with the use of the same operations. Some of the file metadata, like the time and date of last modification, the size, are also stored in separate blocks on the medium, called inodes. The metadata and control information concerning the entire file system are stored in the main block on the physical medium called a *superblock*. Some real file systems don't exactly match this model, but thanks to the VFS they still can be used in Linux. This abstraction layer represents all their elements in a way that makes them fit the described model.

Virtual File System Elements

The Virtual File System code follows the object-oriented programming model although it is entirely written in the C language. The VFS objects are just variables which types are defined by structures that represents classes[2]. Each of the structures has a member which is a pointer to a structure of function pointers. This structure points to functions that implement operations performed on the represented real file system. In other words those functions are methods. The VFS defines four types of objects: the superblock objects which represent superblocks of mounted file systems, the inode objects, which represent files in the file systems, dentry objects which represent single directory entries and file objects that represent opened files. Each object of a given type has its own object (structure) of operations. The `super_operations` object groups the file system methods, the `inode_operations` object groups the file methods.

[2] In the C++ language it is also possible to use the `struct` keyword instead of the `class` for defining a class of an object.

Virtual File System Elements

The `dentry_operations` object groups dentry operations and finally the `file_operations` object groups open file operations. Some of them are "inherited" from the groups of generic functions that implement operations common to all file systems supported by Linux.

Superblock Object

All data about mounted file system are stored in a superblock object. Usually those data correspond to the data stored in a superblock of the external memory device file system. There are however in-memory file systems, like `sysfs` and `procfs`, that don't have a physical superblock. In their cases the content of the superblock object is generated on the fly. The superblock object type is defined by the `struct super_block` structure. It has several members that store such data as: the identifier of the device where the file system is, the maximal size of the file in this file system, the identifier of the file system type, the number of active references to the file system, etc. One of the most important fields of this structure is the `s_op` member, that points to the superblock operations object, also called the *superblock method table*. This object is in fact a structure of the `struct super_operations` type. Each member of this structure points to a function invoked when the kernel has to perform some operation of the superblock.

Superblock Object

For example, decrementing of the reference counter is accomplished by calling the `put_super()` function like this:

```
sb->s_op->put_super(sb);
```

The `sb` variable is a pointer to the superblock object. Because the C language, unlike the C++ language, doesn't have the `this` pointer, the `sb` variable has to be passed to the `put_super()` function, so it knows for which object it has been invoked. Other methods of superblock include: `alloc_inode()` — allocates and initializes the inode object, `destroy_inode()` — deletes the inode object, `read_inode()` — reads the content of the inode block from the storage device and stores it in the inode object, `dirty_inode()` — the function marks the inode object as modified it content may differ from the content of the inode block in the storage device, that it represents, `write_inode()` — writes the data from the inode object, to the inode block in the storage medium, `drop_inode()` — this method is called by the VFS when the last reference to the inode object has been dropped;

Superblock Object

in a Unix-like file system it results in deleting the inode, `evict_inode()` — the method removes the inode block from the storage medium, `put_super()` — the method is called when the file system is unmounted, to decrease the superblock object reference counter; if it drops to zero then the function also deletes the object, `sync_fs()` — the method writes data from the superblock object to the superblock in the storage medium; in other words it updates the superblock, `statfs()` — the method returns statistics about the file system, `remount_fs()` — the method is mounts the file system with new options, `umount_begin()` — the function aborts the operation of mounting file system; it is used by networked file systems like the NFS. Not all methods from this list perform operations on the superblock. Some of them also operate on inodes. Also not all of them have to be implemented by in a code that handles a real file system. The value of the function pointers to the unimplemented methods is NULL. The superblock object is implemented and initialized by the `alloc_super()` kernel function.

Inode Object

The inode objects store data required to perform operations on files and directories associated with those objects. Those data include: the file owner identifier, the so-called real number of the storage device where the file is kept, file access permissions, the size of the file and the inode identifier. In case of the Unix-like file systems the inode objects represent inode blocks. For the other file systems the data for those objects is acquired directly from files or other places in the storage medium. There are also file systems that don't store some of the data needed by inode objects. In that cases default values are used. The inodes are associated not only with regular files but also with special files like the *device files* and FIFOs. In each inode object are two members that point to the method tables. The fist one is called `f_op` and it points to a object of file operations. The other is named `i_op` and it points to the inode operations object.

Inode Object

The inode operations include: `create()` — allocates the new inode object, `lookup()` — it searches directory for an inode block associated with the specified directory entry, `link()` — creates a hard link, `unlink()` — removes a link, `symlink()` — creates a symbolic link, `mkdir()` — creates a directory, `rmdir()` — removes an empty directory, `mknod()` — creates a special file (for example the device file), `rename()` — renames the file, `readlink()` — copies a specified part of the full path associated with a given link, `follow_link()` — translates a symbolic link to the inode it points, `permissions()` — handles the access permissions in some of the file systems, `setattr()` — initializes the event which informs that the content of the inode has been modified, `getattr()` — notifies that the inode object should be updated from the inode block in the storage medium, `setxattr()` — sets the extended attributes, `getxattr()` — gets the value of the specified extended attribute, `listxattr()` — copies the list of extended attributes to a buffer, `removexattr()` — removes a specified extended attribute.

Dentry Object

The dentry objects are associated with each name that occurs in a path. For example, the kernel creates three such objects for the following path: `/usr/java`. The first one is associated with the `/` character, which represents the root (main) directory, the next one represents the `usr` directory, and the last one is associated with the `java` directory. The dentry objects also represent a file names that end some of the paths and mounting points that can be located inside a path. Those objects don't have their counterparts in the storage medium. They are created on the fly when a path is resolved. The dentry objects are necessary for performing operations specific to the directories, like traversing the directory tree. Those objects are represented by structures of the `struct dentry` type. Each dentry object can be in one of the three states: used, unused and negative. The dentry object in the used state is associated with a valid inode object and it has been recently used by the kernel. The dentry object in the unused state is associated with a valid inode object, but it hasn't been used for a while.

Dentry Object

The kernel doesn't delete such an object, unless it is short of memory. Otherwise it keeps the dentry object, because it may be useful in the future. The dentry object in the negative state is not associated with a valid inode object. It means that it refers to a file or directory that has been deleted or that never existed. The kernel also doesn't delete such an object without a reason. Those objects may be useful when the paths that contain entries associated with those object are referenced. They can prevent the kernel from resolving incorrect paths that already have been processed. Dentry objects are allocated and deallocated by the slab allocator. The Linux kernel maintains also a buffer of dentry objects. It consists of three elements: the list of all dentry objects, the list of recently accessed object, which usually stores objects in the used and unused states, and the hash array that applies a hash function to quickly locate a given dentry object in the buffer. The kernel start resolving a path with the last entry.

Dentry Object

If it successfully locates the dentry object associated with the name in the buffer, then there is a chance that the rest of the dentry objects have been already created and it doesn't have to recreate them. The kernel also has a buffer for the inode objects associated with the buffered dentry objects. The dentry object method table is represented by the structure of the `struct dentry_operations` type. The operations include: `d_revalidate()` — the method verifies the validity of the dentry object, `d_hash()` — it is a hash function, `d_compare()` — compared names of two files or directories, `d_delete()` — the method is invoked when the reference counter of the dentry object drops to zero, `d_release()` — it releases the dentry object, `d_iput()` — is invoked when the dentry object looses the inode object associated with it.

File Object

From the user-space process point of view the file object is the most important VFS variable. Those objects are created by the `open()` system call and destroyed by the `close()` system call. The file object points to a dentry object that point to an inode object associated with a file opened by the user process. With a single file can be associated several file objects, depending on how many times it has been opened by user processes. The type of this object is defined by the `struct file` structure that has a member pointing to the structure of the `struct file_operations` type. The latter implements the method table of the file objects. The file methods include: `llseek()` — updates the file pointer, `read()` — reads the file, `write()` — writes the file, `poll()` — puts a user process and wakes it when some activity happens on a file, `unlocked_ioctl()` and `compat_ioctl()` — those two methods are used for perform some operations on device files that cannot be expressed with the use of regular file operations; in older kernels there had been only one such function called `ioctl()` which used the BKL;

File Object

the `unlocked_ioctl()` don't use the BKL, like the `compat_ioctl()`, but the latter also preserves the compatibility of file handling between the 32-bit and 64-bit hardware platforms; in other words it allows the 32-bit file operations to be performed on 64-bit hardware platforms, `mmap()` — it maps a file to the memory, `open()` — opens a file, `flush()` — its behaviour depends on the file system, but it always decrements the file object reference counter, `release()` — invoked when the file object reference counter drops to zero, `fsync()` — it writes all the buffered changes to the storage medium, `aio_fsync()` — it does the same as `fsync()`, but without putting to sleep the user process that issued the operation, `fasync()` — activates or deactivates signals that notify about asynchronous operations, `read_iter()` — it reads the data from a file and stores them in multiple buffers, `write_iter()` — it writes data from multiple buffers to a single file, `sendpage()` — it transfers data between files, `get_unmapped_area()` — the method maps file to an unused area of memory, `lock()` — the method manages the file lock,

File Object

`flock()` — it is used for implementing a system call of the same name which provides the advisory file locking, `check_flags()` — verifies flags set by the `fcntl()` function.

Other Data Structures

Aside from the four object types, the VFS uses several other structures. Structure of the `file_system_type` type store data about the *file system types* supported by Linux. Those structures are used by the `get_sb()` function that reads the content of the superblock of a give file system from the storage medium. For each file system supported by Linux the kernel has one such a structure. When a file system is mounted the kernel creates a variable of the `struct vfsmount` type. This structure stores data about the mounting point, including the flags specifying the operations that can be performed on the file system. With each user process are associated three VFS data structures. The structure of the `struct files_struct` stores data about files opened by the process and their descriptors, including the pointers to the file objects. The structure of the `fs_struct` type stores data about the file system associated with the given process, including the current and the root directory. The structure of the `struct mnt_namespace` type defines a unique view of the mounted file systems for the user process.

Other Data Structures

The two former structures can be shared by related processes. The last one is by default shared by all processes in the system, but it can also be defined for a given process.

Questions

?

The End

Thank You for Your attention!