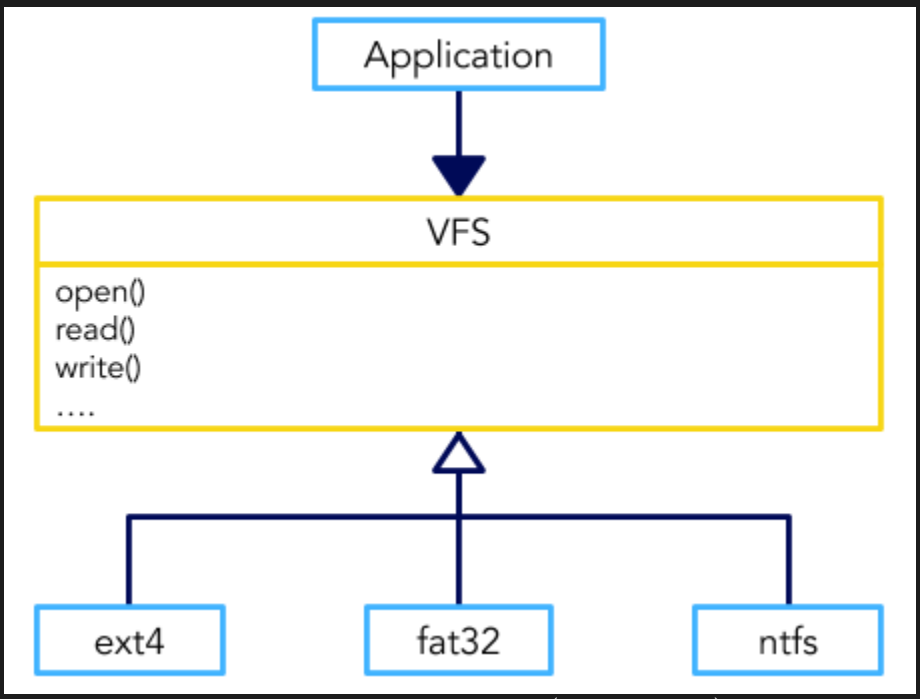
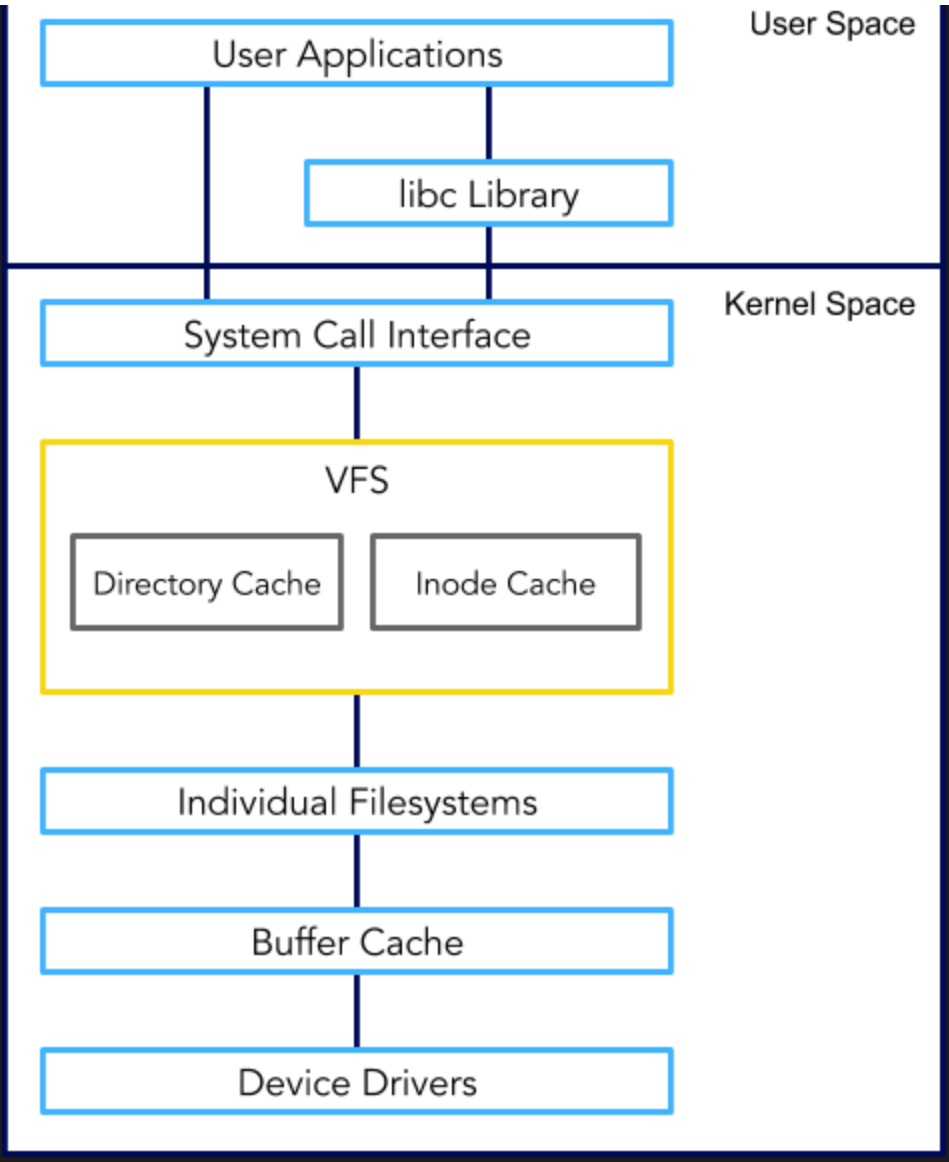
The VFS is sandwiched between two layers: the upper and the lower. The upper layer is the system call layer where a user space process traps into the kernel to request a service (which is usually accomplished via libc wrapper functions) -- thus catalyzing the VFS's processes. The lower layer is a set of function pointers, one set per filesystem implementation, which the VFS calls when it needs an action performed that requires information specific to a particular filesystem. Taken another way, the VFS could be thought of as the glue between the user space applications' requests for file-related services and the filesystem-specific functions which the VFS invokes as needed. From a high-level view, this can be modeled as a standard abstract interface:




Naturally, the lower layer is where Star Lab's AuthFS (and FortiFS) filesystems hook into the VFS. Therefore it is necessary that we understand the VFS if we want to work in the realm of filesystem implementation. By having the VFS require filesystems to define certain callback functions, such as read and write, we can operate on files without worrying about the implementation details. This way, the user space programmer is only concerned with reading from a file or writing to it and not with how the file is physically stored in a USB flash drive, disk drive, over the network, or on any other conceivable medium.

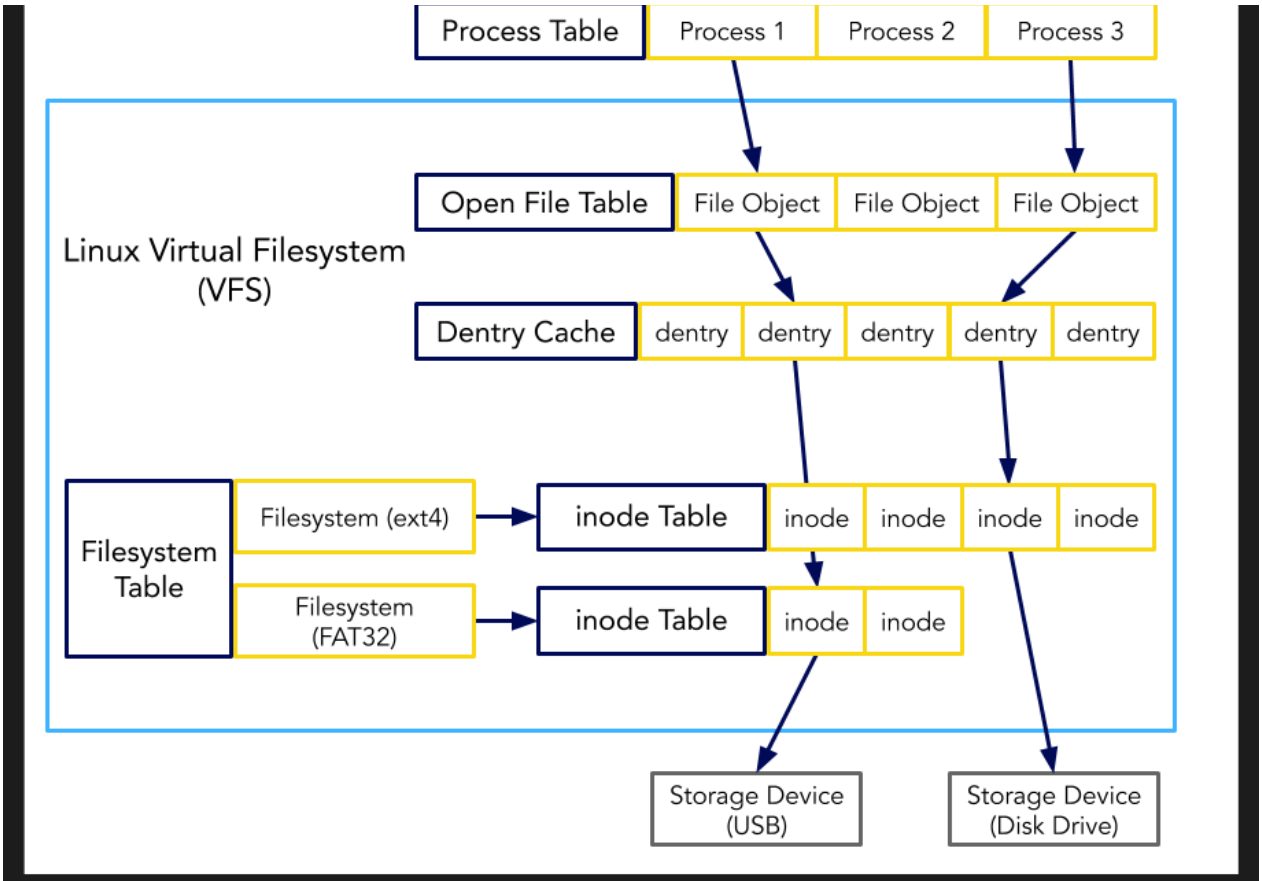
This results in the following architecture:




We will not concern ourselves here with a thorough explanation of the buffer cache or the device drivers. The device drivers are ultimately responsible for taking requests to read and write to a particular device (or medium, such as over the network) and realizing those requests. The buffer cache is a layer of optimization that acts an intermediary between the filesystem drivers and the device drivers.

# HOW IS VFS IMPLEMENTED?

While the previous high-level overview is appropriate for understanding the 10,000 foot view of the VFS, the devil really is in the details as depicted here:




Understanding all of details would make for an unreasonably long article. So, for this post, let's begin to break down the VFS with a tour of the primary data structures involved, namely:

- Filesystem Types
- Superblocks
- Inodes
- Dentries
- Files

# FILESYSTEM TYPES

In order to use a filesystem, the kernel must know the filesystem type, defined in include/linux/fs.h (and shorted here for brevity):

```
struct file_system_type {
    const char                  *name;      // friendly name of the
filesystem -- like 'ext2'
    int                         fs_flags;   // mostly obscure options
    int                         (*init_fs_context)(struct fs_context
*);
    const struct fs_parameter_spec  *parameters;
```

```
    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 hlist_head               fs_supers;
};
```

Note: All data structures mentioned here are shown as they were in Linux v5.7-rc4.

`struct file_system_type` maintains data related to a filesystem but is not associated with any particular 'instance of a filesystem', a phrase which here means a mounted filesystem. Examples of filesystems include:

- ext2/3/4
- fat16/32
- ntfs
- btrfs
- isofs
- reiserfs
- squashfs
- ... and the list goes on.

The VFS maintains a linked-list of known filesystem types, which can be viewed in user space by executing `cat /proc/filesystems`.

A snippet of what our system prints:

```
nodev    sysfs
nodev    tmpfs
nodev    proc
nodev    debugfs
nodev    sockfs
nodev    bpf
nodev    pipefs
nodev    ramfs
nodev    pstore
         ext3
         ext2
         ext4
         vfat
nodev    nfsd
```

You will notice a theme in most kernel data structures in that they are linked to and indexed in many different ways. In `struct file_system_type`, the member `struct file_system_type *next` is a pointer to the next filesystem type that the kernel knows about, i.e., every filesystem type is part of a global list of all filesystem types in the kernel. You can find this global list in `fs/filesystems.c`:

```
static struct file_system_type *file_systems;
static DEFINE_RWLOCK(file_systems_lock);
```

The member `struct hlist_head fs_supers` indicates that each filesystem type contains a hash table of all superblocks of the same filesystem type. We shall describe the superblock shortly.

Filesystem types are registered and unregistered in the kernel via these functions in include/linux/fs.h:

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

Once a filesystem type is known to the kernel, a filesystem of that type may be mounted to a location in the directory tree.

# SUPERBLOCK

The superblock is a structure that represents an instance of a filesystem, i.e., a mounted filesystem. The superblock is defined in include/linux/fs.hs (and greatly shortened here for brevity):

```
struct super_block {
    struct list_head                s_list;     // list of all other
superblocks of this filesystem type
    dev_t                           s_dev;      // device associated with
this mount
    unsigned long                   s_blocksize;
    loff_t                          s_maxbytes;
    struct file_system_type         *s_type;    // struct describing the
type of filesystem this mount represents
    const struct super_operations   *s_op;
    uuid_t                          s_uuid;     // unique ID of this mount
    struct list_head                s_inodes;
    unsigned long                   s_magic;    // magic number of this
filesystem
    struct dentry                   *s_root;
    int                             s_count;
    void                            *s_fs_info;
    const struct dentry_operations  *s_d_op;
};
```

Learn about magic numbers here.

The superblock is typically stored on the storage device itself and loaded into memory when mounted.

There are a few fields we want to pay special attention to. First, `s_list` is a linked list to the other superblocks of the same filesystem type. Second, `s_inodes` is the list of inodes within this

filesystem mount (which we explain in a moment). And third, s_op, which points to a struct that defines a set of functions which provide data about the superblock. `struct super_operations` is one example of many of a group of function pointers which are provided per filesystem type by that filesystem's implementation thus keeping the VFS agnostic to the details of a particular filesystem's inner workings.

# INODE

A filesystem object can be one of the following types:

- socket
- symbolic link
- regular file
- block device
- directory
- character device
- FIFO

An inode, short for index node, exists per object in all filesystems for all of the filesystem object types. It is defined in include/linux/fs.h (and shortened here for brevity):

```
struct inode {
    umode_t                         i_mode;     // access permissions,
i.e., readable or writeable
    kuid_t                          i_uid;      // user id of owner
    kgid_t                          i_gid;      // group id of owner
    unsigned int                    i_flags;
    const struct inode_operations   *i_op;
    struct super_block              *i_sb;
    struct address_space            *i_mapping;
    unsigned long                   i_ino;      // unique number
identifying this inode
    const unsigned int              i_nlink;    // number of hard links
    dev_t                           i_rdev;
    loff_t                          i_size;     // size of inode contents
in bytes
    struct timespec64               i_atime;    // access time
    struct timespec64               i_mtime;    // modify time
    struct timespec64               i_ctime;    // creation time
    unsigned short                  i_bytes;    // bytes consumed
    const struct file_operations    *i_fop;
    struct address_space            i_data;
};
```

Note the fields i_op and i_fop. These are more examples of function pointers which are provided by the implementation of a particular filesystem. `struct inode_operations` defines the set of callback functions that operate on the inode. These functions do things like:

- change permissions
- create files
- make symlinks
- make directories
- rename files `struct file_operations` defines the set of callback operations that can be called on a `struct file` object. We will discuss this data structure later -- but in short, a `struct file` is created for every instance of an open file.

# DENTRY AND THE DENTRY CACHE

A `dentry`, short for directory entry, is the glue that holds inodes and files together by relating inode numbers to filenames. However, in reality, things are a bit more tricky; lets take another look at `struct inode`.

Even though there is an inode for every object in the filesystem, `struct inode` itself does not contain a name or any friendly identifier for that object. The reason for this is because an inode can be referred to by multiple filesystem objects, each with their own unique name (see hard and symbolic links). Therefore, a dentry maps a filename to its respective inode. Moreover, a dentry exists not just for filenames, but also for directories and thus for every component of a path. For instance, the path `/mnt/cdrom/foo` contains dentries for the components `/`, `mnt`, `cdrom`, and `foo`.

The dentry structure is defined in include/linux/dcache.h (and shortened here for brevity):

```
struct dentry {
    struct hlist_bl_node            d_hash;                     // lookup
hash list
    struct dentry                   *d_parent;                     //
parent directory
    struct qstr                     d_name;
    struct inode                    *d_inode;                   // where
the name belongs to
    unsigned char                   d_iname[DNAME_INLINE_LEN];  // small
names
    const struct dentry_operations  *d_op;
    void                            *d_fsdata;                  // fs-
specific data
    struct list_head                d_child;                    // child
of parent list, i.e., our siblings
    struct list_head                d_subdirs;                  // our
children
};
```

We can see that `d_name` contains the name of the file and `d_inode` points to its associated inode.

Lastly, another reason for having dentries is one of optimization: Having to constantly compare strings to lookup filesystem paths is computationally expensive. So after performing a lookup, the kernel caches dentries into the aptly named dentry cache. As a result, file access speed is significantly increased.

# FILE

A file object represents an open file and is defined in include/linux/fs.h (and shortened here for brevity):

```
struct file {
    struct path                     f_path;         // a dentry and a
mount point which locate this file
    struct inode                    *f_inode;       // the inode
underlying this file
    const struct file_operations    *f_op;          // callbacks to
function which can operate on this file
    spinlock_t                      f_lock;
    atomic_long_t                   f_count;
    unsigned int                    f_flags;
    fmode_t                         f_mode;
    struct mutex                    f_pos_lock;
    loff_t                          f_pos           // offset in the file
from which the next read or write shall commence
    struct fown_struct              f_owner;
    void                            *private_data
    struct address_space            *f_mapping;     // callbacks for
memory mapping operations
};
```

We would like to emphasize that this represents an *open* file and contains data such as flags used while opening the file and the offset from which a process can read or write from. When the file is closed, this data structure is removed from memory - operations such as writing data will be delegated to its respective inode.
Although the information presented thus far is not detailed enough to allow one to implement and or modify a filesystem, it will hopefully act as a starting point from which to expand one's knowledge. In particular, the following details will hopefully be addressed in upcoming blogs:

- What exactly are all the callback functions and which ones must we implement? Answering these questions will propel us into the majority of the details that are necessary to know to competently implement or modify filesystems in Linux.
- What are the system calls which involve the VFS and how do they end up calling our callback functions? Tracing system calls through the VFS and into the various callback functions will be instrumental in understanding the VFS's inner workings.
- A superblock represents a mounted filesystem.
- The VFS maintains a list of superblocks.
- An inode represents a filesystem object (i.e. file).
- Each superblock maintains a list of inodes.

- A dentry translates a file path component into inodes.
- The dcache contains all dentries.

---

The flexibility and extensibility of support for Linux file systems is a direct result of an abstracted set of interfaces. At the core of that set of interfaces is the virtual file system switch (VFS).

The VFS provides a set of standard interfaces for upper-layer applications to perform file I/O over a diverse set of file systems. And it does it in a way that supports multiple concurrent file systems over one or more underlying devices. Additionally, these file systems need not be static but may come and go with the transient nature of the storage devices.

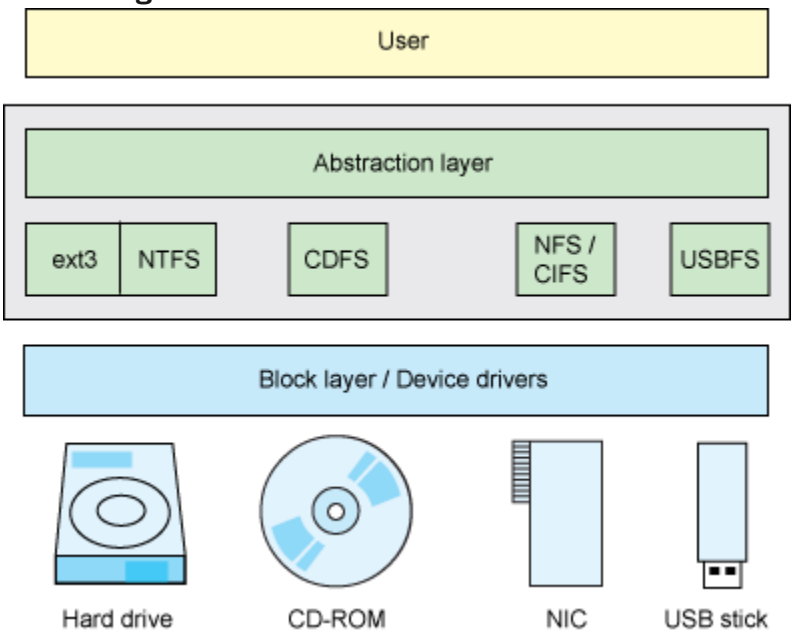
**VFs or VFS?**

You'll find VFS also defined as *virtual file system,* but *virtual file system switch* is a much more descriptive definition, as the layer switches (that is, multiplexes) requests across multiple file systems. The `/proc` file system adds even more confusion here, as it is commonly called a virtual file system.

For example, a typical Linux desktop supports an ext3 file system on the available hard disk, as well as the ISO 9660 file system on an available CD-ROM (otherwise called the *CD-ROM file system,* or CDFS). As CD-ROMs are inserted and removed, the Linux kernel must adapt to these new file systems with different contents and structure. A remote file system can be accessed through the Network File System (NFS). At the same time, Linux can mount the NT File System (NTFS) partition of a Windows®/Linux dual-boot system from the local hard disk and read and write from it.

Finally, a removable USB flash drive (UFD) can be hot plugged, providing yet another file system. All the while, the same set of file I/O interfaces can be used over these devices, permitting the underlying file system and physical device to be abstracted away from the user (see Figure 1).

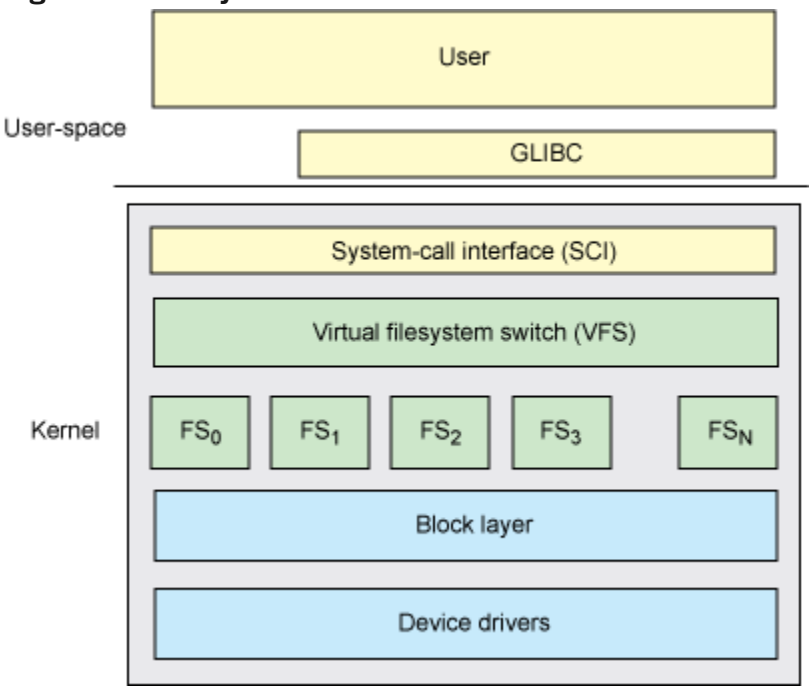
**Figure 1. An abstraction layer provides a uniform interface over different file systems and storage devices**




## Layered abstractions

Now, let's add some concrete architecture to the abstract features that the Linux VFS provides. Figure 2 shows a high-level view of the Linux stack from the point of view of the VFS. Above the VFS is the standard kernel system-call interface (SCI). This interface allows calls from user-space to transition to the kernel (in different address spaces). In this domain, a user-space application invoking the POSIX `open` call passes through the GNU C library (`glibc`) into the kernel and into system call de-multiplexing. Eventually, the VFS is invoked using the call `sys_open`.

**Figure 2. The layered architecture of the VFS**




**Earlier VFS implementations**
Linux wasn't the first operating system to incorporate a virtual layer to support a common file model.

**Earlier VFS implementations**
include Sun's VFS (in SunOS version 2.0, circa 1985) and IBM and Microsoft's "Installable File System" for IBM OS/2. These approaches to virtualizing the file system layer paved the way for Linux's VFS.

The VFS provides the abstraction layer, separating the POSIX API from the details of how a particular file system implements that behavior. The key here is that Open, Read, Write, or Close API system calls work the same regardless of whether the underlying file system is ext3 or Btrfs. VFS provides a common file model that the underlying file systems inherit (they must implement behaviors for the various POSIX API functions). A further abstraction, outside of the VFS, hides the underlying physical device (which could be a disk, partition of a disk, networked storage entity, memory, or any other medium able to store information—even transiently).

In addition to abstracting the details of file operations from the underlying file systems, VFS ties the underlying block devices to the available file systems. Let's now look at the internals of the VFS to see how this works.

## VFS internals

Before looking at the overall architecture of the VFS subsystem, let's have a look at the major objects that are used. This section explores the superblock, the index node (or *inode*), the directory entry (or *dentry*), and finally, the file object. Some additional elements, such as caches, are also important here, and I explore these later in the overall architecture.

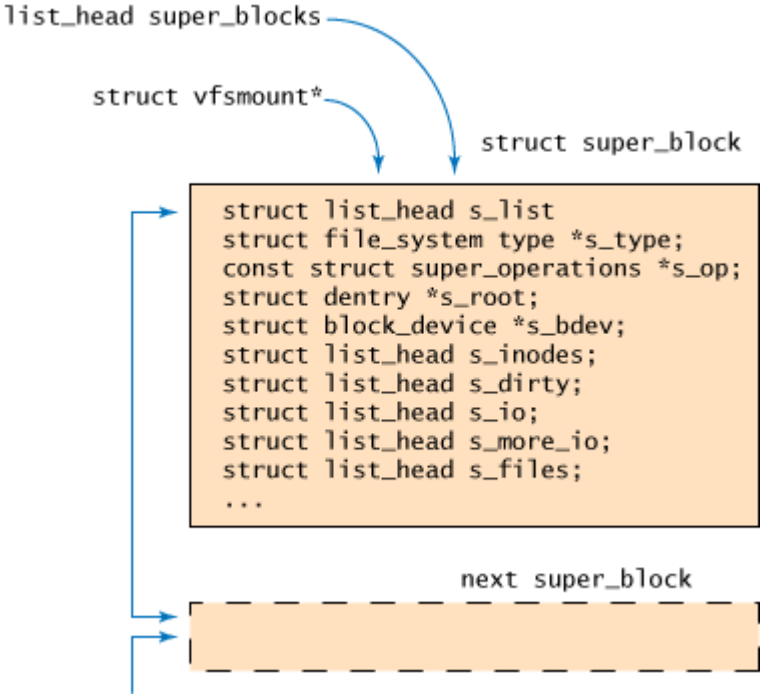
### Superblock

The *superblock* is the container for high-level metadata about a file system. The superblock is a structure that exists on disk (actually, multiple places on disk for redundancy) and also in memory. It provides the basis for dealing with the on-disk file system, as it defines the file system's managing parameters (for example, total number of blocks, free blocks, root index node).

On disk, the superblock provides information to the kernel on the structure of the file system on disk. In memory, the superblock provides the necessary information and state to manage the active (mounted) file system. Because Linux supports multiple concurrent file systems mounted at the same time, each `super_block` structure is maintained in a list (`super_blocks`, defined in ./linux/fs/super.c, with the structure defined in /linux/include/fs/fs.h).

Figure 3 provides a simplified view of the superblock and its elements.
The `super_block` structure refers to a number of other structures that encapsulate other information. The `file_system_type` structure, for example, maintains the name of the file system (such as *ext3*) as well as various locks and functions to get and remove the `super_block`. The `file_system_type` object is managed through the well-known `register_file system` and `unregister_file system` functions (see ./linux/fs/file systems.c). The `super_operations` structure defines a number of functions for reading and writing inodes as well as higher-level operations (such as remounting). The root directory entry (`dentry`) object is cached here also, as is the block device on which this file system resides. Finally, a number of lists are provided for managing inodes, including `s_inodes` (a list of all inodes), `s_dirty` (a list of all dirty inodes), `s_io` and `s_more_io` (parked for writeback), and `s_files` (the list of all opened files for a given file system).

**Figure 3. Simplified view of the super_block structure and its related elements**




Note that within the kernel, another management object called `vfsmount` provides information on mounted file systems. The list of these objects refers to the superblock and defines the mount point, name of the /dev device on which this file system resides, and other higher-level attachment information.

## The index node (inode)

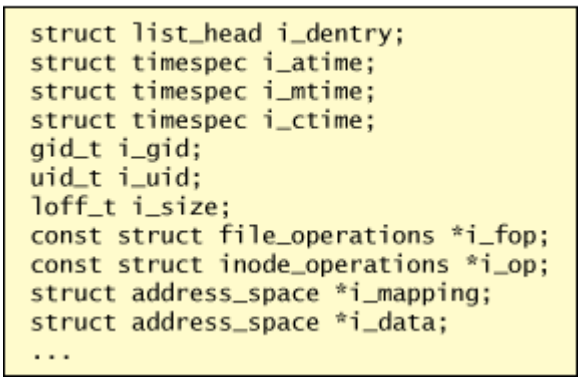
Linux manages all objects in a file system through an object called an *inode* (short for *index node*). An inode can refer to a file or a directory or a symbolic link to another object. Note that because files are used to represent other types of objects, such as devices or memory, inodes are used to represent them also.

Note that the inode I refer to here is the VFS layer inode (in-memory inode). Each file system also includes an inode that lives on disk and provides details about the object specific to the particular file system.

VFS inodes are allocated using the slab allocator (from the `inode_cache`; see resources on the right for a link to more information on the slab allocator). The inode consists of data and operations that describe the inode, its contents, and the variety of operations that are possible on it. Figure 4 is a simple illustration of a VFS inode consisting of a number of lists, one of which refers to the dentries that refer to this inode. Object-level metadata is included here, consisting of the familiar manipulation times (create time, access time, modify time), as are the owner and permission data

(group-id, user-id, and permissions). The inode refers to the file operations that are possible on it, most of which map directly to the system-call interfaces (for example, `open`, `read`, `write`, and `flush`). There is also a reference to inode-specific operations (`create`, `lookup`, `link`, `mkdir`, and so on). Finally, there's a structure to manage the actual data for the object that is represented by an address space object. An *address space object* is an object that manages the various pages for the inode within the page cache. The address space object is used to manage the pages for a file and also for mapping file sections into individual process address spaces. The address space object comes with its own set of operations (`writepage`, `readpage`, `releasepage`, and so on).

**Figure 4. Simplified representation of the VFS inode**




Note that all of this information can be found in ./linux/include/linux/fs.h.

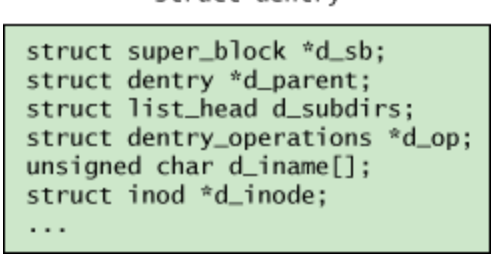
## Directory entry (dentry)

The hierarchical nature of a file system is managed by another object in VFS called a *dentry* object. A file system will have one root dentry (referenced in the superblock), this being the only dentry without a parent. All other dentries have parents, and some have children. For example, if a file is opened that's made up of /home/user/name, four dentry objects are created: one for the root `/`, one for the `home` entry of the root directory, one for the `name` entry of the `user` directory, and finally, one dentry for the `name` entry of the user directory. In this way, dentries map cleanly into the hierarchical file systems in use today.

The dentry object is defined by the dentry structure (in ./linux/include/fs/dcache.h). It consists of a number of elements that track the relationship of the entry to other entries in the file system as well as physical data (such as the file name). A simplified view of the dentry object is shown in Figure 5. The dentry refers to the `super_block`, which defines the particular file system instance in which this object is contained.

Next is the parent dentry (parent directory) of the object, followed by the children dentries contained within a list (if the object happens to be a directory). The operations for a dentry are then defined (consisting of operations such as `hash`, `compare`, `delete`, `release`, and so on). The name of the object is then defined, which is kept here in the dentry instead of the inode itself. Finally, a reference is provided to the VFS inode.

**Figure 5. Simplified representation of the dentry object**

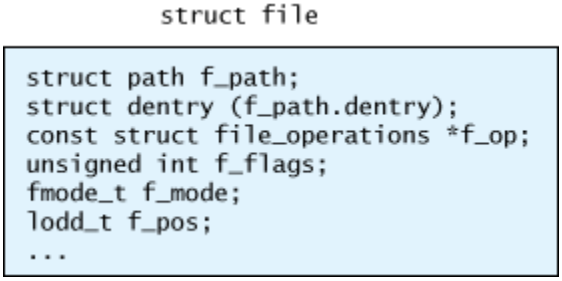



Note that the dentry objects exist only in file system memory and are not stored on disk. Only file system inodes are stored permanently, where dentry objects are used to improve performance. You can see the full description of the dentry structure in ./linux/include/dcache.h.

## File object

For each opened file in a Linux system, a `file` object exists. This object contains information specific to the open instance for a given user. A very simplified view of the file object is provided in Figure 6. As shown, a `path` structure provides reference to both the `dentry` and `vfsmount`. A set of file operations is defined for each file, which are the well-known file operations (`open`, `close`, `read`, `write`, `flush`, and so on). A set of flags and permissions is defined (including group and owner). Finally, stateful data is defined for the particular file instance, such as the current offset into the file.

**Figure 6. Simplified representation of the file object**

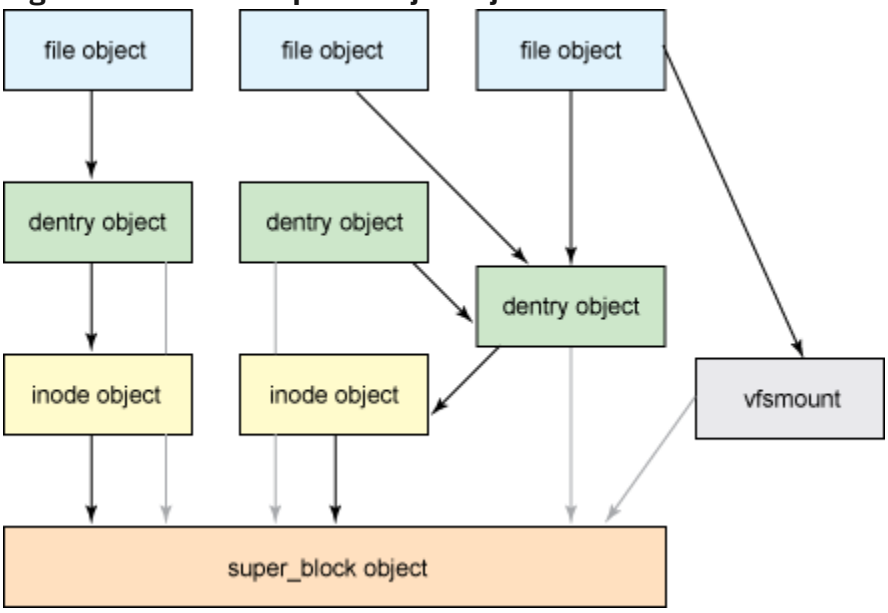


## Object relationships

Now that I've reviewed the various important objects in the VFS layer, let's look at how they relate in a single diagram. Because I've explored the object in a bottom-up fashion so far in this article, I now look at the reverse from the user perspective (see Figure 7).

At the top is the open `file` object, which is referenced by a process's file descriptor list. The `file` object refers to a `dentry` object, which refers to an `inode`. Both the `inode` and `dentry` objects refer to the underlying `super_block` object. Multiple file objects may refer to the same dentry (as in the case of two users sharing the same file). Note also in Figure 7 that a `dentry` object refers to another `dentry` object. In this case, a directory refers to file, which in turn refers to the inode for the particular file.

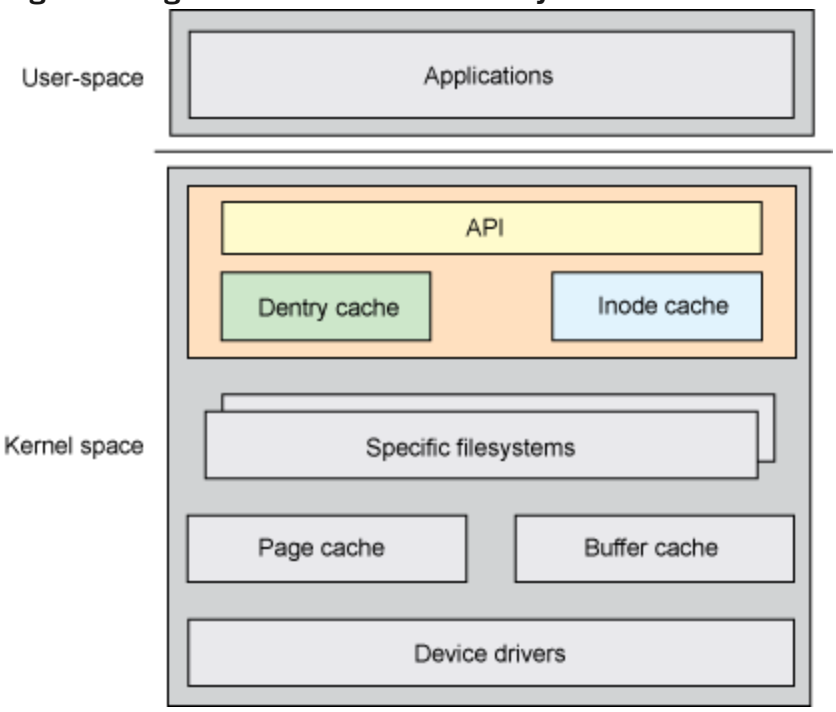
**Figure 7. Relationships of major objects in the VFS**




## The VFS architecture

The internal architecture of the VFS is made up of a dispatching layer that provides the file system abstraction and a number of caches to improve the performance of file system operations. This section explores the internal architecture and how the major objects interact (see Figure 8).

**Figure 8. High-level view of the VFS layer**




The two major objects that are dynamically managed in the VFS include the `dentry` and `inode` objects. These are cached to improve the performance of accesses to the underlying file systems. When a file is opened, the dentry cache is populated with entries representing the directory levels representing the path. An inode for the object is also created representing the file. The dentry cache is built using a hash table and is hashed by the name of the object. Entries for the dentry cache are allocated from the `dentry_cache` slab allocator and use a least-recently-used (LRU) algorithm to prune entries when memory pressure exists. You can find the functions associated with the dentry cache in ./linux/fs/dcache.c (and ./linux/include/linux/dcache.h).

The inode cache is implemented as two lists and a hash table for faster lookup. The first list defines the inodes that are currently in use; the second list defines the inodes that are unused. Those inodes in use are also stored in the hash table. Individual inode cache objects are allocated from the `inode_cache` slab allocator. You can find the functions associated with the inode cache in ./linux/fs/inode.c (and ./linux/include/fs.h). From the implementation today, the dentry cache is the master of the inode cache. When a `dentry` object exists, an `inode` object will also exist in the inode cache. Lookups are performed on the dentry cache, which result in an object in the inode cache.