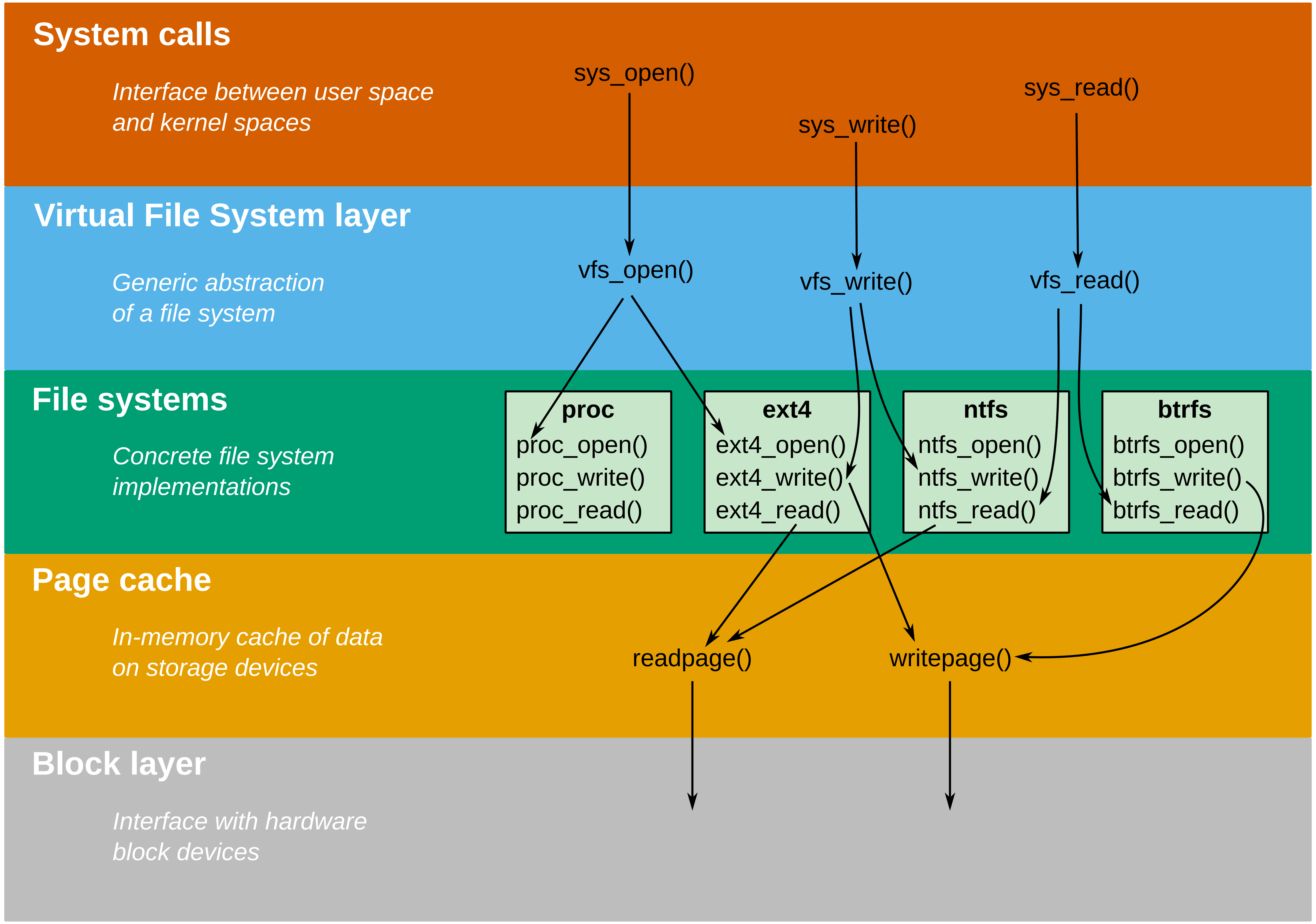
Linux Kernel Programming

Redha Gouicem and Julien Sopena

Lecture 06:

Virtual File System

The Virtual File System Abstraction



Overview of the VFS

The **Virtual File System (VFS)** defines a set of abstractions that are then made concrete by file system implementations.

Objects

- A **file descriptor** represents an instance of an open file
- An **inode** represents a file on the storage device
- The **directory entry** caches the resolution of a file path to its corresponding inode
- A **superblock** represents an instance of a mounted file system

Interfaces

- **File operations** operate on file content (open, read, write, …)
- **Inode operations** operate on file metadata (create, mkdir, unlink, …)
- **Superblock operations** operate on a partition (mount, sync, unmount, …)

File Descriptors

File descriptors represent an instance of an open file.
It contains information about the file on storage as well as the current state of the open file, *e.g.*, *cursor*.
There can be multiple file descriptors for the same file on storage.

A file descriptor is defined as a `struct file` in `include/linux/fs.h`

```c
struct file {
    fmode_t         f_mode;     // mode in which the file is opened (rwxrwxrwx)
    atomic_long_t   f_count;    // number of threads sharing this file descriptor
    loff_t          f_pos;      // position in the file, the "cursor"
    struct inode    *f_inode;   // the inode representing the concrete file
    struct file_operations  *f_op;
    struct address_space    *f_mapping; // mapping in memory for the page cache
    /* ... */
};
```

Inodes

Inodes describe files or directories on the storage device.
Each inode corresponds to one and only one file/directory.
Conversely, each file/directory corresponds to one and only one inode.

An inode is defined as a `struct inode` in `include/linux/fs.h`

```c
struct inode {
    umode_t             i_mode;     // mode of the file on disk
    kuid_t              i_uid;      // user id of the owner of the file
    kgid_t              i_gid;      // group id of the owner of the file
    unsigned long       i_ino;      // inode number

    unsigned int        i_nlink;    // number of links to this file (hard links)
    loff_t              i_size;     // size of the file
    struct timespec64   i_atime;    // date of the last access
    struct timespec64   i_mtime;    // date of the last modification
    struct timespec64   i_ctime;    // date of creation

    struct inode_operations     *i_op;
    struct super_block          *i_sb;      // super block (partition) that contains this inode
    struct address_space        *i_mapping; // mapping in memory for the page cache
};
```

The inode is partially stored on the disk too, in order to preserve information across mounts/reboots, *e.g., file size, timestamps, mode, etc…*

Resolving Paths to Inodes

From a user perspective, a file/directory is identified by its path.
The VFS needs to translate this path into an inode to actually interact with the file.
This resolution operation is costly as it requires costly string operations and walking the directory hierarchy on disk.

To avoid repeating this operation too many times, the VFS builds **directory entries**, or **dentry**, that maintain a relationship between a path and its corresponding inode.

Example: If you open the file located at `/home/lkp/foo/bar`, it will create/query the following *dentries*: `/`, `home`, `lkp`, `foo` and `bar`.

Dentries are defined as `struct dentry` in `include/linux/dcache.h`

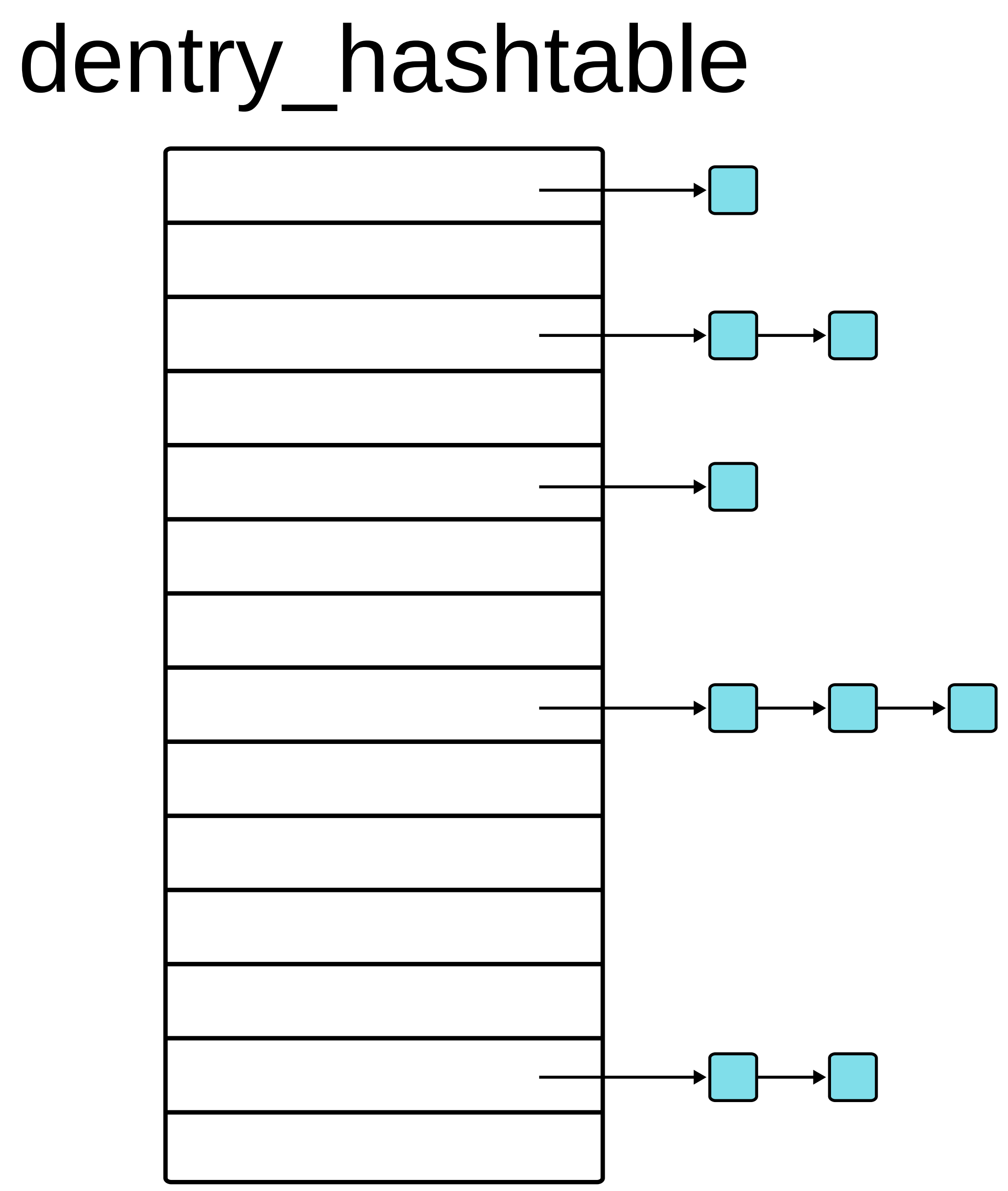
```c
struct dentry {
    struct qstr         d_name;     // file name
    struct inode        *d_inode;   // inode of this path
    struct dentry       *d_parent;  // parent in the fs hierarchy
    struct super_block  *d_sb;      // mounted file system owning this dentry

    struct hlist_bl_node d_hash;    // list for the hash table
};
```

Dentries can be in one of three states:

- **Used:** *dentry* points to a valid inode (`d_inode` points to an inode) and is in the *dentry cache*;
- **Unused:** *dentry* points to a valid inode but not in the *dentry cache*;
- **Negative:** *dentry* does not point to a valid inode (`d_inode` is `NULL`).

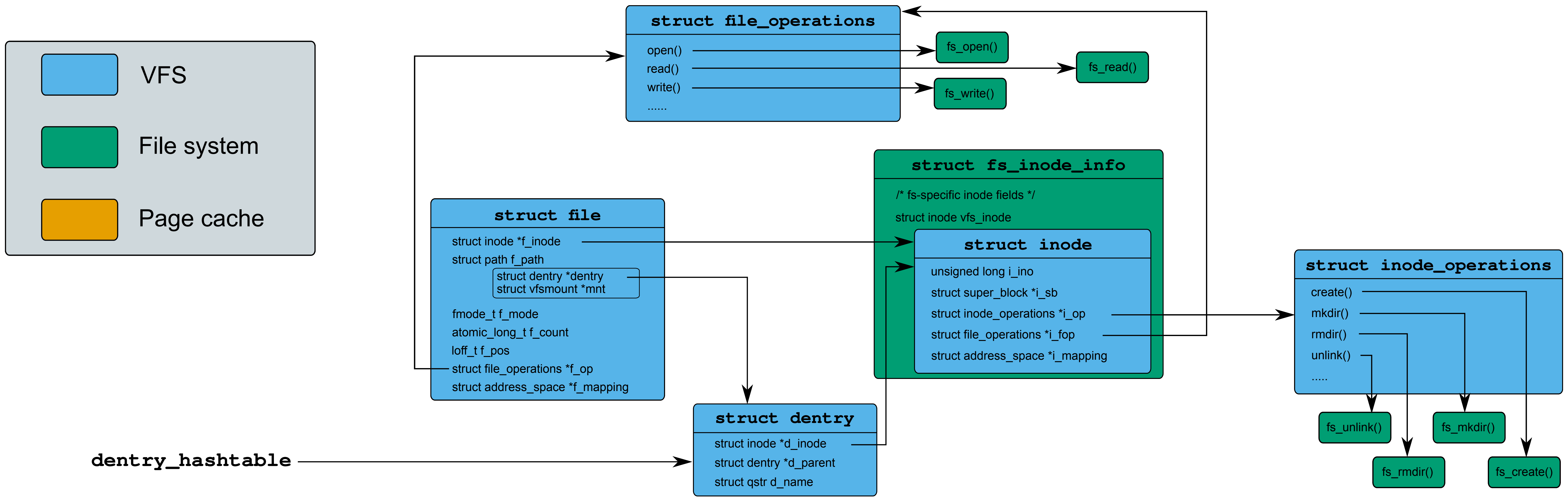
Dentry Cache

The *dentries* are cached into a hash table for fast lookup.
This cache is declared as `static struct hlist_bl_head *dentry_hashtable` in `fs/dcache.c`.



No more details here, you'll have to work with that cache in the exercise.

Relation Between VFS Objects



File System

As shown in the previous figure, implementing a file system means implementing a set of file and inode operations.

File systems may also add their own flavor of inode definition.

Example: The *ext4* file system has this inode definition that extends the VFS inode:

```
struct ext4_inode {
    __le16  i_mode;          /* File mode */
    __le16  i_uid;           /* Low 16 bits of Owner Uid */
    __le32  i_size_lo;       /* Size in bytes */
    __le32  i_atime;         /* Access time */
    __le32  i_ctime;         /* Inode Change time */
    __le32  i_mtime;         /* Modification time */
    __le32  i_dtime;         /* Deletion Time */
    __le16  i_gid;           /* Low 16 bits of Group Id */
    __le16  i_links_count;   /* Links count */
    __le32  i_blocks_lo;     /* Blocks count */
    __le32  i_flags;         /* File flags */
    __le16  i_checksum_hi;   /* crc32c(uuid+inum+inode) BE */
    __le32  i_generation;    /* File version (for NFS) */
    __le32  i_file_acl_lo;   /* File ACL */
};
```

Page Cache

Accesses to storage devices are costly!

Latency for a 1 MB sequential read
(Latency Numbers Everyone Should Know, Google SRE)

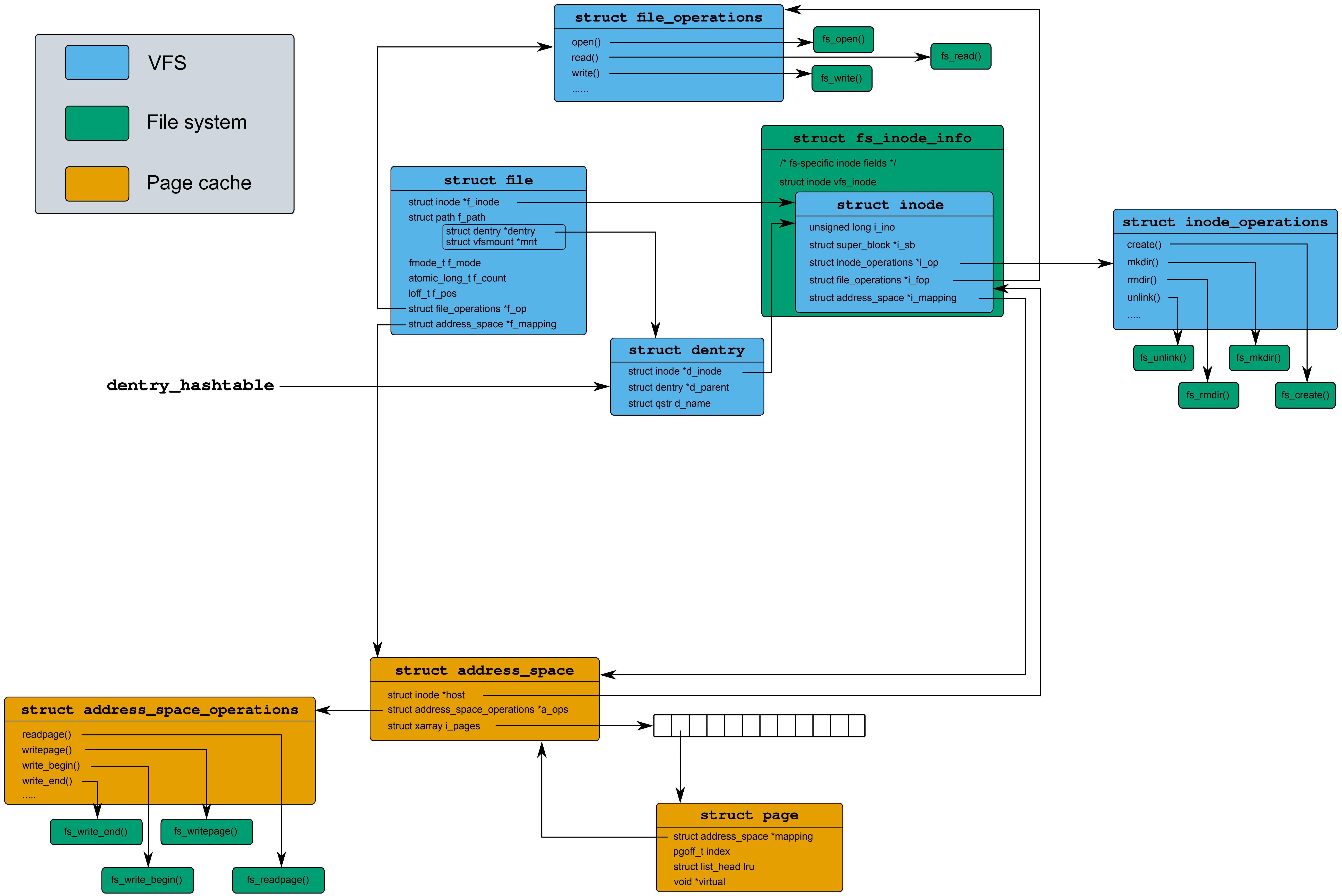
Device	Time (in ns)
Memory	10.000
SSD	1.000.000
HDD	5.000.000

Accessing storage device is **two orders of magnitude** longer than accessing memory!

To alleviate this, Linux has a **page cache** sitting between file systems and storage devices.

- Pages read from block devices are cached in memory for fast access;
- The page cache employs a **write-back** policy: writes are asynchronously propagated to the storage device;
- The page cache does not have a size, it uses all the free memory available. when memory is needed, it evicts pages with an **LRU policy**.

Relation Between VFS Objects and the Page Cache



Super Block

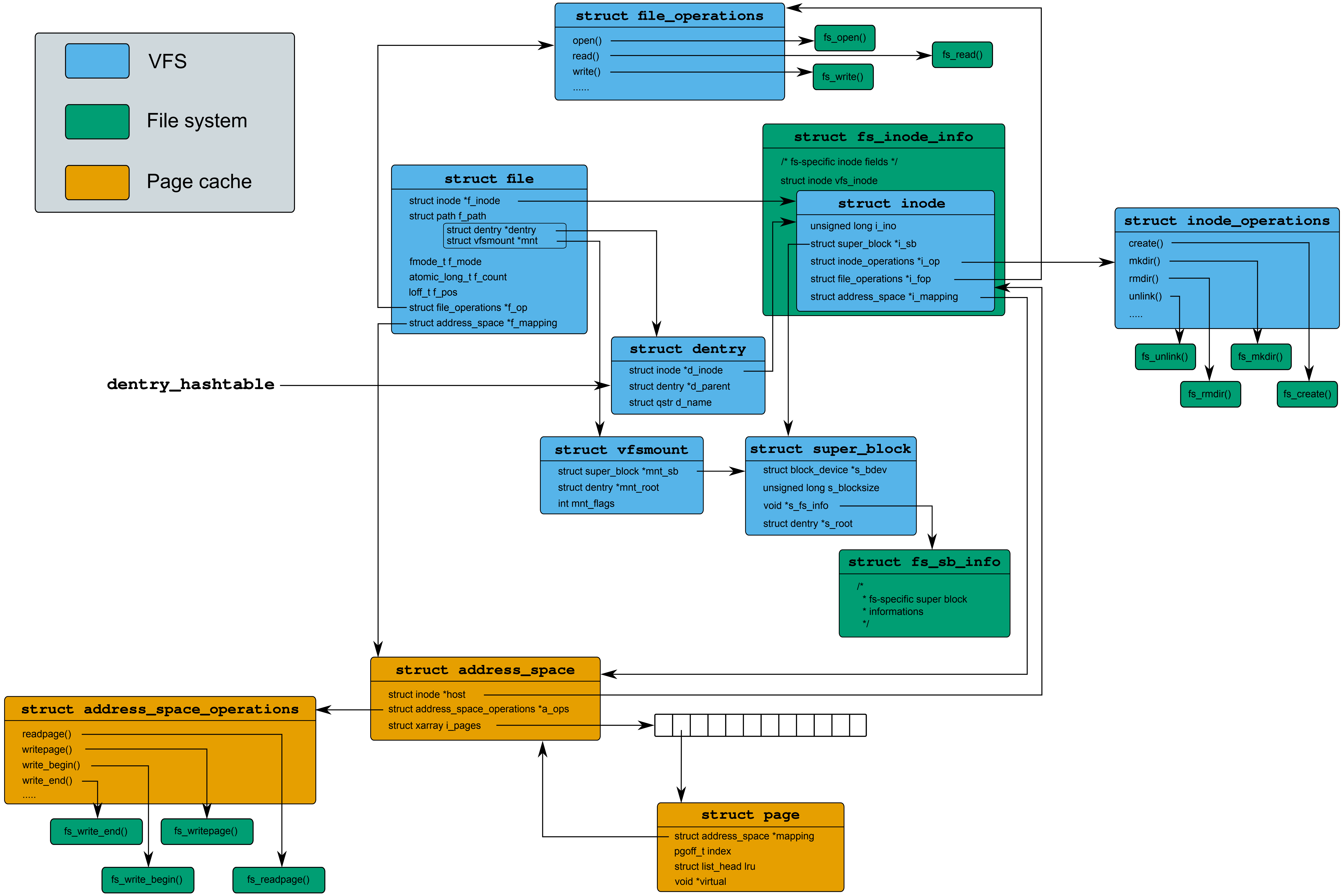
A **super block** describes a file system partition.
When you mount a file system partition, a `struct super_block` object is created and populated with information read from storage.

This structure is defined in `include/linux/fs.h`.

```c
struct super_block {
    struct block_device     *s_bdev;       // the block device containing this partition
    unsigned long           s_blocksize;   // size of a block
    loff_t                  s_maxbytes;    // max file size
    struct file_system_type *s_type;       // file system descriptor
    struct super_operations *s_op;         // fs ops (alloc_inode, sync, umount)
    struct dentry           *s_root;       // dentry of the root of the mount point (the / of this partition)
    unsigned long           s_magic;       // magic number of this file system
    void                    *s_fs_info;    // file system-specific private info
    char                    s_id[32];      // short name
    uuid_t                  s_uuid;        // UUID
    /* ... */
};
```

```c
struct super_operations {
    /* inode handling */
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*free_inode)(struct inode *);
    int (*write_inode) (struct inode *, struct writeback_control *wbc);
    /* partition handling */
    void (*put_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*statfs) (struct dentry *, struct kstatfs *);
    void (*umount_begin) (struct super_block *);
    /* ... */
};
```

Relation Between VFS Objects



Implementing a File System

To implementing a file system, you need to:

1. Design the layout on your physical storage device
   - Design your super block and how inodes and blocks are managed
   - Design your inode content (what information must be stored)
2. Design the operations on your file system
   - File operations (open, read, write, …)
   - Inode operations (create, unlink, mkdir, …)
   - Super block operations (alloc_inode, sync, …)
   - Page cache operations if you want to use it

For **1.**, this mostly means implementing the `mkfs` utility for your new file system.
For **2.**, this is your kernel implementation as a module.

File system in User SpacE

Another way of implementing a file system in Linux is through the **File system in User SpacE (FUSE)** API.

With FUSE, you can implement everything in user space and register your FUSE file system with the kernel.

The VFS will then redirect system calls targeting your file system to your code in user space.