# Virtual File System

W4118 Operating Systems I

columbia-os.github.io

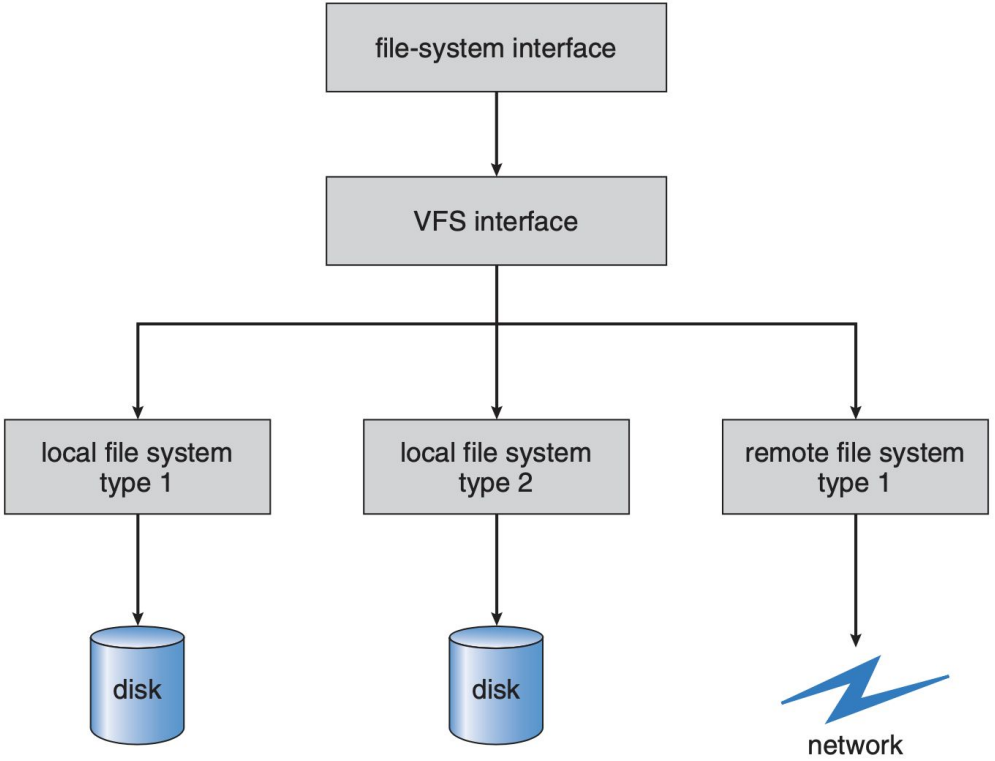
Credits to Jae and Hans

# Virtual File System (VFS)

Many file systems and device types can coexist on the same system.

Different levels of the stack have different interfaces:

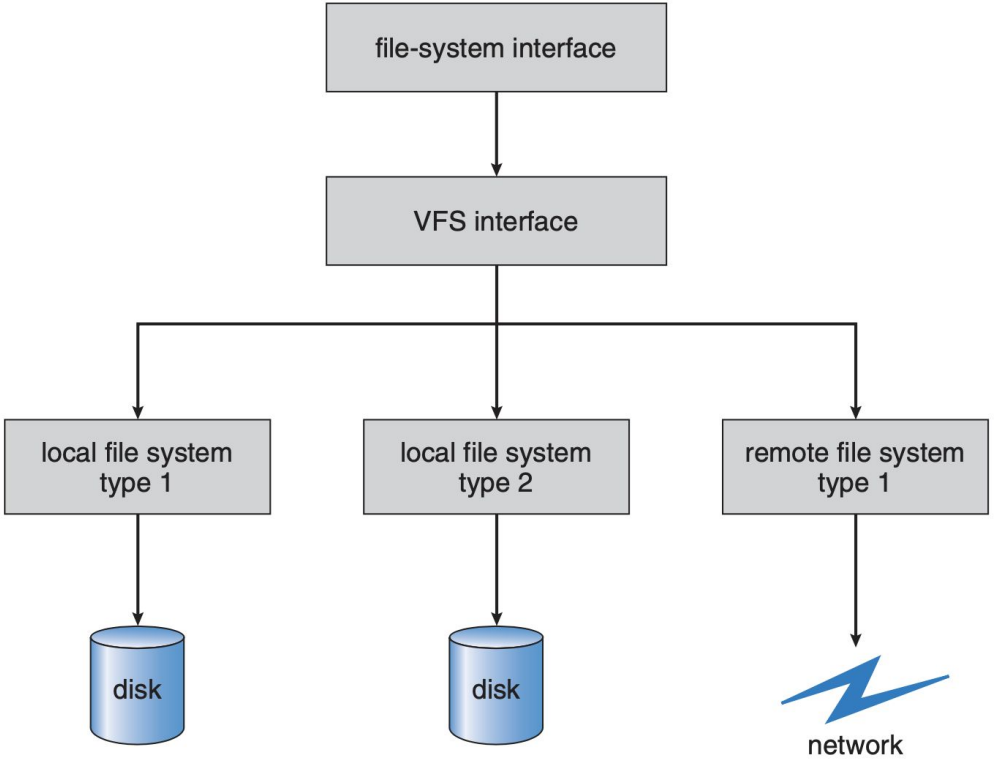
- File System Interface
- VFS Interface
- Storage Level



# Virtual File System (VFS)

**File System Interface:**

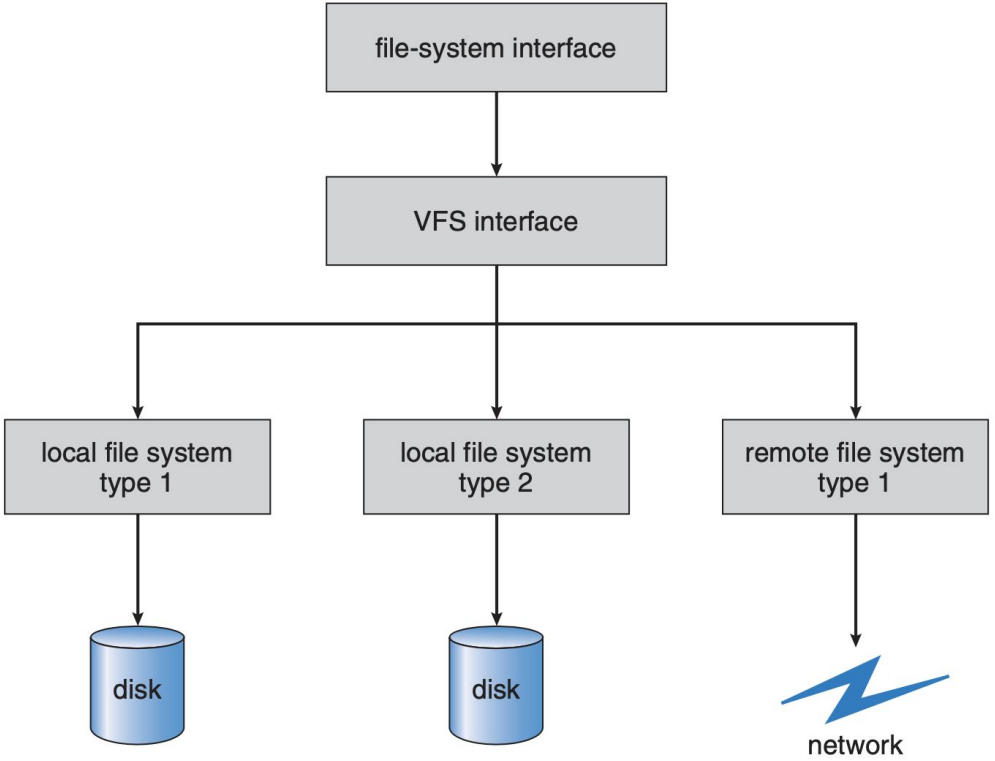
- API for userspace programs to interact with files
- `open()`, `close()`, `read()`, etc.
- Uses file descriptor to refer to a file
- Does not expose implementation details to the users



# Virtual File System (VFS)

**Storage Level:**

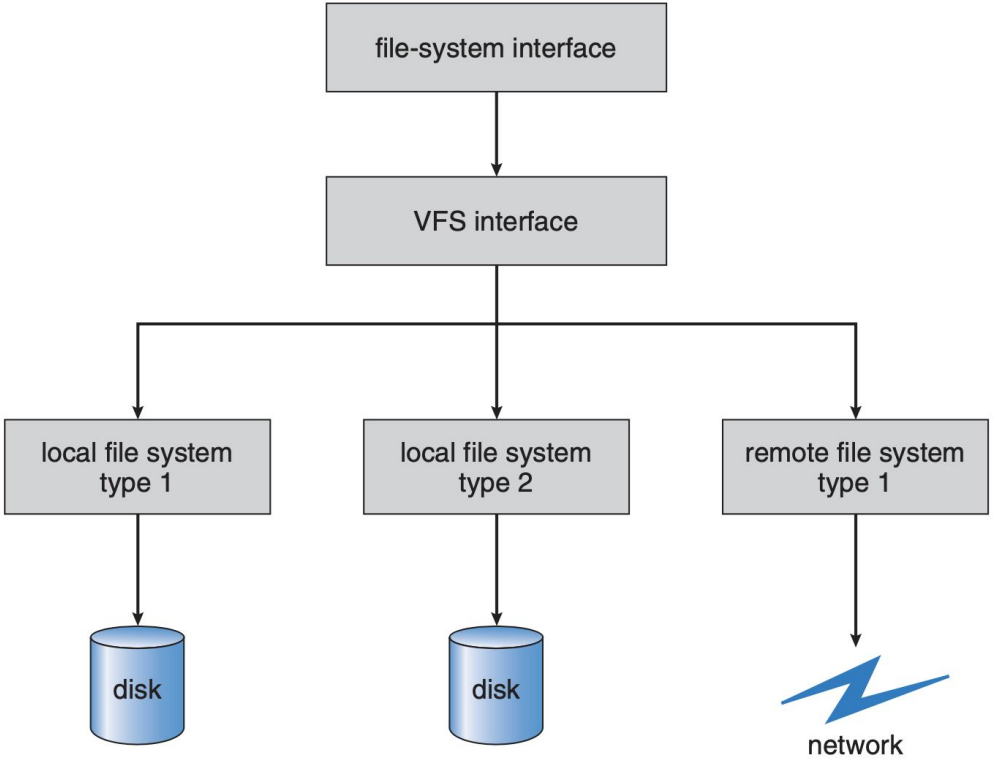
- Determines how data are stored in the disk
- Userspace programs are not burdened with these details
- Can even store data remotely, over the network



# Virtual File System (VFS)

**VFS Interface:**

- Abstraction layer that can support multiple file systems
- Specifies an interface (similar to `struct sched_class`) that a given FS implements to hook into the kernel
- VFS dispatches operations to a specific FS using the interface, e.g.., `dir->inode_op->mkdir()`



# VFS Data Structures

`struct file`: Represents an instance of an open file

- Pointed to by per-process fdtable entry, allows for open file sharing by copying the pointer
- Stores flags, current position, etc.
- Refers to dentry via `struct path f_path` (which refers to the inode)

VFS interface: `struct file_operations *f_op`

- read, write, seek, etc.

# VFS Data Structures

`struct dentry`: Basically a “hard link”: contains name of link and inode number

Break up an absolute path into dentries, one per component, e.g., `/home/kkaffes/foo` has `/`, `home` , `kkaffes`, `foo`

Path resolution is expensive to open `/home/kkaffes/foo` you need to:

- consult the dentry for `/` to find the root inode
- find the root data block, iterate through it to find dentry for `home`
- consult the dentry for `home` to find the inode
- find the corresponding data block, iterate through it to find dentry for `kkaffes`
- consult the dentry `kkaffes` to find the inode
- find the corresponding data block, iterate through to find the dentry for `foo`
- consult the dentry `foo` to find the inode
- find the corresponding data block, and finally read the file contents!

VFS interface: `const struct dentry_operations *d_op`

- manage dentries through dentry cache (create/remove/hash/etc), more on this later

# VFS Data Structures

`struct inode`: Unique descriptor of a file or directory

`i_ino`: inode # unique per mounted file system

Can refer to fs-specific data via `i_private` (will be used for HW8)

VFS interface: `const struct inode_operations *i_op`

- read, write, seek, etc.

# VFS Data Structures

`struct super_block`: Descriptor of a mounted filesystem.

VFS interface: `const struct super_operations *s_op`

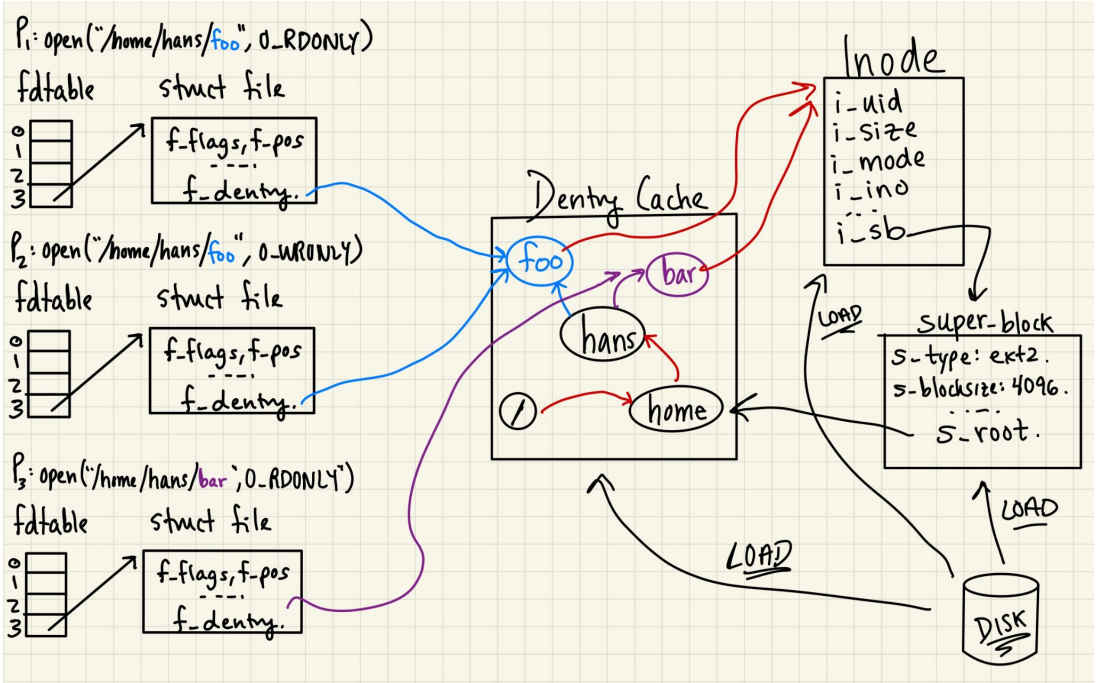
- inode management, journaling, syncing metadata

# Dentry Cache

Linux kernel makes path resolution efficient by employing a dentry cache (dcache)

1. Mount an instance of ext2 at `/home`

`s_root` field of `super_block` refers to the root dentry of the mount



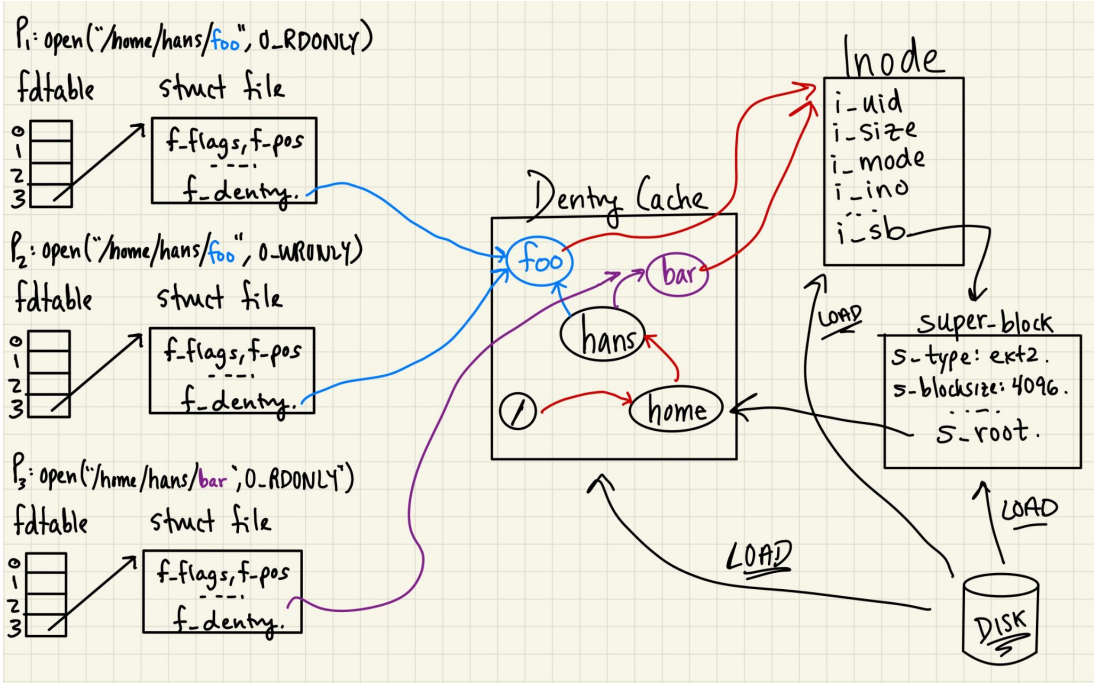
# Dentry Cache

Linux kernel makes path resolution efficient by employing a dentry cache (dcache)

2. P1 opens `/home/hans/foo` for reading

Need to read several inodes/dentries from disk

Along the way, cache them in the dcache



# Dentry Cache

Linux kernel makes path resolution efficient by employing a dentry cache (dcache)

3. P3 opens `/home/hans/bar`

Different file than P1 and P2

`/home/hans/` path resolution cached in dcache

Need to read in `hans/` directory data block to find dentry for `bar`

…only to find it refers to the same inode as `foo`

`bar` and `foo` are hard links to the same inode!