



# Virtual File System

---

Don Porter  
CSE 506

# History



- ✦ Early OSes provided a single file system
  - ✦ In general, system was pretty tailored to target hardware
- ✦ In the early 80s, people became interested in supporting more than one file system type on a single system
  - ✦ Any guesses why?
  - ✦ Networked file systems – sharing parts of a file system transparently across a network of workstations

# Modern VFS



- ✦ Dozens of supported file systems
  - ✦ Allows experimentation with new features and designs transparent to applications
  - ✦ Interoperability with removable media and other OSES
- ✦ Independent layer from backing storage
  - ✦ Pseudo FSes used for configuration (/proc, /devtmps...) only backed by kernel data structures
- ✦ And, of course, networked file system support

# User's perspective



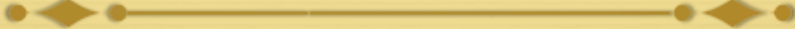
- ✦ Single programming interface
  - ✦ (POSIX file system calls – open, read, write, etc.)
- ✦ Single file system tree
  - ✦ A remote file system with home directories can be transparently mounted at /home
- ✦ Alternative: Custom library for each file system
  - ✦ Much more trouble for the programmer

# What the VFS does



- ✦ The VFS is a substantial piece of code, not just an API wrapper
- ✦ Caches file system metadata (e.g., file names, attributes)
  - ✦ Coordinates data caching with the page cache
- ✦ Enforces a common access control model
- ✦ Implements complex, common routines, such as path lookup, file opening, and file handle management

# FS Developer's Perspective



- ✦ FS developer responsible for implementing a set of standard objects/functions, which are called by the VFS
  - ✦ Primarily populating in-memory objects from stable storage, and writing them back
- ✦ Can use block device interfaces to schedule disk I/O
  - ✦ And page cache functions
  - ✦ And some VFS helpers
- ✦ Analogous to implementing Java abstract classes

# High-level FS dev. tasks



- ✦ Translate between volatile VFS objects and backing storage (whether device, remote system, or other/none)
  - ✦ Potentially includes requesting I/O
- ✦ Read and write file pages

# Opportunities



- ✦ VFS doesn't prescribe all aspects of FS design
  - ✦ More of a lowest common denominator
- ✦ Opportunities: (to name a few)
  - ✦ More optimal media usage/scheduling
  - ✦ Varying on-disk consistency guarantees
  - ✦ Features (e.g., encryption, virus scanning, snapshotting)



# Core VFS abstractions



- ✦ super block – FS-global data
  - ✦ Early/many file systems put this as first block of partition
- ✦ inode (index node) – metadata for one file
- ✦ dentry (directory entry) – file name to inode mapping
- ✦ file – a file handle – refers to a dentry and a cursor in the file (offset)

# Super blocks



- ✦ SB + inodes are *extended* by FS developer
- ✦ Stores all FS-global data
  - ✦ Opaque pointer (`s_fs_info`) for fs-specific data
- ✦ Includes many hooks for tasks such as creating or destroying inodes
- ✦ Dirty flag for when it needs to be synced with disk
- ✦ Kernel keeps a circular list of all of these

# Inode



- ✦ The second object extended by the FS
  - ✦ Huge – more fields than we can talk about
- ✦ Tracks:
  - ✦ File attributes: permissions, size, modification time, etc.
  - ✦ File contents:
    - ✦ Address space for contents cached in memory
    - ✦ Low-level file system stores block locations on disk
  - ✦ Flags, including dirty inode and dirty data

# Inode history



- ✦ Name goes back to file systems that stored file metadata at fixed intervals on the disk
  - ✦ If you knew the file's index number, you could find its metadata on disk
- ✦ Hence, the name 'index node'
- ✦ Original VFS design called them 'vnode' for virtual node (perhaps more appropriately)
- ✦ Linux uses the name inode

# Embedded inodes



- ✦ Many file systems embed the VFS inode in a larger, FS-specific inode, e.g.,:

```
struct donfs_inode {  
    int ondisk_blocks[];  
  
    /* other stuff */  
  
    struct inode vfs_inode;  
  
}
```

- ✦ Why? Finding the low-level data associated with an inode just requires simple (compiler-generated) math

# Linking



- ✦ An inode uniquely identifies a file for its lifespan
  - ✦ Does not change when renamed
- ✦ Model: Inode tracks “links” or references
  - ✦ Created by open file handles and file names in a directory that point to the inode
  - ✦ Ex: renaming the file temporarily increases link count and then lower it again
- ✦ When link count is zero, inode (and contents) deleted
  - ✦ There is no ‘delete’ system call, only ‘unlink’

# Linking, cont.



- ✦ “Hard” link (link system call/ln utility): creates a second name for the same file; modifications to either name changes **contents**.
  - ✦ This is not a copy
- ✦ Common trick for temporary files:
  - ✦ create (1 link)
  - ✦ open (2 links)
  - ✦ unlink (1 link)
  - ✦ File gets cleaned up when program dies
    - ✦ (kernel removes last link)

# Inode 'stats'



- ✦ The 'stat' word encodes both permissions and type
- ✦ High bits encode the type: regular file, directory, pipe, char device, socket, block device, etc.
  - ✦ Unix: Everything's a file! VFS involved even with sockets!
- ✦ Lower bits encode permissions:
  - ✦ 3 bits for each of User, Group, Other + 3 special bits
  - ✦ Bits: 2 = read, 1 = write, 0 = execute
  - ✦ Ex: 750 – User RWX, Group RX, Other nothing



# Special bits



- ✦ For directories, 'Execute' means search
  - ✦ X-only permissions means I can find readable subdirectories or files, but can't enumerate the contents
  - ✦ Useful for sharing files in your home directory, without sharing your home directory contents
    - ✦ Lots of information in meta-data!
- ✦ Setuid bit
  - ✦ Mostly relevant for executables: Allows anyone who runs this program to execute with owner's uid
  - ✦ Crude form of permission delegation

# More special bits



## ✦ Group inheritance bit

- ✦ In general, when I create a file, it is owned by my default group
- ✦ If I create in a 'g+s' directory, the directory group owns the file
- ✦ Useful for things like shared git repositories

## ✦ Sticky bit

- ✦ Restricts deletion of files

# File objects



- ✦ Represent an open file; point to a dentry and cursor
  - ✦ Each process has a table of pointers to them
  - ✦ The int fd returned by open is an offset into this table
- ✦ These are VFS-only abstractions; the FS doesn't need to track which process has a reference to a file
- ✦ Files have a reference count. Why?
  - ✦ Fork also copies the file handles
  - ✦ If your child reads from the handle, it advances your (shared) cursor

# File handle games



- ✦ dup, dup2 – Copy a file handle
  - ✦ Just creates 2 table entries for same file struct, increments the reference count
- ✦ seek – adjust the cursor position
  - ✦ Obviously a throw-back to when files were on tapes
- ✦ fcntl – Like ioctl (misc operations), but for files
- ✦ CLOSE\_ON\_EXEC – a bit that prevents file inheritance if a new binary is exec'ed (set by open or fcntl)

# Dentries



- ✦ These store:
  - ✦ A file name
  - ✦ A link to an inode
  - ✦ A parent pointer (null for root of file system)
- ✦ Ex: `/home/porter/vfs.pptx` would have 4 dentries:
  - ✦ `/`, `home`, `porter`, & `vfs.pptx`
  - ✦ Parent pointer distinguishes `/home/porter` from `/tmp/porter`
- ✦ These are also VFS-only abstractions
  - ✦ Although inode hooks on directories can populate them

# Why dentries?



- ✦ A simple directory model might just treat it as a file listing  $\langle \text{name, inode} \rangle$  tuples
- ✦ Why not just use the page cache for this?
  - ✦ FS directory tree traversal very common; optimize with special data structures
- ✦ The dentry cache is a complex data structure we will discuss in much more detail later

# Summary of abstractions



- ✦ Super blocks – FS- global data
- ✦ Inodes – stores a given file
- ✦ File (handle) – Essentially a  $\langle$ dentry, offset $\rangle$  tuple
- ✦ Dentry – Essentially a  $\langle$ name, parent dentry, inode $\rangle$  tuple

# More on the user's perspective

- ✦ Let's wrap today by discussing some common FS system calls in more detail
- ✦ Let's play it as a trivia game
  - ✦ What call would you use to...

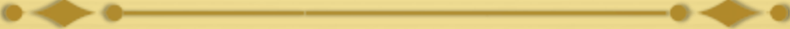


# Create a file?



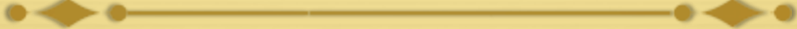
- ✦ `creat`
- ✦ More commonly, open with the `O_CREAT` flag
  - ✦ Avoid race conditions between creation and open
- ✦ What does `O_EXCL` do?
  - ✦ Fails if the file already exists

# Create a directory?



- ✦ mkdir
- ✦ But I thought everything in Unix was a file!?!
  - ✦ This means that *sometimes* you can read/write an existing handle, even if you don't know what is behind it.
  - ✦ Even this doesn't work for directories

# Remove a directory



✦ `rmdir`

# Remove a file



✦ unlink

# Read a file?



- ✦ `read()`
- ✦ How do you change cursor position?
  - ✦ `lseek` (or `pread`)

# Read a directory?



✦ readdir or getdents

# Shorten a file



- ✦ truncate/ftruncate
- ✦ Can also be used to create a file full of zeros of arbitrary length
  - ✦ Often blocks on disk are demand-allocated (laziness rules!)

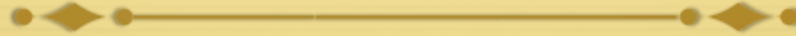
# What is a symbolic link?




- ✦ A special file type that stores the name of another file
- ✦ How different from a hard link?
  - ✦ Doesn't raise the link count of the file
  - ✦ Can be “broken,” or point to a missing file
- ✦ How created?
  - ✦ `symlink` system call or `ln -s` command



Let's step it up a bit



# How does an editor save a file?



- ✦ Hint: we don't want the program to crash with a half-written file
- ✦ Create a backup (using open)
- ✦ Write the full backup (using read old/ write new)
- ✦ Close both
- ✦ Do a rename(old, new) to atomically replace

# How does 'ls' work?



- ✦ `dh = open(dir)`
- ✦ for each file (`while readdir(dh)`)
  - ✦ Print file name
- ✦ `close(dh)`

# What about that cool colored text?

- ✦ `dh = opendir(dir)`
- ✦ for each file (`while readdir(dh)`)
  - ✦ `stat(file, &stat_buf)`
  - ✦ if (`stat & execute bit`) `color == green`
  - ✦ else if ...
  - ✦ Print file name
  - ✦ Reset color
- ✦ `closedir(dh)`

# Summary



- ✦ Today's goal: VFS overview from many perspectives
  - ✦ User (application programmer)
  - ✦ FS implementer
    - ✦ Used many page cache and disk I/O tools we've seen
- ✦ Key VFS objects
- ✦ Important to be able to pick POSIX fs system calls from a line up
  - ✦ Homework: think about pseudocode from any simple command-line file system utilities you type this weekend