

CSE 506: Operating Systems

Virtual File System

History

- Early OSes provided a single file system
 - In general, system was tailored to target hardware
- In the early 80s, desire for more than one file system
 - Any guesses why?
 - Networked file systems
 - Sharing parts of a file system across a network of workstations

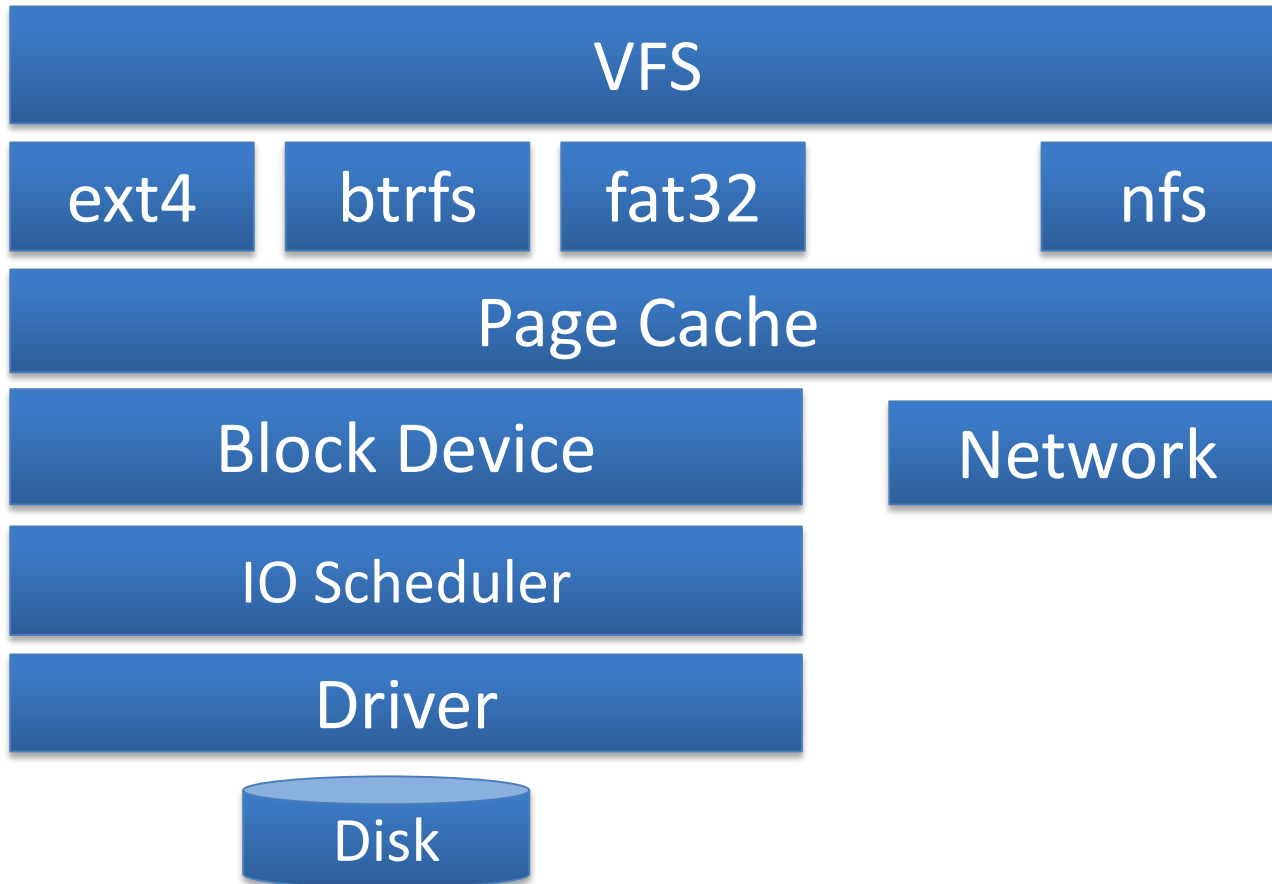
Modern VFS

- Dozens of supported file systems
 - Allows new features and designs transparent to apps
 - Interoperability with removable media and other OSes
- Independent layer from backing storage
 - In-memory file systems (*ramdisks*)
 - Pseudo file systems used for configuration
 - (/proc, /dev, ...) only backed by kernel data structures
- And, of course, networked file system support

More detailed diagram

User

Kernel



User's perspective

- Single programming interface
 - (POSIX file system calls – open, read, write, etc.)
- Single file system tree
 - Remote FS can be transparently mounted (e.g., 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., names, attributes)
 - Coordinates data caching with the page cache
- Enforces a common access control model
- Implements complex, common routines
 - path lookup
 - opening files
 - file handle management

FS Developer's Perspective

- FS developer responsible for...
 - Implementing standard objects/functions called by the VFS
 - Primarily populating in-memory objects
 - Typically from stable storage
 - Sometimes 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 VFS objects and backing storage (whether device, remote system, or other/none)
 - Potentially includes requesting I/O
- Read and write file pages
- VFS doesn't prescribe all aspects of FS design
 - More of a lowest common denominator

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) – name to inode mapping
- file – *file descriptor* – dentry and cursor (file offset)

Super blocks

- SB + inodes are extended by file system developer
- Stores all FS-global data
 - Opaque pointer (`s_fs_info`) for FS-specific data
- Includes many hooks
 - 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
 - When there are multiple FSes (in today's systems: always)

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

- Original file systems stored files at fixed intervals
 - If you knew the file's index number
 - you could find its metadata on disk
 - Think of a portion of the disk as a big array of metadata
- Hence, the name 'index node'
- Original VFS design called them 'vnode'
 - virtual node (perhaps more appropriately)
 - Linux uses the name inode

Embedded inodes

- Many FSes embed VFS inode in FS-specific inode

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

- Why?
 - Finding the low-level from inode is simple
 - Compiler translates references to simple math

Linking

- An inode uniquely identifies a file for its lifespan
 - Does not change when renamed
- Model: inode tracks “links” or references on disk
 - Count “1” for every reference on disk
 - Created by file names in a directory that point to the inode
 - What happens when a file is renamed?
 - renaming file temporarily increases link count and then lowers it (at least in some implementations)
- 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 new name for the same inode
 - Opening either name opens the same file
 - This is not a copy
- Open files create an in-memory reference to a file
 - If an open file is unlinked, the directory entry is deleted
 - inode and data retained until all in-memory references are deleted
 - Famous feature: `rm` on large open file when out of quota
 - Still out of quota

Common trick for temporary files

- How to clean up temp file when program crashes?
 - create (1 link)
 - open (1 link, 1 ref)
 - unlink (0 link)
 - File gets cleaned up when program dies
 - Kernel removes last reference on exit
 - Happens regardless if exit is clean or not
 - Except if the kernel crashes / power is lost
 - Need something like fsck to “clean up” inodes without dentries
 - » Dropped into `lost+found` directory

inode 'stats'

- The 'stat' word encodes both permissions and type
- High bits encode the type:
 - regular file, directory, pipe, device, socket, 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
 - How about the “sticky” bit? “suid” bit?
 - `chmod` has more pleasant syntax `[ugo][+-][rwx]`

Special bits

- For directories, ‘Execute’ means search
 - X-only allows to find readable subdirectories or files
 - Can’t enumerate the contents
 - Useful for sharing files in your home directory
 - Without sharing your home directory contents
- Setuid bit
 - Program executes with owner’s UID
 - Crude form of permission delegation

More special bits

- Group inheritance bit
 - When I create a file, it is owned by my default group
 - When I create in a 'g+s' directory, directory group owns file
 - Useful for things like shared git repositories
- Sticky bit
 - Prevents non-owners from deleting or renaming files
 - Ex: CSE506 submission directory

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
- VFS-only abstractions
 - FS doesn't track which process has a reference to a file
- Files have a reference count. Why?
 - Fork also copies the file handles
 - Particularly important for `stdin`, `stdout`, `stderr`
 - If child reads from the handle, it advances (shared) cursor
 - These days, hard to tell if this is a “feature” or a “bug”

File handle games

- `dup ()`, `dup2 ()` – Copy a file handle
 - Creates 2 table entries for same file struct
 - Increments the reference count
- `seek ()` – adjust the cursor position
 - Back when files were on tape...
- `fcntl ()` – Set flags on file (`ioctl ()` for inodes)
 - `CLOSE_ON_EXEC` – bit prevents inheritance on `exec ()`
 - Set by `open ()` or `fcntl ()`

Dentries

- These store:
 - A file name
 - A link to an inode
 - A parent ptr (null for root of file system)
- Ex: `/home/myuser/vfs.pptx` may have 4 dentries:
 - `/`, `home`, `myuser`, and `vfs.pptx`
 - Parent ptr distinguishes `/home/myuser` from `/tmp/myuser`
- Also VFS-only abstraction
 - Although inode hooks on directories can populate them

Why dentries?

- Simple directory model can treat it as a file
 - Contents are a list of <name, inode> tuples
- Why not just use the page cache?
 - FS directory tree traversal very common
 - Optimize with special data structures
 - No need to re-parse and traverse on-disk layout format
- The dentry cache is a complex data structure
 - We will discuss in more detail later

Symbolic Links

- Special file type that stores a string
 - String usually assumed to be a filename
 - Created with `symlink()` system call
- How different from a hard link
 - Completely
 - Doesn't raise the link count of the file
 - Can be “broken,” or point to a missing file (just a string)
- Sometimes abused to store short strings

```
[myself@newcastle ~/tmp]% ln -s "silly example" mydata
```

```
[myself@newcastle ~/tmp]% ls -l
```

```
lrwxrwxrwx 1 myself mygroup 23 Oct 24 02:42 mydata -> silly example
```


How does a text editor save a file?

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

Quick review: dentry

- What purpose does a dentry serve?
 - Maps a path name to an inode
 - More in 2 slides on how to find a dentry
- dentries are cached in memory
 - Only “recently” accessed parts of dir are in memory
 - Others may need to be read from disk
 - dentries can be freed to reclaim memory (like pages)

dentry Caching

- 3 cases for a dentry:
 - In memory (exists)
 - Not in memory (doesn't exist)
 - Not in memory (on disk/evicted for space or never used)
- How to distinguish last 2 cases?
 - Case 2 can generate a lot of needless disk traffic
 - “Negative” dentry – Dentry with a NULL inode pointer

dentry Tracking

- dentries are stored in four data structures:
 - A hash table (for quick lookup)
 - A LRU list (for freeing cache space wisely)
 - A child list of subdirectories (mainly for freeing)
 - An alias list (to do reverse mapping of inode -> dentries)
 - Recall that many names can point to one inode

Summary of `open ()` Implementation

- Key kernel tasks:
 - Map a human-readable path name to an inode
 - Check access permissions, from / to the file
 - Possibly create or truncate the file (`O_CREAT`, `O_TRUNC`)
 - Create a file struct
 - Allocate a descriptor
 - Point descriptor at file struct
 - Return descriptor

open () arguments

```
int open(char *path, int flags, int mode);
```

- Path: file name
- Flags: many (see manual page)
- Mode: If creating file, what perms? (e.g., 0755)
- Return value: File handle index (≥ 0 on success)
 - Or (0 -errno) on failure

Absolute vs. Relative Paths

- Each process has root and working directories
 - Stored in `pcb->root` (or `pcb->fs`) and `pcb->cwd`
 - These are dentry pointers (not strings)
- Why store a current root directory?
 - Some programs are “`chroot` jailed”
 - Should not be able to access anything outside of the jail

More on paths

- An absolute path starts with the ‘/’ character
 - E.g., /home/myself/foo.txt, /lib/libc.so
- A relative path starts with anything else:
 - E.g., vfs.pptx, ../../etc/apache2.conf
- First character dictates where to start searching
 - Only two options: root (absolute) or cwd (relative)

Search

- Execute in a loop looking for next piece
 - Treat ‘/’ character as component delimiter
 - Each iteration looks up part of the path
- Ex: ‘/home/myself/foo’ would look up...
 - ‘home’, ‘myself’, then ‘foo’, starting at ‘/’

Iteration 1

- For searched dentry (/), dereference the inode
- Check access permission (mode is stored in inode)
 - Use `permission()` function pointer on inode
 - Can be overridden by a file system
- If ok, look at next path component (/home)

Detail (2)

- Some special cases:
 - If next component is a '.', just skip it
 - If next component is a '..', move up to parent
 - Catch special case where current dentry is root
 - Treat this as a no-op
- If not a '.' or '..':
 - Compute a hash value to find bucket in `d_hash` table
 - Hash of path from root (e.g., '/home/foo', not 'foo')
 - Search the `d_hash` bucket at this hash value

Detail (3)

- If no dentry in the hash bucket
 - Call `lookup()` method on parent inode (provided by FS)
 - Probably will read the dentry from disk
 - Or the network, or kernel data structures, ...
- If dentry found, check if it is a symlink
 - If so, call `inode->readlink()` (also provided by FS)
 - Get the path stored in the symlink
 - Then continue next iteration
 - First char decides to start at root or at cwd again
- If not a symlink, check if it is a directory
 - If not a directory and not last element, we have a bad path

Iteration 2

- We have dentry/inode for /home, now finding myself
- Check permission in /home
- Hash /home/myself, find dentry
- Confirm not '.', '..', or a symlink
- Confirm is a directory
- Repeat with dentry/inode for /home/myself
 - Search for foo

Symlink Loops

- What if /home/myself/foo is a symlink to 'foo'?
 - Kernel gets in an infinite loop
- Can be more subtle:
 - foo -> bar
 - bar -> baz
 - baz -> foo

Preventing infinite symlink recursion

- More heuristics
- If more than 40 symlinks resolved
 - quit with `-ELOOP`
- If more than 6 symlinks in a row without non-symlink
 - quit with `-ELOOP`
- Can prevent execution of legitimate 41 symlink path
 - Better than an infinite loop

Back to `open()`

- Key tasks:
 - Map a human-readable path name to an inode
 - Check access permissions, from / to the file
 - Possibly create or truncate the file (`O_CREAT`, `O_TRUNC`)
 - Create a file descriptor
- We've seen how first few steps are done

Creation

- Handled as part of search; last item is special
 - Usually, if an item isn't found, search returns an error
- If last item (foo) exists and `O_EXCL` flag set, fail
 - If `O_EXCL` is not set, return existing dentry
- If it does not exist, call FS create method
 - Make a new inode and dentry
 - Then open it
- Why is Create a part of Open?
 - Avoid races in “if (!exist()) create(); open();”

File descriptors

- Descriptors index into per-process array of struct file
- struct file stores
 - dentry pointer
 - cursor into the file
 - permissions (cache of inode's value)
 - reference count
- `open ()` marks a free table entry as 'in use'
 - If full, create a new table 2x the size and copies old one
 - Allocate a new file struct and put a pointer in table

Once `open()`, can `read()`

```
int read(int fd, void *buf, size_t bytes);
```

- `fd`: File descriptor index
- `buf`: Buffer kernel writes the read data into
- `bytes`: Number of bytes requested
- Returns: bytes read (if ≥ 0), or $-\text{errno}$

Summary of `read()` Implementation

- Translate `int fd` to a struct file (if valid)
 - Check cached permissions in the file
 - Increase reference count
 - FS `read()` routine might context switch away temporarily
- Do `read()` routine associated with file (FS-specific)
 - Probe the page cache for data
 - Access storage if needed
 - Can even do both: `read()` is allowed to return less than requested
- Drop refcount, return bytes read

Copying data to user

- Kernel needs to be sure that buffer is a valid address
 - Validate that `buf` is a valid address
 - And that `buf size` \geq `bytes` requested
- How to do it?
 - Can walk appropriate page table entries
- What could go wrong?
 - Concurrent `munmap` from another thread
 - Page might be lazy allocated by kernel

Trick for Validating User Buffers

- What if we don't do all of this validation?
 - Looks like kernel had a page fault
 - Usually *really bad*
- Idea: set kernel flag `in_copy_to_user`
 - If a page fault happens for a user address, don't panic
 - Just handle demand faults
 - If the page is really bad
 - Set indicator that write loop should be aborted
 - Indicator can be left in a register
 - » Tricky context switch code while exiting the page fault

Zero-Copy

- How many memory copies needed for a large read?
 - One in page cache
 - One in user space
- What if we then write this to network?
 - One more for NIC driver
- Avoided extra copies if read/write is page-granularity
 - Steal physical page containing buffer from process
 - Replace it with physical page from page cache
 - Mark it as COW in process, just in case
 - Avoids unnecessary copies in common case