Virtual File System (VFS)

Nima Honarmand

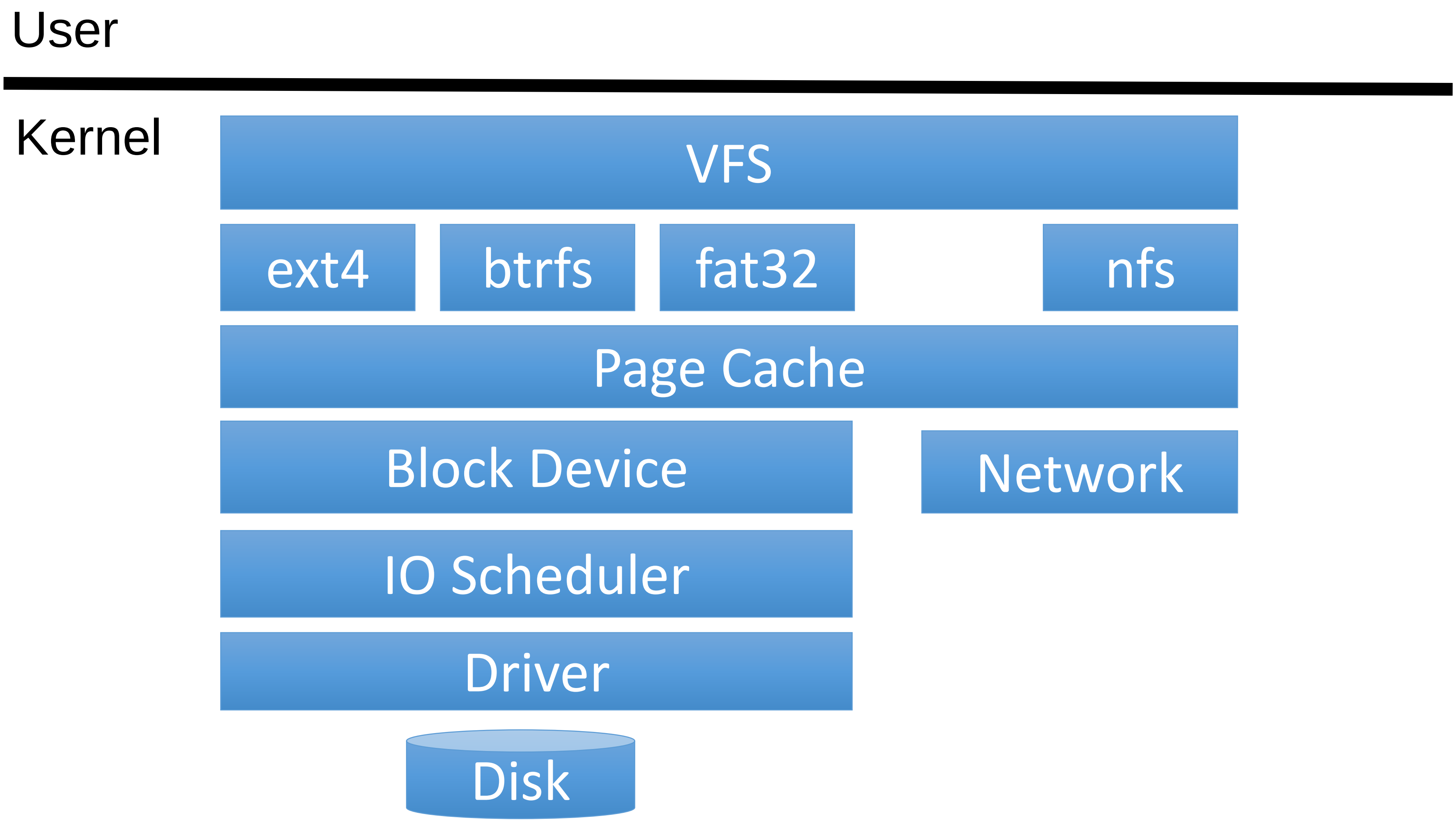
History

- Early OSes provided a single file system
  - In general, system was tailored to target hardware
- People became interested in supporting more than one file system type on a single system
  - Especially to support 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
  - On-disk FS
  - Network FS
  - In-memory FS (***ramdisks***)
  - Pseudo file systems used for configuration
    - (/proc, /devtmps…) only backed by kernel data structures

More Detailed Diagram



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 and API 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

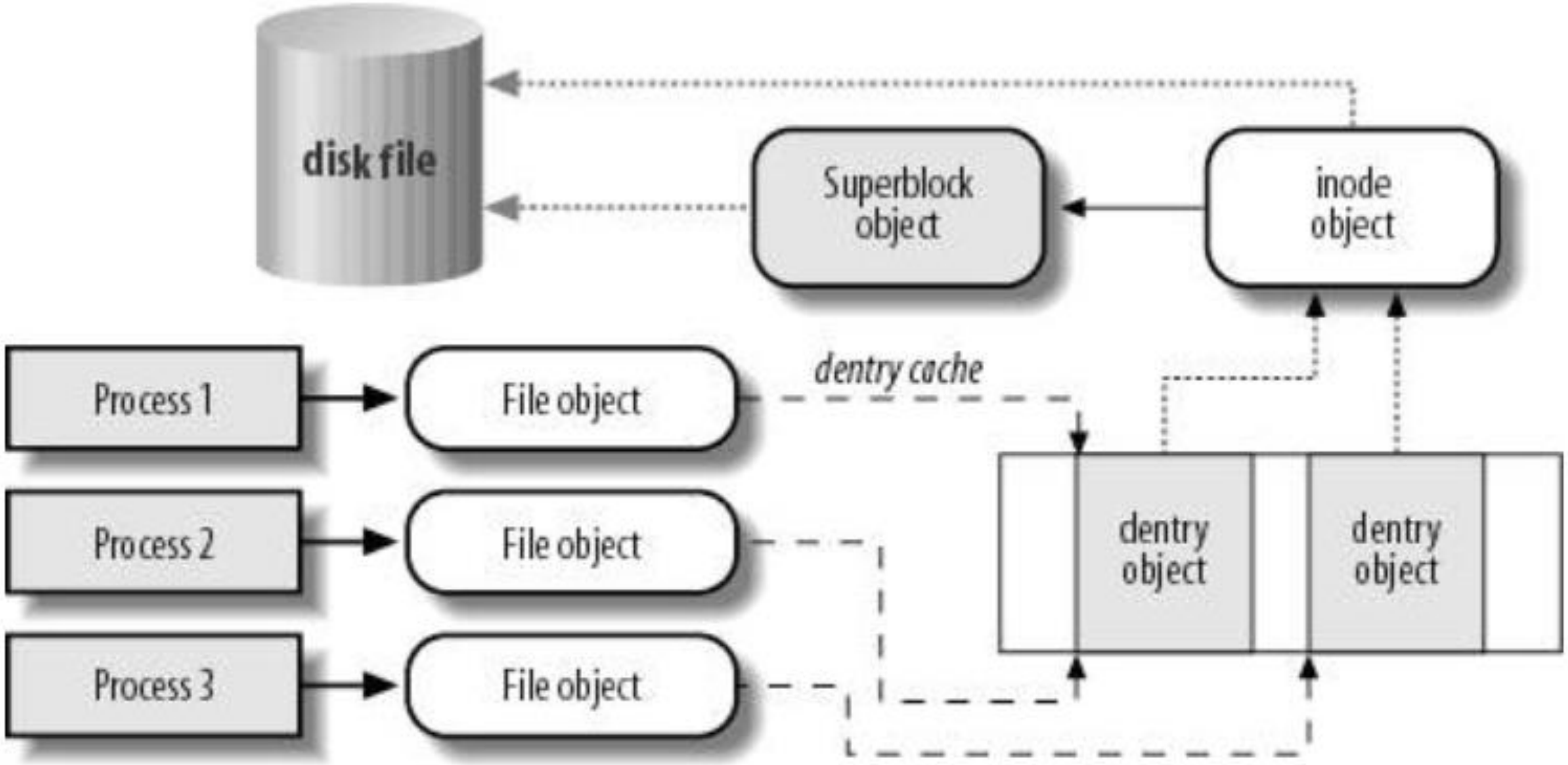
High-Level FS Developer Tasks

- 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
- 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

- ***superblock***: FS-global data
  - 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 object***: represents an `open()`ed file
  - Keeps pointer to dentry and cursor (file offset)
- Superblock and inodes are extended by file system developer

Core VFS Abstractions



Source: Understanding Linux kernel, 3rd Ed

Superblock

- 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 list of all of these
  - When there are multiple FSes (in today's systems: almost 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 appropriate)
  - 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 (1)

- 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
- When link count is zero, inode (and contents) deleted
  - There is no ‘delete’ system call, only ‘***unlink***’

Linking (2)

- "***Hard***" link (`link()` system call/`ln` utility)
  - Creates a new name for the same inode
    - Opening either name opens the *<u>same</u>* file
  - This is *<u>not</u>* a copy
- Open files create an in-memory reference to a file
  - If an open file is unlinked, <u>the directory entry is deleted</u>
    - 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

Example: Common Trick for Temp Files

- How to clean up temp file when program crashes?
  - create (1 link)
  - open (1 link, 1 ref)
  - unlink (0 link, 1 ref)
  - 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

Interlude: 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
```

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 [ugs][+-][rwx]

Special Bits

- For directories, ‘Execute’ means ‘entering’
  - 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
  - `chmod u+s <file>`
  - Program executes with owner’s UID
  - Crude form of permission delegation
  - Any examples?
    - passwd, sudo

More Special Bits

- Group inheritance bit
  - `chmod g+s <directory>`
  - Normally, 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
  - `chmod +t <directory>`
  - Prevents non-owners from deleting or renaming files in a directory with sticky bit

File Objects

- Represents an <u>open</u> file (a.k.a. `struct file`)
  - Each process has a table of pointers to them
  - The `int fd` returned by open is an offset into this table
  - ***File Descriptor Table***
- File object stores state relevant for an open file
  - reference count of the object (like most other kernel objects)
  - dentry pointer
  - cursor into the file
  - file access mode (read-only, read/write, etc.), flags, etc.
  - cache of some inode fields (such as file_operations, permissions, etc.)
- Why a reference count?
  - Fork copies the file descriptors but the file object is shared
    - Particularly important for stdin, stdout, stderr
- VFS-only abstraction
  - FS doesn't track which process has a reference to a file

File Handle Games

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

dentry

- Essentially map a path name to an inode
  - These store:
    - A file name
    - A link to an inode
    - A pointer to parent dentry (null for root of file system)
- Ex: /home/myuser/vfs.pptx may have 4 dentries:
  - /, home, myuser, and vfs.pptx
- Also VFS-only abstraction
  - Although inode hooks on directories can populate them
- Why dentries? 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

dentry Caching and Tracking

- 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 page-cache pages)

- 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

Synthesis Example: open()

- 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 object
  - Allocate a descriptor
    - Point descriptor at file object
  - 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 a ***root*** and ***working*** directory
  - Stored in `current->fs->fs` and `current->fs>pwd`
  - Specifically, these are dentry pointers (not strings)
- Why have a current root directory?
  - Some programs are ***chroot***-jailed and should not be able to access anything outside of the directory
- First character of pathname dictates which dentry to use to start searching (`fs` or `pwd`)
  - An absolute path starts with the '/' character (e.g., /lib/libc.so)
  - A relative path starts with anything else (e.g., ../vfs.pptx)

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
  - Remember: dentry for / is stored in `current->fs->fs`
- 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)
  - Compute a hash value to find bucket in denry hash table
    - Hash of path from root (e.g., '/home/foo', not 'foo')
  - Search the hash bucket to find entry for /home

Detail

- If no dentry in the hash bucket
  - Call `lookup()` method on parent inode (provided by FS)
    - Probably will read the directory content from disk
- 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
- Check for 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
- To prevent infinite symlink recursion, quit (with `-ELOOP`) if
  - more than 40 symlinks resolved, or
  - more than 6 symlinks in a row without non-symlink
- 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

Back to `open()`: file 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 Descriptor Table

- Recap: descriptors index into per-process array of pointers to file objects
  - File descriptor table
- `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 object and put a pointer in the table

Once `open()`'d, 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 –errno
- Reminder: discussed the implementation in “Page Cache” lecture

More on User's Perspective

How to…

- …create a file?
  - `create()` system call
  - Also, more commonly, `open()` with the `O_CREAT` flag
  - What does `O_EXCL` do?
    - If called with `O_EXCL|O_CREATE` and the file already exists, `open()` fails
    - Avoids race conditions between creation and open
- …create a directory?
  - `mkdir()`

More on User's Perspective

How to…

- …remove a directory?
  - `rmdir()`
- …remove a file?
  - `unlink()`
- …read a file?
  - `read()`
  - Use `lseek()` to change the cursor position
  - Use `pread()` to read from an offset w/o changing cursor
- …read a directory?
  - `readdir()` or `getdents()`

How Does an Editor Save a File?

- Hint: don't want half-written file in case of crash
  - Create a *temp* file (using `open`)
  - Copy *old* to *temp* (using `read` *old* / `write` *temp*)
  - Apply writes to *temp*
  - Close both *old* and *temp*
  - Do a `rename(temp, old)` to atomically replace
    - Drawback?
    - Hint 1: what if there was a second hard link to *old*?
    - Hint 2: what if *old* and *temp* have different permissions?