Linux Filesystems

*nix file cmd utils ⟶ implementation in storage

- ➔ What all happens in the background when “ls -ltr file.c” is executed?
    - ◆ Where is file access permissions, access history etc. stored?
    - ◆ Who has the responsibility of enforcing access policies?
    - ◆ How is the file located?

Support for multiple file systems

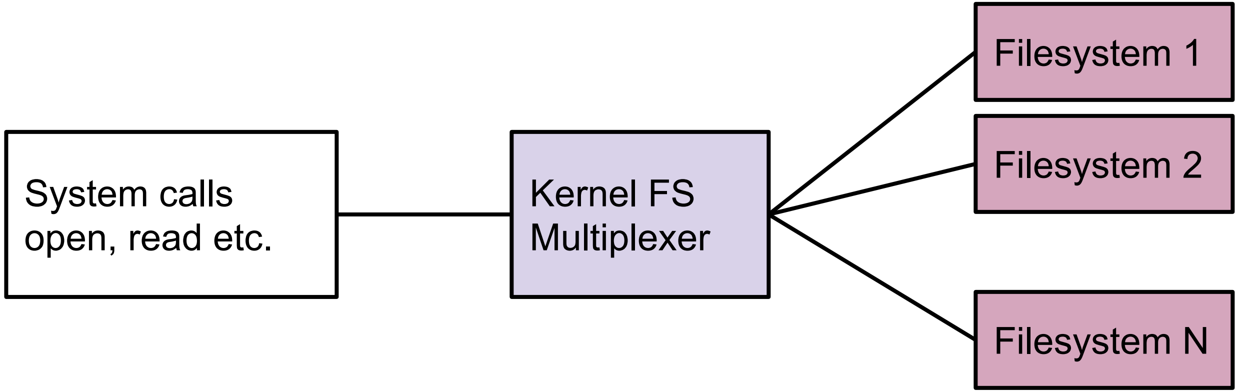
- One file system per OS is restrictive, why?
  - In unix systems, file is a heavily used abstraction: regular files, device files, sockets
  - Remote file systems
- If you think your new FS idea has potential for improvements
  - you should not have an excuse, “but you see, I have to change the existing file system”
- Support for multiple file systems require some careful interfacing
  - POSIX compliant file system calls - standards matter!
  - File systems can not bear the burden of policy enforcement
    - Process, user, quota etc.

Process (user) view   vs.   reality

- ➔ User views the file system as a big fat tree
  - ◆ Can open a file with a relative/absolute path
- ➔ Most of the times more than one file systems underneath

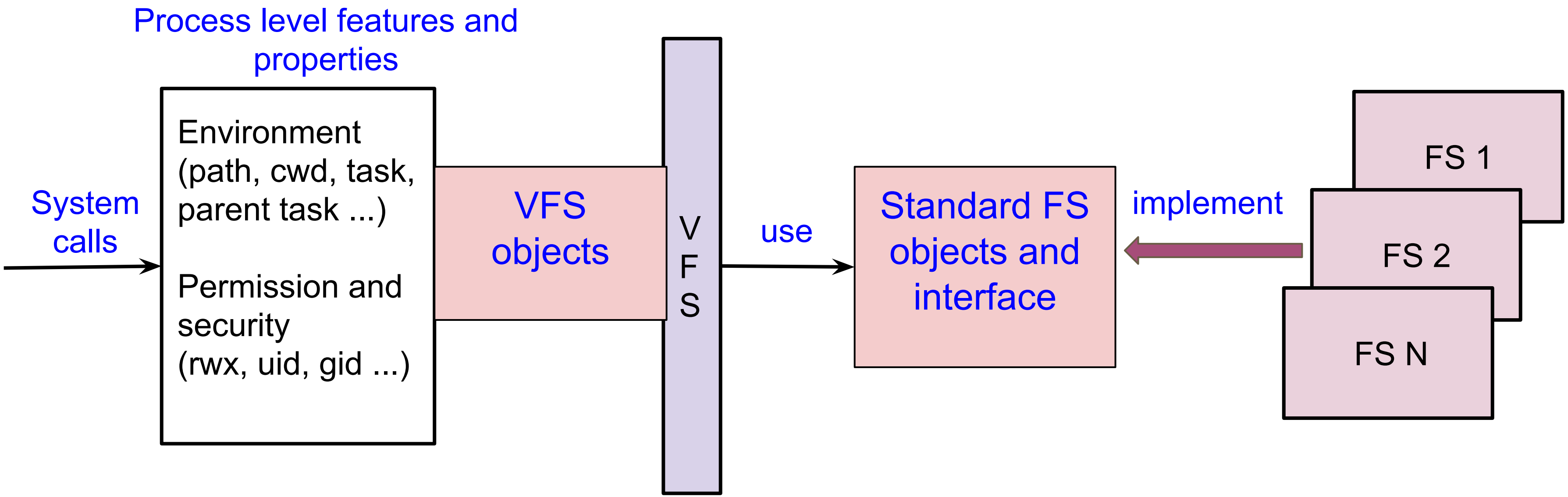
- ➔ Multiple processes can open the same file
  - ◆ Different access modes
  - ◆ Different file position pointers
- ➔ At storage level, it is the same file

Let us decide the responsibilities!



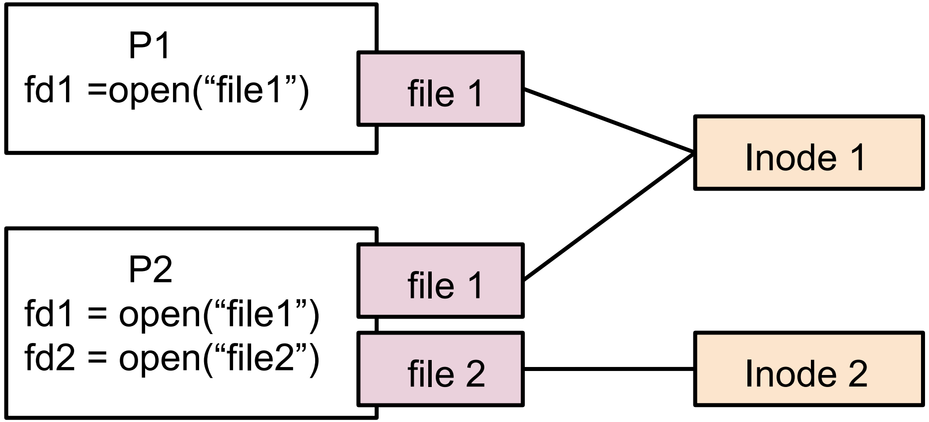
- ➔ Path name translation e.g., open (/home/user1/data/file.c)
- ➔ create, delete, chown, chmod …
- ➔ Open, read, write, truncate…
- ➔ Multiplexer or filesystem?

Linux virtual file system (VFS)


Process level features and properties
System calls
Environment (path, cwd, task, parent task ...)
Permission and security (rwx, uid, gid ...)
VFS objects
V F S
use
Standard FS objects and interface
implement
FS 1
FS 2
FS N

- ➔ Object and interface choices guided by API requirement (mostly)
- ➔ Sometimes unix tradition determines the interfacing

Process view of a file



- ➔ Every file opened has a state
  - ◆ Represented in “struct file”
  - ◆ Steps of creation : fd = open(“some.txt”, O_RDWR)
  - ◆ Operations (read, write etc.) implemented by ?
- ➔ File object must be stored (persistently), true or false
- ➔ One physical file → many file objects

Process view: All open files and FS information

- ➔ All open files information
  - ◆ Struct files_struct, contains a list of “struct file”
  - ◆ Task has a pointer to this structure

- ➔ FS Information required by a process to get started on real file operations
- ➔ FS struct
  - ◆ Root directory
  - ◆ Current directory
  - ◆ Default file permissions

VFS - FS interface: inode

- A traditional representation of a file in unix systems
  - Permissions, access time, file size, block layout (e.g., indexed allocation)
  - Unique for every file in the file system
- Most file systems implement a similar on-disk version
- Linux VFS compulsion
  - “Don’t care if you represent a file on disk in a different way, you show me the way I want to see a file”
- Operations
  - Create, truncate, permissions …

VFS - FS interface: superblock

- ➔ Every file system registered with VFS must have a super block
  - ◆ FS is not a real on-disk FS, does not matter, VFS requires it anyway
- ➔ Device information, block size, ...
- ➔ Operations: alloc inode, destroy inode ...
- ➔ More on super block latter

VFS - FS interface: dentry

- Dentry represents a specific element in a file path
  - Both for file and directory
- May not have an equivalent on-disk state
- Explicit representation of parent dir - subdirectory relationships
- Dentry cache: speed up path translation
- A dentry can be
  - Used and valid
  - Unused but valid
  - Invalid (also called negative)
- There can be a dentry for non-existent path!

Path translation example