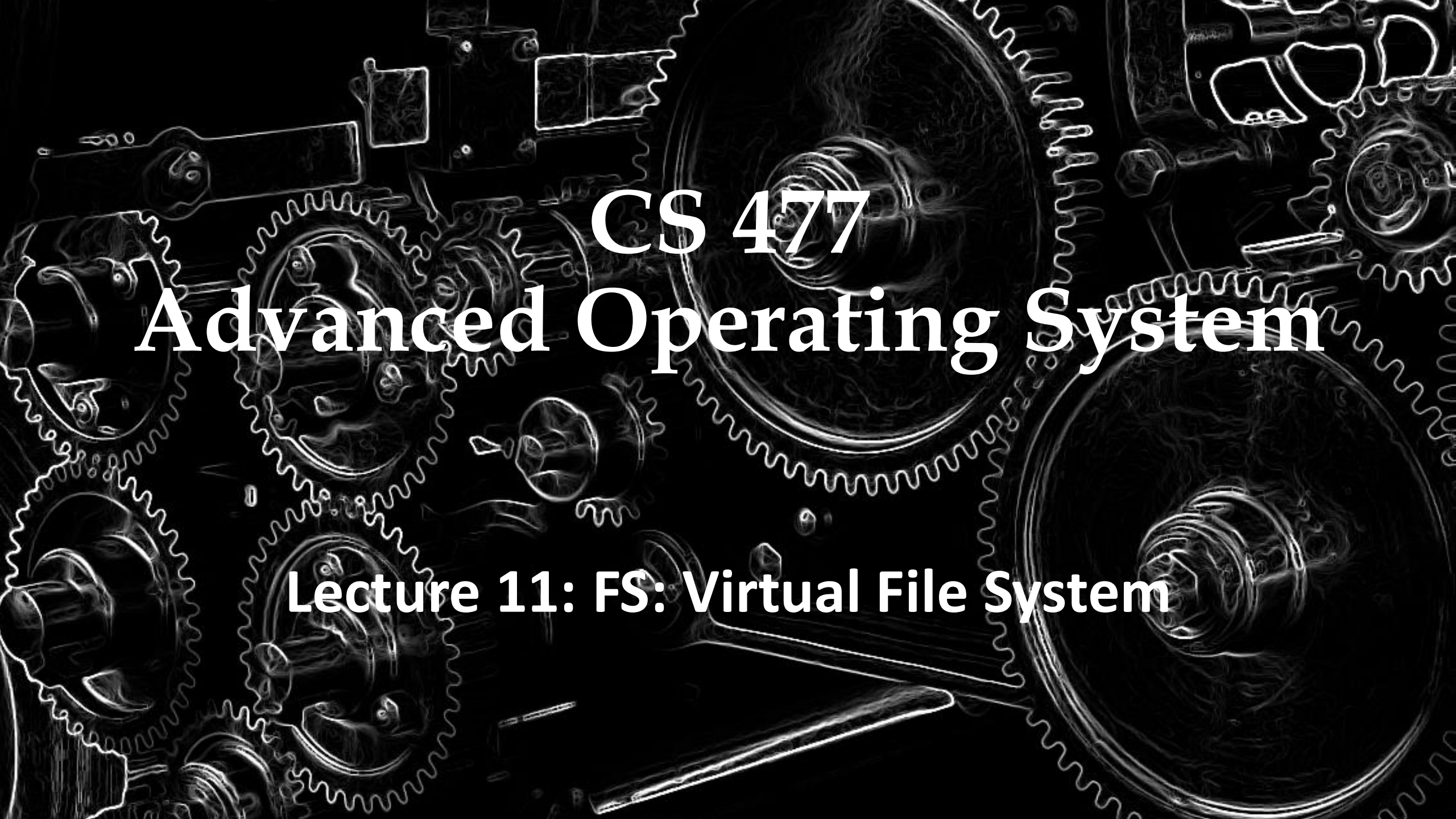
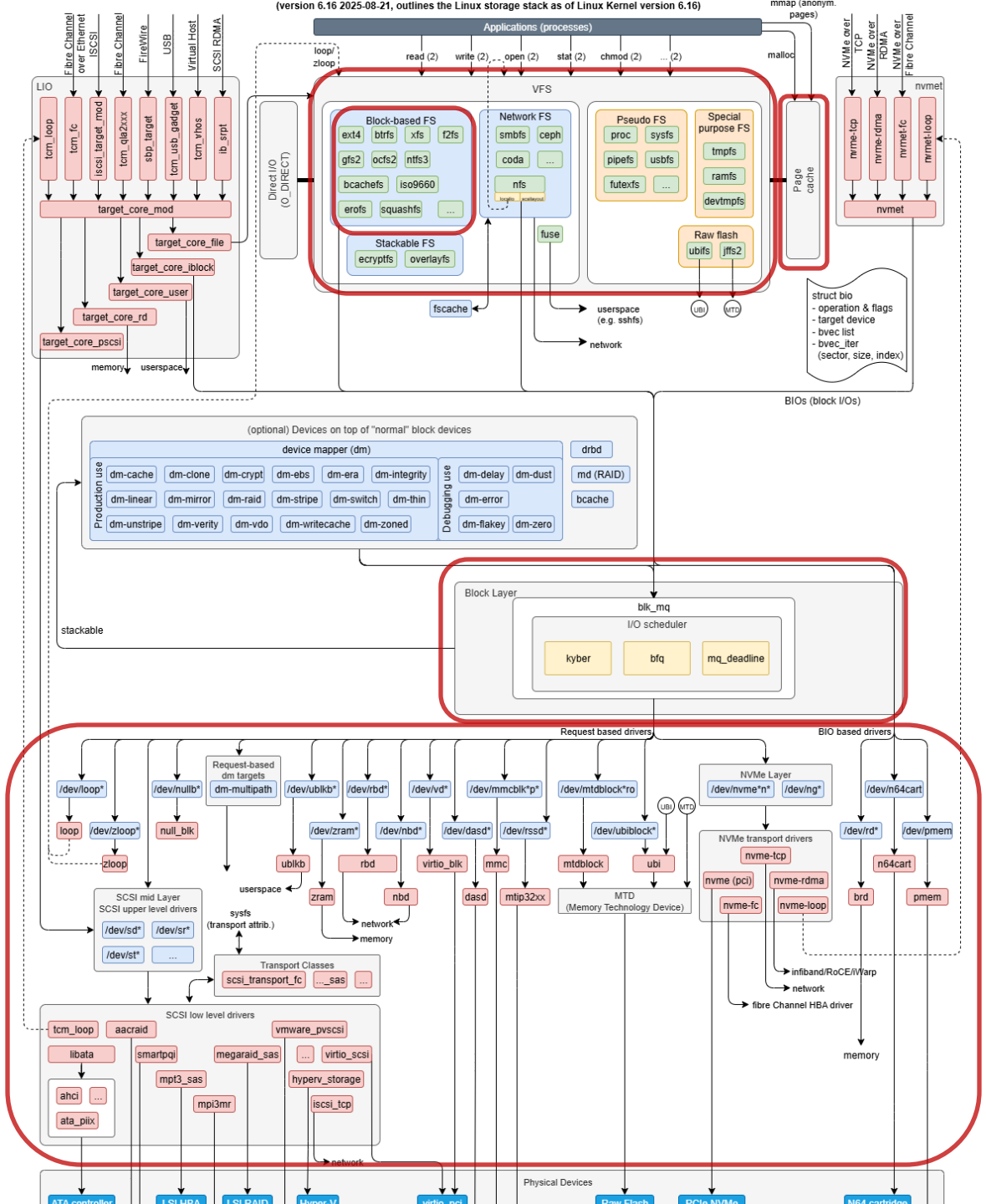
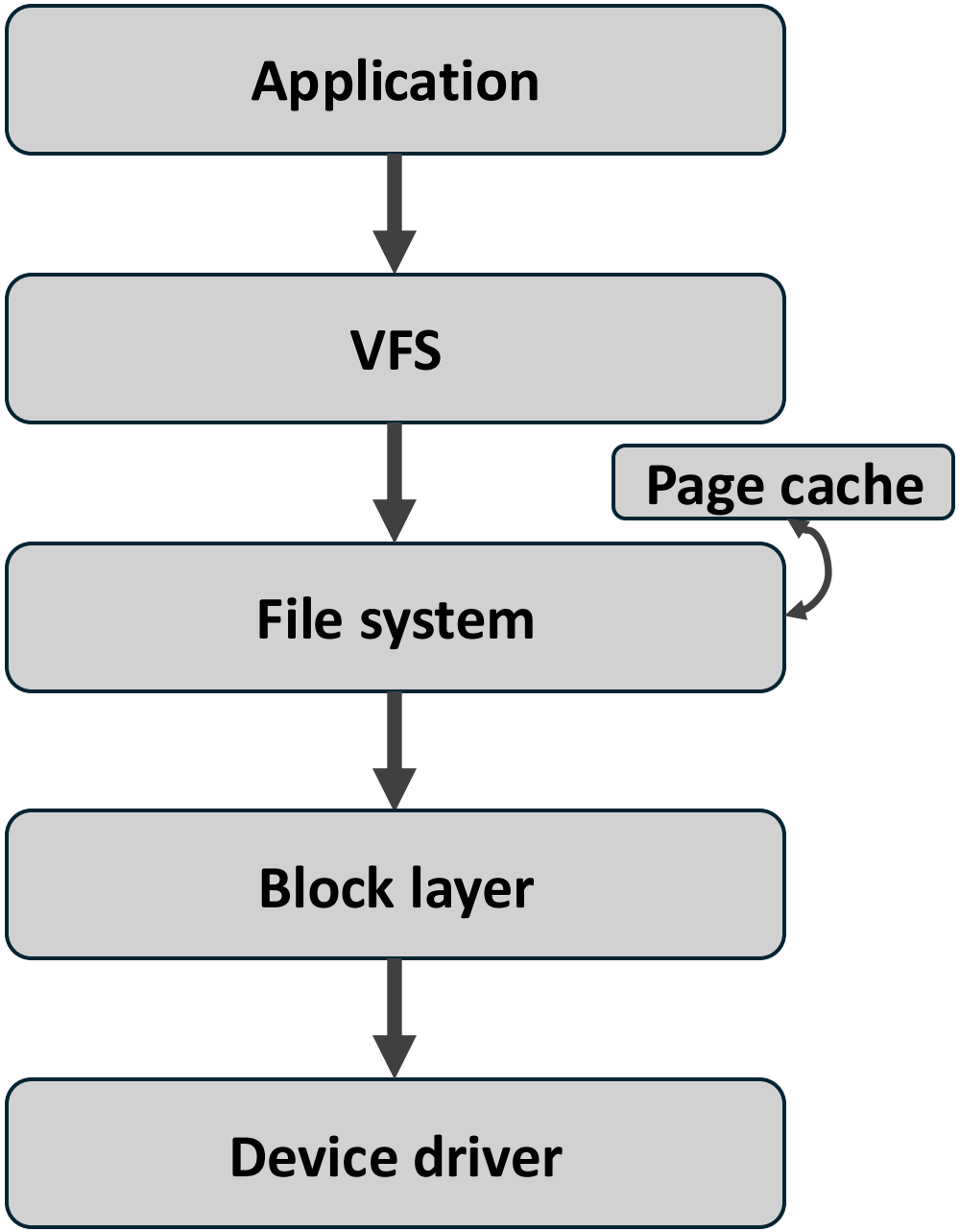


# CS 477
# Advanced Operating System

## Lecture 11: FS: Virtual File System


(version 6.16 2025-08-21, outlines the Linux storage stack as of Linux Kernel version 6.16)
Applications (processes)
mmap (anonym. pages)
read (2)
write (2)
open (2)
stat (2)
chmod (2)
... (2)
malloc
VFS
loop/ zloop
LIO
Fibre Channel over Ethernet
iSCSI
Fibre Channel
FireWire
USB
Virtual Host
SCSI RDMA
tcm_loop
tcm_fc
iscsi_target_mod
tcm_qla2xxx
sbp_target
tcm_usb_gadget
tcm_vhos
ib_srpt
target_core_mod
target_core_file
target_core_iblock
target_core_user
target_core_rd
target_core_pscsi
memory
userspace
Direct I/O (O_DIRECT)
Block-based FS
ext4
btrfs
xfs
f2fs
gfs2
ocfs2
ntfs3
bcachefs
iso9660
erofs
squashfs
...
Network FS
smbfs
ceph
coda
...
nfs
Stackable FS
ecryptfs
overlayfs
fuse
Pseudo FS
proc
sysfs
pipefs
usbfs
futexfs
...
Special purpose FS
tmpfs
ramfs
devtmpfs
Raw flash
ubifs
jffs2
Page cache
fscache
userspace (e.g. sshfs)
network
UBI
MTD
struct bio
- operation & flags
- target device
- bvec list
- bvec_iter (sector, size, index)
BIOs (block I/Os)
nvmet
NVMe over TCP
NVMe over RDMA
NVMe over Fibre Channel
nvme-tcp
nvme-rdma
nvmet-fc
nvmet-loop
(optional) Devices on top of "normal" block devices
device mapper (dm)
Production use
dm-cache
dm-clone
dm-crypt
dm-ebs
dm-era
dm-integrity
dm-linear
dm-mirror
dm-raid
dm-stripe
dm-switch
dm-thin
dm-unstripe
dm-verity
dm-vdo
dm-writecache
dm-zoned
Debugging use
dm-delay
dm-dust
dm-error
dm-flakey
dm-zero
drbd
md (RAID)
bcache
stackable
Block Layer
blk_mq
I/O scheduler
kyber
bfq
mq_deadline
Request based drivers
BIO based drivers
/dev/loop*
loop
/dev/zloop*
zloop
/dev/nullb*
null_blk
Request-based dm targets
dm-multipath
/dev/ublkb*
ublkb
userspace
/dev/zram*
zram
sysfs (transport attrib.)
network
memory
/dev/rbd*
rbd
/dev/nbd*
nbd
/dev/vd*
virtio_blk
/dev/dasd*
dasd
/dev/mmcblk*p*
mmc
/dev/rssd*
mtip32xx
/dev/mtdblock*ro
mtdblock
/dev/ubiblock*
ubi
MTD (Memory Technology Device)
NVMe Layer
/dev/nvme*n*
/dev/ng*
NVMe transport drivers
nvme-tcp
nvme (pci)
nvme-rdma
nvme-fc
nvme-loop
infiband/RoCE/iWarp
network
fibre Channel HBA driver
/dev/n64cart
n64cart
/dev/rd*
brd
/dev/pmem
pmem
memory
SCSI mid Layer
SCSI upper level drivers
/dev/sd*
/dev/sr*
/dev/st*
...
Transport Classes
scsi_transport_fc
..._sas
...
SCSI low level drivers
tcm_loop
aacraid
vmware_pvscsi
libata
smartpqi
megaraid_sas
...
virtio_scsi
mpt3_sas
hyperv_storage
ahci
...
ata_piix
mpi3mr
iscsi_tcp
network
Physical Devices


Application
VFS
Page cache
File system
Block layer
Device driver

# Today's agenda

- Storage stack design philosophy
- VFS and four key objects
  - File system types and mounting
  - Caching in VFS
  - Polymorphism in VFS
- Process-VFS relationship
- Path lookup example

# The "Everything is a File" Philosophy

- **Uniform interface:** Same syscalls for everything: `open/read/write/close`
- **Powerful tools:** Utilities (cat, grep, cp) work universally, without caring **what** they are reading
- **Composability:** The pipe (|) operator becomes the ultimate "glue"
  - `cat /dev/random | head -n 1`
- **Example of files:**
  - **Files:** A file on a disk (eg, `/etc/passwd`)
  - **Devices:** A hardware device (eg, `/dev/sda`)
  - **Kernel data:** Process info (eg, `/proc/self/status`)
  - **Network:** A network socket (via a file descriptor)

**Q. What problems do you think arise when Linux needs to support ext4, FAT32, NFS, and /proc all at the same time?**

# The core problem: A messy reality

**ext4 (Disk)**

Uses inodes, complex data blocks, and journals for reliability.

**FAT32 (USB)**

Simple file allocation table. No permissions, simple block chains.

**NFS (Network)**

The “file” is on another computer. All data is fetched via RPCs.

**procfs (`/proc`)**

The “file” doesn't exist at all. Data is generated on-the-fly by the kernel.

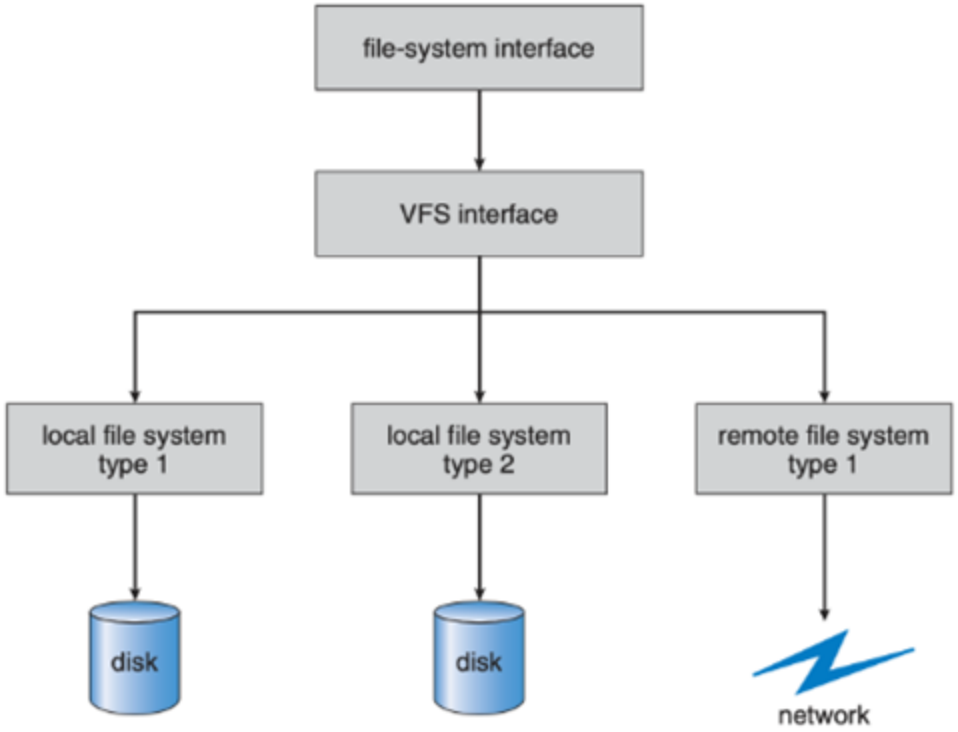
**Q. How can `cat` use the same `read()` syscall for all four?**

# VFS as an abstraction layer

**The "adapter" layer**

- VFS → a generic translation layer
- Does not know about **disks** or **networks**
- Only knows about abstract objects
  - **User space:** open/read/write calls
  - **VFS (kernel): generic translation layer**
  - **File system (kernel):** Plugins like ext4, NFS, procfs
  - **Drivers (kernel):** block layer / network stack

VFS is the API that all "concrete" file systems must implement

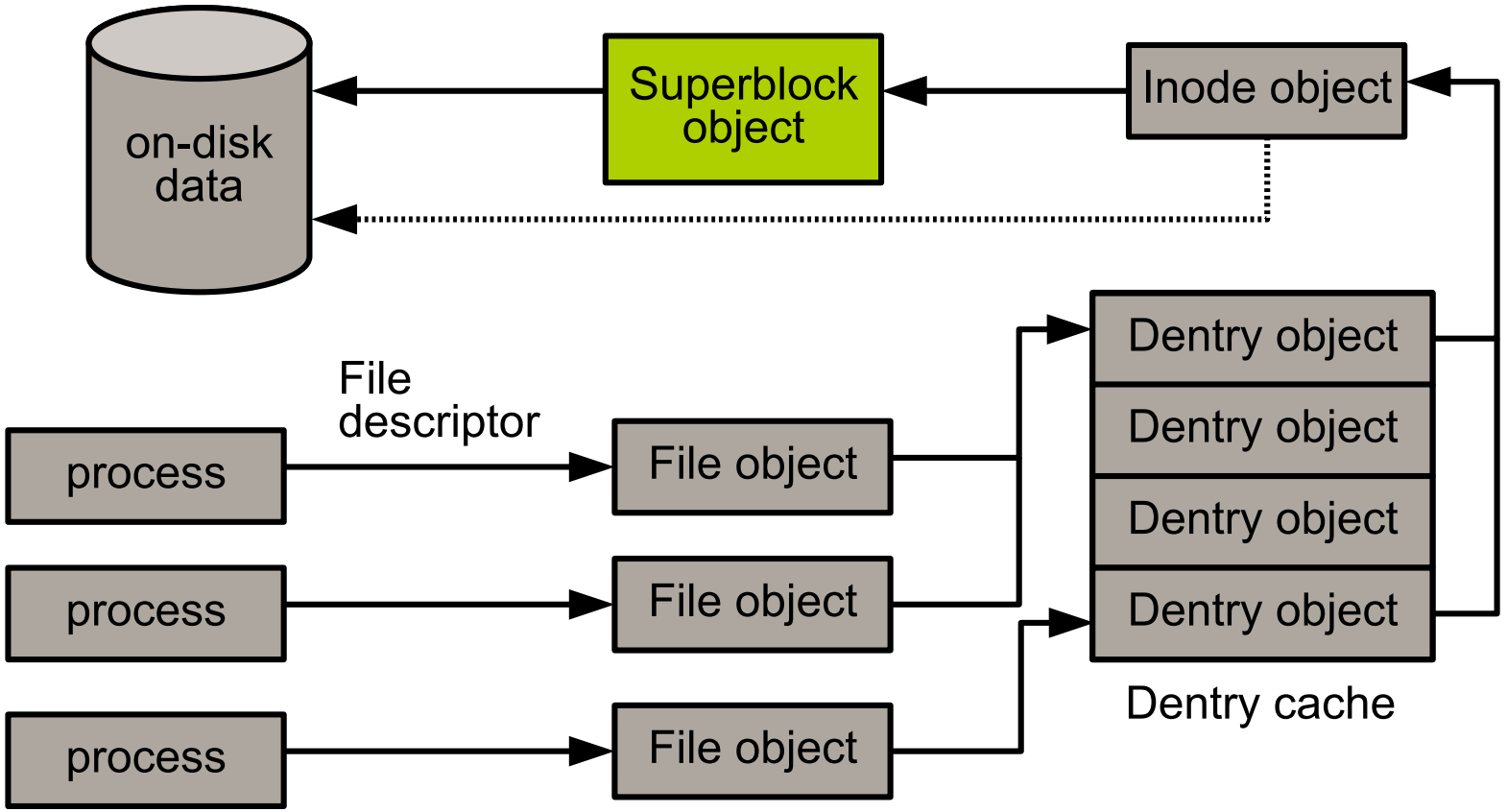


# Four key VFS objects

1. **Superblock:** file system instance (mounted partition)
2. **Inode:** file metadata (permission, size)
3. **Dentry:** path component or directory entry (name)
4. **File:** open file descriptor (handle)

**superblock (partition)**
↓
**inode (file metadata)**
↓
**dentry (name) ← multiple can point to same inode (hard links!)**
↓
**file (open descriptor) ← per-process**

# 1. Superblock

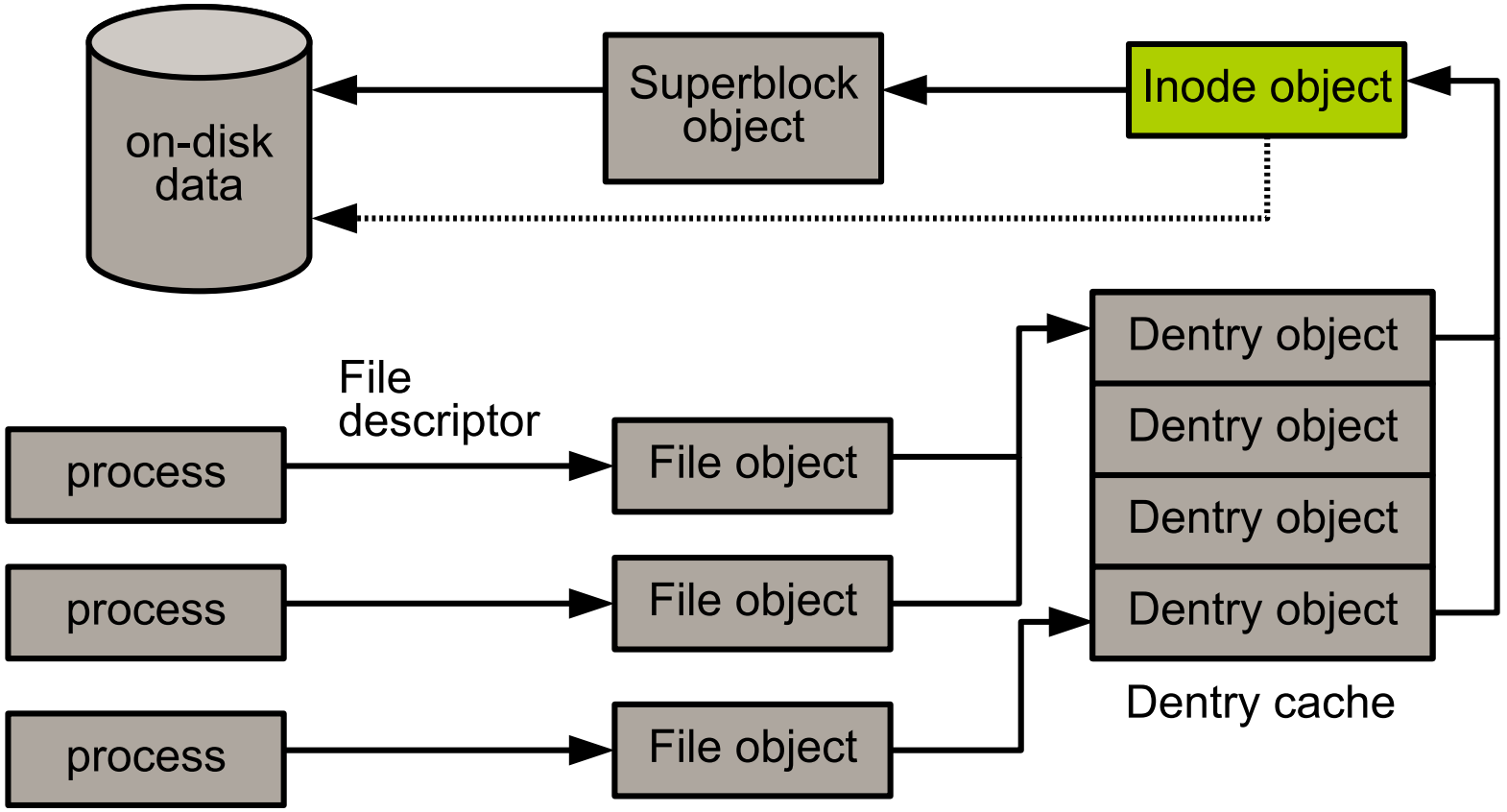


# 1. Superblock: Factory for FS instance

```
struct super_block {
  struct file_system_type *s_type;   // ext4? NFS?
  struct super_operations *s_op;     // operations table
  struct dentry *s_root;             // root directory "/"
  unsigned long s_blocksize;         // block size
  // ... many other fields ...
};
```

- Represents a **single mounted file system instance**
- Contains global information about the filesystem (partition)
- Created by the filesystem and given to VFS at mount time:
  - Disk-based filesystem stores it in a special location
  - Other filesystems have a way to generate it at mount time
- **`struct super_block`** defined in `include/linux/fs.h`

# 2. Inode


on-disk data
Superblock object
Inode object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache
File descriptor
process
process
process
File object
File object
File object

# 2. Inode: the *file blueprint*

```c
struct inode {
  unsigned long i_ino; // inode number
  umode_t i_mode; // file type & permissions
  uid_t i_uid; // owner
  loff_t i_size; // file size in bytes
  struct timespec i_atime; // access time
  struct timespec i_mtime; // modification time
  const struct inode_operations *i_op; // operations
  const struct file_operations *i_fop; // file operations
  struct address_space *i_mapping; // page cache!
  // ... many other fields ...
};
```

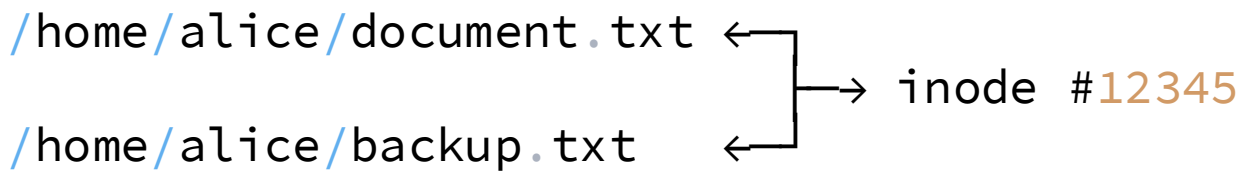
- Represents all metadata of a **single file** or **directory**
- **One inode in memory, no matter how many times it is opened!**

# 2. Inode (contd ...)

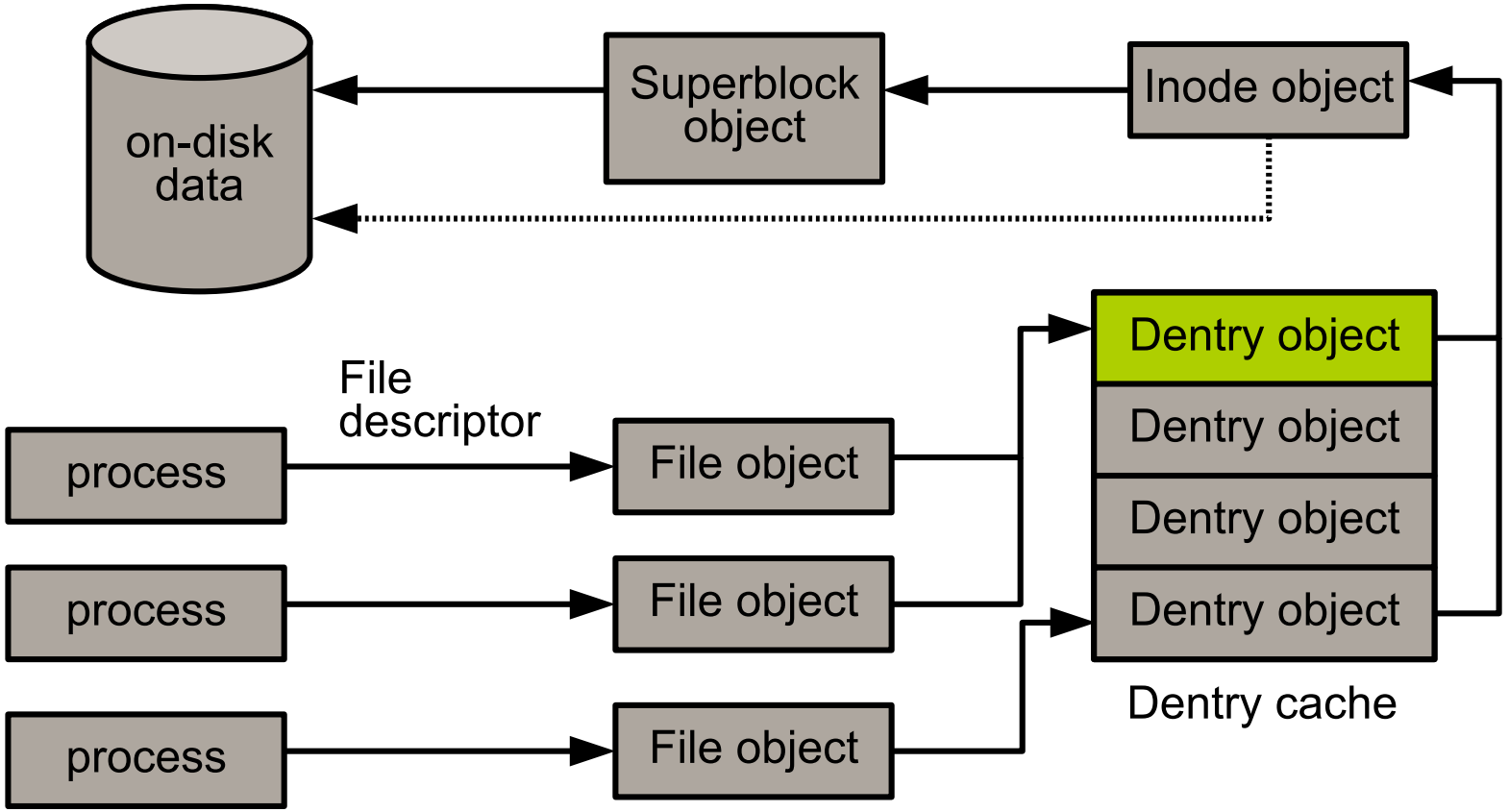
- Also contains information on how to manipulate the file
- Must be produced by the filesystem on-demand when a file/directory is accessed:
  - Read from disk in Unix-like filesystem
  - Reconstructed from on-disk information for other filesystems

**Q. Why is the filename NOT stored in the inode?**

**Hardlinks:** multiple files can point to the same inode



# 3. Dentry


on-disk data
Superblock object
Inode object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache
File descriptor
process
process
process
File object
File object
File object

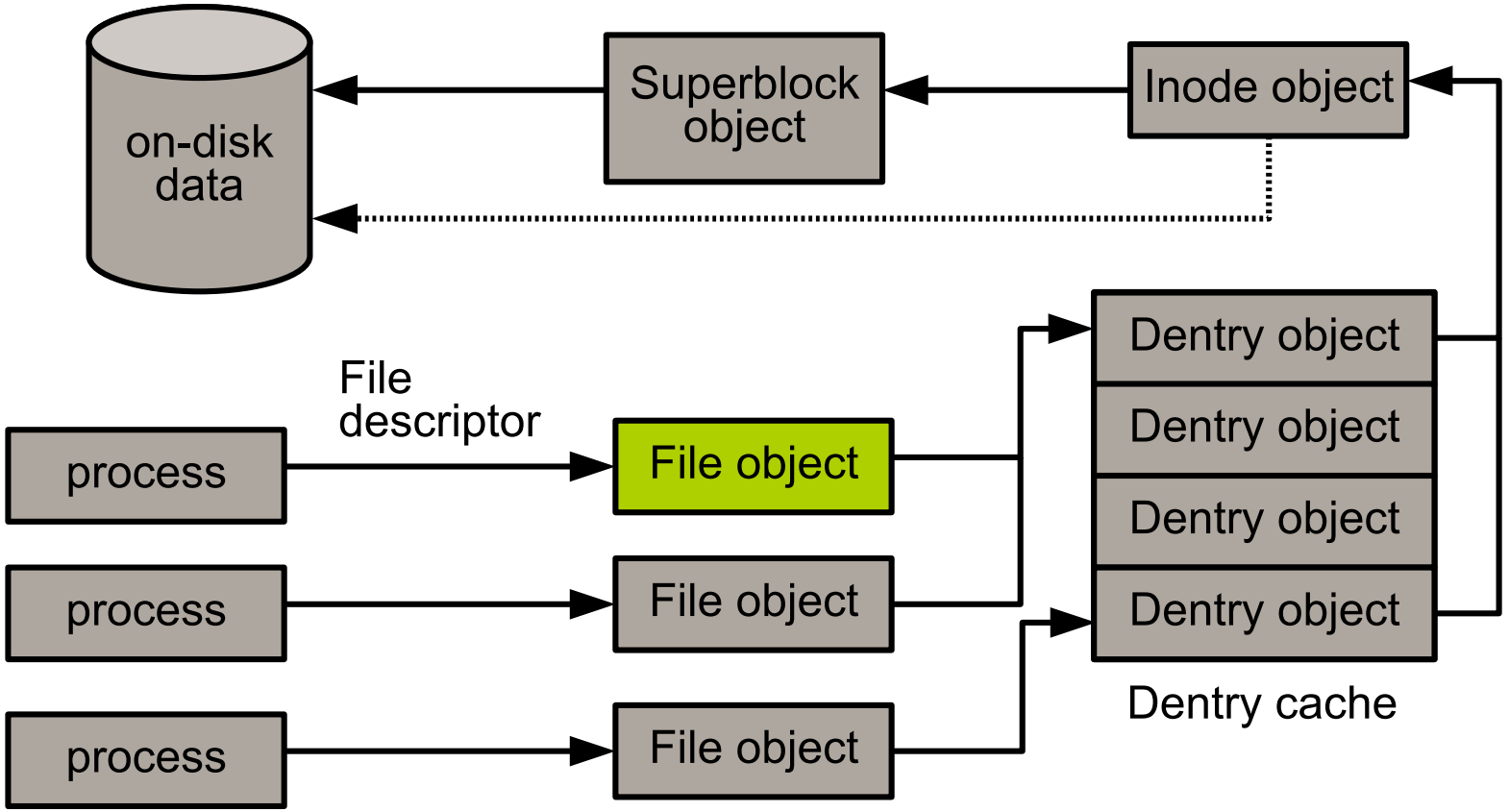
# 3. Dentry: the *signpost*

```
struct dentry {
  struct qstr d_name; // the filename!
  struct inode *d_inode; // pointer to inode
  struct dentry *d_parent; // parent directory
  struct list_head d_subdirs; // children
  struct dentry_operations *d_op; // operations
  // ... other fields ...
};
```

- Represents a **path component** and links it to an inode
  - Stores file/directory name
  - Stores its location in the directory
  - Performs directory specific ops → pathname lookups
- **`/home/cs477/test.txt`**
  - One dentry associated with each of: '/', 'home', 'cs477', and 'test.txt'

**Q. When you delete a file, do you the inode or the dentry?**

# 4. File


on-disk data
Superblock object
Inode object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache
File descriptor
process
process
process
File object
File object
File object

# 4. File object: the bookmark

```
struct file {
  struct path f_path; // contains dentry
  const struct file_operations *f_op; // operations
  loff_t f_pos; // current file offset
  unsigned int f_flags; // O_RDONLY, etc.
  struct address_space *f_mapping; // page cache
  // ... other fields ...
};
```

- Open file handle (a session)
  - Created on **`open()`** and destroyed on **`close()`**
- No corresponding on-disk structure (live view maintained by the kernel)

**Q. If two processes open the same file, how many file objects exist? How many inodes?**

**Two file objects, point to the same unique dentry (point to a unique inode)**

# Today's agenda

- Storage stack design philosophy
- VFS and four key objects
  - **File system types and mounting**
  - Caching in VFS
  - Polymorphism in VFS
- Process-VFS relationship
- Path lookup example

# File system data structures: `file_system_type`

```
struct file_system_type {
  const char *name; // "ext4", "nfs", etc.
  int fs_flags; // FS_REQUIRES_DEV, etc.
  // Called when mounting
  struct dentry *(*mount)(struct file_system_type *, int, const char *, void *);
  void (*kill_sb)(struct super_block *);
  struct module *owner; // THIS_MODULE
  struct file_system_type *next; // linked list
  struct hlist_head fs_supers; // all superblocks
};
```

- One **`file_system_type`** per filesystem **type** (ext4, NFS, etc)
- Lives for the entire kernel **lifetime** (or module lifetime)
- Registered via **`register_filesystem`**

```
file_system_type("ext4") ──┬──→ superblock (USB drive mounted at /mnt/usb)
                           └──→ superblock (HDD partition at /home)
file_system_type("nfs") ─────→ superblock (network share at /mnt/nfs)
```

# File system data structures: `vfsmount`

```c
struct vfsmount {
  struct dentry *mnt_root; // root of mounted tree
  struct super_block *mnt_sb; // pointer to superblock
  int mnt_flags; // MNT_NOSUID, etc.
};
```

- Represent a specific instance of the filesystem: a mount point

```
Global VFS tree:
        / (root vfsmount)
        ├── home/
        ├── mnt/
        │   └── usb/ ← mount point (new vfsmount starts here)
        │   ├── file1.txt
        │   └── file2.txt
```

**Q. What happens when you do cd /mnt/usb? How does VFS know to switch to the USB drive's filesystem?**

# Dentry contains information about mount point

- Dentry for `/mnt/usb` has a flag saying "this is a mount point"
- VFS follows the **`vfsmount`** to the new superblock

```
dentry("/mnt/usb") → d_mounted flag set
                  ↓
              vfsmount → new superblock (USB drive)
                       → mnt_root (root dentry of USB)
```

# Why VFS is not slow?

# Latency numbers

L1 cache reference ..... 0.5 ns
Branch mispredict ..... 5 ns
L2 cache reference ..... 7 ns
Mutex lock/unlock ..... 25 ns
Main memory reference ..... **100 ns**
Compress 1K bytes with Zippy ..... 3,000 ns = 3 µs
Send 2K bytes over 1 Gbps network ..... 20,000 ns = 20 µs
SSD random read ..... 150,000 ns = 150 µs
Read 1 MB sequentially from memory ... 250,000 ns = 250 µs
Round trip within same datacenter ..... 500,000 ns = 0.5 ms
Read 1 MB sequentially from SSD*..... **1,000,000 ns = 1 ms**
Disk seek ..... **10,000,000 ns = 10 ms**
Read 1 MB sequentially from disk ..... **20,000,000 ns = 20 ms**
Send packet CA->Netherlands->CA ..... 150,000,000 ns = 150 ms

Source: Latency numbers every programmer should know

# Why caching

- Disk access is several orders of magnitude slower than memory access
- Data accessed once will, with a high likelihood, find itself accessed again in the near future → **temporal locality**

# Three VFS caches

1. **Dentry cache (dcache)**
   - Hash table: **`dentry_hashtable`**
   - Caches path lookups
   - Most important for performance
2. **Inode cache**
   - Hash table: **`inode_hashtable`**
   - Caches files metadata
   - Pinned by dentries in **`dcache`**
3. **Page cache**
   - Caches file data (not metadata)
   - Lives in **`inode->i_mapping`**

# Dentry cache: mechanism

- Hash table lookup: **`d_lookup()`**
  - `Key = (parent dentry, filename)`
- Extremely fast, most path lookups hit cache
  - Kernel finds paths in DRAM and rarely has to read directory from the disk
- Dynamically grows on demand and shrinks under pressure
  - Tunable knob: **`vfs_cache_pressure`**

# Dentry cache: states

Manages cache lifetime:

1. **USED:** `d_count > 0`
   - Has valid inode, and actively used (open file, current working directory)
   - **Cannot be evicted**
2. **UNUSED:** `d_count = 0`
   - Has valid inode, and kept in cache for performance
   - **Can be evicted, if needed** (LRU list to maintain such objects)
3. **NEGATIVE:** `d_count > 0, d_inode = NULL`
   - Caches “files does not exist”
   - Also in the LRU list

**Q. Why Linux cache NEGATIVE dentries (files that don’t exist)? Isn’t that a waste of memory**

# The importane of negative dentry

```
$ gcc myprogram.c
Shell searches for "gcc" in PATH:
/usr/local/bin/gcc  → NEGATIVE (cached!)
/usr/bin/gcc        → FOUND
```

Without negative dentry cache:

- Every command would do expensive "file not found" lookups
- PATH with 10 entries = 9 wasted disk lookups

With negative dentry cache:

- First time: 9 disk lookups + 9 negative dentries cached
- Subsequent times: 9 instant cache hits + 1 disk lookup

Negative dentry cache is why shells are fast!

# Page cache (or buffer cache)

- Physical pages in RAM holding disk content (blocks)
  - Disk → backing store
  - Works for regular files, memory-mapped files, and block device files

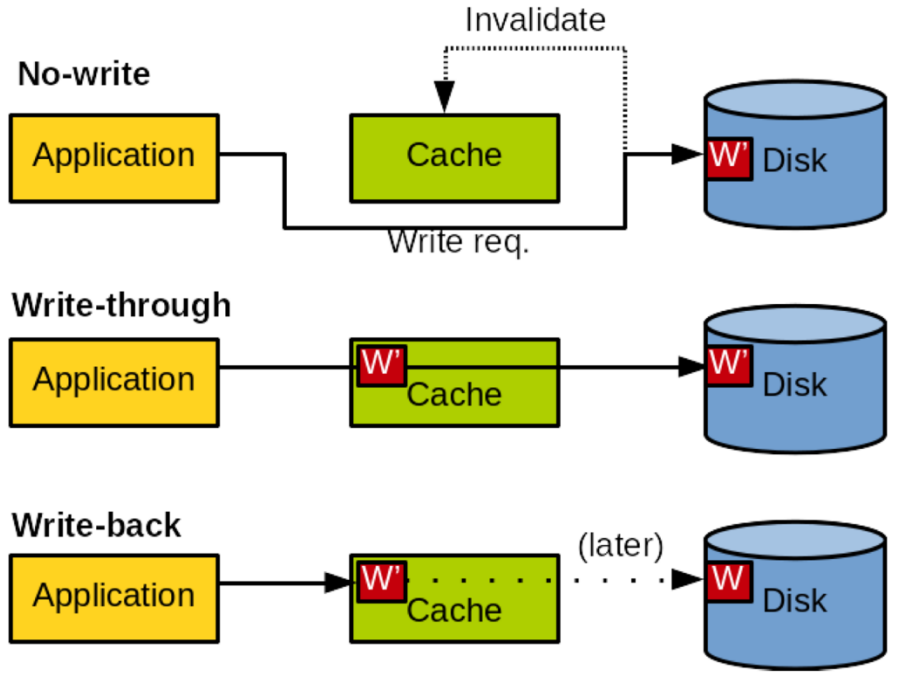
- Dynamic size
  - Grows to consume free memory unused by kernel and processes
  - Shrinks to relieve memory pressure

# Page cache

- Buffered IO operations (without **`O_DIRECT`**), the page cache of a file is first checked
- **Cache hit:** if data is in the page cache, copy from/to user memory
- **Cache miss**: otherwise, VFS asks the concrete file system (e.g., ext4) to read data from disk
  - Read/write operations populate the page cache

# Write caching policies

- **No-write:** does not cache write operations
- **Write-through:** write operations immediately go through to disk
  - Keeps the cache coherent
  - No need to invalidate cached data → simple
- **Write-back:** write operations update page cache but disk is not immediately updated → **Linux page cache policy**
  - Pages written are marked dirty using a tag in the Xarray
  - Periodically, write dirty pages to disk → writeback
  - Page cache absorbs temporal locality to reduce disk access



# Cache eviction

- **When data should be removed from the cache?**
  - Need more free memory (memory pressure)
- **Which data should be removed from the cache?**
  - Ideally, evict cache pages that will not be accessed in the future
  - **Eviction policy**: deciding what to evict
  - MGLRU: Multi-generational LRU policy (discuss in the next class)

# Today's agenda

- Storage stack design philosophy
- VFS and four key objects
  - File system types and mounting
  - Caching in VFS
  - **Polymorphism in VFS**
- Process-VFS relationship
- Path lookup example

# VFS “polymorphism” via function pointers

```
// VFS doesn't know about ext4, NFS, procfs, etc.
// It just calls through function pointers:
ssize_t vfs_read(struct file *file, char *buf, size_t count, loff_t *pos) {
  return file->f_op->read(file, buf, count, pos);
//                ↑
//               This points to:
//                     - ext4_file_read() for ext4 files
//                     - nfs_file_read() for NFS files
//                     - proc_file_read() for /proc files
}
```

- VFS uses C-structs to achieve polymorphism
- Corresponds to virtual functions in C++, but they are manual

# VFS operation table

| Object | Operations struct | Key funtions |
|---|---|---|
| superblock | `alloc_inode()` | Create a new inode |
| | `write_inode()` | Sync inode to disk |
| | `put_super()` | Unmount cleanup |
| | | |
| inode | `create()` | Create a file |
| | `lookup()` | find file in a directory |
| | `mkdir()` | create directory |
| | `link()` | create hard link |
| | `unlink()` | delete a file |

# VFS operation table (contd …)

| Object | Operations struct | Key funtions |
|---|---|---|
| dentry | `d_revalidate()` | Check if still valid |
| | `d_hash()` | Custom hash function |
| | `d_compare()` | Custom name cmparison |
| | | |
| file | `open()` | Open a file |
| | `read()` | Read data |
| | `write()` | Write data |
| | `llseek()` | Change position of the file pointer |
| | `mmap()` | Memory map a file |
| | `release()` | Close a file |

# Example 1: ext4 (disk file system)

```
static struct file_system_type ext4_fs_type = {
    .owner = THIS_MODULE,
    .name = "ext4",
    .mount = ext4_mount, // ← Reads from disk
    .kill_sb = kill_block_super,
    .fs_flags = FS_REQUIRES_DEV, // ← NEEDS a block device
};
```

- `.mount` reads on-disk superblock from a device file (/dev/sda1), validates magic number, block size, etc, and loads root inode from disk
- `FS_REQUIRES_DEV` tells VFS: "I need a /dev/sda1 device"
- `.read` operation uses page cache for reading/writing data

**Q. What happens when you `cat` a file on ext4 for the first time vs. the second time?**

# Page cache comes to the rescue

- First time: Page cache miss → disk IO → slow
- Second time: Page cache hit → memory read → faster (up to 100x)

```
cat /home/alice/file.txt (first time)
        ↓
VFS: file->f_op->read()
        ↓
ext4_file_read()
        ↓
Check page cache → MISS
        ↓
Read from disk (slow!)
        ↓
Store in page cache
        ↓
Return to user
```

```
cat /home/alice/file.txt (second time)
        ↓
VFS: file->f_op->read()
        ↓
ext4_file_read()
        ↓
Check page cache → HIT!
        ↓
Return from memory (fast!)
```

# Example 2: NFS (network file system)

```
static struct file_system_type nfs_fs_type = {
   .name = "nfs",
   .mount = nfs_mount, // ← RPC to server, not disk!
   // .kill_sb is DIFFERENT & NO FS_REQUIRES_DEV flag!
};
static const struct inode_operations nfs_dir_inode_operations = {
   .lookup = nfs_lookup, // ← RPC to server
   .create = nfs_create, // ← RPC to server
   .link = nfs_link, // ← RPC to server
   // ...
};
```

- Same interface,  different backend

| Operation | ext4 | NFS |
|---|---|---|
| `.mount()` | Read disk superblock | RPC to server |
| `.lookup()` | Read disk directory | RPC to server |
| `.read()` | Read disk blocks | RPC to server |
| `FS_REQUIRES_DEV?` | Yes | No |

# Example: `cp /mnt/nfs/file.txt /mnt/disk/file.txt`

`NFS backend` `ext4`

**Users don’t know or care! VFS abstracts it all!**

```
cp /mnt/nfs/file.txt /mnt/disk/file.txt
Step 1: Open source (NFS)
       → nfs_file_operations.open()
Step 2: Read from NFS → nfs_file_operations.read()
       → Makes RPC to server
Step 3: Open destination (ext4)
       → ext4_file_operations.open()
Step 4: Write to ext4
       → ext4_file_operations.write()
       → Writes to disk
Step 5: Close both
       → nfs_file_operations.release()
       → ext4_file_operations.release()
```

# Example 3: procfs (pseudo file system)

```c
/* Example: /proc/cpuinfo */
static int cpuinfo_open(struct inode *inode, struct file *file) {
  return seq_open(file, &cpuinfo_op);
}
static const struct file_operations proc_cpuinfo_operations = {
  .open = cpuinfo_open,
  .read = seq_read, // ← seq_read is generic helper
  .llseek = seq_lseek,
  .release = seq_release,
};
```

- No storage at all! Data is generated on the fly

# Example: `cat /proc/cpuinfo`

```
Step 1: VFS calls .open()
  → cpuinfo_open()
  → Sets up "sequence file" helper
Step 2: VFS calls .read()
  → seq_read() (generic helper)
  → Calls cpuinfo_op.show()
  → This function:
       1. Reads kernel CPU structures
       2. Formats as text
       3. Copies to user buffer
  → NO disk I/O at all!
Step 3: User sees:
       processor : 0
       vendor_id :
       GenuineIntel cpu family: 6
       ...
```

# Power of VFS abstraction

- Disk file system (ext4): `read()` → disk → page cache → user
- Network file system (NFS): `read()` → RPC → cache → user
- Pseudo file system (proc): `read()` → generate on the fly → user

- All use the same VFS interface!

**Q. Why doesn't `/proc/cpuinfo` get cached in the page cache like regular files?**

# Procfs data changes

- CPU info can change over time:
  - CPU is hot-plugged
  - Frequency scaling changes
  - Kernel parameters changes
- The file is **virtual**: reflects current kernel static, not **static disk data**

**Q. But procfs does use seq_file, which has some internal buffering; why not use the page cache?**

ext4: page cache persists across reads (durable)

procfs: seq_file buffer is per-open file (temporary)

# Process-VFS relationship

# Three per-process structures

```
struct task_struct {
  // ... many other fields ...
  struct files_struct *files; // open file descriptors
  struct fs_struct *fs; // filesystem context
  struct nsproxy *nsproxy; // namespaces (includes mnt_namespace)
};
```

### 1. files_struct (Open file fescriptors)

- **fd 0 (stdin)** → file object
- **fd 1 (stdout)** → file object
- **fd 2 (stderr)** → file object
- **fd 3** → file object
- **fd N** → file object

each file objects points to dentry → inode

### 2. fs_struct (filesystem fontext)

- **root:** dentry of root directory
- **pwd:** dentry of current directory
- **umask:** default permissions

### 3. mnt_namespace (mount namespace)

- List of vfsmounts visible to this process
- (Used for container isolation!)

# 1. files_struct (include/linux/fdtable.h)

```
struct files_struct {
  atomic_t count; // reference count
  struct fdtable *fdt; // descriptor table
  // contains array: struct file *fd_array[]
};
```

- Array of struct file * pointers
- Index = file descriptor number
- Shared by threads, copied by processes

# 2. fs_struct (include/linux/fs_struct.h)

```
struct fs_struct {
  int users; // reference count
  struct path root; // root directory
  struct path pwd; // current working directory
  int umask; // default permissions mask
};
```

- Determines "**/**" and "**.**" for the process
- **chroot()** changes the permission
- **chdir()** changes the present working directory (**pwd**)

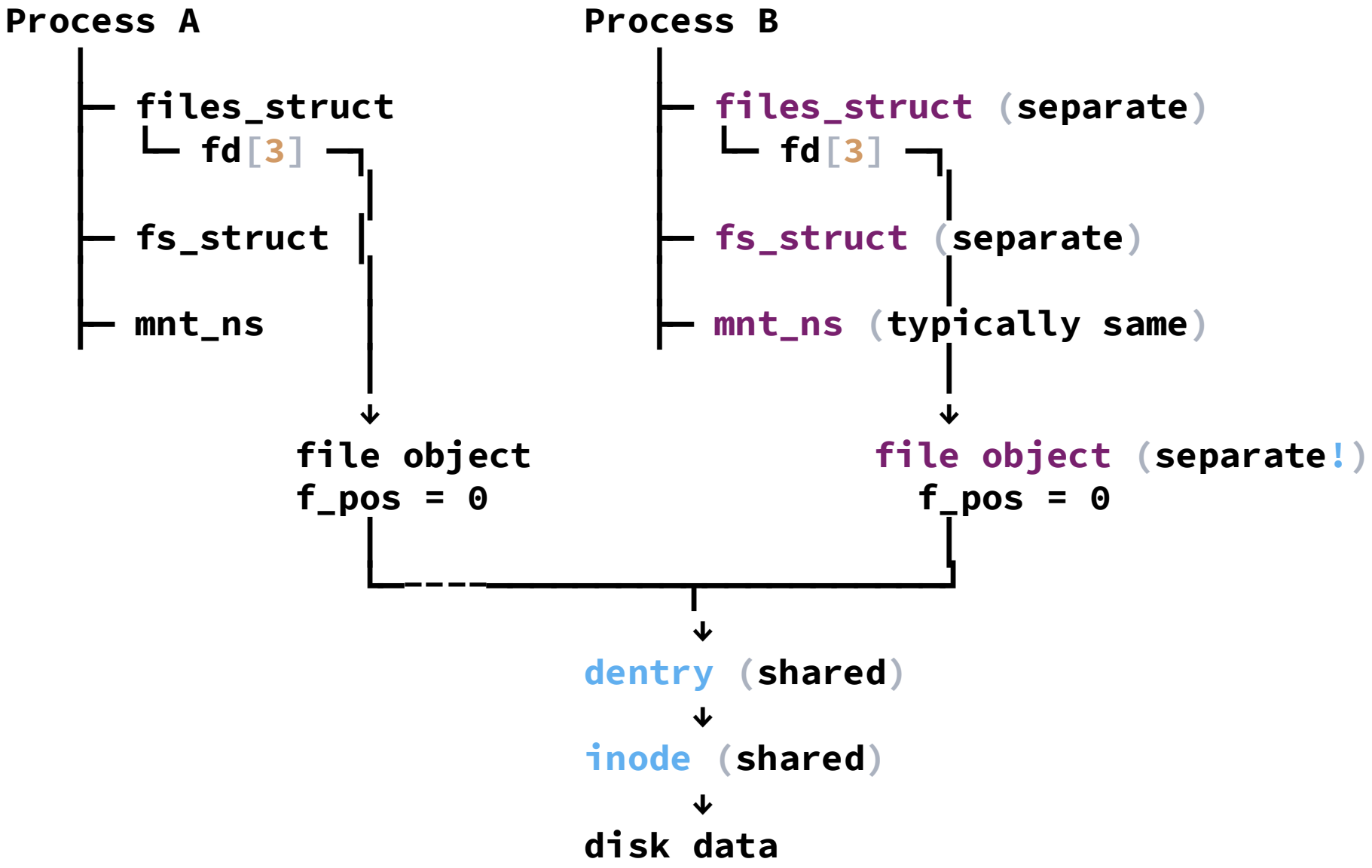
# 2. fs_struct (include/linux/fs_struct.h)

```
struct mnt_namespace {
  atomic_t count; // reference count
  struct mount *root; // root mount
  struct list_head list; // all mounts
  // ... other fields ...
};
```

- Determines which file systems are visible
- Enables container isolation (Docker)
- Inherited from parent, can be unshared via **`unshare(CLONE_NEWNS)`**

**Q. Two processes open the same file. Which structures are shared, and which are separate?**

# Structures shared and separate from a process view



# Complete path lookup!

# Walkthrough: `open("/home/cs477/test.txt")`

## Q. How many inode lookups? How many dentry cache lookups?

```
Step 1: Start at root
├─ current->fs->root → dentry("/")
└─ Lookup "home" in dcache
   ├─ Key = (dentry("/"), "home")
   └─ Result: MISS (assume cold cache)

Step 2: Call inode_operations->lookup()
├─ dir_inode = dentry("/")->d_inode
├─ dir_inode->i_op->lookup(dir_inode, dentry_lookup, ...)
└─ ext4_lookup() reads disk, finds inode #456 for "home"

Step 3: Create & cache dentry for "home"
├─ new_dentry = d_alloc()
├─ new_dentry->d_name = "home"
├─ new_dentry->d_inode = inode #456
├─ new_dentry->d_parent = dentry("/")
└─ d_add() → inserts into dentry_hashtable
```

# Walkthrough: `open("/home/cs477/test.txt")`

```
Step 4: Repeat for "lkp"
├─ Lookup "lkp" in dcache
│  └─ Key = (dentry("/home"), "lkp")
│  └─ Result: MISS
├─ Call inode(#456)->i_op->lookup()
├─ Find inode #789 for "lkp"
└─ Cache dentry("/home/lkp")

Step 5: Repeat for "test.txt"
├─ Lookup "test.txt" in dcache
│  └─ Key = (dentry("/home/lkp"), "test.txt")
│  └─ Result: MISS
├─ Call inode(#789)->i_op->lookup()
├─ Find inode #1234 for "test.txt"
└─ Cache dentry("/home/lkp/test.txt")
```

# Walkthrough: `open("/home/cs477/test.txt")`

```
Step 6: Create file object
├─ file_obj = alloc_file()
├─ file_obj->f_path.dentry = dentry("/home/lkp/test.txt")
├─ file_obj->f_path.mnt = current_vfsmount
├─ file_obj->f_op = inode(#1234)->i_fop (ext4_file_operations)
├─ file_obj->f_pos = 0
└─ file_obj->f_flags = O_RDONLY

Step 7: Call open operation
├─ file_obj->f_op->open(inode, file_obj)
└─ ext4_file_open() does filesystem-specific initialization

Step 8: Install file descriptor
├─ fd = get_unused_fd() // find free slot, e.g., 3
├─ current->files->fdt->fd[3] = file_obj
└─ return 3 to user space
```

# Walkthrough: `open("/home/cs477/test.txt")`

Cached!

```
Step 1: Lookup "/" → dcache HIT
Step 2: Lookup "home" → dcache HIT (inode #456 already cached)
Step 3: Lookup "lkp" → dcache HIT (inode #789 already cached)
Step 4: Lookup "test.txt" → dcache HIT (inode #1234 already cached)
Step 5: Create new file object (f_pos = 0 for this process)
Step 6: Call file_obj->f_op->open()
Step 7: Install fd and return
```

- Result: No disk IO, pure memory operations
- Comparison:
  - Cold cache: 3 disk reads (for 3 directory lookups)
  - Warm cache: 0 disk reads

# Further reading

- Linux storage stack explained
- VFS
- MGLRU