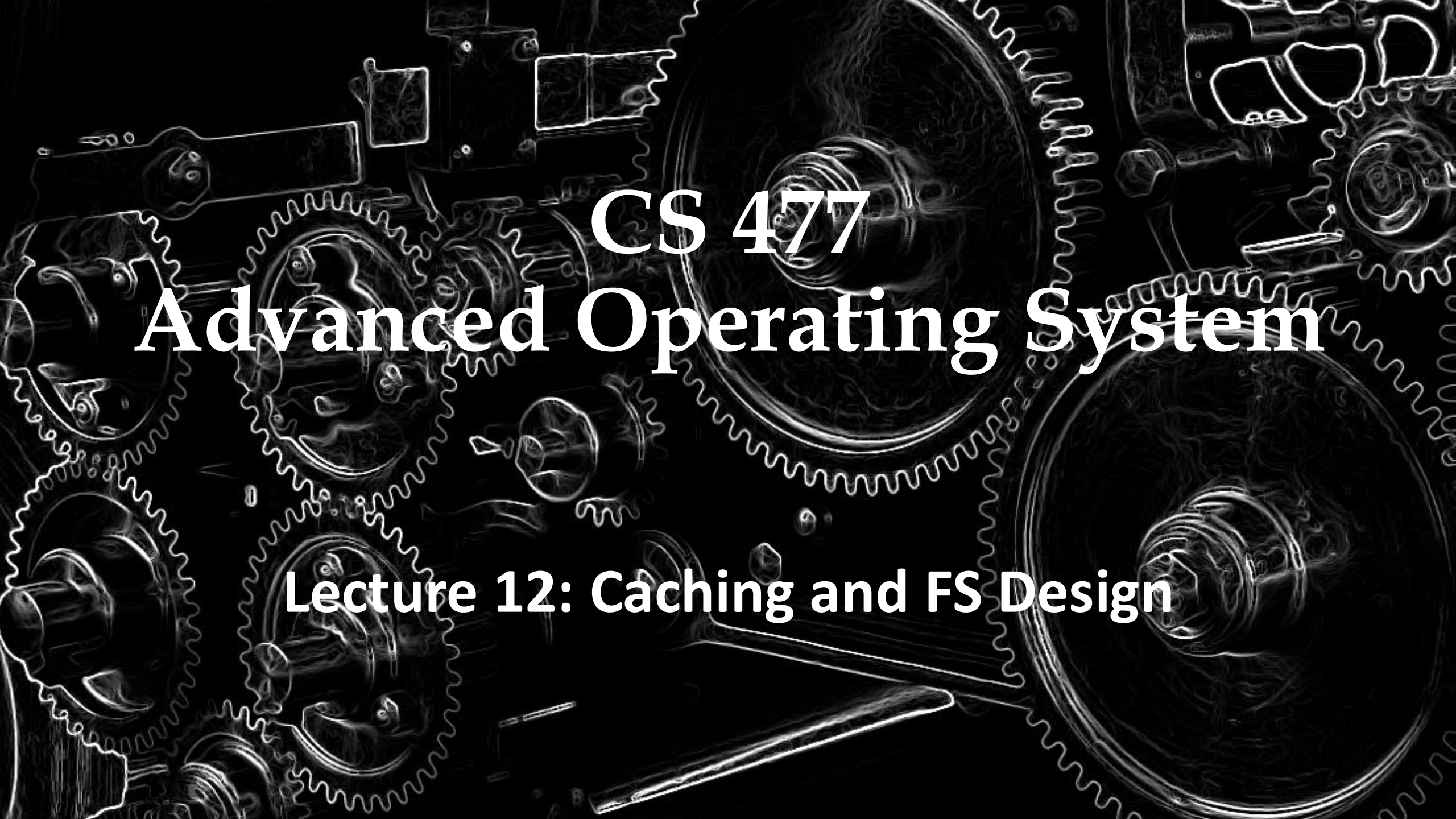
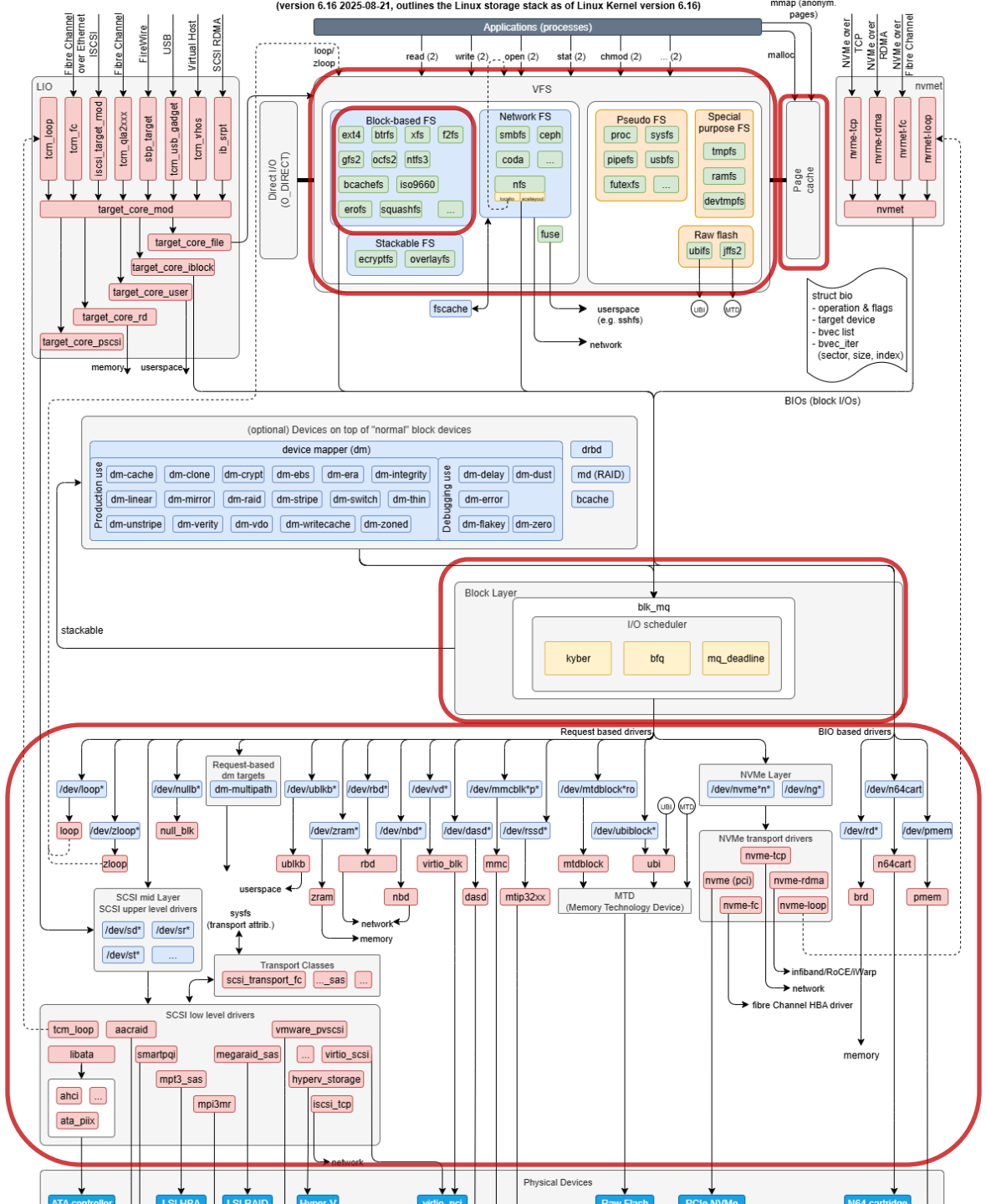


CS 477
Advanced Operating System

Lecture 12: Caching and FS Design

Today's agenda

- VFS recap
- Page caching
- File system design choices



Application

VFS

Page cache

File system

Block layer

Device driver

Recap: 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

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 context)

- **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**)

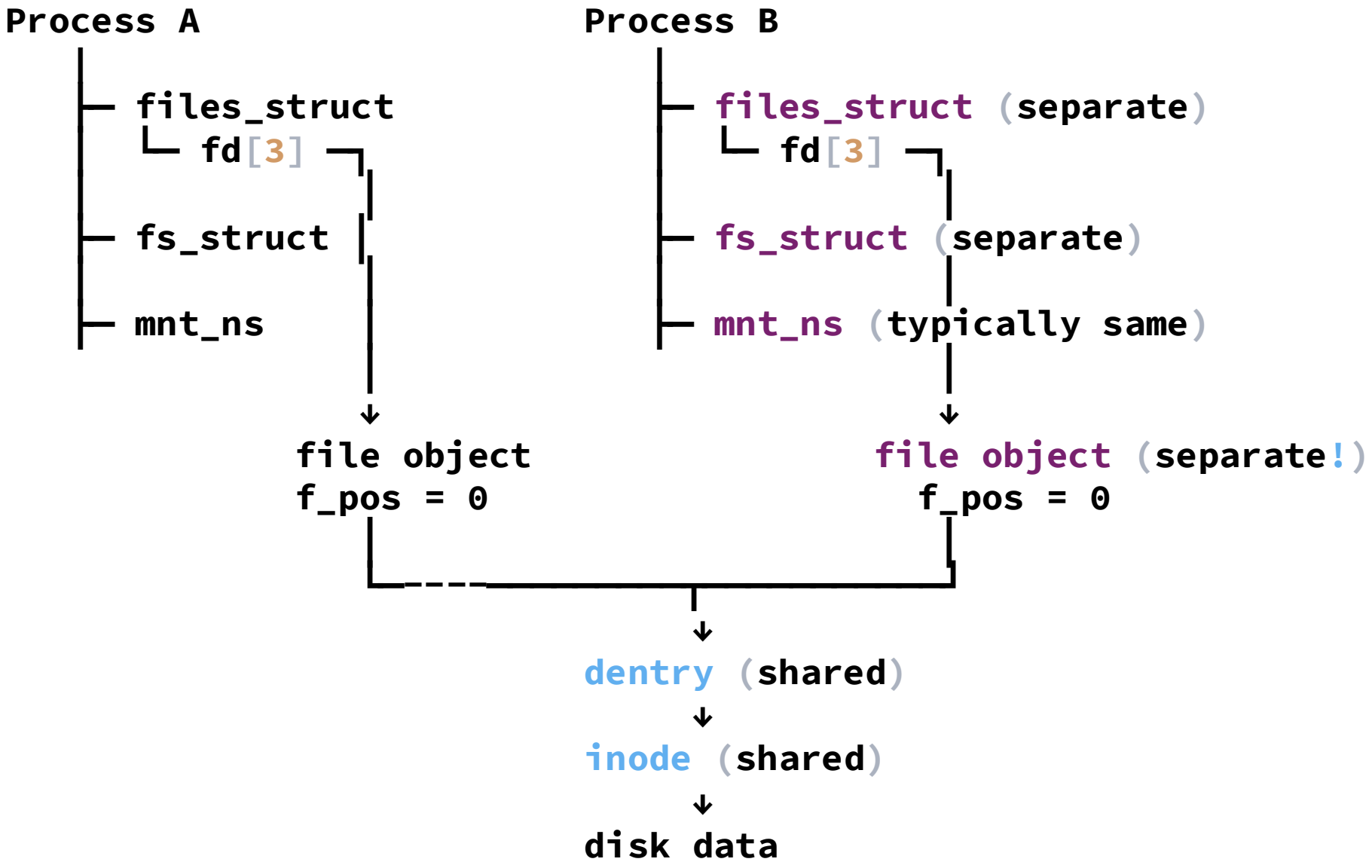
3. mnt_namespace (include/linux/mount.h)

```
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 "cs477"
├─ Lookup "cs477" in dcache
│  └─ Key = (dentry("/home"), "cs477")
│  └─ Result: MISS
├─ Call inode(#456)->i_op->lookup()
├─ Find inode #789 for "cs477"
└─ Cache dentry("/home/cs477")

Step 5: Repeat for "test.txt"
├─ Lookup "test.txt" in dcache
│  └─ Key = (dentry("/home/cs477"), "test.txt")
│  └─ Result: MISS
├─ Call inode(#789)->i_op->lookup()
├─ Find inode #1234 for "test.txt"
└─ Cache dentry("/home/cs477/test.txt")
```

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

```
Step 6: Create file object
├─ file_obj = alloc_file()
├─ file_obj->f_path.dentry = dentry("/home/cs477/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 ”cs477" → 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

Today's agenda

- VFS recap
- **Page caching**
- File systems mapping files to disk
- Some file systems comparison

The speed problem

L1 cache reference ..... 0.5 ns
L2 cache reference ..... 7 ns
Main memory reference ..... **100 ns**
SSD random read ..... 150,000 ns = 150 µs
Read 1 MB sequentially from SSD*..... **1,000,000 ns = 1 ms**

Q. If 10% of reads hit disk for reading, what's the effective latency?

A. .9 * 100ns + .1 * 150µs = 15.09µs

Source: Latency numbers every programmer should know

Where does cached data (page cache) live?

```
File: struct file (per open)
       ↓ f_mapping
Cache: struct address_space (per file)
       ↓ i_pages
Pages: Radix/XArray tree of pages
```

- One **`address_space`** per file
- Shared across all processes
- Contains **ALL** cached pages for that file

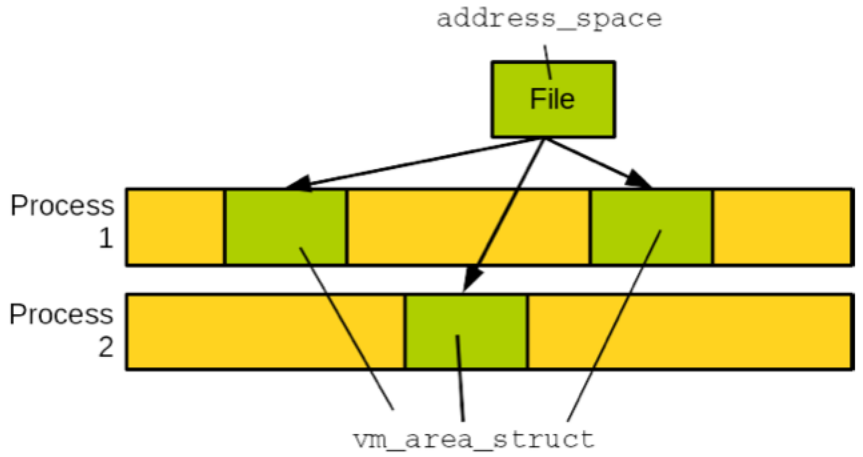
`address_space`: The page cache anchor (mapper)

- Core kernel abstraction → manages link between a cached object (usually a file) and its data in physical memory
  - an **`address_space`** → a file → access a page cache for a file
  - an **`address_space`** → one or more **`vm_area_struct`**
- Only one **`address_space`** structure for **`inode`** regardless of processes
  - Ensures data consistency among processes
- **Logical vs. virtual**
  - **`vm_area_struct`**: virtual address

  (where data appears in a process,

  `address_space`: logical offsets

  (where data sits in the file)



address_space for page cache

```
/* include/linux/fs.h */
struct address_space {
  struct inode *host;                 // Owning inode
  struct xarray i_pages;              // Page cache tree

  rwlock_t invalidate_lock;
  gfp_t gfp_mask;                     // Allocation flags
  atomic_t i_mmap_writable;           // Writable mmap count

  unsigned long nrpages;              // Total cached pages
  pgoff_t writeback_index;

  const struct address_space_operations *a_ops;
  //                                        ↑
  // Key operations: readpage, writepage, etc.
  errseq_t wb_err;                    // Writeback errors
};
```

Why is `i_pages` an Xarray, not a linked list?

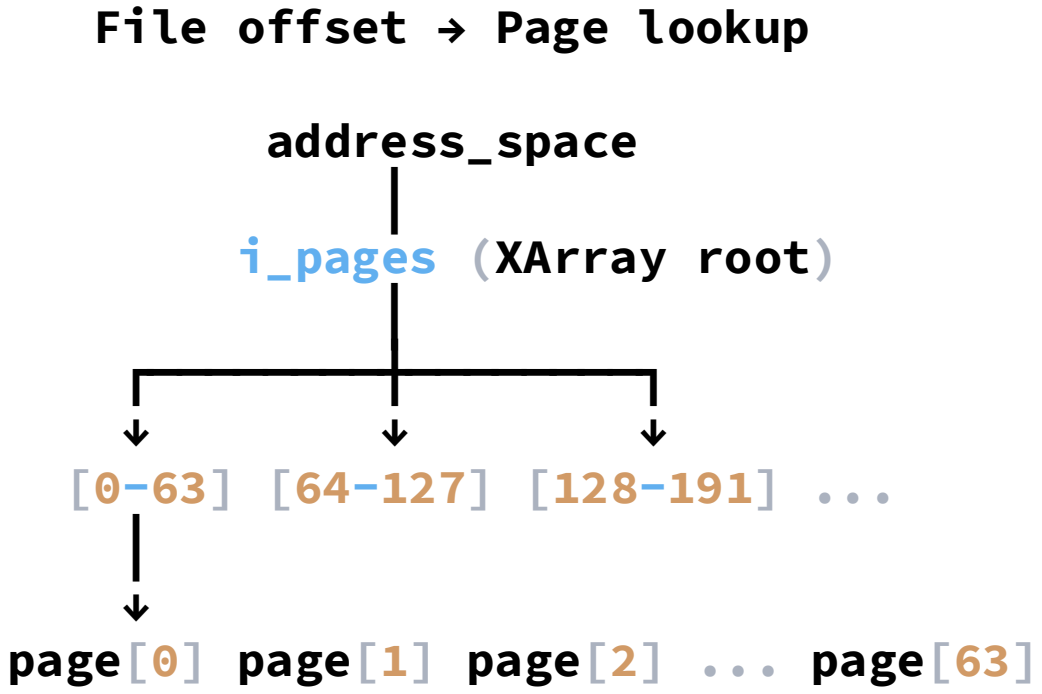
O(log n) lookup due to XArray

Example: Read offset 100KB

page_index = 100Kb / 4KB = 25

→Lookup page[25] in Xarray

→O(log n) instead of O(n)



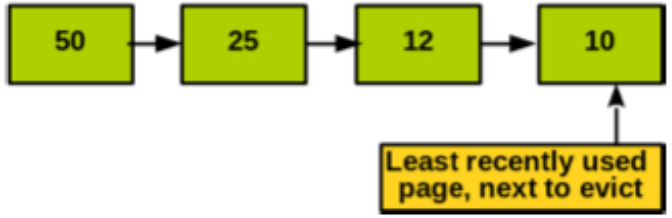
VFS write() path code walkthrough

```c
// 1. System call entry: fs/read_write.c
ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count) {
  struct file *f = fdget_pos(fd);
  return vfs_write(f.file, buf, count, &f.file->f_pos);
}

// 2. VFS layer: calls file operations
ssize_t vfs_write(struct file *file, ...) {
  return file->f_op->write_iter(kiocb, iter);
  // ↑ ext4_file_write_iter() for ext4
}

// 3. Generic buffered write: mm/filemap.c
ssize_t generic_perform_write(struct kiocb *iocb, ...) {
  // Find or create page in cache
  page = grab_cache_page_write_begin(mapping, index);
  // Copy user data to page copy_from_user(page_address(page) + offset, buf, len);
  // Mark page as dirty
  set_page_dirty(page); // ← Key step!
  // NOT written to disk yet!
}
```

After **`write()`** returns, is data on disk?

Page cache eviction policy: LRU

- Least recently used (LRU)
  - Keep track of when each page is accessed
  - Evict the page with the oldest timestamp



- Failure cases of LRU policy
  - Many files are accessed once and then never again
  - LRU puts them at the top of LRU list → not optimal
  - Example:
    - DB: 10 GB working set, frequently accessed pages (hot)
    - Backup: Scans all files once for backup

Q. How to address this limitation?

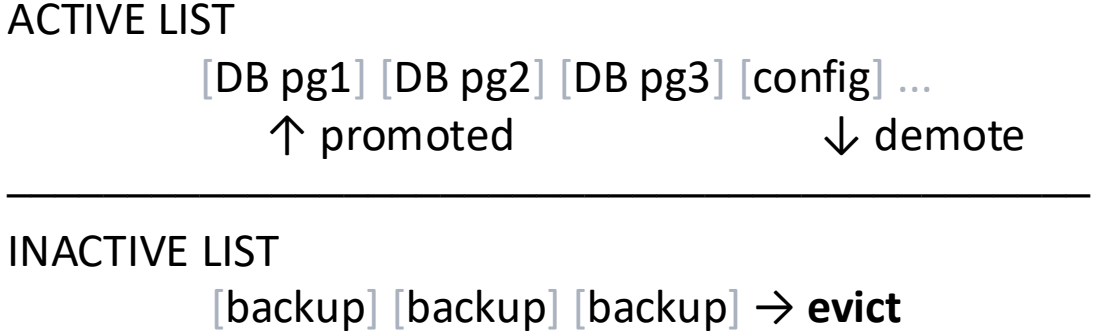
The two-list strategy

- **Active list**
  - Pages in the active list: **hot**
  - Not available for eviction
- **Inactive list**
  - Pages in the inactive list: **cold**
  - Available for eviction

```
/* Two-list LRU: scan resistance */
struct lruvec {
  struct list_head lists[NR_LRU_LISTS];
  // LRU_INACTIVE_FILE ← Eviction candidates
  // LRU_ACTIVE_FILE ← Hot pages
};
```

The two-list strategy: Page life cycle

1. First access: → inactive list
   `unmark_page_accessed(page);`
2. Second **access** (while still in cache):
   → Promote to active list
   `activate_page(page);`
3. Memory pressure:
   → Demote from active list to inactive list
   → Evict from Inactive



One-time backup scan: stays in inactive list → evicted
Database page: promoted to active → protected!

Q. Suppose you run a web server, which serves some pages hourly, others once per day. Should they have the same eviction priority?

Traditional LRU problems

1. Binary decision: active or inactive?
   - Too coarse grained
2. Scan pollution:

   ```
   find / -name "*.log"
   ```

   - Millions of pages accessed once
   - Evict frequently-used pages
3. No age information
   - Page accessed 1 second ago = page accessed 1 hour ago
   - Both in the inactive list

Multi-generation LRU (MGLRU)

```c
/* mm/vmscan.c - Multi-generational LRU */
#define MAX_NR_GENS 4 // Typical: 4 generations
struct lru_gen_folio {
  unsigned long max_seq; // Youngest generation
  unsigned long min_seq; // Oldest generation

  /* Per-generation lists */
  struct list_head folios[MAX_NR_GENS];
};
```

Why 4 generations? Why not 2 or 10?

Trade-off between **precision and overhead**

Key idea: **fine-grained aging**

Gen 0 (youngest) ← Just accessed

Gen 1 ← Recently used

Gen 2 ← Older

Gen 3 (oldest) ← Evict from here

How does MGLRU know pages are accessed?

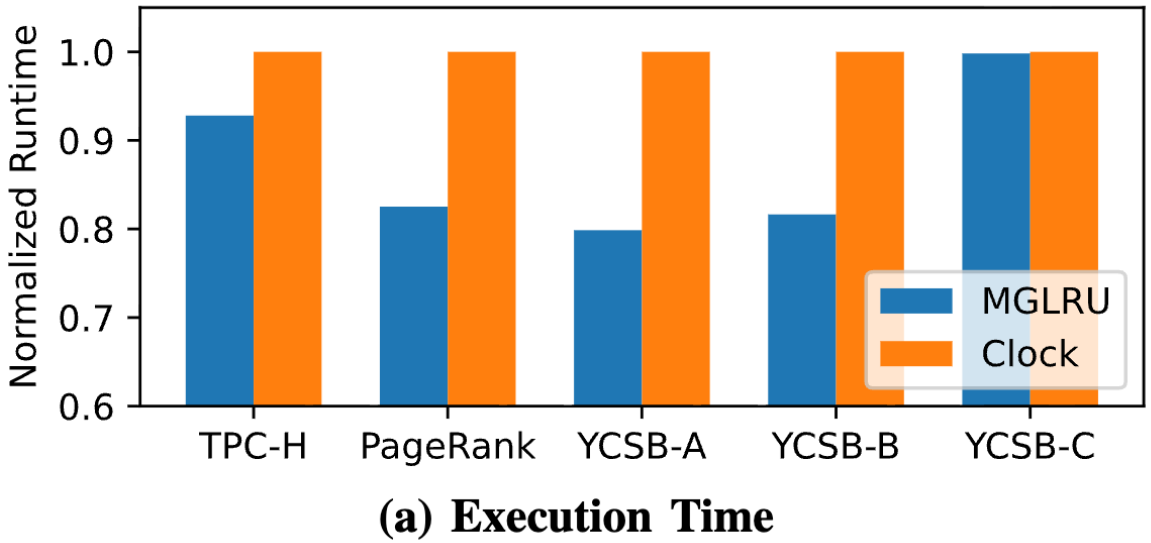
```
// 1. Hardware Access Bit (x86 PTE)
struct page_table_entry {
  unsigned long present : 1;
  unsigned long rw : 1;
  unsigned long accessed : 1; // ← MMU sets this! ...
};

// 2. Periodic Scanning
void lru_gen_age_node(struct pglist_data *pgdat) {
  // Walk page tables
  // Check accessed bits
  // Promote to younger generation if accessed
  // Clear accessed bit
}

// 3. Generation Transition
Age 0 → Age 1 → Age 2 → Age 3 → Evict (if not accessed)
```

So-called MGLRU benefits

1. Better scan resistence:

   `find / -name "*.log"`

   - One time-access → stays in Gen 3
   - Does not evict hot pages from Gen 0—1
2. Workload-aware
   1. Database: Keep working set in Gen 0—1
   2. File server: Promote popular files quickly
   3. Container VM: Seprarate generations per cgroup
3. Performance



(a) Execution Time

Characterizing Emerging Page Replacement Policies for Memory-Intensive Applications

Reading data using page cache

```c
// fs/read_write.c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
  // Calls file->f_op->read_iter()
  return file->f_op->read_iter(iocb, iter);
}

// mm/filemap.c - Generic implementation
ssize_t generic_file_read_iter(...) {
  // 1. Check page cache first
  page = find_get_page(mapping, index);
  if (page) {
    // Cache HIT - copy to user
    lru_gen_mark_accessed(page); // ← Update MGLRU!
    copy_page_to_iter(page, iter);
  } else {
    // Cache MISS - read from disk
    page = page_cache_alloc();
    a_ops->readpage(file, page); // → ext4_readpage()
    lru_gen_add_folio(page, 0); // ← Add to Gen 0
  }
}
```

When does this **cached page get flushed to the** disk?

Flush data by tracking dirty pages

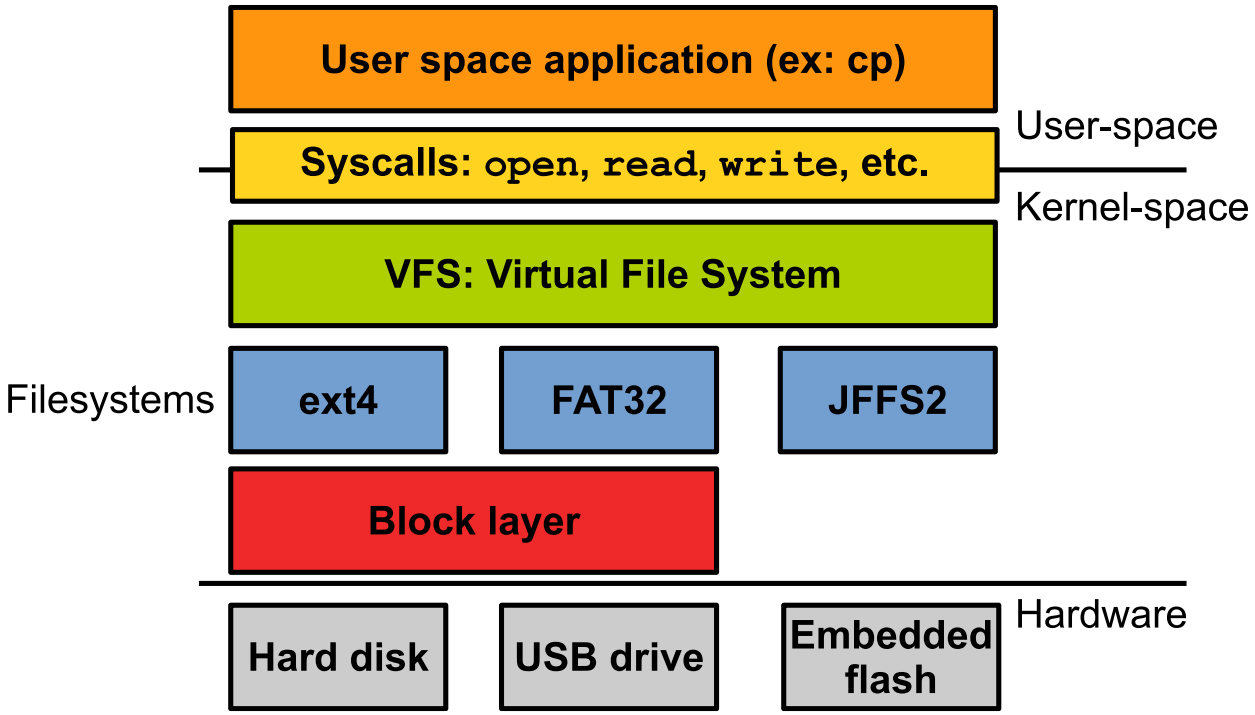
- Track the **dirty bit** in each cached page
  - **`PG_DIRTY`**: modified in cache
  - **`PG_WRITEBACK`**: IO in progress
- 3 triggers to write dirty pages
  1. **`sync`**()/**`fsync`**(): explicitly from the application
  2. Memory pressure: other applications need RAM
  3. Flusher threads: periodic background flushing
     - One flusher per backing device (disk)
     - Wakes up every **`dirty_writeback_interval`** (5 sec default)
     - Asynchronous background flushing (**`dirty_background_ratio`**);
       Can synchronously flush data if exceed some dirty page ratio (**`dirty_ratio`**);
       And after how long dirty pages must be forced (**`dirty_expire_centisecs`**)

Check `/proc/sys/vm/*`

Today's agenda

- VFS recap
- Page caching
- **File systems mapping files to disk**
- Some file systems comparison

Connecting VFS to physical storage

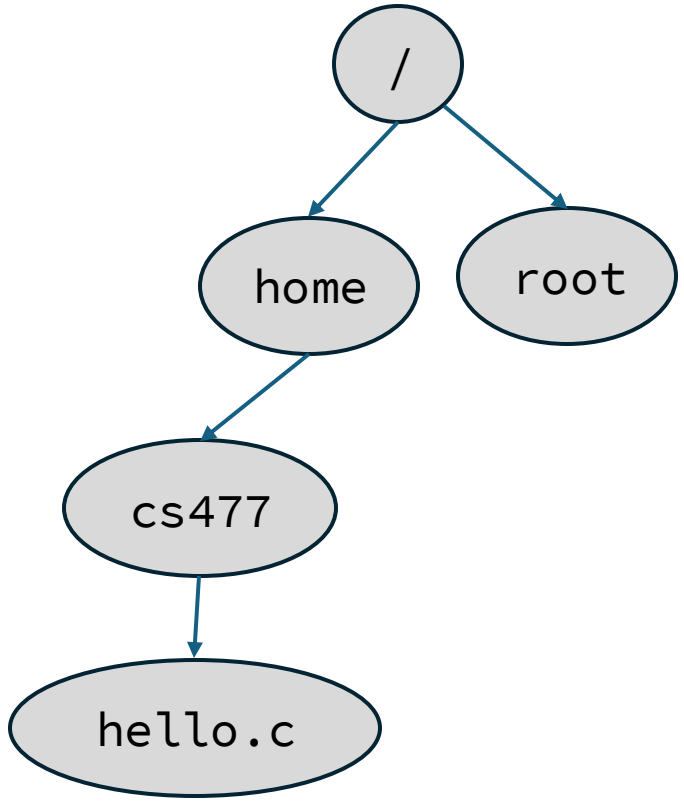



```
User:       read(fd, buf, 4096)
                  ↓
VFS:        file → inode → address_space
                  ↓
Question:   Which DISK BLOCK?
                  ↓
FS:         Inode block map → Block 50000
                  ↓
Block:      submit_bio() → disk
```

Q. How does FS map a file offset to a disk block?

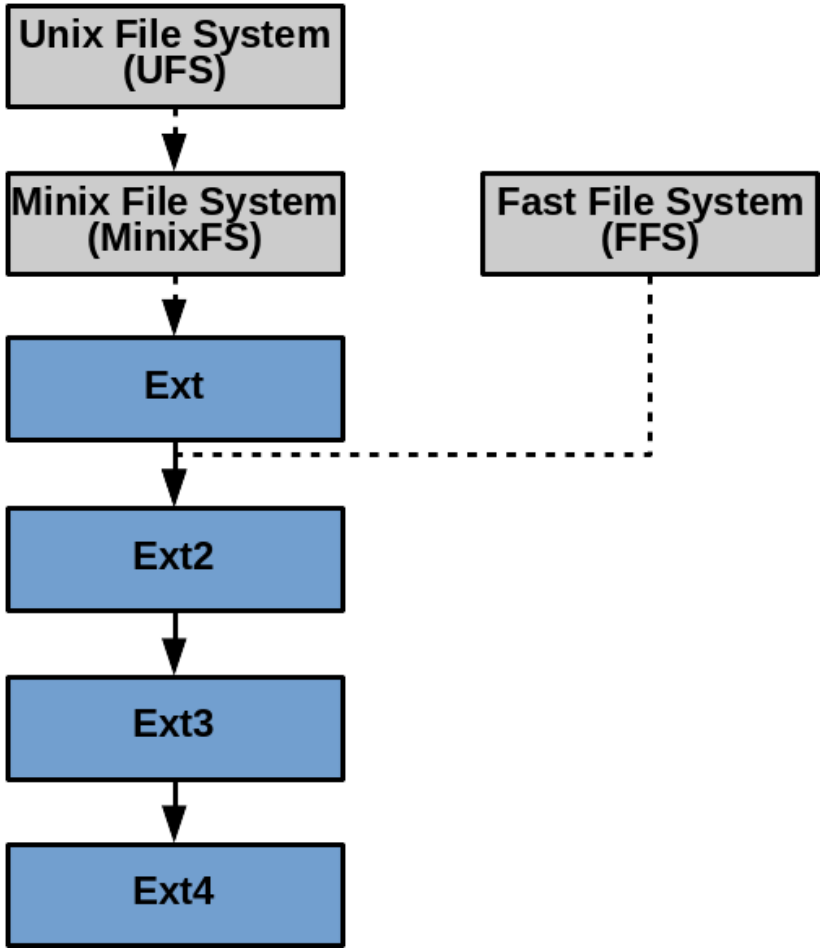
What file systems provide …

```c
int main(int argc, char *argv[]) {
  int fd; char buffer[4096];
  struct stat stat_buf; DIR *dir;
  struct dirent *entry;

  /* 1. Path name → inode mapping */
  fd = open("/home/cs477/hello.c", O_RDONLY);

  /* 2. File offset → disk block address mapping */
  pread(fd, buffer, sizeof(buffer), 0);

  /* 3. File metadata operation */
  fstat(fd, &stat_buf);
  printf("file size = %d\n", stat_buf.st_size);

  /* 4. Directory operation */
  dir = opendir("/home");
  entry = readdir(dir); printf("dir = %s\n", entry->d_name);
  return 0;
}
```



Tree hierarchy

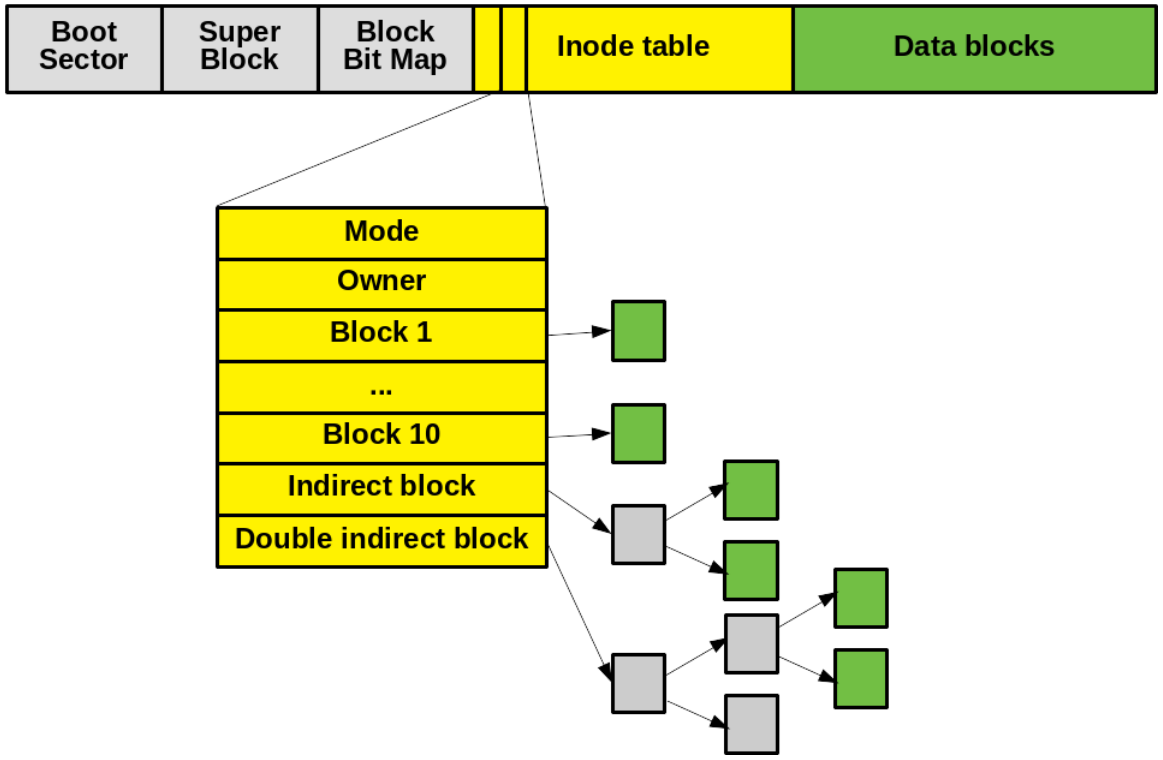
History of file system design



- File systems have to care about:
  - Mapping to map file content to data blocks
  - When to allocate data
  - Representing tree hierarchy efficiently (directory)
  - Being crash consistent
- Ensure several performance metrics:
  - Latency, throughput, fragmentation

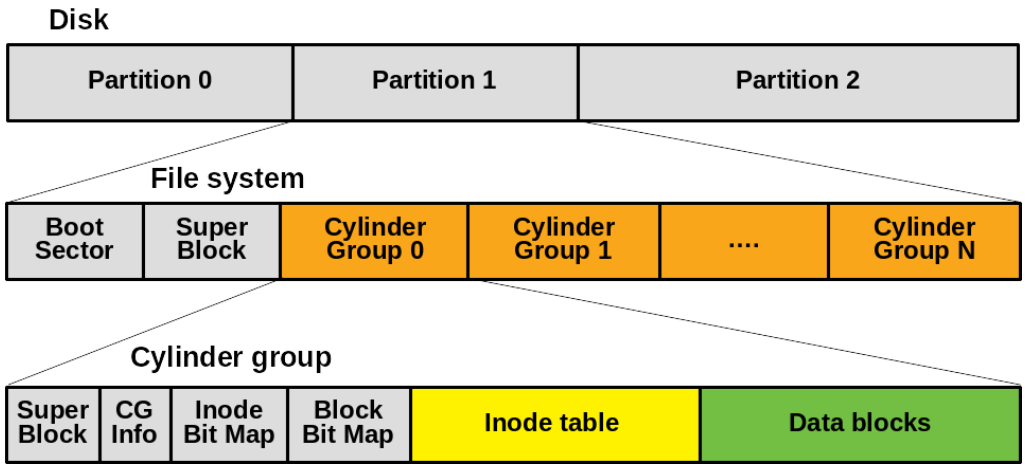
Unix file system (UFS)

- Original Unix file system designed in 1974
  - Authors: Dennis Ritche & Ken Thompson (invented C, Unix OS)
- Issue: long seek time between inode table and data blocks
  - Seek to inode table (outer tracks)
  - Read inode
  - Seek to data blocks (anywhere)
  - Read data
  - HDD: ~10ms per seek
  - Two seeks per file: 20ms overhead



Fast file system (FFS)

- File system of BSD UNIX (1984)
- Used the idea of **locality**
- **Cylinder group: one or more consecutive cylinders**
  - Disk block allocation heuristics to reduce seek time
  - Try to locate inode and associated data in the same cylinder group
- Several in-place file systems follow this design



Allocation policy:

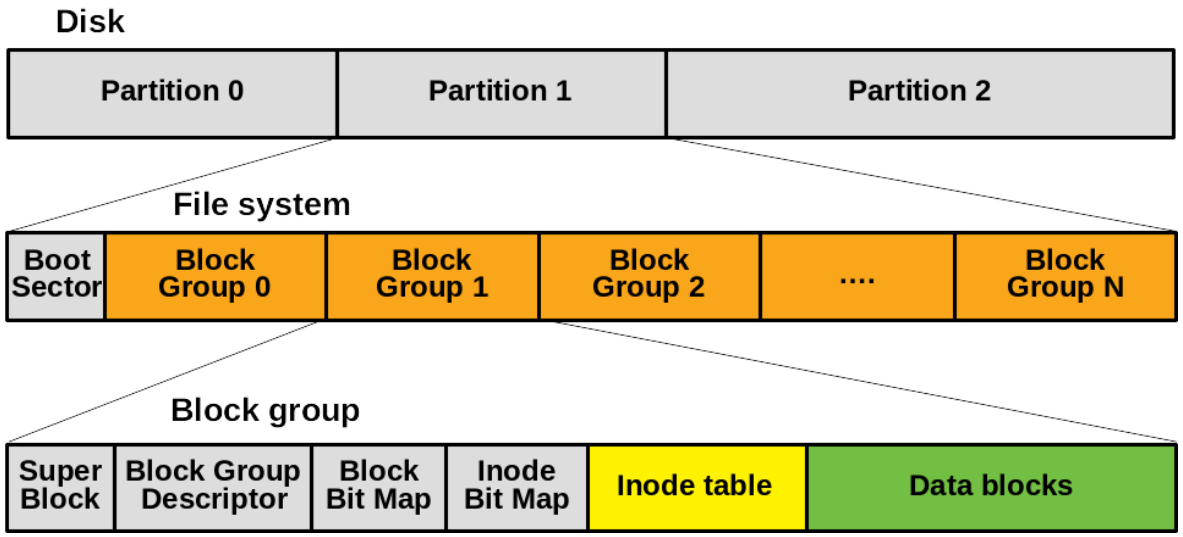
1. Put file's inode in parent dir's CG
2. Put file's data block near inode
3. Spread directories across CG

Result: inode + data in same CG

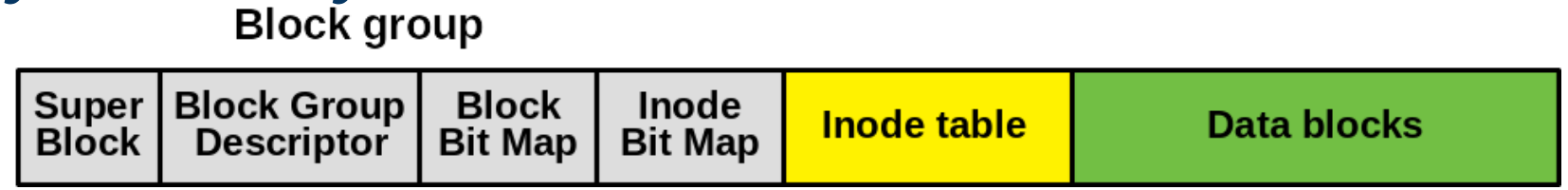
- Read time: 10ms → 1ms

Ext2 file system

- The second extended version (ext2) of the first Linux file system (ext)
  - Block group to reduce disk seek time
  - Abstracted from physical geometry
  - Same locality benefits
  - All block groups have same size and stored sequentially (128MB block)



Ext2 file system: layout information



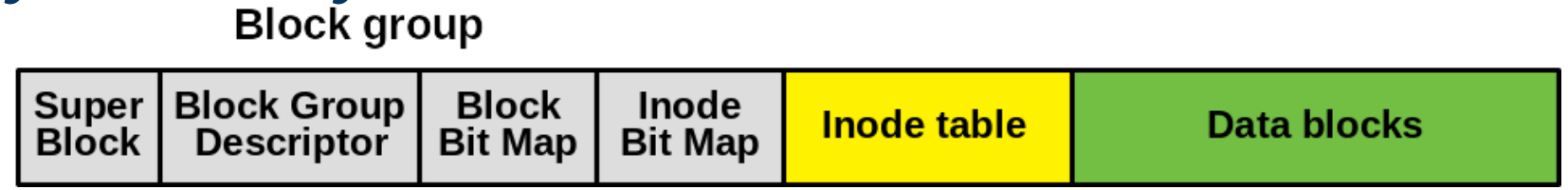
- Superblock: stores information describing the file system

```
struct ext2_super_block {
  __le32 s_inodes_count; // Total inodes
  __le32 s_blocks_count; // Total blocks
  __le32 s_free_blocks_count; // Free blocks
  __le32 s_free_inodes_count; // Free inodes
  __le32 s_log_block_size; // Block size (log2)
  __le32 s_blocks_per_group; // Blocks per group
  __le32 s_inodes_per_group; // Inodes per group
  __le16 s_magic; // 0xEF53
  __le16 s_state; // Mount state
  // ... 100+ more fields
};
```

What is this **__le32**?

little-endian format; disk structures must work across all architectures

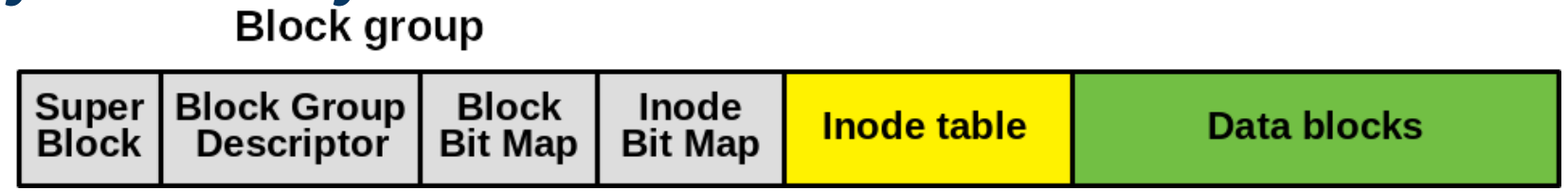
Ext2 file system: layout information



- Group descriptor: describes each block group information

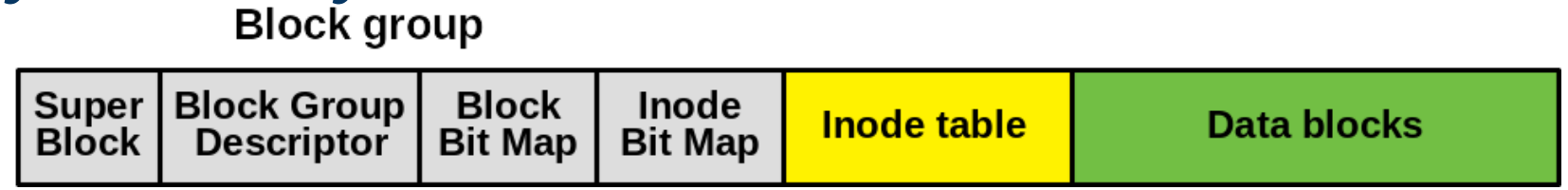
```
/* Group Descriptor: Per-group metadata */
struct ext2_group_desc {
  __le32 bg_block_bitmap; // Block bitmap location
  __le32 bg_inode_bitmap; // Inode bitmap location
  __le32 bg_inode_table; // Inode table location
  __le16 bg_free_blocks_count; // Free blocks in group
  __le16 bg_free_inodes_count; // Free inodes in group
  __le16 bg_used_dirs_count; // Directories in group
};
```

Ext2 file system: layout information



- Inode bitmap: Inode usage information
  - 0 → free; 1 → in use
  - 100th bit in inode bitmap → using 100th inode in the inode table
- Block bitmap: data block usage information
  - 0 → free; 1 → in use
  - 100th bit in inode bitmap → using 100th data block in the data group

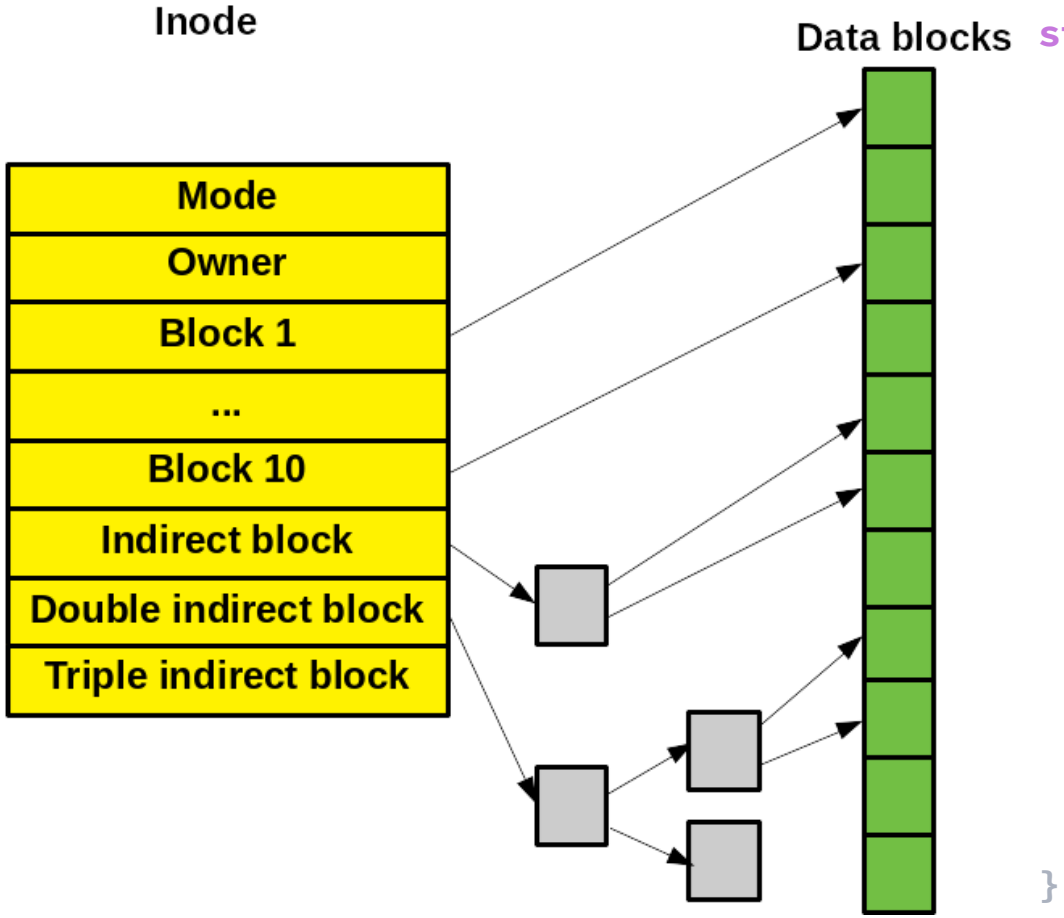
Ext2 file system: layout information



- Inode table: fixed-size array of inodes
  - Inode number = starting inode number of a block group + indoe location in inode table
- Data blocks: actual data contents

Ext2 file system: layout information

- Inode indirect block map: file offset → disk block address mapping

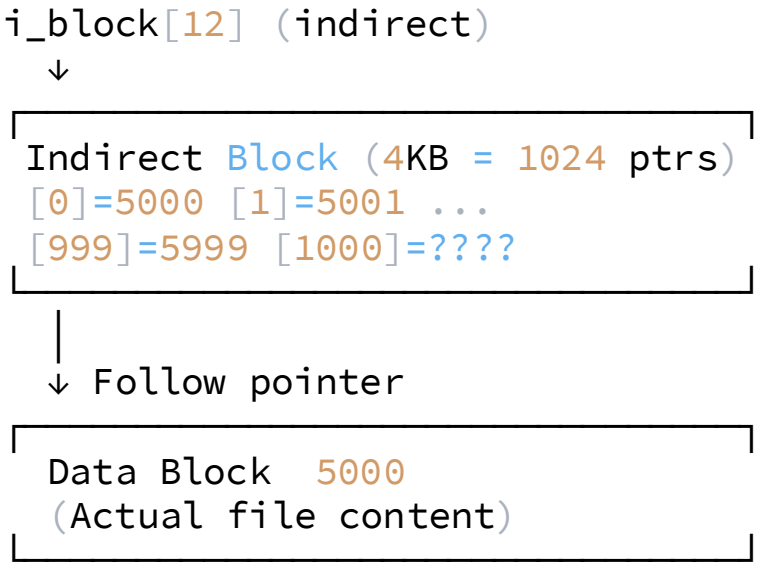



```c
struct ext2_inode {
    __le16 i_mode; // File type + permissions
    __le16 i_uid; // Owner UID
    __le32 i_size; // File size (bytes)
    __le32 i_atime; // Access time
    __le32 i_ctime; // Change time
    __le32 i_mtime; // Modify time
    __le32 i_dtime; // Delete time
    __le16 i_gid; // Owner GID
    __le16 i_links_count; // Hard link count
    __le32 i_blocks; // Block count (512-byte units)
    __le32 i_flags; // Flags
    __le32 i_block[15]; // Block pointers!
    /*
     * [0-11]: Direct blocks → 48 KB
     * [12]: Indirect → 4 MB
     * [13]: Double indirect → 4 GB
     * [14]: Triple indirect → 4 TB
     */
};
```

Q. How many disk reads for indirect, double indirect, and triple indirect blocks?

Indirect block addressing

- Reading file block 1000 (offset 4 KB)
- Cost: 2 disk reads (indirect + data)
- For double indirect: 3 reads
- For triple indirect: 4 reads



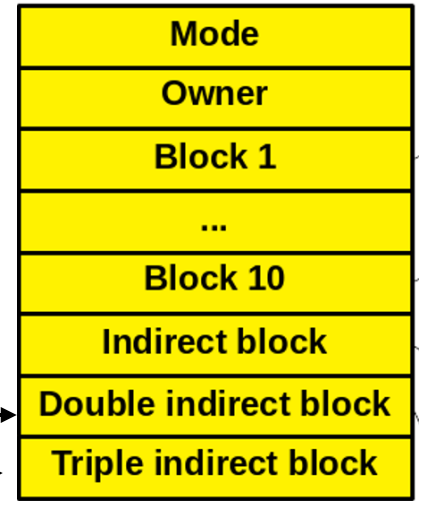
(assuming this information i_block information is already cached in DRAM)

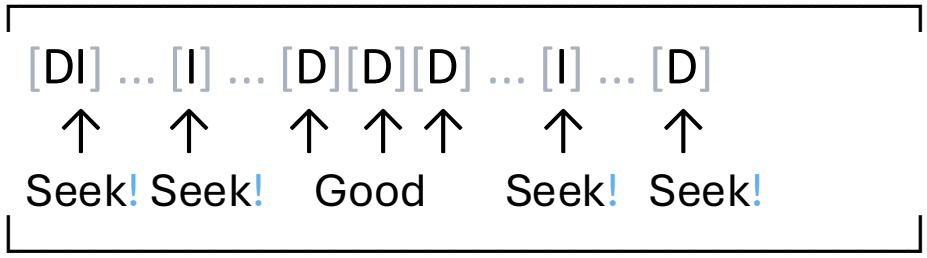
Q. What are the issues with indirect blocks?

Performance issues with indirect blocks

Example: 10GB file (sequential read)

- 10GB = 2.6 M blocks
  - Each double-indirect covers 1024^2 (= 1M) blocks, 4 GB
    - Need 3 double-indirect blocks
    - 1-double indirect blocks
    - 2-double indirect blocks from triple-indirect block table
  - Metadata overhead: 3 * 1024: ~12 MB for 10 GB file
- **Worse:** Metadata scattered on disk; affects seek time and deletion





Q. Can we do better?

Extents (ext4)

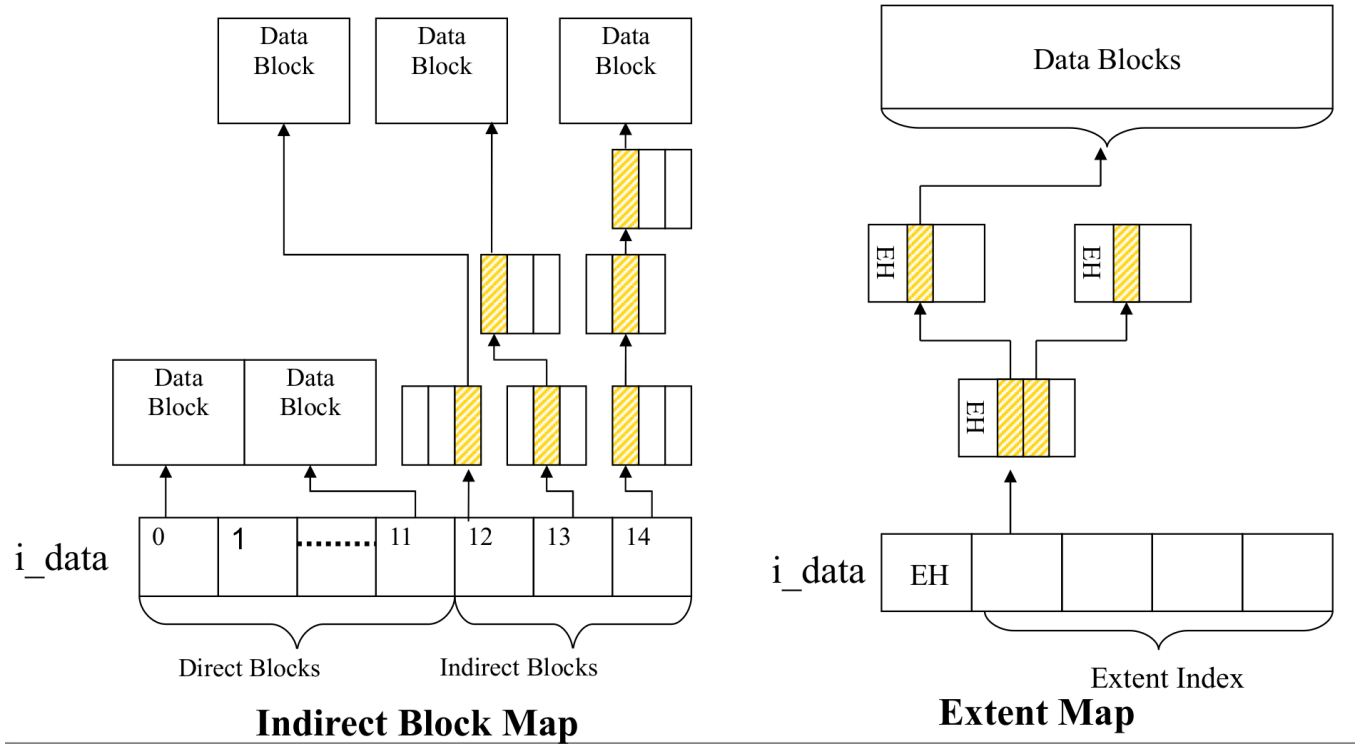
```
/* Extent: Map a contiguous range of blocks using a single descriptor */
struct ext4_extent {
  __le32 ee_block; // Logical block (file offset)
  __le16 ee_len; // Number of blocks
  __le16 ee_start_hi; // Physical block (high 16)
  __le32 ee_start_lo; // Physical block (low 32)
};
```

Example: 100 MB contiguous file

- Indirect blocks: 25,600 block pointers → 100 KB metadata
- Extents: maintains a tuple of **{logical offset, length, physical address}**
  - {length=0, len=25600, phys=500000} (need only 12 bytes)
  - Can store up to 128 MB chunks contiguously

- Benefits:
  1. Less metadata
  2. Contiguous allocation
  3. Faster sequential access
  4. Lower fragmentation

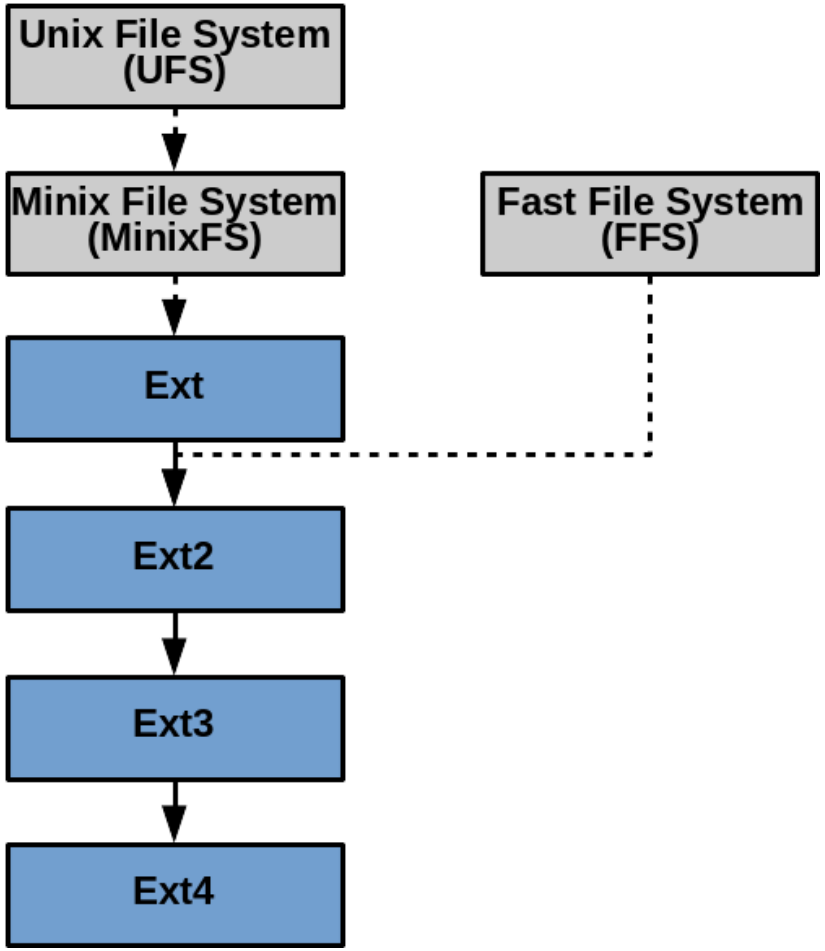
Extent tree visualization




```
struct ext4_inode {
 __le32 i_block[15]; // Reused!
};
```

- Small files are kept in inode (<= 4 extents)
- Large files use the extent tree
  - Max depth: 5 levels ➔ max file size: 16 TB (with 4 KB blocks)

History of file system design



- File systems have to care about:
  - Mapping to map file content to data blocks
  - **When to allocate data**
  - Representing tree hierarchy efficiently (directory)
  - Being crash consistent
- Ensure several performance metrics:
  - Latency, throughput, fragmentation

Allocating file data

- ext2/3 file systems use an immediate allocation strategy

```
for (i = 0; i < 1000; i++) {
  write(fd, buf, 4096);  // Each write → allocate block!
}
```

- Result: 1000 separate allocations → potentially fragmented!

Q. How to address this problem?

- Use the idea of **delayed allocation**

```
for (i = 0; i < 1000; i++) {
  write(fd, buf, 4096);
  // Just copy to page cache
  // Mark dirty, don't allocate yet
}
// ... later, flusher runs ...
// See 4MB pending → allocate one extent!
```

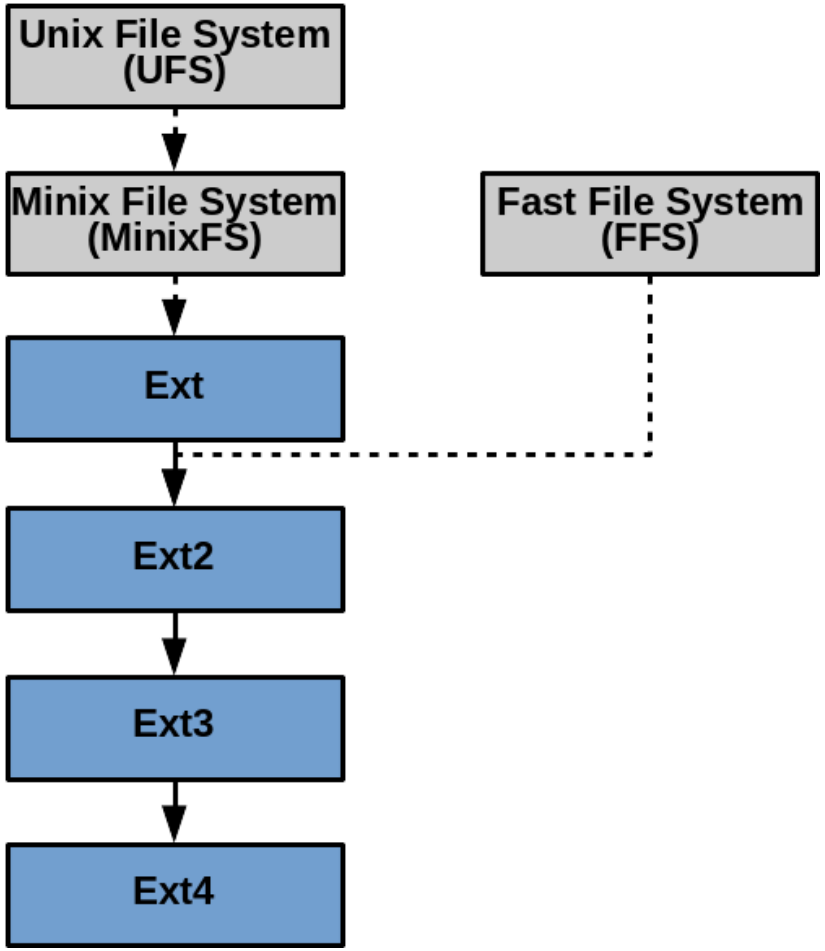
- **Result:** 1 contiguous 4 MB allocation

Delay allocation advantages

- Scenario 1: Temporary file deleted early
  - **`create() → write() → delete()`**
    `(never flushed)`
  - Result: 0 disk IO

- Scenario 2: Growing a file
  - ext2/3: Allocate 1 block at a time
    - Many small extents
  - ext4: wait, see total size
    - Allocate large extent

History of file system design



- File systems have to care about:
  - Mapping to map file content to data blocks
  - When to allocate data
  - **Representing tree hierarchy efficiently (directory)**
  - Being crash consistent
- Ensure several performance metrics:
  - Latency, throughput, fragmentation

Directory: simple linked list

- Directory access is **linear time**

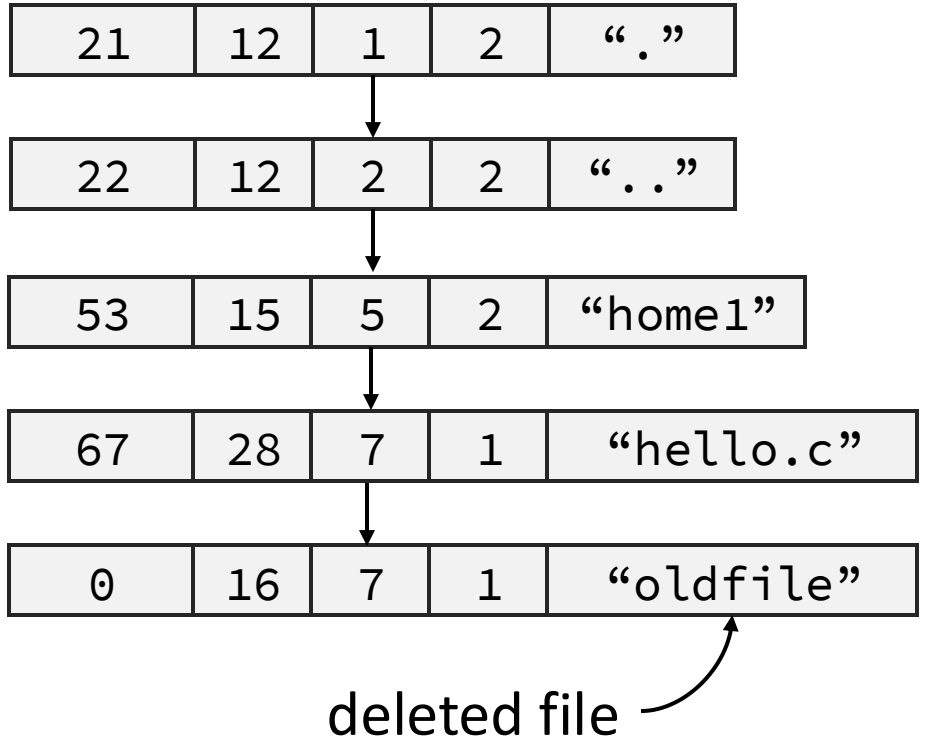
Q. What is the issue with linear time?

- Example: 100K files in a directory
  - Cost: O(n) comparisons, can take minutes
- This easily happens:
  - Mail servers have 1M files in a directory

```
$ touch /var/spool/mail/newfile
```

  - First, scan the entire directory to check if newfile exists
  - Then, add entry
  - Total cost: O(n) for every file operation

```
struct ext2_dir_entry_2 {
  __le32 inode;          // Inode number
  __le16 rec_len;        // Record length
  __u8 name_len;         // Name length
  __u8 file_type;        // File type
  char name[];           // Filename
};
```



Q. How do we minimize the scan time?

Hashed btree (Htree) for directories

- Compute hash of filename and store in the tree as its key
  - Hashing converts variable string to fixed int
  - Bucket A (block 10): hashes 0–0x3fffffff
  - Bucket B (block 11): 0x4000000—0x7fffffff
  - The leaf block (block 10) contains actual directory entries (name, inode number)

```
Root Block (index)
hash < 0x40000000 → Block 10
hash < 0x80000000 → Block 11
hash < 0xC0000000 → Block 12
hash ≥ 0xC0000000 → Block 13
```

↓ hash("hello.c") = 0x3A7F2B1C

```
Block 10 (leaf entries)
[hello.c, inode=100] [holiday.c, i=200]
```

- O(log n) lookup!
  - Brings the lookup time from minutes to milliseconds!

Summary: from inodes to extents

- UFS (1974): Simple but slow (metadata far from data)
- FFS (1984): Cylinder groups → locality (10x performance improvement)
- ext2 (1993): FFS for Linux
  - Block groups, bitmap, indirect blocks
- ext4 (2000):
  - Extents → less metadata, contiguous
  - Delayed allocation → less fragmentation
  - HTree → O(log n) directory lookup

Further reading

- An introduction to Linux's EXT4 filesystem
- Ext4: The Next Generation of Ext2/3
- Ext4 Disk Layout
- Chapter 9. The Extended Filesystem Family in Professional Linux Kernel Architecture
- Chapter 18. The Ext2 and Ext3 Filesystems in Understanding Linux Kernel