🎯 Project Charter: Filesystem Implementation

What You Are Building

You are building a fully functional, on-disk, inode-based filesystem that integrates directly with the Linux kernel via FUSE. Unlike a simple memory-based simulation, your filesystem will store data in a persistent 4KB-block-aligned disk image, support multi-level indirect block pointers for large files, provide a hierarchical directory tree, and implement a circular write-ahead journal (WAL) to ensure metadata remains consistent even if the power is cut mid-write.

Why This Project Exists

Most developers treat the filesystem as a reliable, infinite, byte-addressable array, but storage hardware is actually a rigid, finite collection of blocks. Building a filesystem from scratch forces you to bridge this gap manually. You will learn to negotiate the "read-modify-write" cycles inherent in SSDs, implement the same indirection logic used in CPU page tables, and master the atomic commit protocols that prevent data corruption in production databases like PostgreSQL and SQLite.

What You Will Be Able to Do When Done

- **Design on-disk layouts:** Calculate exactly where superblocks, bitmaps, and inode tables live on a raw device.
- **Manage multi-level indirection:** Implement direct, single-indirect, and double-indirect pointers to address files up to 4GB.
- **Implement Path Resolution:** Build the recursion logic that translates strings like `/home/user/file.txt` into specific inode identifiers.
- **Expose Userspace Filesystems:** Use the FUSE API to mount your code as a real Linux mount point accessible by `ls`, `cp`, and `vim`.
- **Guarantee Crash Consistency:** Build a write-ahead journal that can replay committed transactions after a simulated system crash.

Final Deliverable

A complete C or Rust system consisting of ~4,000 lines of code across two primary binaries:

1. `mkfs`: A formatting tool that initializes raw files into valid filesystem images.
2. `myfs`: A FUSE daemon that mounts the image. The system is complete when you can mount your filesystem, copy a git repository into it, `SIGKILL` the process during a write, and see the filesystem recover perfectly upon remounting.

Is This Project For You?

You should start this if you:

- Are comfortable with C or Rust, specifically manual memory management and pointers.
- Understand bitwise operations (masks, shifts) for bitmap manipulation.
- Have a basic grasp of POSIX File I/O (`open`, `lseek`, `read`, `write`).

Come back after you've learned:

- **Struct Packing:** Understanding how compilers align data in memory (e.g., `__attribute__((packed))`).

- **Basic Data Structures:** You must be comfortable with linked lists and tree-like traversal.

Estimated Effort

Phase	Time
Phase 1: Block Layer and mkfs	~12 hours
Phase 2: Inode Management	~12 hours
Phase 3: Directory Operations	~12 hours
Phase 4: File Read/Write Operations	~12 hours
Phase 5: FUSE Integration	~12 hours
Phase 6: Write-Ahead Journaling	~15 hours
Total	**~75 hours**

Definition of Done

The project is complete when:

- The `mkfs` tool generates an image that passes a validation script checking superblock magic numbers and root directory invariants.
- The filesystem can be mounted via FUSE and supports standard Unix commands (`ls -la` , `mkdir` , `rm` , `cp` , `mv`).
- The system correctly handles "sparse files," where seeking past EOF and writing does not consume physical disk blocks for the gap.
- A "Crash Test" script successfully triggers a `SIGKILL` during a metadata-heavy operation, and `journal_recover` restores the filesystem to a consistent state on the next mount.
- All internal consistency checks (assertions on block numbers and link counts) pass under a concurrent workload of at least 4 threads.

📚 Before You Read This: Prerequisites & Further Reading

> **Read these first.** The Atlas assumes you are familiar with the foundations below. Resources are ordered by when you should encounter them — some before you start, some at specific milestones.

I. Core Architecture & The Inode

Paper: McKusick, M. K., et al. (1984). *A Fast File System for UNIX*. **Code**: Linux Kernel: fs/ext2/inode.c — `ext2_get_block` is the classic implementation of the indirection tree. **Best Explanation**: OSTEP: Chapter 40 - File System Implementation. **Why**: This chapter provides the clearest visual breakdown of how bitmapped allocation and inode tables interact in a "Simple File System." **Pedagogical Timing**: Read **BEFORE Milestone 1**. It establishes the mental model of the disk as an array of blocks that you will spend the rest of the project implementing.

Code: Linux Kernel: fs/ext4/extents.c — Specifically the `ext4_ext_binsearch` function. **Best Explanation**: Ext4 Disk Layout (Kernel.org Wiki). **Why**: This is the industry successor to the indirection tree you build in Milestone 2, essential for understanding modern large-scale storage. **Pedagogical Timing**: Read **AFTER Milestone 2**. Once you've struggled with the math of triple-indirection, you will immediately appreciate why "extents" are a more efficient way to track contiguous data.

II. VFS & Path Resolution

Paper: Kleiman, S. R. (1986). *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*. **Code**: Linux Kernel: fs/namei.c — The `link_path_walk` function is the actual engine behind `path_resolve`. **Best Explanation**: The Path Lookup Series (LWN.net) — Specifically "Part 2: Components of the path." **Why**: Neil Brown provides the most exhaustive explanation of the "walking" process and the subtle complexity of symlinks and mount points. **Pedagogical Timing**: Read **DURING Milestone 3**. Use it to debug your logic for `..` and trailing slashes in your path resolution engine.

III. Userspace Integration (FUSE)

Paper: Vangoor, B. R., et al. (2017). *To FUSE or Not to FUSE: Performance of User-Level File Systems*. **Code**: libfuse: example/hello.c. **Best Explanation**: FUSE Kernel Documentation. **Why**: It explains the protocol that lives inside `/dev/fuse`, clarifying how requests move from the VFS to your C code. **Pedagogical Timing**: Read **BEFORE Milestone 5**. You need to understand that FUSE is a message-passing protocol before you try to debug context switches and locking.

IV. Persistence & Crash Safety

Paper: Prabhakaran, V., et al. (2005). *Analysis and Evolution of Journaling File Systems*. **Code**: Linux Kernel: fs/jbd2/transaction.c — Look for `jbd2_journal_commit_transaction`. **Best Explanation**: OSTEP: Chapter 42 - Crash Consistency: FSCK and Journaling. **Why**: It provides a narrative walkthrough of the "Why" behind write-ahead logging using the same block-level terminology as this project. **Pedagogical Timing**: Read **BEFORE starting Milestone 6**. This is a conceptually heavy module; the OSTEP chapter acts as the necessary map before you write the code.

Explanation: SQLite: Atomic Commit In SQLite — Section 3.0 "The Rollback Journal." **Why**: This is a perfect example of how a database implements the same "intent-to-write" logic at the application level to ensure atomicity. **Pedagogical Timing**: Read **AFTER Milestone 6**. It will help you realize that your filesystem's journal and a database's WAL are essentially the same machine built for different purposes.

V. Hardware Interaction (The Physical Layer)

Spec: NVMe Express Base Specification, Revision 2.0 — Section 3.1.3 "Logical Block Address (LBA)." **Best Explanation**: The LBA is a Lie: An Introduction to SSDs (Cody Littlewood). **Why**: It shatters the illusion that writing to "Block 5" means writing to the 5th physical slot on an SSD, explaining the "Flash Translation Layer." **Pedagogical Timing**: Read **AFTER Milestone 4**. You've implemented software-level indirection; now see how the hardware implements its own hidden indirection for wear leveling.

VI. Memory Management Parallelism

Code: jemalloc: src/bitmap.c. **Best Explanation**: Understanding the Linux Virtual Memory Manager (Mel Gorman) — Chapter 6 "Page Frame Allocation." **Why**: This shows that the bitmap allocator you built in Milestone 1 is identical to how the Linux kernel manages physical RAM. **Pedagogical Timing**: Read **AFTER Milestone 1**. It validates that the patterns you are learning (bitmaps, alignment, metadata) are universal in systems engineering.

Filesystem Implementation: An Interactive Atlas

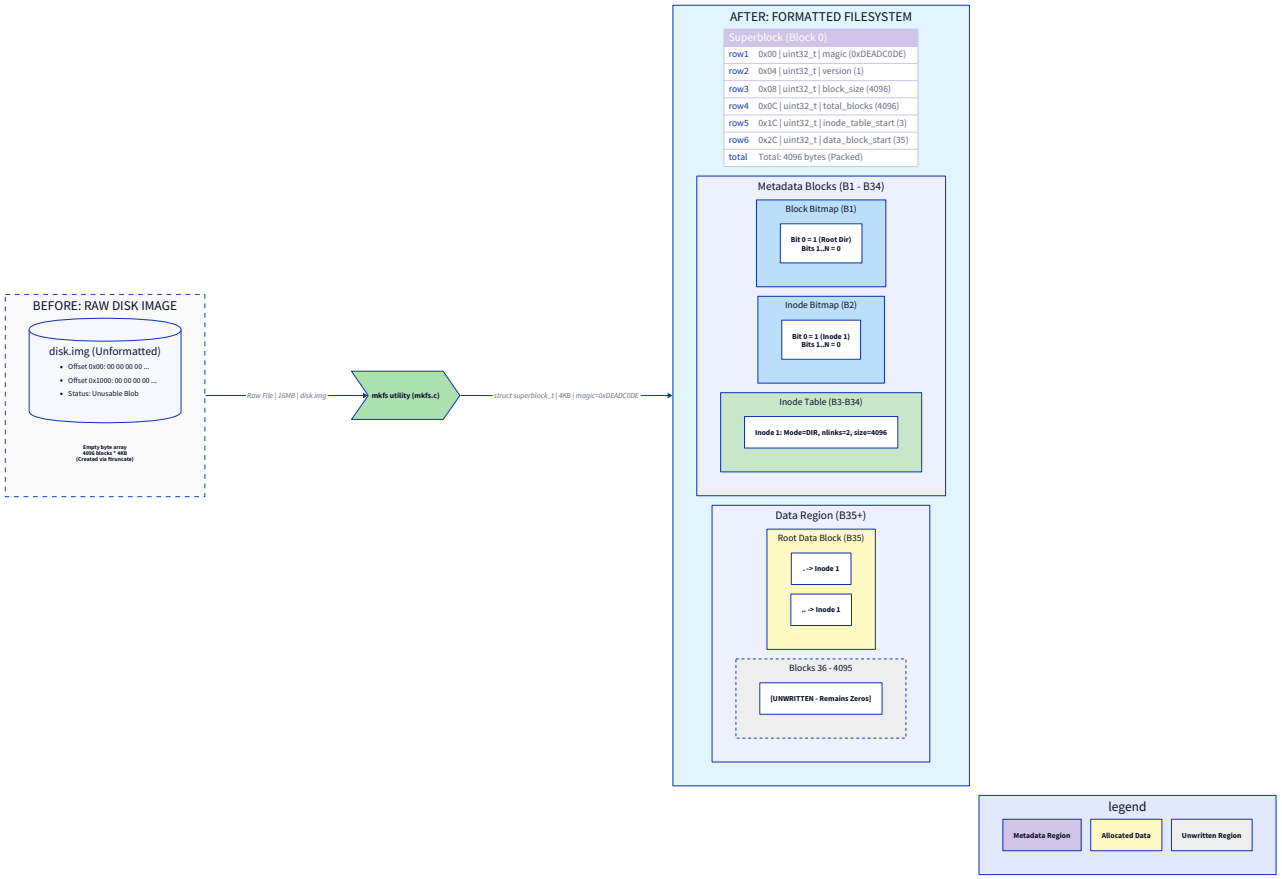
This project builds a complete inode-based filesystem from raw blocks up through FUSE mounting and write-ahead journaling. You will start with nothing but a flat file acting as a disk image, and layer increasingly sophisticated abstractions: block I/O, bitmap allocation, inode metadata with multi-level indirection, directory trees with path resolution, full POSIX-like file operations, real OS integration via FUSE, and finally crash-consistent journaling. Each layer reveals how the operating system transforms a dumb array of 4KB blocks into the rich tree of files and directories that every program takes for granted.

The atlas is structured so that each milestone shatters a common misconception about how filesystems work, then cascades that insight into 3-5 connected domains. By the end, you won't just have a working filesystem — you'll understand why databases use WAL, why SSDs have FTLs, why `fsync` matters, and why every storage system on earth is really just a block allocator with opinions.

Milestone 1: Block Layer and mkfs

The Disk Does Not Know You Exist

You have written thousands of files. You have called `fopen`, `fwrite`, `fclose` without thinking twice. You have an intuition — a reasonable one — that when you write "Hello" to a file, those five bytes land somewhere on the disk in sequence, and the OS just keeps track of where they are. That intuition is completely wrong. The disk has no concept of your file. It has no concept of bytes. It has no concept of names, paths, or even the idea that one chunk of data "belongs together." The disk is a machine that answers exactly two questions: "Give me block number N" and "Store these bytes at block number N." Block N is always the same size — 4,096 bytes — and the disk treats all of them identically. Block 7 is not more important than block 7,000,000. They are interchangeable containers. Everything you have ever experienced as a "file" — its name, its size, its permissions, the fact that its contents are logically contiguous even when physically scattered — is a fiction maintained entirely by software. The filesystem is the author of that fiction. And this milestone is where you learn to write it from scratch.



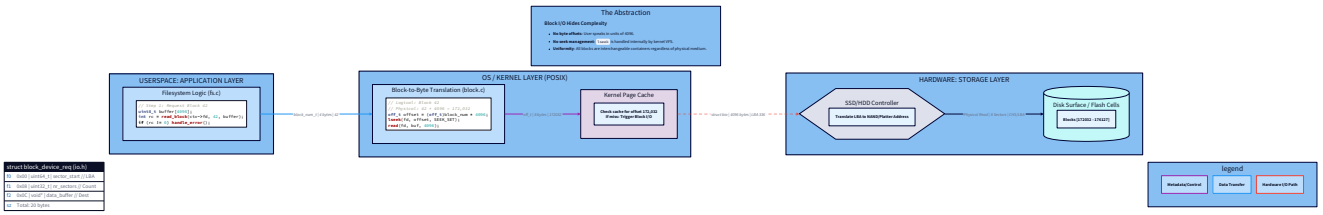
The Fundamental Tension: Fixed Blocks vs. Variable Reality

[[EXPLAIN:block-device-abstraction-—-why-disks-speak-in-fixed-size-blocks,-not-bytes|Block device abstraction — why disks speak in fixed-size blocks, not bytes]] Here is the core hardware constraint you are negotiating with for this entire project: **Storage hardware speaks in fixed, aligned blocks. Software demands arbitrary byte ranges.** A spinning magnetic disk reads and writes in sectors (typically 512 bytes, modern "Advanced Format" drives: 4,096 bytes). The read/write head sweeps across a platter — you cannot tell it to read 5 bytes starting at byte offset 7,234. You must read the entire sector containing those bytes, modify what you need in memory, and write the entire sector back. This is called a **read-modify-write cycle**, and you will encounter it in every layer of this project. SSDs are even more constrained. NAND flash cells can only be written in "pages" (4–16 KB) and only erased in larger "blocks" (128–512 KB). The SSD's internal controller — called the Flash Translation Layer (FTL) — hides these constraints behind a block interface identical to what a spinning disk presents.

> The numbers that matter:
>
> - A read of any size within one 4KB block = 1 disk I/O operation
> - Two reads that span a 4KB boundary = 2 disk I/O operations
> - Misaligned access that crosses block boundaries = double the cost Your filesystem inherits this constraint from below. You will define a block size of 4,096 bytes (4KB). Everything you store — superblock metadata, bitmaps, inode tables, directory entries, file data — must be packed into an integer number of 4KB blocks. Nothing lives between blocks. Nothing spans blocks unless you explicitly manage it. This is the deal. Let's build around it.

The Disk Image: A File That Pretends to Be a Disk

You do not have a spare disk lying around for development. You don't need one. A regular file on your host OS can serve as a perfectly faithful disk simulation. You will open a file (`disk.img`), and every 4KB region within that file corresponds to one block.



This is your block device abstraction. Two functions. Nothing else:

```c
int read_block(int fd, uint32_t block_num, void *buf);

int write_block(int fd, uint32_t block_num, const void *buf);
```

`fd` is the open file descriptor of your disk image. `block_num` is the logical address. `buf` is exactly `BLOCK_SIZE` (4,096) bytes. The implementation is a seek followed by a read or write:

```c
#include <unistd.h>

#include <stdint.h>

#include <string.h>

#include <errno.h>

#define BLOCK_SIZE   4096

#define SECTOR_SIZE  512   /* kept for reference; we use BLOCK_SIZE units */

int read_block(int fd, uint32_t block_num, void *buf) {

    off_t offset = (off_t)block_num * BLOCK_SIZE;

    if (lseek(fd, offset, SEEK_SET) == (off_t)-1) {

        return -errno;

    }

    ssize_t n = read(fd, buf, BLOCK_SIZE);

    if (n != BLOCK_SIZE) {

        return (n < 0) ? -errno : -EIO;

    }

    return 0;

}

int write_block(int fd, uint32_t block_num, const void *buf) {

    off_t offset = (off_t)block_num * BLOCK_SIZE;

    if (lseek(fd, offset, SEEK_SET) == (off_t)-1) {

        return -errno;

    }

    ssize_t n = write(fd, buf, BLOCK_SIZE);

    if (n != BLOCK_SIZE) {

        return (n < 0) ? -errno : -EIO;

    }

    return 0;

}
```

Every single operation in this project — reading a superblock, scanning a bitmap, fetching an inode, reading file data — will eventually call one of these two functions. They are the bedrock. Everything above them is metadata management.

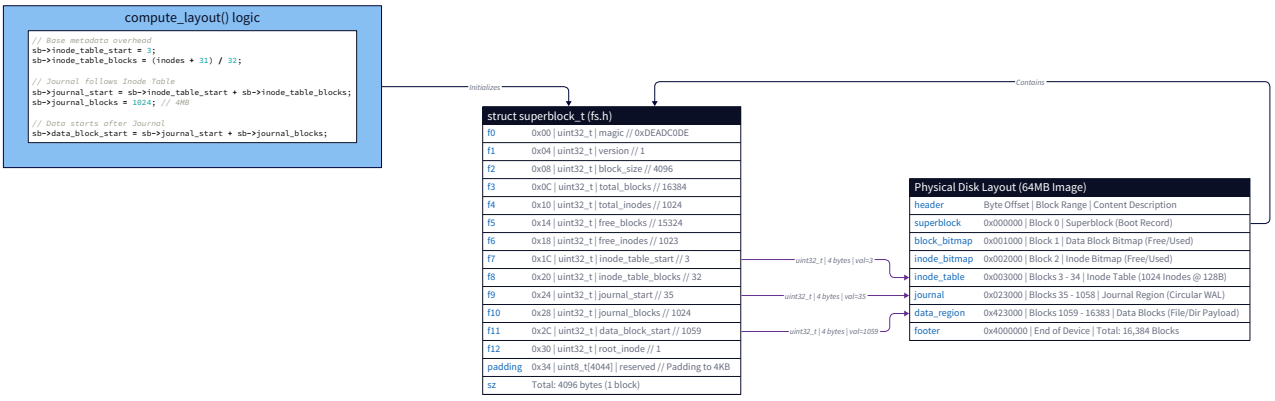
> **Hardware Soul**: What happens when `read_block` executes? Your host OS's VFS layer intercepts the `read` syscall. The kernel checks the page cache — if the 4KB chunk is already cached in RAM, no disk I/O occurs; the data is copied from kernel memory to your buffer in nanoseconds. If it's not cached (cold miss), the kernel submits a block I/O request, the disk controller fetches the sector(s), the data lands in a kernel buffer, and then gets copied to your process. That cold miss costs 50–150 µs on an SSD, or 5–10 ms on a spinning disk. Your block abstraction hides this, but the latency is always there underneath. **Pitfall to avoid**: Always validate `block_num < total_blocks` before calling into these functions. A bug that passes `block_num = 3,000,000` into a disk image with only 1,000 blocks will happily seek to byte offset 12 billion and either hit EOF or extend the file. Add a bounds check:

```c
/* Always called before read_block / write_block */
static int validate_block_num(uint32_t block_num, uint32_t total_blocks) {
    if (block_num >= total_blocks) {
        return -EINVAL;
    }
    return 0;
}
```

On-Disk Layout: The Filesystem as a Database

You have a flat array of 4KB blocks. Your filesystem needs to store several kinds of data:

1. **The filesystem's own description** — how big is it? how many inodes? where do things live? → **Superblock**
2. **Which blocks are free and which are in use?** → **Block Bitmap**
3. **Which inode numbers are free and which are in use?** → **Inode Bitmap**
4. **The actual inode structures** (file metadata: size, permissions, block pointers) → **Inode Table**
5. **The journal** (for crash recovery in Milestone 6) → **Journal Region**
6. **The actual file and directory data** → **Data Blocks** These must be arranged in a specific order in the block array, and every component must know where to find every other component. The superblock is the bootstrap record that makes this possible — it stores the positions and sizes of everything else.



Here is the layout we will use:

Block Range	Contents
Block 0	Superblock
Block 1	Block Bitmap (tracks data blocks)
Block 2	Inode Bitmap
Blocks 3–N	Inode Table (N depends on inode count)
Blocks N+1–M	Journal Region
Blocks M+1–end	Data Blocks
The exact values of N and M are computed from the superblock fields — which means you need to understand the superblock first.	

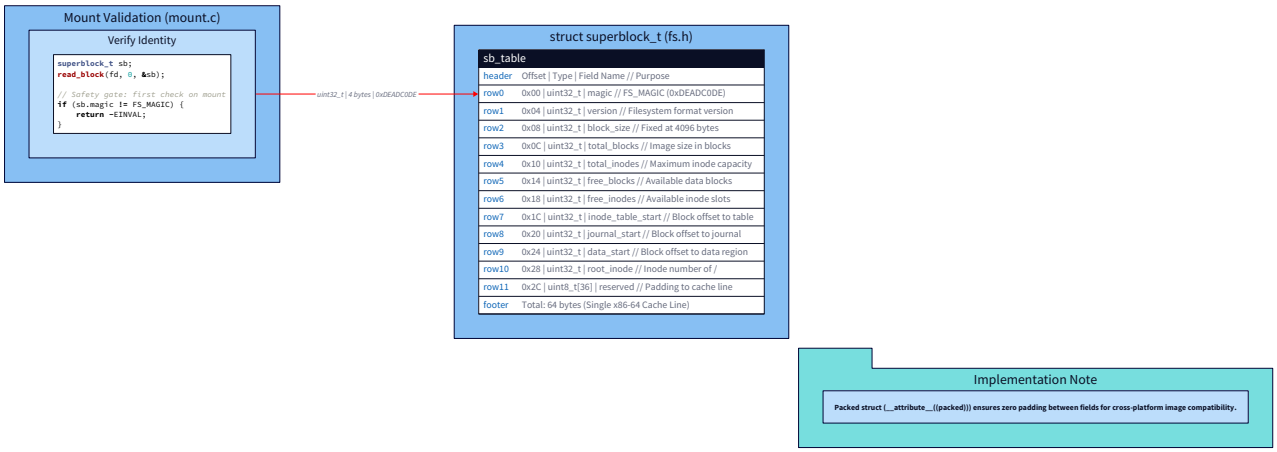
The Superblock: Your Filesystem's Birth Certificate

[[EXPLAIN:on-disk-serialization-—-packing-structs-to-exact-byte-layouts-with-fixed-width-types|On-disk serialization — packing structs to exact byte layouts with fixed-width types]] The superblock lives in block 0. It is the first thing read when you mount the filesystem, and the source of truth for every other layout calculation. If the superblock is corrupted, the filesystem is unreadable — there is no way to find anything else. Here is every field and why it exists:

```c
#include <stdint.h>

#define FS_MAGIC        0xDEADC0DE   /* arbitrary identifier; proves this is OUR filesystem */

#define FS_VERSION      1

#define BLOCK_SIZE      4096

#define INODE_SIZE      128          /* bytes per inode; must divide evenly into BLOCK_SIZE */

#define INODES_PER_BLOCK (BLOCK_SIZE / INODE_SIZE)   /* 32 inodes per block */

/*

 * Superblock — always resides in block 0.

 *

 * BYTE LAYOUT (all little-endian, packed):

 *   Offset  0: magic           (4 bytes)

 *   Offset  4: version         (4 bytes)

 *   Offset  8: block_size      (4 bytes)

 *   Offset 12: total_blocks    (4 bytes)

 *   Offset 16: total_inodes    (4 bytes)

 *   Offset 20: free_blocks     (4 bytes)

 *   Offset 24: free_inodes     (4 bytes)

 *   Offset 28: inode_table_start (4 bytes)

 *   Offset 32: inode_table_blocks (4 bytes)

 *   Offset 36: journal_start   (4 bytes)

 *   Offset 40: journal_blocks  (4 bytes)

 *   Offset 44: data_block_start (4 bytes)

 *   Offset 48: root_inode      (4 bytes)

 *   ...reserved...             (4044 bytes to fill block)

 *   Total: 4096 bytes (1 block)

 */

typedef struct __attribute__((packed)) {

    uint32_t magic;              /* must equal FS_MAGIC to be a valid filesystem */

    uint32_t version;            /* filesystem format version */

    uint32_t block_size;         /* always BLOCK_SIZE (4096) — stored for validation */

    uint32_t total_blocks;       /* total number of blocks in the image */
```

```
    uint32_t total_inodes;       /* maximum number of inodes */
    uint32_t free_blocks;        /* count of currently unallocated data blocks */
    uint32_t free_inodes;        /* count of currently unallocated inodes */
    uint32_t inode_table_start;  /* block number where inode table begins */
    uint32_t inode_table_blocks; /* how many blocks the inode table occupies */
    uint32_t journal_start;      /* block number where journal region begins */
    uint32_t journal_blocks;     /* how many blocks the journal occupies */
    uint32_t data_block_start;   /* block number where data blocks begin */
    uint32_t root_inode;         /* inode number of the root directory (/) */
    uint8_t  reserved[BLOCK_SIZE - 52]; /* pad to exactly one block */
} superblock_t;
```

Why `__attribute__((packed))` ? The C compiler inserts invisible padding bytes between struct fields to align them to their natural size (a `uint32_t` is normally aligned to a 4-byte boundary). When you write this struct to disk and read it back on a different system — or even the same system after recompilation — padding bytes may shift. `__attribute__((packed))` tells GCC/Clang to remove all padding: every field occupies exactly the bytes you declared. This is mandatory for any struct that lives on disk. The tradeoff: packed struct access may generate slower unaligned memory reads on some architectures. For disk I/O, this is irrelevant.



The magic number deserves special attention. `0xDEADC0DE` is not just vanity — it is a safety gate. When you implement mounting, the very first check is:

```c
if (sb.magic != FS_MAGIC) {
    fprintf(stderr, "Not a valid filesystem image (bad magic: 0x%08X)\n", sb.magic);
    return -EINVAL;
}
```

Without this check, your filesystem code will happily interpret a random file, a truncated image, or a different filesystem format as valid — and then corrupt it. Every filesystem has a magic number: ext4 uses `0xEF53`, FAT32 uses `0x28` at offset 66, XFS uses `"XFSB"` as ASCII bytes. **Layout calculations** — all positions are derived from the superblock, computed once during `mkfs` and stored:

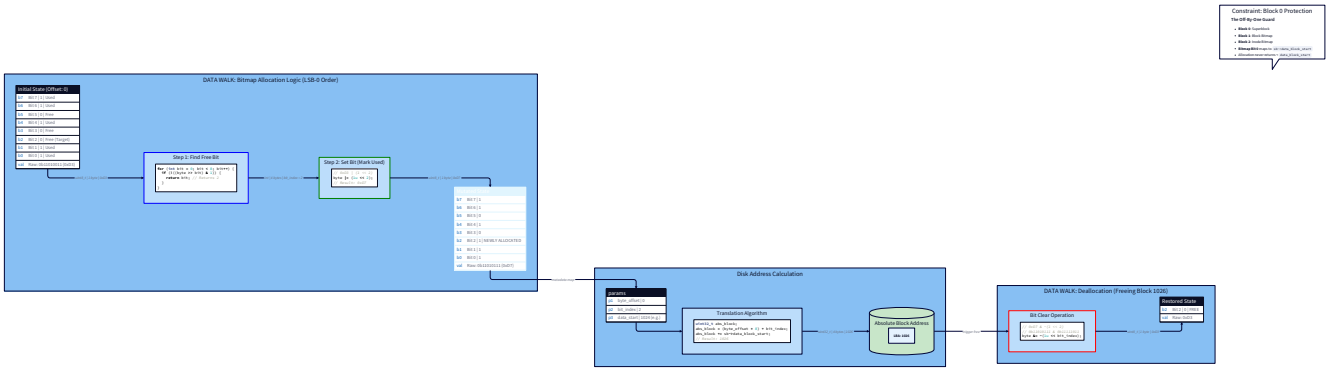
```c
/* Given: total_blocks, total_inodes
 * Compute: all region positions */
static void compute_layout(uint32_t total_blocks, uint32_t total_inodes,
                           superblock_t *sb) {
    sb->block_size      = BLOCK_SIZE;
    sb->total_blocks    = total_blocks;
    sb->total_inodes    = total_inodes;
    /* Fixed positions */
    /* Block 0: superblock (already implicit) */
    /* Block 1: block bitmap */
    /* Block 2: inode bitmap */
    uint32_t inode_table_start = 3;
    uint32_t inode_table_blocks = (total_inodes + INODES_PER_BLOCK - 1)
                                  / INODES_PER_BLOCK;
    /* e.g., 1024 inodes / 32 per block = 32 blocks for inode table */
    uint32_t journal_start  = inode_table_start + inode_table_blocks;
    uint32_t journal_blocks = 1024;  /* 4MB journal; tune as needed */
    uint32_t data_start = journal_start + journal_blocks;
    sb->inode_table_start  = inode_table_start;
    sb->inode_table_blocks = inode_table_blocks;
    sb->journal_start      = journal_start;
    sb->journal_blocks     = journal_blocks;
    sb->data_block_start   = data_start;
    /* Free counts: subtract everything used by metadata */
    sb->free_blocks = total_blocks - data_start;  /* data blocks only */
    sb->free_inodes = total_inodes - 1;           /* inode 1 reserved for root */
    sb->root_inode  = 1;
    sb->magic       = FS_MAGIC;
    sb->version     = FS_VERSION;
}
```

> **Off-by-one protection**: Notice `free_blocks = total_blocks - data_start`. This is exact — it means blocks 0 through `data_start - 1` are all metadata and are never treated as allocatable data blocks. The block bitmap only tracks data blocks, so bitmap bit 0 corresponds to disk block `data_block_start`, not disk block 0. This offset must be applied consistently everywhere — allocating from the bitmap gives you a *data block index*, which you add to `data_block_start` to get the actual disk block number.

Bitmaps: One Bit per Block, Two States of Truth

[[EXPLAIN:bitmap-data-structure-—-bit-level-allocation-tracking|Bitmap data structure — bit-level allocation tracking]] To track which blocks and inodes are free, you need a data structure that is:

1. **Compact** — tracking 1,000,000 blocks should not cost 1,000,000 bytes
2. **Fast to scan** — finding the first free block must be quick
3. **Easy to update** — marking a block used or free must be a simple write A bitmap satisfies all three. One bit per tracked item: `0` means free, `1` means used. 4KB of bitmap = 4096 × 8 = 32,768 bits = 32,768 blocks tracked. A filesystem with 1 million data blocks needs only 32 bitmap blocks (~128KB) to track them all.



Your block bitmap lives in block 1. It is allocated exactly one block (4KB = 32,768 bits), which limits your filesystem to 32,768 data blocks (128MB). For a learning project, this is fine. Real filesystems use multiple bitmap blocks per block group (ext4) or more sophisticated free space trees (XFS, Btrfs).

```c
/*
 * Bitmap operations.
 * The bitmap is stored in memory as uint8_t[BLOCK_SIZE].
 * Each bit corresponds to one block or inode.
 *
 * Bit layout within a byte (standard, big-endian bit order):
 *   byte[0] bit 7 → index 0
 *   byte[0] bit 6 → index 1
 *   ...
 *   byte[0] bit 0 → index 7
 *   byte[1] bit 7 → index 8
 *   ...
 *
 * We use the simpler little-endian bit order here:
 *   byte[i / 8] bit (i % 8)
 */
/* Test whether bit at index is set (1 = used) */
static inline int bitmap_test(const uint8_t *bitmap, uint32_t index) {
    return (bitmap[index / 8] >> (index % 8)) & 1;
}
/* Set bit at index (mark as used) */
static inline void bitmap_set(uint8_t *bitmap, uint32_t index) {
    bitmap[index / 8] |= (1u << (index % 8));
}
/* Clear bit at index (mark as free) */
static inline void bitmap_clear(uint8_t *bitmap, uint32_t index) {
    bitmap[index / 8] &= ~(1u << (index % 8));
}
/*
 * Find the first free bit in the bitmap.
 * Returns the index of the first 0 bit, or -1 if all bits are set (full).
```

```
 *
 * Optimization: scan byte-by-byte first; skip fully-used bytes (0xFF) fast.
 */
static int bitmap_find_free(const uint8_t *bitmap, uint32_t total_items) {
    uint32_t num_bytes = (total_items + 7) / 8;
    for (uint32_t byte_idx = 0; byte_idx < num_bytes; byte_idx++) {
        if (bitmap[byte_idx] == 0xFF) {
            continue;  /* all 8 bits used — skip entire byte */
        }
        /* At least one free bit in this byte; find which one */
        for (int bit = 0; bit < 8; bit++) {
            uint32_t index = byte_idx * 8 + bit;
            if (index >= total_items) {
                return -1;  /* ran past the valid range */
            }
            if (!((bitmap[byte_idx] >> bit) & 1)) {
                return (int)index;
            }
        }
    }
    return -1;  /* bitmap full */
}
```

Block allocation in full context — combining disk I/O with bitmap logic:

```c
/*
 * Allocate a data block. Returns disk block number, or -1 on failure.
 * The block bitmap is in disk block 1 (BLOCK_BITMAP_BLOCK).
 */
#define BLOCK_BITMAP_BLOCK  1
#define INODE_BITMAP_BLOCK  2
int alloc_block(int fd, superblock_t *sb) {
    uint8_t bitmap[BLOCK_SIZE];
    /* Step 1: Load the block bitmap from disk */
    if (read_block(fd, BLOCK_BITMAP_BLOCK, bitmap) != 0) {
        return -1;
    }
    /* Step 2: Find a free bit */
    int data_idx = bitmap_find_free(bitmap, sb->free_blocks +
                                    (sb->total_blocks - sb->data_block_start
                                     - sb->free_blocks));
    /* Simpler: track total data blocks separately */
    uint32_t data_blocks_total = sb->total_blocks - sb->data_block_start;
    data_idx = bitmap_find_free(bitmap, data_blocks_total);
    if (data_idx < 0) {
        return -1;  /* disk full */
    }
    /* Step 3: Mark it used */
    bitmap_set(bitmap, (uint32_t)data_idx);
    /* Step 4: Write bitmap back to disk — MUST happen before returning */
    if (write_block(fd, BLOCK_BITMAP_BLOCK, bitmap) != 0) {
        return -1;
    }
    /* Step 5: Update superblock free count */
    sb->free_blocks--;
    /* Note: caller must persist the superblock update too */
```

```c
    /* Step 6: Return the DISK block number (offset by data_block_start) */
    return (int)(sb->data_block_start + (uint32_t)data_idx);
}
/*
 * Free a data block. disk_block_num is the actual disk block number.
 */
int free_block(int fd, superblock_t *sb, uint32_t disk_block_num) {
    if (disk_block_num < sb->data_block_start ||
        disk_block_num >= sb->total_blocks) {
        return -EINVAL;  /* cannot free a metadata block */
    }
    uint8_t bitmap[BLOCK_SIZE];
    if (read_block(fd, BLOCK_BITMAP_BLOCK, bitmap) != 0) {
        return -EIO;
    }
    uint32_t data_idx = disk_block_num - sb->data_block_start;
    bitmap_clear(bitmap, data_idx);
    if (write_block(fd, BLOCK_BITMAP_BLOCK, bitmap) != 0) {
        return -EIO;
    }
    sb->free_blocks++;
    return 0;
}
```

The inode bitmap is identical in structure, just stored in block 2 and tracking inode numbers rather than data blocks. Implement `alloc_inode` and `free_inode` the same way, substituting `INODE_BITMAP_BLOCK` and `sb->total_inodes`.

> **Hardware Soul — Bitmap Scan Performance**: The `bitmap_find_free` function scans sequentially. This is intentional: sequential memory access is extremely cache-friendly. Your entire block bitmap fits in a single 4KB block — which fits in a single L1 cache line fetch (actually 64 L1 cache lines at 64 bytes each, but the sequential scan will prefetch them automatically). Scanning 32,768 bits sequentially is ~4,000 byte comparisons — at 32 bytes per cycle, under 200 CPU cycles, under 100ns. The disk read to load the bitmap is 1000× slower. The bitmap scan itself is never the bottleneck. **Why not a free list instead?** A linked list of free blocks (used by early Unix filesystems) has O(1) allocation but requires updating two disk blocks per allocation (the list head block and the freed/allocated block). Scanning a full bitmap has O(n) worst case but O(1) in practice for nearly-empty filesystems. The bigger win: bitmaps are trivially crash-recoverable — you can reconstruct the correct bitmap by scanning all inodes.

Designing for Crash Safety: The Two-Write Problem

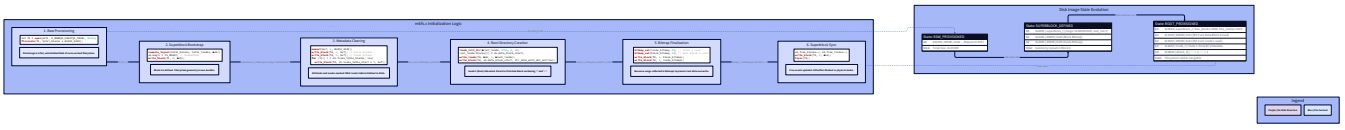
Before writing `mkfs`, you need to internalize one consequence of working with blocks that will affect every write operation you ever implement: **A crash between two writes leaves the filesystem in an inconsistent state.** Example: to allocate block 500, you must:

1. Set bit 500 in the block bitmap (marks it used)
2. Update the inode's block pointer to point to 500 If the power dies between step 1 and step 2, you now have a block that is marked as used but that no inode points to — a **leaked block**. If the power dies between step 2 and step 1 (if you do it in the other order), you have a block that an inode thinks it owns but that is also marked free — which means it could be allocated to a different file, causing **data corruption**. This is the crash consistency problem. Milestone 6 solves it properly with journaling. For now, establish the discipline: **always write to the bitmap before writing to the structure that references it**. When in doubt, bitmap first.

---

mkfs: Formatting the Raw Image

`mkfs` stands for "make filesystem." It is the tool that transforms a raw, unformatted file (a flat blob of zeros) into a valid filesystem that your code can mount and operate on. Every filesystem has one: `mkfs.ext4`, `mkfs.xfs`, `mkfs.fat`. Your `mkfs` must perform these operations in order:



1. Create (or truncate) the disk image file to the desired size
2. Compute the on-disk layout (call `compute_layout`)
3. Write the superblock to block 0
4. Zero the block bitmap (block 1) — all blocks free
5. Zero the inode bitmap (block 2) — all inodes free
6. Zero the inode table blocks
7. Allocate inode 1 for the root directory
8. Initialize the root directory (a special file whose data contains directory entries for `.` and `..`)
9. Flush everything to disk

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdint.h>

#include <errno.h>

/* Minimal inode structure — full version in Milestone 2 */

#define N_DIRECT_PTRS  12

typedef struct __attribute__((packed)) {

    uint16_t mode;          /* file type + permissions */

    uint16_t uid;

    uint32_t size;          /* file size in bytes */

    uint32_t atime;

    uint32_t mtime;

    uint32_t ctime;

    uint16_t nlinks;        /* hard link count */

    uint16_t _pad;

    uint32_t blocks[N_DIRECT_PTRS];  /* direct block pointers */

    uint32_t single_indirect;

    uint32_t double_indirect;

    uint8_t  reserved[12];  /* pad to 128 bytes */

} inode_t;

_Static_assert(sizeof(inode_t) == 128, "inode_t must be exactly 128 bytes");

/* Minimal directory entry — full version in Milestone 3 */

typedef struct __attribute__((packed)) {

    uint32_t inode_num;     /* 0 = free/unused entry */

    uint16_t rec_len;       /* total length of this entry */

    uint8_t  name_len;      /* length of name field */

    uint8_t  file_type;     /* 0=unknown, 1=regular, 2=directory */

    char     name[256];     /* null-padded, NOT null-terminated necessarily */
```

```c
} dirent_t;

/* File type constants (stored in mode high bits) */
#define S_IFDIR  0040000
#define S_IFREG  0100000
#define DEFAULT_DIR_MODE  (S_IFDIR | 0755)

/*
 * Write an inode to the inode table on disk.
 * inode_num: 1-based inode number (0 is reserved/invalid)
 */
int write_inode(int fd, const superblock_t *sb, uint32_t inode_num,
                const inode_t *inode) {
    if (inode_num == 0 || inode_num > sb->total_inodes) {
        return -EINVAL;
    }
    /* Convert inode number to disk position */
    uint32_t idx          = inode_num - 1;  /* 0-based index into inode table */
    uint32_t block_offset = idx / INODES_PER_BLOCK;
    uint32_t block_num    = sb->inode_table_start + block_offset;
    uint32_t slot_in_block = idx % INODES_PER_BLOCK;
    uint8_t buf[BLOCK_SIZE];
    if (read_block(fd, block_num, buf) != 0) {
        return -EIO;
    }
    /* Copy inode into the correct slot within this block */
    memcpy(buf + slot_in_block * INODE_SIZE, inode, INODE_SIZE);
    return write_block(fd, block_num, buf);
}

/*
 * mkfs: initialize a fresh filesystem image.
 *
 * image_path: path to create the disk image
```

```
 * total_blocks: desired size in 4KB blocks (e.g., 4096 = 16MB image)
 * total_inodes: maximum number of inodes
 */
int mkfs(const char *image_path, uint32_t total_blocks, uint32_t total_inodes) {
    /* --- Step 1: Create and size the image file --- */
    int fd = open(image_path, O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        return -1;
    }
    /* Extend the file to total_blocks * BLOCK_SIZE bytes */
    if (ftruncate(fd, (off_t)total_blocks * BLOCK_SIZE) != 0) {
        perror("ftruncate");
        close(fd);
        return -1;
    }
    /* --- Step 2: Compute layout and initialize superblock --- */
    superblock_t sb;
    memset(&sb, 0, sizeof(sb));
    compute_layout(total_blocks, total_inodes, &sb);
    /* --- Step 3: Write superblock to block 0 --- */
    uint8_t block_buf[BLOCK_SIZE];
    memset(block_buf, 0, BLOCK_SIZE);
    memcpy(block_buf, &sb, sizeof(sb));
    if (write_block(fd, 0, block_buf) != 0) {
        fprintf(stderr, "Failed to write superblock\n");
        close(fd);
        return -1;
    }
    /* --- Step 4: Zero the block bitmap (block 1) --- */
    memset(block_buf, 0, BLOCK_SIZE);
```

```c
    if (write_block(fd, BLOCK_BITMAP_BLOCK, block_buf) != 0) {
        close(fd);
        return -1;
    }
    /* --- Step 5: Zero the inode bitmap (block 2) --- */
    if (write_block(fd, INODE_BITMAP_BLOCK, block_buf) != 0) {
        close(fd);
        return -1;
    }
    /* --- Step 6: Zero the inode table --- */
    for (uint32_t i = 0; i < sb.inode_table_blocks; i++) {
        memset(block_buf, 0, BLOCK_SIZE);
        if (write_block(fd, sb.inode_table_start + i, block_buf) != 0) {
            close(fd);
            return -1;
        }
    }
    /* --- Step 7: Mark inode 1 as used in inode bitmap --- */
    uint8_t inode_bmap[BLOCK_SIZE];
    memset(inode_bmap, 0, BLOCK_SIZE);
    bitmap_set(inode_bmap, 0);  /* bit 0 → inode number 1 (1-based) */
    if (write_block(fd, INODE_BITMAP_BLOCK, inode_bmap) != 0) {
        close(fd);
        return -1;
    }
    /* --- Step 8: Allocate a data block for root directory's entries --- */
    /* Manually allocate block 0 of data blocks for root dir content */
    uint8_t block_bmap[BLOCK_SIZE];
    memset(block_bmap, 0, BLOCK_SIZE);
    bitmap_set(block_bmap, 0);  /* first data block used by root dir */
    if (write_block(fd, BLOCK_BITMAP_BLOCK, block_bmap) != 0) {
```

```
    close(fd);
    return -1;
}
uint32_t root_data_block = sb.data_block_start;  /* data block index 0 */
/* --- Step 9: Create root directory inode --- */
inode_t root_inode;
memset(&root_inode, 0, sizeof(root_inode));
root_inode.mode    = DEFAULT_DIR_MODE;
root_inode.nlinks  = 2;      /* '.' and the reference from parent (itself) */
root_inode.size    = BLOCK_SIZE;
root_inode.blocks[0] = root_data_block;
uint32_t now = (uint32_t)time(NULL);
root_inode.atime = root_inode.mtime = root_inode.ctime = now;
if (write_inode(fd, &sb, sb.root_inode, &root_inode) != 0) {
    close(fd);
    return -1;
}
/* --- Step 10: Write '.' and '..' entries into root's data block --- */
uint8_t dir_block[BLOCK_SIZE];
memset(dir_block, 0, BLOCK_SIZE);
/* Entry for '.' — points to root inode itself */
dirent_t *dot = (dirent_t *)dir_block;
dot->inode_num  = sb.root_inode;
dot->name_len   = 1;
dot->file_type  = 2;  /* directory */
dot->rec_len    = sizeof(dirent_t);
memcpy(dot->name, ".", 1);
/* Entry for '..' — root's parent is also root */
dirent_t *dotdot = (dirent_t *)(dir_block + sizeof(dirent_t));
dotdot->inode_num = sb.root_inode;
dotdot->name_len  = 2;
```

```
    dotdot->file_type = 2;
    dotdot->rec_len   = sizeof(dirent_t);
    memcpy(dotdot->name, "..", 2);
    if (write_block(fd, root_data_block, dir_block) != 0) {
        close(fd);
        return -1;
    }
    /* --- Step 11: Persist the updated superblock (free counts changed) --- */
    sb.free_blocks--;  /* one data block used by root dir */
    sb.free_inodes--;  /* inode 1 used by root dir */
    memset(block_buf, 0, BLOCK_SIZE);
    memcpy(block_buf, &sb, sizeof(sb));
    if (write_block(fd, 0, block_buf) != 0) {
        close(fd);
        return -1;
    }
    printf("mkfs: created %s (%u blocks, %u inodes)\n",
           image_path, total_blocks, total_inodes);
    printf("  Superblock at block 0\n");
    printf("  Block bitmap at block %u\n", BLOCK_BITMAP_BLOCK);
    printf("  Inode bitmap at block %u\n", INODE_BITMAP_BLOCK);
    printf("  Inode table: blocks %u–%u (%u blocks)\n",
           sb.inode_table_start,
           sb.inode_table_start + sb.inode_table_blocks - 1,
           sb.inode_table_blocks);
    printf("  Journal: blocks %u–%u\n",
           sb.journal_start,
           sb.journal_start + sb.journal_blocks - 1);
    printf("  Data blocks: %u–%u (%u available)\n",
           sb.data_block_start, total_blocks - 1,
           sb.free_blocks);
```

```
    close(fd);

    return 0;

}

int main(int argc, char *argv[]) {

    if (argc != 4) {

        fprintf(stderr, "Usage: mkfs <image> <total_blocks> <total_inodes>\n");

        fprintf(stderr, "  Example: mkfs disk.img 4096 1024\n");

        return 1;

    }

    uint32_t total_blocks = (uint32_t)atoi(argv[2]);

    uint32_t total_inodes = (uint32_t)atoi(argv[3]);

    return mkfs(argv[1], total_blocks, total_inodes) == 0 ? 0 : 1;

}
```

Verification: Reading Back What You Wrote

After `mkfs` runs, write a simple verification function that reads the image back and validates every invariant:

```c
int verify_filesystem(const char *image_path) {
    int fd = open(image_path, O_RDONLY);
    if (fd < 0) { perror("open"); return -1; }
    /* Read and validate superblock */
    uint8_t buf[BLOCK_SIZE];
    read_block(fd, 0, buf);
    superblock_t *sb = (superblock_t *)buf;
    if (sb->magic != FS_MAGIC) {
        fprintf(stderr, "FAIL: bad magic number 0x%08X\n", sb->magic);
        close(fd); return -1;
    }
    printf("OK: magic number 0x%08X\n", sb->magic);
    printf("OK: %u total blocks, %u total inodes\n",
           sb->total_blocks, sb->total_inodes);
    printf("OK: data blocks start at block %u\n", sb->data_block_start);
    /* Verify root inode exists and is a directory */
    uint32_t root_ino = sb->root_inode;
    uint32_t idx          = root_ino - 1;
    uint32_t block_num    = sb->inode_table_start + idx / INODES_PER_BLOCK;
    uint32_t slot         = idx % INODES_PER_BLOCK;
    read_block(fd, block_num, buf);
    inode_t *root = (inode_t *)(buf + slot * INODE_SIZE);
    if ((root->mode & S_IFDIR) == 0) {
        fprintf(stderr, "FAIL: root inode is not a directory\n");
        close(fd); return -1;
    }
    if (root->nlinks != 2) {
        fprintf(stderr, "FAIL: root nlinks = %u, expected 2\n", root->nlinks);
        close(fd); return -1;
    }
    printf("OK: root inode %u is a directory with nlinks=%u\n",
```

```
           root_ino, root->nlinks);

    /* Verify inode bitmap marks root inode as used */

    read_block(fd, INODE_BITMAP_BLOCK, buf);

    if (!bitmap_test(buf, root_ino - 1)) {

        fprintf(stderr, "FAIL: root inode not marked used in bitmap\n");

        close(fd); return -1;

    }

    printf("OK: root inode bit set in inode bitmap\n");

    /* Read root directory's data block, find '.' entry */

    read_block(fd, root->blocks[0], buf);

    dirent_t *dot = (dirent_t *)buf;

    if (dot->inode_num != root_ino || dot->name[0] != '.') {

        fprintf(stderr, "FAIL: first entry in root is not '.'\n");

        close(fd); return -1;

    }

    printf("OK: root directory has '.' → inode %u\n", dot->inode_num);

    close(fd);

    printf("Filesystem image: VERIFIED\n");

    return 0;

}
```

Run this immediately after `mkfs`. If it passes, every piece of data you wrote survives a round-trip to disk and back — the most basic contract of a storage system.

Three-Level View: What Happens When `write_block` Executes

Let's trace what actually happens when your code calls `write_block(fd, 1, bitmap)` — writing the block bitmap to disk.

Level	What Happens
Application (your code)	Calls `write(fd, buf, 4096)` via your `write_block` wrapper. Control transfers to the kernel via a syscall interrupt.
OS / Kernel	The VFS layer receives the write. The kernel allocates a 4KB page in the **page cache** (kernel memory) and copies your `buf` there. The page is now "dirty." The kernel may not write to disk immediately — it batches writes for efficiency. The `write` syscall returns as soon as the data is in the page cache.
Hardware	Eventually (on `fsync`, or when the kernel decides to flush), the block I/O scheduler submits a write request to the disk driver. The disk DMA controller transfers the 4KB from kernel memory to the disk's write buffer. The disk confirms completion. The page cache page is marked clean.

> **The subtle danger**: When `write_block` returns `0`, your data is in the kernel page cache — not necessarily on disk. A power loss between the `write` call and the disk flush loses your data. Later in this project (and in Milestone 6), you will use `fsync(fd)` to force the kernel to flush all dirty pages to disk before returning. For `mkfs`, call `fsync(fd)` before `close(fd)` to guarantee the freshly formatted image is actually on disk.

```c
/* Add this before close() in mkfs */
if (fsync(fd) != 0) {
    perror("fsync");
    /* non-fatal for mkfs in development, but fatal in production */
}
```

Key Design Decisions

There are real choices to make at this layer. Here are the most impactful: **Fixed inode count vs. dynamic inode allocation**

Option	Pros	Cons	Used By
Fixed count at mkfs time (chosen ✓)	Simple bitmap; O(1) allocation; layout fully known at format time	Running out of inodes with space remaining (classic ext2/ext3 problem on mail servers)	ext2, ext3, your implementation
Dynamic inode allocation	Inodes grow as needed; no `df -i` exhaustion	Complex; requires indirect structures for inode lookup	XFS, ext4 (flexible block groups)
Bitmap per region vs. free list			
Option	Pros	Cons	Used By
---	---	---	---
Bitmap (chosen ✓)	Compact; crash-recoverable (rebuild from inodes); sequential scan	O(n) scan for allocation; fragmentation over time	ext2/3/4, most simple filesystems
Free list / extent list	O(1) allocation; naturally tracks contiguous free regions	Requires careful crash recovery; complexity	XFS (B-tree free space), NTFS
Block size: 4KB vs. alternatives			
4KB is the dominant choice because it matches the virtual memory page size on x86-64 (and ARM). This means: block-aligned I/O never triggers partial-page reads; memory-mapped files (`mmap`) map naturally; the OS page cache handles blocks without padding. Larger blocks (8KB, 16KB) reduce metadata overhead for large files but waste space for small files (a 1-byte file consumes an entire 8KB block). Smaller blocks (1KB, 2KB) reduce internal fragmentation but require more metadata to track the same data.			

Knowledge Cascade: One Concept, Ten Connections

You just built a block allocator backed by a bitmap that sits on top of a flat address space of fixed-size chunks. This is not a filesystem-specific idea. It is one of the fundamental patterns of systems programming: → Database Page Management PostgreSQL, SQLite, and MySQL all store data in "pages" — fixed-size blocks (8KB in Postgres, 4KB in SQLite by default). Their internal page managers maintain a freelist or bitmap of free pages. The "heap file" in Postgres is literally a sequence of

8KB pages, with a "free space map" (FSM) tracking available space per page. Your superblock is Postgres's `pg_control` file. Your bitmap is the FSM. Building a filesystem gives you the mental model to read database internals with clarity. → The SSD Flash Translation Layer (FTL) When you call `write_block(fd, 500, buf)`, you are writing to logical block 500. But inside your SSD, logical block 500 does not map to a fixed physical NAND page. The FTL maintains its own mapping table (logical-to-physical, LPT), performs wear leveling (distributing writes evenly across cells), and manages its own garbage collection when pages need to be erased. Your filesystem sits on top of a block abstraction that is itself implemented by another system with its own block abstraction. Indirection all the way down. → Memory Allocators (jemalloc, tcmalloc, mimalloc) `malloc()` faces the same problem from a different angle: carve variable-size allocations from fixed-size OS pages (4KB). jemalloc's slab allocator tracks free slots within a page using... a bitmap. Each `jemalloc` "run" is a region of pages dedicated to a specific allocation size class, with a bitmap marking which slots are occupied. Your `bitmap_find_free` and `bitmap_set` are functionally identical to what jemalloc executes on every `malloc`. The problem is universal; the solution is the same. → Virtual Memory and Physical Frame Allocation The Linux kernel maintains a "buddy allocator" for physical memory pages, with bitmaps at each order level tracking which frames are free. When your process calls `mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS, -1, 0)`, the kernel runs a bitmap scan nearly identical to your `bitmap_find_free`, then marks the physical frame allocated, and records the mapping in your process's page table. Your filesystem's block allocator is the on-disk analogue of the kernel's physical frame allocator. → Disk Image Forensics You now understand on-disk layout precisely enough to examine a raw disk image with `hexdump -C disk.img | head -100` and identify the magic number at offset 0, the total block count at offset 12, and the free block count at offset 20. Forensic tools like `Autopsy`, `The Sleuth Kit`, and `foremost` work by reading raw disk images and interpreting known filesystem structures at known byte offsets — exactly what your `verify_filesystem` function does. With the knowledge from this milestone, you can recover files from a "deleted" (but not overwritten) partition by finding inodes in the inode table and following their block pointers. → RAID and Storage Systems RAID controllers manage arrays of physical blocks with their own parity metadata. A RAID-5 controller maintains a "stripe map" tracking which data blocks correspond

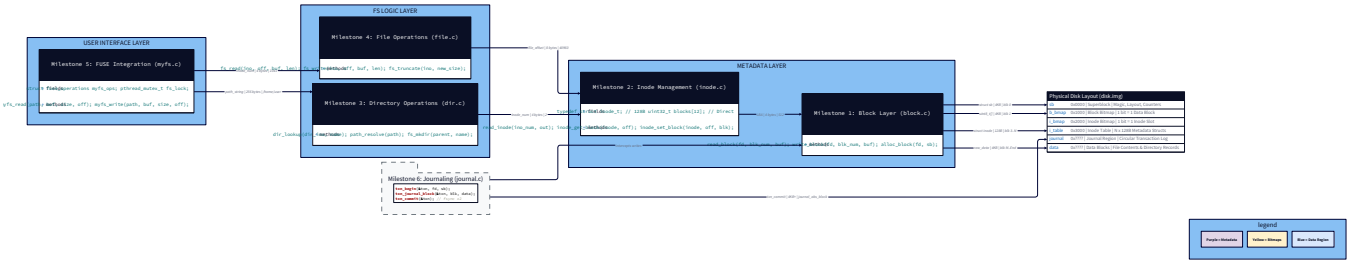
to which parity blocks — another form of block-level metadata management. The same free/used tracking problem appears at every storage abstraction layer.

Before You Move On: Pitfall Checklist

Run through this checklist before calling Milestone 1 complete:

- [ ] **Magic number validation**: does your mount code refuse images with a bad magic number?
- [ ] **Block 0 protection**: can you accidentally allocate block 0 (the superblock) as a data block? Add a guard: `assert(alloc_block(...) >= sb->data_block_start)`.
- [ ] **Bitmap index vs. disk block number**: are you consistently converting between "data block index" (0-based offset into data region) and "disk block number" (absolute block address)?
- [ ] **`fsync` on mkfs exit**: is your freshly formatted image actually on disk or still in the page cache?
- [ ] **`__attribute__((packed))` on all disk structs**: did you verify `sizeof(superblock_t) == BLOCK_SIZE` with `_Static_assert`?
- [ ] **Root inode link count = 2**: `.` and `..` each count as one hard link. This is a filesystem invariant that tools like `fsck` check.
- [ ] **Superblock free count accuracy**: after `mkfs`, `sb.free_blocks` must equal `total_blocks - data_block_start - 1` (one block used by root directory data). Off-by-one here causes cascading errors in Milestone 2.

What You've Built and What Comes Next



You started with nothing. A flat file. You gave it a grammar: block 0 is the superblock, blocks 1 and 2 are bitmaps, then the inode table, then the journal, then data. You built tools to allocate and free blocks and inodes with one-bit precision. You formatted a raw image into a valid filesystem with a root directory. But look at what the root directory actually contains right now: two entries (`.` and `..`) sitting in one data block. The "inode" for the root directory contains a block pointer to that data block. The "inode" has a size, a mode, and a link count — but those fields are barely populated. In Milestone 2, you build the full inode structure: the direct and indirect block pointer tree that lets a single inode track a file from 1 byte to several gigabytes. That's where the on-disk layout stops being about *where* things live, and

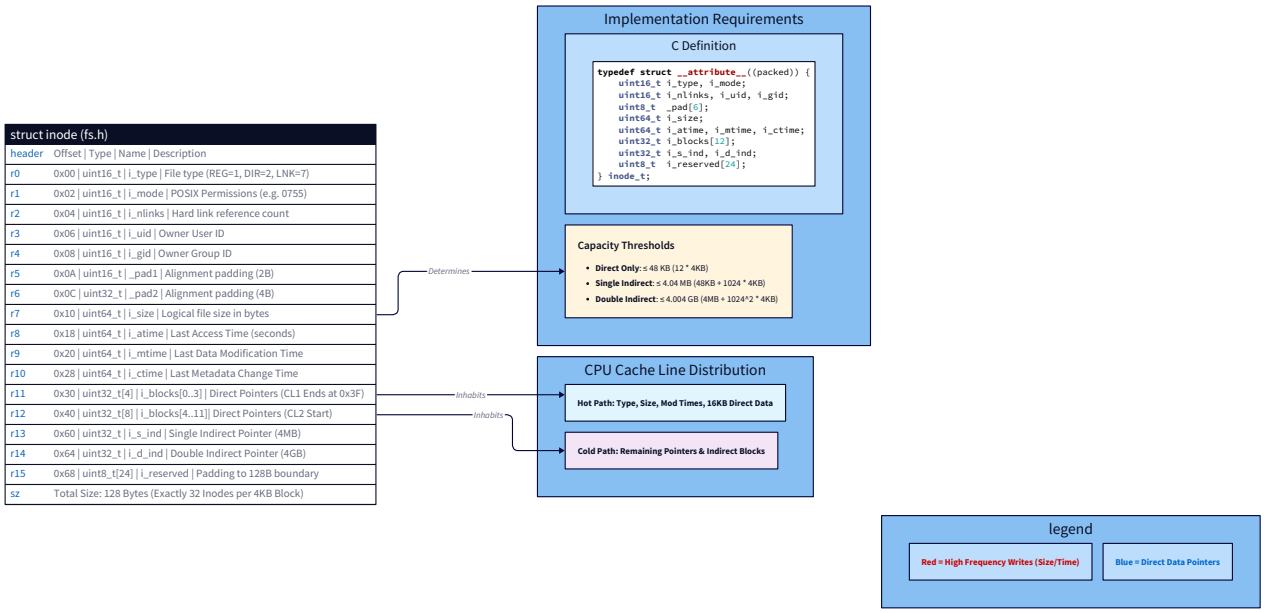
starts being about *how much* they can grow. The disk doesn't know you exist. But now you know how to make it remember anything you want.

Milestone 2: Inode Management

A File Is Not Its Name, Not Its Data, Not Its Location

You have used files your entire programming career. You have opened them, read them, written to them, deleted them. You have a mental model — reasonable, intuitive, and almost entirely wrong about what is actually happening on disk. Here is the model most developers carry: a file is a named container. The name tells you where it is. The contents are stored somewhere nearby. The file "has" its data the way a folder holds documents. Now consider this: you can have two names for the same file (hard links), and deleting one name does not delete the file. You can have a file that is 10 gigabytes according to `ls -l` but occupies only 4 kilobytes on disk. You can rename a file without moving a single byte of its data. You can have a file with no name at all, still open, still readable, living in limbo between deletion and the last `close()`. None of this makes sense under the "file is a named container" model. It makes perfect sense once you understand the inode. The fundamental design of Unix-derived filesystems separates three things that seem inseparable:

1. **The name** — lives in a directory entry. Just a string pointing to an inode number.
2. **The metadata** — lives in the inode. Size, permissions, timestamps, owner, block pointers.
3. **The data** — lives in data blocks. The inode points to them; they do not know they belong to any file. The inode is the **identity** of a file. Names come and go. Data blocks are allocated and freed. The inode is the fixed center around which everything else orbits. This milestone, you build the inode — 128 bytes of metadata that can describe a file from 0 bytes to several gigabytes.



The Fundamental Tension: Fixed Metadata, Variable Data

Here is the physical constraint that makes this design challenging: The inode must be fixed-size. The files it describes have variable size. An inode occupies exactly 128 bytes in the inode table (or 256 bytes in ext4, but always fixed). This size cannot change after the filesystem is formatted. Yet a text file might be 47 bytes, while a disk image might be 40 gigabytes. The same inode structure must represent both. If you tried to embed the block pointer list directly in the inode scaled to the file size, you would need variable-size inodes — which makes the inode table impossible to index directly (you cannot do `inode_table_block + inode_num * sizeof(inode_t)` if inodes have different sizes). You would need a separate map of inode locations, which is just pushing the indirection one level up. The solution is indirection. For small files, the inode contains direct pointers to data blocks — fast, simple, no extra I/O. For medium files, the inode contains a pointer to a *block full of pointers* — one extra disk read, many more blocks reachable. For large files, the inode contains a pointer to a *block full of pointers to blocks full of pointers* — two extra disk reads, files up to several gigabytes. This is the key insight to internalize before you write a single line of code: the inode does not contain data. The inode contains a tree of pointers to data. The tree has a fixed root (the inode fields), but it can grow additional levels as the file grows. Let's build it.

The Inode Structure: 128 Bytes That Describe a Universe

Every field in the inode exists for a specific operational reason. There is no dead weight here — this is 128 bytes under tight space pressure. [[EXPLAIN:on-disk-serialization-—-packing-structs-to-exact-byte-layouts-with-fixed-width-types|On-disk serialization — packing structs to exact byte layouts with fixed-width types]]

```c
#include <stdint.h>

#include <time.h>

#define BLOCK_SIZE          4096

#define INODE_SIZE          128

#define INODES_PER_BLOCK    (BLOCK_SIZE / INODE_SIZE)   /* 32 */

#define N_DIRECT            12     /* number of direct block pointers */

/*

 * Inode structure — always exactly 128 bytes on disk.

 *

 * BYTE LAYOUT (all fields little-endian, packed):

 *   Offset  0: mode           (2 bytes) — file type + permissions

 *   Offset  2: uid            (2 bytes) — owner user ID

 *   Offset  4: gid            (2 bytes) — owner group ID

 *   Offset  6: nlinks         (2 bytes) — hard link count

 *   Offset  8: size           (4 bytes) — file size in bytes

 *   Offset 12: atime          (4 bytes) — last access time (Unix timestamp)

 *   Offset 16: mtime          (4 bytes) — last modification time

 *   Offset 20: ctime          (4 bytes) — last metadata change time

 *   Offset 24: blocks[0..11]  (48 bytes) — 12 direct block pointers

 *   Offset 72: single_indirect (4 bytes)

 *   Offset 76: double_indirect (4 bytes)

 *   Offset 80: reserved       (48 bytes) — pad to 128 bytes

 *   Total: 128 bytes exactly

 */

typedef struct __attribute__((packed)) {

    uint16_t mode;                  /* file type bits + permission bits */

    uint16_t uid;                   /* owner UID */

    uint16_t gid;                   /* owner GID */

    uint16_t nlinks;                /* hard link count; free inode when this hits 0 */

    uint32_t size;                  /* file size in bytes (logical, not disk usage) */

    uint32_t atime;                 /* last access time */
```

```
    uint32_t mtime;                /* last data modification time */
    uint32_t ctime;                /* last inode change time (chmod, chown, link) */
    uint32_t blocks[N_DIRECT];     /* direct block pointers: blocks[0] is file offset 0 */
    uint32_t single_indirect;      /* points to a block containing 1024 block pointers */
    uint32_t double_indirect;      /* points to a block of single-indirect pointers */
    uint8_t  reserved[48];         /* future use; must be zeroed */
} inode_t;
_Static_assert(sizeof(inode_t) == INODE_SIZE,
               "inode_t must be exactly 128 bytes");
```

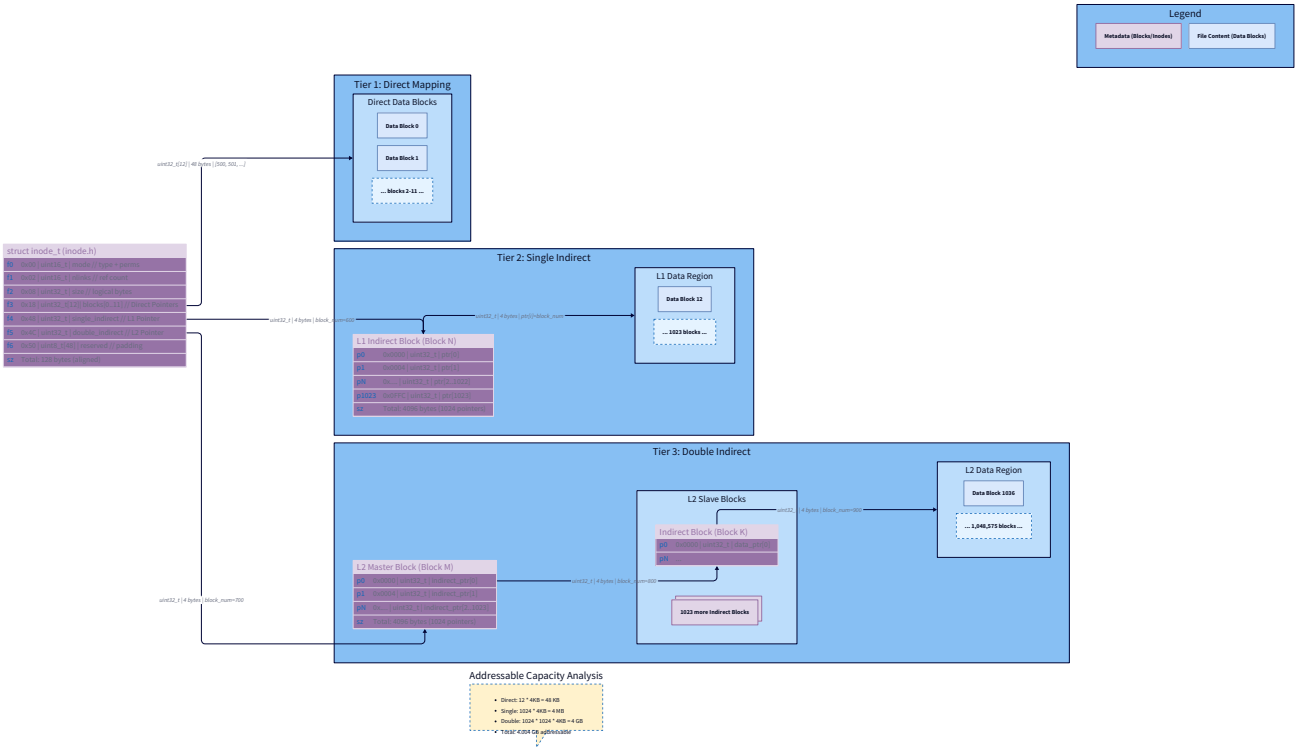
Let's examine each field's purpose carefully: **`mode` (2 bytes)**: This single 16-bit field encodes two things at once. The high bits identify the *file type* (regular file, directory, symbolic link, device, etc.). The low 12 bits encode *permissions* (read/write/execute for owner, group, and others — the familiar `rwxr-xr-x` pattern). The POSIX standard defines these bit positions:

C

```
/* File type bits (high 4 bits of mode, shifted into bits 12-15) */
#define S_IFMT   0170000  /* mask for file type bits */
#define S_IFREG  0100000  /* regular file */
#define S_IFDIR  0040000  /* directory */
#define S_IFLNK  0120000  /* symbolic link */
#define S_IFBLK  0060000  /* block device */
#define S_IFCHR  0020000  /* character device */
#define S_IFIFO  0010000  /* named pipe (FIFO) */
/* Permission bits (low 9 bits) */
#define S_IRUSR  0400     /* owner read */
#define S_IWUSR  0200     /* owner write */
#define S_IXUSR  0100     /* owner execute */
#define S_IRGRP  0040     /* group read */
#define S_IROTH  0004     /* others read */
/* ... etc. */
/* Helper macros to test file type */
#define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR)
#define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG)
```

A regular file with `rwxr-xr-x` permissions: `mode = 0100755` (octal). A directory with `rwxr-xr-x`: `mode = 0040755`. **`nlinks` (2 bytes)**: The hard link count. Every directory entry pointing to this inode increments this counter. When you `rm` a file, the directory entry is removed and `nlinks` decrements. Only when `nlinks` hits 0 *and* no process has the file open should the inode be freed. This is reference counting — the same principle used by C++'s `shared_ptr` and Rust's `Arc`. The inode is the shared resource; directory entries are the shared references. **`size` (4 bytes)**: The *logical* file size in bytes. This is what `ls -l` shows you. It is **not** the amount of disk space the file consumes. A sparse file with 1 byte written at offset 999,999,999 has `size = 1,000,000,000` but allocates only 2 data blocks (8KB). The relationship between `size` and actual disk usage is the heart of sparse files, discussed below. **`atime`, `mtime`, `ctime` (4 bytes each)**: Unix timestamps (seconds since January 1, 1970).

- `atime` updates on every read (often disabled in modern Linux with `noatime` mount option for performance — every read causing a write is expensive)
- `mtime` updates when file *data* changes
- `ctime` updates when the inode *metadata* changes (permissions, link count, owner) — note: `ctime` is NOT "creation time"; Unix has no native creation time field in the traditional inode **`blocks[N_DIRECT]` (48 bytes)**: Twelve `uint32_t` values. Each is a disk block number, or zero if that block is not allocated. `blocks[0]` contains the data starting at file offset 0. `blocks[1]` contains the data at file offset 4096. `blocks[11]` contains data at offset 45,056. Together they address the first 48KB of any file. **`single_indirect` (4 bytes)**: A pointer to an *indirect block* — a 4KB block whose entire contents are an array of `uint32_t` block pointers. Each pointer in that indirect block points to a data block. A 4KB indirect block holds `4096 / 4 = 1024` pointers. So the single-indirect pointer can reach an additional `1024 × 4KB = 4MB` of file data. **`double_indirect` (4 bytes)**: A pointer to a block containing 1024 *single-indirect block pointers*. Each of those points to a block with 1024 data block pointers. Total capacity: `1024 × 1024 × 4KB = 4GB`.



> **Why stop at double-indirect?** Most filesystems add a triple-indirect pointer for files up to 4TB. For this project, double-indirect gives you 4GB+ files which is sufficient. ext2/ext3 use triple-indirect. ext4 abandoned this entire scheme in favor of "extents" — contiguous run-length encoded block ranges — which are far more efficient for large sequential files. We'll note where extents fit in the knowledge cascade.

Reading and Writing Inodes: The Inode Table

Milestone 1 established that inodes live in the inode table — a contiguous region of blocks starting at `sb->inode_table_start`. With 32 inodes per block (`INODES_PER_BLOCK`), inode number `N` lives in:

```
block  = inode_table_start + (N - 1) / INODES_PER_BLOCK
offset = ((N - 1) % INODES_PER_BLOCK) * INODE_SIZE
```

The `N - 1` converts from 1-based inode numbers (inode 0 is reserved/invalid) to 0-based array indices.

```c
/*
 * Read inode from disk.
 * inode_num: 1-based inode number (must be >= 1 and <= total_inodes)
 * out: caller-allocated inode_t to populate
 * Returns 0 on success, negative errno on failure.
 */
int read_inode(int fd, const superblock_t *sb,
               uint32_t inode_num, inode_t *out) {
    if (inode_num == 0 || inode_num > sb->total_inodes) {
        return -EINVAL;
    }
    uint32_t idx           = inode_num - 1;
    uint32_t block_num     = sb->inode_table_start + idx / INODES_PER_BLOCK;
    uint32_t slot_in_block = idx % INODES_PER_BLOCK;
    uint8_t buf[BLOCK_SIZE];
    int ret = read_block(fd, block_num, buf);
    if (ret != 0) return ret;
    memcpy(out, buf + slot_in_block * INODE_SIZE, INODE_SIZE);
    return 0;
}
/*
 * Write inode to disk.
 * This is a read-modify-write: we read the block, update one inode slot,
 * then write the whole block back. We cannot write just the 128-byte inode
 * because the block device only supports full-block writes.
 */
int write_inode(int fd, const superblock_t *sb,
                uint32_t inode_num, const inode_t *inode) {
    if (inode_num == 0 || inode_num > sb->total_inodes) {
        return -EINVAL;
    }
```

```
    uint32_t idx          = inode_num - 1;
    uint32_t block_num    = sb->inode_table_start + idx / INODES_PER_BLOCK;
    uint32_t slot_in_block = idx % INODES_PER_BLOCK;
    uint8_t buf[BLOCK_SIZE];
    int ret = read_block(fd, block_num, buf);
    if (ret != 0) return ret;
    memcpy(buf + slot_in_block * INODE_SIZE, inode, INODE_SIZE);
    return write_block(fd, block_num, buf);
}
```

> **Hardware Soul — Read-Modify-Write on Inode Updates**: Every `write_inode` call is actually a read-modify-write cycle. You read 4096 bytes (one block containing 32 inodes), modify 128 bytes in memory, then write 4096 bytes back. This means every inode update costs 2 disk I/Os instead of 1. Real filesystems address this through a page cache: the block is likely already in RAM from a recent read, so the "read" is a cache hit at ~100ns. Only the write hits disk. But the write-amplification factor of 32× (writing 4096 bytes to modify 128) is real and unavoidable at the block device level. This is one reason ext4 packs multiple inode modifications into a single transaction.

Inode Allocation and Deallocation

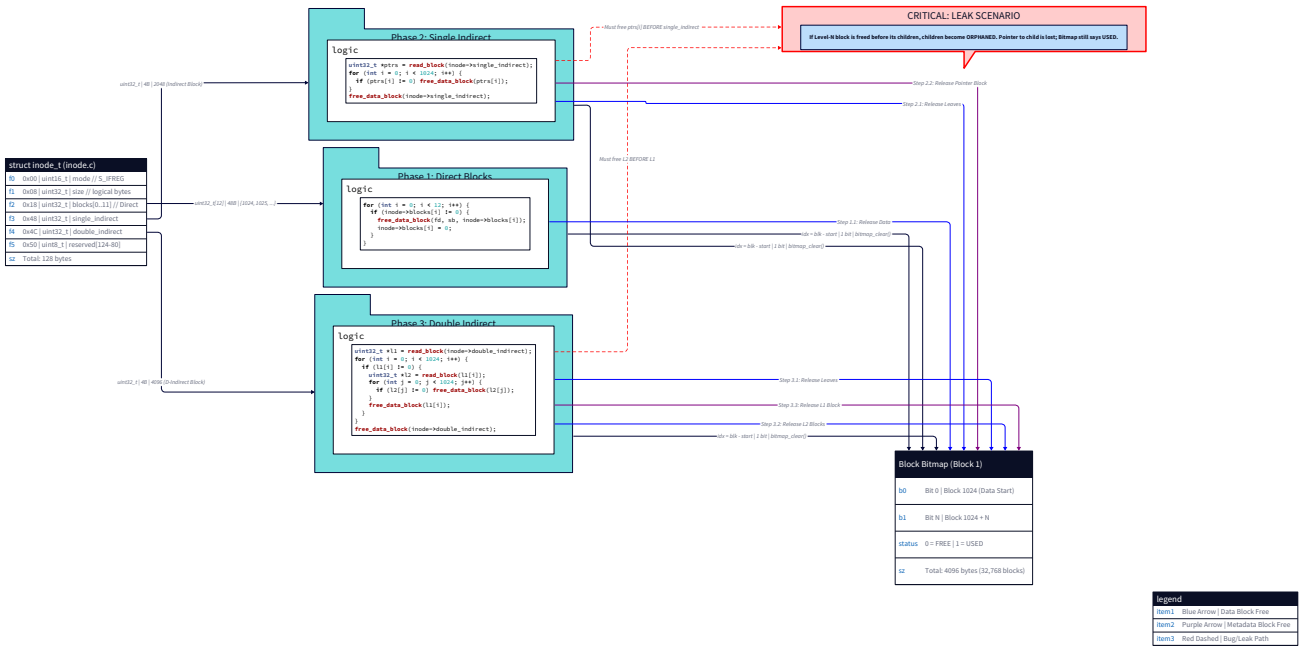
Allocating an inode follows the same bitmap pattern as block allocation from Milestone 1:

```c
#define INODE_BITMAP_BLOCK  2

/*
 * Allocate a new inode.
 * Returns the inode number (1-based) on success, -1 if none free.
 * The caller is responsible for initializing and writing the inode.
 */
int alloc_inode(int fd, superblock_t *sb) {
    if (sb->free_inodes == 0) return -1;
    uint8_t bitmap[BLOCK_SIZE];
    if (read_block(fd, INODE_BITMAP_BLOCK, bitmap) != 0) return -1;
    /* Find first free bit (0 = free) */
    uint32_t total_inodes = sb->total_inodes;
    for (uint32_t byte_idx = 0; byte_idx < (total_inodes + 7) / 8; byte_idx++) {
        if (bitmap[byte_idx] == 0xFF) continue;
        for (int bit = 0; bit < 8; bit++) {
            uint32_t idx = byte_idx * 8 + bit;
            if (idx >= total_inodes) return -1;
            if (!((bitmap[byte_idx] >> bit) & 1)) {
                /* Found free inode */
                bitmap[byte_idx] |= (1u << bit);
                if (write_block(fd, INODE_BITMAP_BLOCK, bitmap) != 0) return -1;
                sb->free_inodes--;
                return (int)(idx + 1);   /* convert 0-based index → 1-based number */
            }
        }
    }
    return -1;
}
```

Deallocation is where the real complexity lives. To free an inode, you must:

1. Free all **direct** data blocks (up to 12)
2. If `single_indirect != 0`: read that block, free all non-zero block pointers within it, then free the indirect block itself

3. If `double_indirect != 0`: read that block, for each non-zero entry (which is itself a single-indirect block), read that block, free all its data block pointers, free the single-indirect block, then free the double-indirect block
4. Clear the inode in the inode table (zero it out)
5. Clear the inode's bit in the inode bitmap Steps 2 and 3 describe **recursive block freeing** — following the pointer tree to its leaves and freeing bottom-up. Missing this is the most common bug in inode implementations: you free the data blocks and the inode, but orphan the indirect pointer blocks, permanently leaking them until a `fsck` scan.



```c
/*
 * Free a single data block by its disk block number.
 * Clears the bit in the block bitmap and updates sb->free_blocks.
 */
static int free_data_block(int fd, superblock_t *sb, uint32_t disk_block_num) {
    if (disk_block_num == 0) return 0;    /* null pointer — nothing to free */
    if (disk_block_num < sb->data_block_start ||
        disk_block_num >= sb->total_blocks) {
        return -EINVAL;   /* never free metadata blocks */
    }
    uint8_t bitmap[BLOCK_SIZE];
    if (read_block(fd, BLOCK_BITMAP_BLOCK, bitmap) != 0) return -EIO;
    uint32_t data_idx = disk_block_num - sb->data_block_start;
    bitmap[data_idx / 8] &= ~(1u << (data_idx % 8));
    if (write_block(fd, BLOCK_BITMAP_BLOCK, bitmap) != 0) return -EIO;
    sb->free_blocks++;
    return 0;
}
/*
 * Free all blocks pointed to by a single-indirect block,
 * then free the indirect block itself.
 *
 * indirect_block_num: disk block number of the single-indirect block.
 *                     If 0, this is a null pointer (sparse region) — do nothing.
 */
static int free_single_indirect(int fd, superblock_t *sb,
                                uint32_t indirect_block_num) {
    if (indirect_block_num == 0) return 0;
    uint8_t buf[BLOCK_SIZE];
    if (read_block(fd, indirect_block_num, buf) != 0) return -EIO;
    /* The indirect block contains 1024 uint32_t block pointers */
```

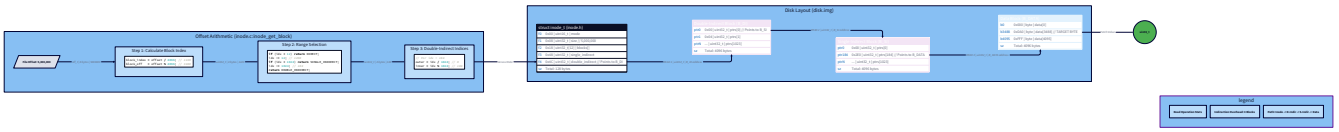
```c
    uint32_t *ptrs = (uint32_t *)buf;
    uint32_t ptrs_per_block = BLOCK_SIZE / sizeof(uint32_t);  /* 1024 */
    for (uint32_t i = 0; i < ptrs_per_block; i++) {
        if (ptrs[i] != 0) {
            free_data_block(fd, sb, ptrs[i]);
        }
    }
    /* Now free the indirect block itself */
    return free_data_block(fd, sb, indirect_block_num);
}
/*
 * Free all blocks reachable through a double-indirect block,
 * then free the double-indirect block itself.
 */
static int free_double_indirect(int fd, superblock_t *sb,
                                uint32_t dindirect_block_num) {
    if (dindirect_block_num == 0) return 0;
    uint8_t buf[BLOCK_SIZE];
    if (read_block(fd, dindirect_block_num, buf) != 0) return -EIO;
    uint32_t *ptrs = (uint32_t *)buf;
    uint32_t ptrs_per_block = BLOCK_SIZE / sizeof(uint32_t);
    for (uint32_t i = 0; i < ptrs_per_block; i++) {
        if (ptrs[i] != 0) {
            /* Each entry here is itself a single-indirect block */
            free_single_indirect(fd, sb, ptrs[i]);
        }
    }
    return free_data_block(fd, sb, dindirect_block_num);
}
/*
 * Free an inode and all its associated data blocks.
```

```c
 * After this call, inode_num may be reallocated to a new file.
 */
int free_inode(int fd, superblock_t *sb, uint32_t inode_num) {
    inode_t inode;
    if (read_inode(fd, sb, inode_num, &inode) != 0) return -EIO;
    /* Step 1: Free all direct data blocks */
    for (int i = 0; i < N_DIRECT; i++) {
        if (inode.blocks[i] != 0) {
            free_data_block(fd, sb, inode.blocks[i]);
        }
    }
    /* Step 2: Free single-indirect chain */
    free_single_indirect(fd, sb, inode.single_indirect);
    /* Step 3: Free double-indirect chain */
    free_double_indirect(fd, sb, inode.double_indirect);
    /* Step 4: Zero the inode in the table */
    inode_t zeroed;
    memset(&zeroed, 0, sizeof(zeroed));
    if (write_inode(fd, sb, inode_num, &zeroed) != 0) return -EIO;
    /* Step 5: Clear inode bitmap bit */
    uint8_t bitmap[BLOCK_SIZE];
    if (read_block(fd, INODE_BITMAP_BLOCK, bitmap) != 0) return -EIO;
    uint32_t idx = inode_num - 1;
    bitmap[idx / 8] &= ~(1u << (idx % 8));
    if (write_block(fd, INODE_BITMAP_BLOCK, bitmap) != 0) return -EIO;
    sb->free_inodes++;
    return 0;
}
```

> **Pitfall — Bitmap Load on Every Block Free**: The `free_data_block` function above reads and writes the bitmap for every single block freed. Freeing an inode with 12 direct blocks + 1024 indirect blocks = 1036 blocks freed = 2072 bitmap I/Os. This is acceptable for correctness but catastrophically slow for production. Real implementations load the bitmap once, clear all bits in memory, then write it back once. For now, correctness first. When you integrate with Milestone 6 journaling, you will batch these operations anyway.

The Block Pointer Tree: File Offset to Disk Block

The most important operation on an inode — used by every read and write — is **translating a file offset to a disk block number**. Given "byte offset 150,000 in this file, which disk block is it in?", the function must navigate the pointer tree.



Let's work through the math first, then implement it:

```
File offset 0 to 49,151:
  Block index 0–11 → inode.blocks[0] through inode.blocks[11]
  (12 direct blocks × 4096 bytes = 49,152 bytes)
File offset 49,152 to 4,243,455:
  Block index 12 to 1035 → single-indirect region
  (1024 blocks × 4096 = 4,194,304 bytes = 4MB)
  To find block at file offset X in single-indirect range:
    single_idx = (X / BLOCK_SIZE) - N_DIRECT
    read inode.single_indirect block
    return ptrs[single_idx]
File offset 4,243,456 and beyond:
  Block index 1036 and beyond → double-indirect region
  block_index = X / BLOCK_SIZE
  dbl_idx = block_index - N_DIRECT - PTRS_PER_BLOCK   /* offset into double-indirect */
  outer = dbl_idx / PTRS_PER_BLOCK   /* which single-indirect block within the double */
  inner = dbl_idx % PTRS_PER_BLOCK   /* which data block within that single-indirect */
  read inode.double_indirect block → get ptrs[outer] (a single-indirect block)
  read ptrs[outer] block → get ptrs[inner] (the data block)
```

```c
#define PTRS_PER_BLOCK      (BLOCK_SIZE / sizeof(uint32_t))    /* 1024 */
#define DIRECT_LIMIT        ((uint64_t)N_DIRECT * BLOCK_SIZE)
#define SINGLE_LIMIT        (DIRECT_LIMIT + (uint64_t)PTRS_PER_BLOCK * BLOCK_SIZE)
#define DOUBLE_LIMIT        (SINGLE_LIMIT + (uint64_t)PTRS_PER_BLOCK * PTRS_PER_BLOCK * BLOCK_SIZE)
/*
 * Translate a file byte offset to a disk block number.
 *
 * offset: byte offset within the file
 * inode:  the file's inode
 * out_block_num: set to the disk block number (0 if the block is a hole)
 *
 * Returns 0 on success, negative errno on I/O error.
 * Returns 0 with *out_block_num = 0 for sparse regions (valid, not an error).
 */
int inode_get_block(int fd, const inode_t *inode,
                    uint64_t offset, uint32_t *out_block_num) {
    uint64_t block_index = offset / BLOCK_SIZE;
    *out_block_num = 0;    /* default: hole / unallocated */
    if (offset >= DOUBLE_LIMIT) {
        return -EFBIG;    /* beyond our addressing capacity */
    }
    /* --- Direct region --- */
    if (block_index < N_DIRECT) {
        *out_block_num = inode->blocks[block_index];
        return 0;
    }
    /* --- Single-indirect region --- */
    if (block_index < N_DIRECT + PTRS_PER_BLOCK) {
        if (inode->single_indirect == 0) {
            *out_block_num = 0;    /* entire region is a hole */
            return 0;
```

```
        }

        uint8_t buf[BLOCK_SIZE];

        if (read_block(fd, inode->single_indirect, buf) != 0) return -EIO;

        uint32_t *ptrs = (uint32_t *)buf;

        uint32_t single_idx = (uint32_t)(block_index - N_DIRECT);

        *out_block_num = ptrs[single_idx];

        return 0;

    }

    /* --- Double-indirect region --- */

    if (inode->double_indirect == 0) {

        *out_block_num = 0;    /* entire double-indirect region is a hole */

        return 0;

    }

    uint64_t dbl_offset = block_index - N_DIRECT - PTRS_PER_BLOCK;

    uint32_t outer_idx  = (uint32_t)(dbl_offset / PTRS_PER_BLOCK);

    uint32_t inner_idx  = (uint32_t)(dbl_offset % PTRS_PER_BLOCK);

    /* Read double-indirect block → get outer single-indirect block pointer */

    uint8_t buf[BLOCK_SIZE];

    if (read_block(fd, inode->double_indirect, buf) != 0) return -EIO;

    uint32_t *outer_ptrs = (uint32_t *)buf;

    uint32_t single_block = outer_ptrs[outer_idx];

    if (single_block == 0) {

        *out_block_num = 0;    /* this single-indirect slot is a hole */

        return 0;

    }

    /* Read single-indirect block → get data block pointer */

    if (read_block(fd, single_block, buf) != 0) return -EIO;

    uint32_t *inner_ptrs = (uint32_t *)buf;

    *out_block_num = inner_ptrs[inner_idx];

    return 0;

}
```

The corresponding "set block" operation — assigning a disk block number to a given file offset — is used by the write path in Milestone 4. It must create indirect and double-indirect blocks if they don't yet exist:

```c
/*
 * Set the disk block number for a given file offset.
 * If indirect blocks need to be created to hold the pointer, allocates them.
 *
 * Returns 0 on success, negative errno on failure.
 */
int inode_set_block(int fd, superblock_t *sb, inode_t *inode,
                    uint64_t offset, uint32_t disk_block_num) {
    uint64_t block_index = offset / BLOCK_SIZE;
    if (offset >= DOUBLE_LIMIT) return -EFBIG;
    /* --- Direct region --- */
    if (block_index < N_DIRECT) {
        inode->blocks[block_index] = disk_block_num;
        return 0;
    }
    /* --- Single-indirect region --- */
    if (block_index < N_DIRECT + PTRS_PER_BLOCK) {
        /* Allocate the indirect block if it doesn't exist yet */
        if (inode->single_indirect == 0) {
            int blk = alloc_block(fd, sb);
            if (blk < 0) return -ENOSPC;
            /* Zero the new indirect block */
            uint8_t zeros[BLOCK_SIZE];
            memset(zeros, 0, BLOCK_SIZE);
            if (write_block(fd, (uint32_t)blk, zeros) != 0) return -EIO;
            inode->single_indirect = (uint32_t)blk;
        }
        uint8_t buf[BLOCK_SIZE];
        if (read_block(fd, inode->single_indirect, buf) != 0) return -EIO;
        uint32_t *ptrs = (uint32_t *)buf;
        uint32_t single_idx = (uint32_t)(block_index - N_DIRECT);
```

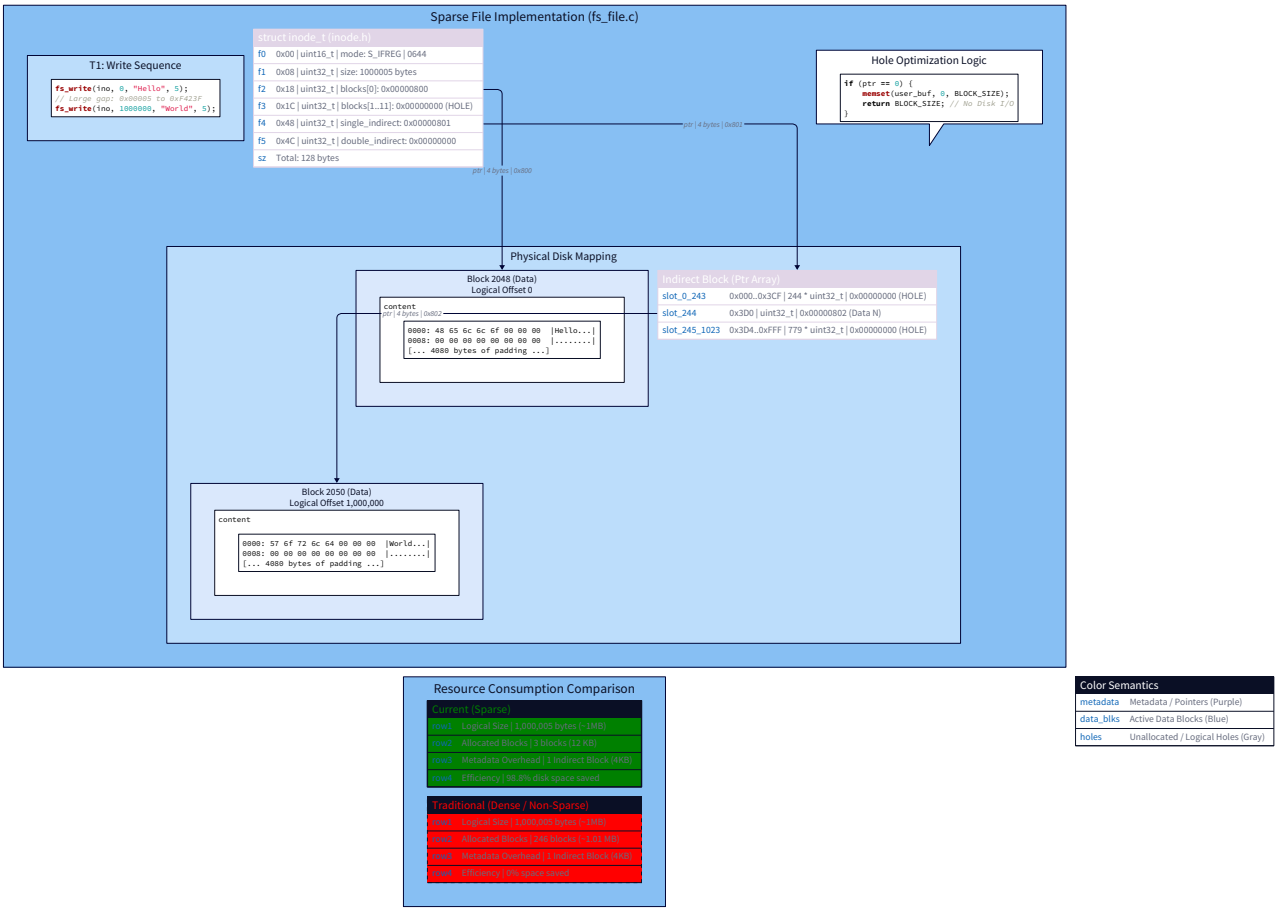
```
        ptrs[single_idx] = disk_block_num;
        return write_block(fd, inode->single_indirect, buf);
    }
    /* --- Double-indirect region --- */
    uint64_t dbl_offset = block_index - N_DIRECT - PTRS_PER_BLOCK;
    uint32_t outer_idx  = (uint32_t)(dbl_offset / PTRS_PER_BLOCK);
    uint32_t inner_idx  = (uint32_t)(dbl_offset % PTRS_PER_BLOCK);
    /* Allocate the double-indirect block if needed */
    if (inode->double_indirect == 0) {
        int blk = alloc_block(fd, sb);
        if (blk < 0) return -ENOSPC;
        uint8_t zeros[BLOCK_SIZE];
        memset(zeros, 0, BLOCK_SIZE);
        if (write_block(fd, (uint32_t)blk, zeros) != 0) return -EIO;
        inode->double_indirect = (uint32_t)blk;
    }
    uint8_t buf[BLOCK_SIZE];
    if (read_block(fd, inode->double_indirect, buf) != 0) return -EIO;
    uint32_t *outer_ptrs  = (uint32_t *)buf;
    uint32_t single_block = outer_ptrs[outer_idx];
    /* Allocate the single-indirect block within the double-indirect if needed */
    if (single_block == 0) {
        int blk = alloc_block(fd, sb);
        if (blk < 0) return -ENOSPC;
        uint8_t zeros[BLOCK_SIZE];
        memset(zeros, 0, BLOCK_SIZE);
        if (write_block(fd, (uint32_t)blk, zeros) != 0) return -EIO;
        outer_ptrs[outer_idx] = (uint32_t)blk;
        single_block = (uint32_t)blk;
        if (write_block(fd, inode->double_indirect, buf) != 0) return -EIO;
    }
```

```
    /* Now read the single-indirect block and set the data pointer */
    if (read_block(fd, single_block, buf) != 0) return -EIO;
    uint32_t *inner_ptrs = (uint32_t *)buf;
    inner_ptrs[inner_idx] = disk_block_num;
    return write_block(fd, single_block, buf);
}
```

[[EXPLAIN:multi-level-indirection-—-pointer-to-pointer-to-data-pattern|Multi-level indirection — pointer-to-pointer-to-data pattern]]

Sparse Files: When Nothing Is Something

Look at `inode_get_block` again. Notice what happens when `block_index < N_DIRECT` and `inode->blocks[block_index] == 0`: you return `*out_block_num = 0` with no error. A zero block pointer means the block is **unallocated** — it is a *hole* in the file. When a read operation encounters a zero block pointer, it must return zero bytes without doing any disk I/O.



This is how sparse files work. Consider:

```c
int create_sparse_file_example(int fd, superblock_t *sb, uint32_t inode_num) {
    inode_t inode;
    memset(&inode, 0, sizeof(inode));
    inode.mode = S_IFREG | 0644;
    inode.nlinks = 1;
    /* Write one byte at offset 0 */
    uint32_t block0 = (uint32_t)alloc_block(fd, sb);
    uint8_t buf[BLOCK_SIZE];
    memset(buf, 0, BLOCK_SIZE);
    buf[0] = 'A';
    write_block(fd, block0, buf);
    inode.blocks[0] = block0;
    /* Write one byte at offset 999,999,999 — deep in double-indirect range */
    /* inode_set_block will allocate the indirect blocks as needed */
    uint32_t blockN = (uint32_t)alloc_block(fd, sb);
    memset(buf, 0, BLOCK_SIZE);
    buf[999999999 % BLOCK_SIZE] = 'Z';
    write_block(fd, blockN, buf);
    inode_set_block(fd, sb, &inode, 999999999, blockN);
    /* File size: 1,000,000,000 bytes */
    inode.size = 1000000000;
    /* But disk usage: 2 data blocks + ~2 indirect blocks = ~16KB */
    write_inode(fd, sb, inode_num, &inode);
    return 0;
}
```

Reading byte 500,000,000 (which is between the two written offsets) will encounter a null pointer and return a zero byte — no disk I/O for the hole. The file appears to be 1GB, but occupies almost no disk space. This is why `ls -l` and `du` report different sizes:

```
$ ls -l vm_disk.img
-rw-r--r-- 1 user user 10737418240 Mar 1 12:00 vm_disk.img   # 10GB
$ du -sh vm_disk.img
4.0K  vm_disk.img   # Only 4KB actually allocated
```

Three-Level View — Where Zero Bytes Come From

Level	What Happens When You Read a Sparse Hole
Application	Calls `read(fd, buf, len)`, expects data
Filesystem	`inode_get_block` returns 0 for the block pointer. Instead of calling `read_block`, the read handler calls `memset(buf, 0, BLOCK_SIZE)` — zero-fill in RAM
Hardware	No disk I/O at all. The zeros are generated entirely in CPU registers and copied to the output buffer. Reads from holes are **faster** than reads from allocated blocks

---

Timestamps: The Three Clocks

Every inode carries three timestamps, and getting them right matters — tools like `rsync`, `make`, and backup software depend on correct timestamp semantics.

```c
#include <time.h>

/*
 * Update inode timestamps based on the operation performed.
 *
 * Note: In production, use clock_gettime(CLOCK_REALTIME) for
 * nanosecond precision. We use time() for simplicity.
 */
static uint32_t now_ts(void) {
    return (uint32_t)time(NULL);
}

/* Called when file data is read */
void inode_touch_atime(inode_t *inode) {
    inode->atime = now_ts();
    /* Note: atime update requires a write_inode() call by the caller.
     * This is why noatime is popular: every read causes a metadata write. */
}

/* Called when file data is written */
void inode_touch_mtime(inode_t *inode) {
    uint32_t ts = now_ts();
    inode->mtime = ts;
    inode->ctime = ts;   /* data change also changes inode */
}

/* Called when inode metadata changes (chmod, chown, link, rename) */
void inode_touch_ctime(inode_t *inode) {
    inode->ctime = now_ts();
}
```

The `ctime` field trips up developers who assume it means "creation time." Unix filesystems traditionally have no creation time (some modern filesystems add `crtime` or `btime` as an extension). `ctime` is the "inode change time" — it updates whenever anything in the inode changes, including the link count, permissions, or owner. This means that `chmod` and `chown` update `ctime` but not `mtime`. The `atime` update-on-every-

read is one of the most notorious filesystem performance problems. A disk that serves mostly reads will perform writes for atime on every read — doubling I/O load on the inode block. Linux introduced the `relatime` mount option (update atime only if it's older than mtime/ctime) and `noatime` (never update atime). Your FUSE integration in Milestone 5 will expose this as a mount option.

Block Capacity Math: Exactly How Big Can Files Get?

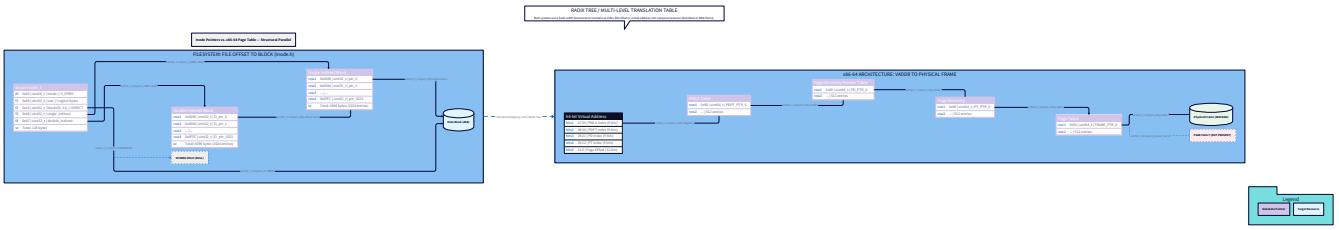
Let's work through the arithmetic precisely, because it comes up in every production filesystem conversation:

```c
/*
 * Maximum file size addressable by this inode scheme.
 *
 * Direct:          12 blocks × 4096 bytes         =        49,152 bytes  (~48 KB)
 * Single-indirect: 1024 blocks × 4096 bytes        =     4,194,304 bytes  (~4 MB)
 * Double-indirect: 1024 × 1024 × 4096 bytes        = 4,294,967,296 bytes  (~4 GB)
 *
 * Total:                                           = 4,299,210,752 bytes  (~4.004 GB)
 *
 * If we added triple-indirect (ext2/ext3 style):
 * Triple-indirect: 1024 × 1024 × 1024 × 4096      = 4,398,046,511,104 bytes (~4 TB)
 */
#define MAX_FILE_BLOCKS  (N_DIRECT + PTRS_PER_BLOCK + \
                          (uint64_t)PTRS_PER_BLOCK * PTRS_PER_BLOCK)
#define MAX_FILE_SIZE    ((uint64_t)MAX_FILE_BLOCKS * BLOCK_SIZE)
```

Note that `size` is stored as `uint32_t` in our inode (4 bytes, max ~4.29 GB). For files larger than 4GB, you would need `uint64_t` for the size field — which is how ext4, XFS, and every modern filesystem implement it. The 32-bit size field is a design limitation of our simplified implementation; it matches the original ext2 design.

The Structural Parallel: Inode Pointers and Page Tables

Here is one of the most illuminating cross-domain connections in systems programming: **Your double-indirect block pointer tree is the same structure as a 3-level CPU page table.**



When the CPU translates a virtual address to a physical address, it walks a multi-level page table:

```
Virtual address (48 bits on x86-64):
  Bits 47-39: PML4 index    → which entry in the top-level table
  Bits 38-30: PDPT index    → which entry in the 2nd-level table
  Bits 29-21: PD index      → which entry in the 3rd-level table
  Bits 20-12: PT index      → which entry in the 4th-level table (page table)
  Bits 11-0:  Page offset   → byte within the physical page (4KB)
```

Compare this to your file offset translation:

```
File offset (38 bits for our 4GB max):
  Bits 37-22: Double-indirect index → which entry in the outer pointer block
  Bits 21-12: Single-indirect index → which entry in the inner pointer block
  Bits 11-0:  Block offset          → byte within the 4KB data block
```

The structural identity is exact: both systems use a tree of fixed-size pointer arrays to translate an index into a physical resource. The MMU does this in hardware with a register (CR3 on x86) pointing to the top-level table. Your filesystem does it in software with a disk block number stored in the inode. The key property both share: **each level of indirection multiplies capacity by the fanout** (1024 for filesystem blocks, 512 for page table entries). This is a **radix tree** — fixed-width indexing that provides O(1) lookup with bounded maximum depth.

> 🔭 **Deep Dive**: x86-64 page table structure. If you want to see the hardware version of exactly what you just built, read the Intel Software Developer's Manual Volume 3, Chapter 4 ("Paging"). Section 4.5 covers 4-level paging with full page-walk diagrams. Every concept maps 1:1 to your double-indirect pointer tree.

Reference Counting: The Link Count Contract

The `nlinks` field implements **reference counting** for inodes. This is the same pattern used by `shared_ptr` in C++, `Arc<T>` in Rust, Python's garbage collector, and every operating system's file descriptor table. The contract:

- `nlinks` starts at 1 when an inode is created (one directory entry points to it)
- Every `link()` call (creating a hard link) increments `nlinks`
- Every `unlink()` call (removing a directory entry) decrements `nlinks`
- When `nlinks == 0` AND no process has the file open: free the inode
- When `nlinks == 0` but processes still have it open: defer freeing until last `close()` The "open but unlinked" case is important: you can `unlink("myfile")` on an open file, and the file continues to exist (and be readable/writable) until the last file descriptor pointing to it is closed. Only then does `nlinks` hitting 0 trigger actual deallocation. Your filesystem's "open file table" (implemented in Milestone 5) tracks open file descriptors separately from `nlinks`. For directories specifically:
- A new directory starts with `nlinks = 2` (one for its own `.` entry, one for the parent's entry pointing to it)
- Each subdirectory increments the parent's `nlinks` by 1 (because the subdirectory's `..` entry points to the parent)

- This is why deeply nested directories have high link counts, and why `find -maxdepth N` can use `nlinks` to prune the search

---

Putting It Together: Inode Initialization

When a new file or directory is created, you need to initialize the inode with correct defaults:

```c
/*
 * Initialize a newly allocated inode for a regular file.
 * Caller must call write_inode() afterward to persist it.
 */
void inode_init_file(inode_t *inode, uint16_t mode, uint16_t uid, uint16_t gid) {
    memset(inode, 0, sizeof(*inode));
    inode->mode   = (S_IFREG | (mode & 0777));
    inode->uid    = uid;
    inode->gid    = gid;
    inode->nlinks = 1;
    inode->size   = 0;
    uint32_t ts   = now_ts();
    inode->atime  = inode->mtime = inode->ctime = ts;
    /* All block pointers zeroed by memset — file starts with no allocated blocks */
    /* single_indirect and double_indirect are also 0 — both are null */
}
/*
 * Initialize a newly allocated inode for a directory.
 * nlinks starts at 2: one for the parent entry pointing to it,
 * one for the '.' entry within the directory itself.
 */
void inode_init_dir(inode_t *inode, uint16_t mode, uint16_t uid, uint16_t gid) {
    memset(inode, 0, sizeof(*inode));
    inode->mode   = (S_IFDIR | (mode & 0777));
    inode->uid    = uid;
    inode->gid    = gid;
    inode->nlinks = 2;
    inode->size   = BLOCK_SIZE;   /* directories are sized in block units */
    uint32_t ts   = now_ts();
    inode->atime  = inode->mtime = inode->ctime = ts;
}
```

Validation: Testing the Inode Implementation

Before moving to Milestone 3, verify your inode layer thoroughly:

```c
/*
 * Test suite for inode layer.
 * Run after mkfs to verify the inode implementation works correctly.
 */
void test_inode_layer(int fd, superblock_t *sb) {
    printf("\n=== Inode Layer Tests ===\n");
    /* Test 1: Allocate an inode */
    int ino = alloc_inode(fd, sb);
    assert(ino > 0);
    printf("OK: allocated inode %d\n", ino);
    /* Test 2: Initialize and write it */
    inode_t inode;
    inode_init_file(&inode, 0644, 1000, 1000);
    assert(write_inode(fd, sb, (uint32_t)ino, &inode) == 0);
    /* Test 3: Read it back and verify fields */
    inode_t read_back;
    assert(read_inode(fd, sb, (uint32_t)ino, &read_back) == 0);
    assert(S_ISREG(read_back.mode));
    assert(read_back.nlinks == 1);
    assert(read_back.size == 0);
    printf("OK: inode round-trip verified\n");
    /* Test 4: Set and get a direct block pointer */
    int blk = alloc_block(fd, sb);
    assert(blk >= 0);
    assert(inode_set_block(fd, sb, &inode, 0, (uint32_t)blk) == 0);
    uint32_t got_block;
    assert(inode_get_block(fd, &inode, 0, &got_block) == 0);
    assert(got_block == (uint32_t)blk);
    printf("OK: direct block pointer set/get\n");
    /* Test 5: Sparse file — get block for unallocated offset */
    assert(inode_get_block(fd, &inode, 4096 * 100, &got_block) == 0);
```

```
    assert(got_block == 0);    /* must be 0 — hole */
    printf("OK: sparse hole returns block 0\n");
    /* Test 6: Single-indirect region */
    int si_blk = alloc_block(fd, sb);
    uint64_t si_offset = (uint64_t)N_DIRECT * BLOCK_SIZE;   /* first single-indirect slot */
    assert(inode_set_block(fd, sb, &inode, si_offset, (uint32_t)si_blk) == 0);
    assert(inode.single_indirect != 0);   /* indirect block must have been allocated */
    assert(inode_get_block(fd, &inode, si_offset, &got_block) == 0);
    assert(got_block == (uint32_t)si_blk);
    printf("OK: single-indirect block pointer set/get\n");
    /* Test 7: Double-indirect region */
    uint64_t di_offset = (uint64_t)(N_DIRECT + PTRS_PER_BLOCK) * BLOCK_SIZE;
    int di_blk = alloc_block(fd, sb);
    assert(inode_set_block(fd, sb, &inode, di_offset, (uint32_t)di_blk) == 0);
    assert(inode.double_indirect != 0);
    assert(inode_get_block(fd, &inode, di_offset, &got_block) == 0);
    assert(got_block == (uint32_t)di_blk);
    printf("OK: double-indirect block pointer set/get\n");
    /* Test 8: Free inode — all blocks should be reclaimed */
    uint32_t free_before = sb->free_blocks;
    write_inode(fd, sb, (uint32_t)ino, &inode);
    assert(free_inode(fd, sb, (uint32_t)ino) == 0);
    /* After freeing: all direct blocks + indirect blocks reclaimed */
    printf("OK: inode freed, %u blocks reclaimed\n",
           sb->free_blocks - free_before);
    printf("=== All Inode Tests Passed ===\n\n");
}
```

Design Decisions: Why This Pointer Scheme?

The direct/indirect/double-indirect scheme is not the only way to map files to blocks. Here is how the alternatives compare:

Option	Pros	Cons	Used By
Direct + Indirect (chosen ✓)	Simple traversal logic; sparse files for free; proven design	Extra I/Os for large files; fragmentation grows with file age	ext2, ext3, original Unix FFS
Extents	Contiguous runs eliminate per-block metadata; large sequential files have 1 metadata read regardless of size; much lower fragmentation	Extent trees are complex; sparse files harder to represent	ext4, XFS (B-tree extents), NTFS, HFS+
Block map array	Trivially simple	Cannot represent large files without enormous inode	FAT (cluster chains), very small filesystems
B-tree block map	Self-balancing; handles arbitrary fragmentation gracefully	Complex implementation; high constant overhead	Btrfs (copy-on-write B-tree), ZFS

> ext4 replaced the indirect pointer scheme with extents — an extent is a tuple `(start_block, length)` that describes a contiguous run of blocks. A 1GB sequential file needs only one extent record instead of 262,144 separate block pointers. This collapses the tree to nearly constant depth for sequential files. The extent tree is a B-tree embedded in the inode's block pointer fields. Building the indirect pointer scheme first (as you're doing) is the right starting point — extents are an optimization of the same conceptual model.

Knowledge Cascade: One Inode, Ten Worlds

You have built a fixed-size metadata structure with a tree of pointers to variable-size content. This pattern is not unique to filesystems. It is the fundamental pattern for indexing variable-size data with fixed-size metadata. → B-Trees and Database Indexes Your indirect block pointer tree is a radix tree with fixed fanout (1024). A B-tree is the same concept generalized: internal nodes are "indirect blocks" holding pointers to children; leaf nodes are "data blocks" holding actual values. The branching factor varies by page size and key size. PostgreSQL's B-tree index pages, InnoDB's clustered index, and SQLite's B-tree pages all implement this same "fixed-size nodes, variable-depth tree" principle. ext4's extent tree is literally a B-tree embedded in the inode. Understanding your double-indirect pointers gives you the mental model to read any B-tree implementation and immediately see the parallel. → x86-64 Virtual Memory Page Tables As shown above, the CPU's page table walker performs the same operation as your `inode_get_block` — traversing a multi-level array of pointers to translate an index into a physical resource. CR3 register is the "inode"; page directory entries are the "double-indirect block"; page table entries are the "single-indirect block"; physical page frames are the "data blocks." If you

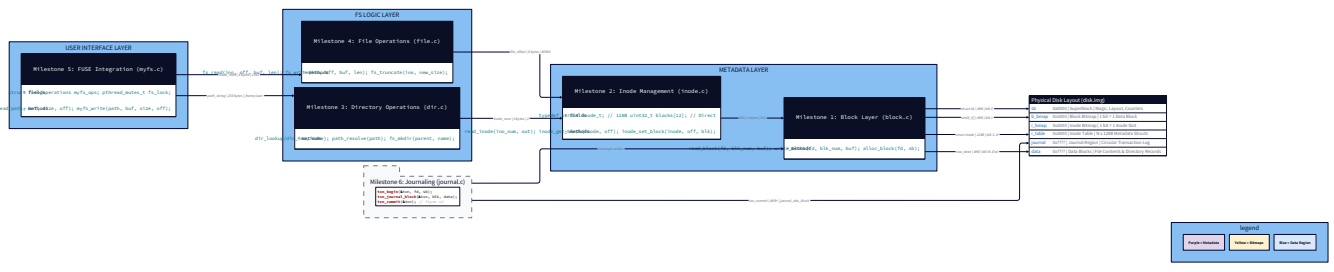
understand your `inode_get_block` function, you understand how the MMU hardware walks page tables. This insight unlocks reading `/proc/PID/maps`, understanding TLB shootdowns, and reasoning about memory-mapped file performance. → Sparse Files in Production Systems Virtual machine disk images (QCOW2, VMDK, VDI) rely on sparse file support from the host filesystem. A 100GB VM disk image containing mostly zeroes occupies only the space for actual writes on an ext4 or XFS host. `cp --sparse=always` and `rsync --sparse` are tools that preserve this sparseness. Backing databases (QCOW2 copy-on-write) work by allocating blocks on first write — exactly what your `inode_set_block` does when a block is first written. Understanding null pointers = holes means understanding how VM provisioning achieves "thin provisioning." → Reference Counting in Language Runtimes `nlinks` in the inode solves the same problem as `std::shared_ptr<T>::use_count()` in C++, `Arc::strong_count()` in Rust, CPython's `ob_refcnt`, and Swift's ARC. The pattern is universal: a shared resource tracks the number of references to it; when that count reaches zero, the resource is freed. The filesystem version has one complication that smart pointers don't: the "open but unlinked" case, where a process can hold a file descriptor to a file whose `nlinks` has dropped to zero. This is the filesystem equivalent of a "weak reference" keeping a resource alive — the `nlinks = 0` condition is the destructor trigger, but only fires when no "strong holds" (open file descriptors) remain. → Git Object Storage and Content-Addressable Systems Git's object store faces an identical design challenge: fixed-size metadata (a commit object: author, timestamp, message, parent pointers, tree pointer) pointing to variable-size content (file trees and blobs). Git's "tree objects" map filenames to blob hashes — exactly like your inode's block pointer array maps block indices to disk block numbers. Git's blob objects are like your data blocks. The metadata/content separation in your inode design is the same abstraction that makes content-addressable storage, Docker image layers, and Merkle trees work. When you understand inodes, you understand why Git's "pack files," Docker's "layers," and your filesystem's "data blocks" are all solving the same structural problem. → Object Storage Metadata (S3, GCS) Cloud object storage systems maintain metadata records for each object: size, creation time, content-type, ETag (hash), owner, access permissions. This metadata is small (hundreds of bytes) and lives in a separate system from the object data (potentially terabytes). The metadata-to-data separation is architecturally identical to your

inode. When you upload a 5TB file to S3, a few hundred bytes of metadata are written to a distributed key-value store, while the data itself flows through a separate data plane to redundant storage servers. Your inode is the on-disk instantiation of this split that cloud engineers implement at global scale.

Before You Move On: Pitfall Checklist

- [ ] **Never use block 0 as a data block**: `alloc_block` returns absolute disk block numbers starting at `data_block_start`. If you ever see `inode->blocks[i] == 0` on an allocated block, something is wrong — 0 is the superblock, not a data block. The null/hole sentinel works because legitimate data blocks always have numbers ≥ `data_block_start`.
- [ ] **Indirect block deallocation**: Call `free_single_indirect` and `free_double_indirect` in your `free_inode`. Audit this path by counting `sb->free_blocks` before and after freeing a large file — the delta should match the number of data blocks plus indirect pointer blocks.
- [ ] **Off-by-one in boundary calculations**: The single-indirect region starts at block index `N_DIRECT` (12), not `N_DIRECT + 1`. The double-indirect region starts at block index `N_DIRECT + PTRS_PER_BLOCK` (1036). An off-by-one here silently corrupts a different block than you intend.
- [ ] **`_Static_assert` on inode size**: Add `_Static_assert(sizeof(inode_t) == 128, "inode_t size mismatch")`. If you add a field and forget to shrink `reserved[]`, the compiler will catch it rather than silently misaligning every inode in the table.
- [ ] **Zero the entire inode on allocation**: `memset(&inode, 0, sizeof(inode))` before `inode_init_*`. Uninitialized garbage in `reserved[]` or unused pointer slots causes spurious non-zero pointer values that `inode_get_block` will try to read as block numbers.
- [ ] **Update `ctime` on inode writes**: Any operation that modifies the inode (permissions change, link count change, block pointer addition) must update `ctime`. Forgetting this makes `stat()` return stale metadata that confuses rsync and backup tools.
- [ ] **Bitmap flush before inode write**: When allocating blocks to satisfy `inode_set_block`, write the updated bitmap to disk before writing the updated inode. If you crash between the two writes, a bitmap that says the block is used but no inode pointing to it is a leaked block (safe). A bitmap that says the block is free but an inode points to it is double-allocation (catastrophic).

What You've Built and What Comes Next



You now have a complete inode layer. A fixed-size structure that can represent any file from 0 bytes to 4GB. A pointer tree with three levels of indirection that costs

zero extra disk reads for small files, one extra read for medium files, and two extra reads for large files. Sparse files where null pointers cost nothing. Reference counting that properly tracks all names pointing to a file. And the ability to serialize all of this to and from exact byte positions on disk. But an inode has no name. Inodes don't know what they're called — that knowledge lives in the directory. A directory is a special kind of file whose data blocks contain structured records mapping names to inode numbers. In Milestone 3, you'll build those directory entries, implement the path resolution algorithm that translates `/home/user/projects/file.txt` into a sequence of inode lookups, and implement `mkdir`, `rmdir`, `link`, and `unlink` with their correct link count semantics. The inode's `mode` field already knows whether it's a directory (`S_IFDIR`). Now it's time to give the directory its voice.

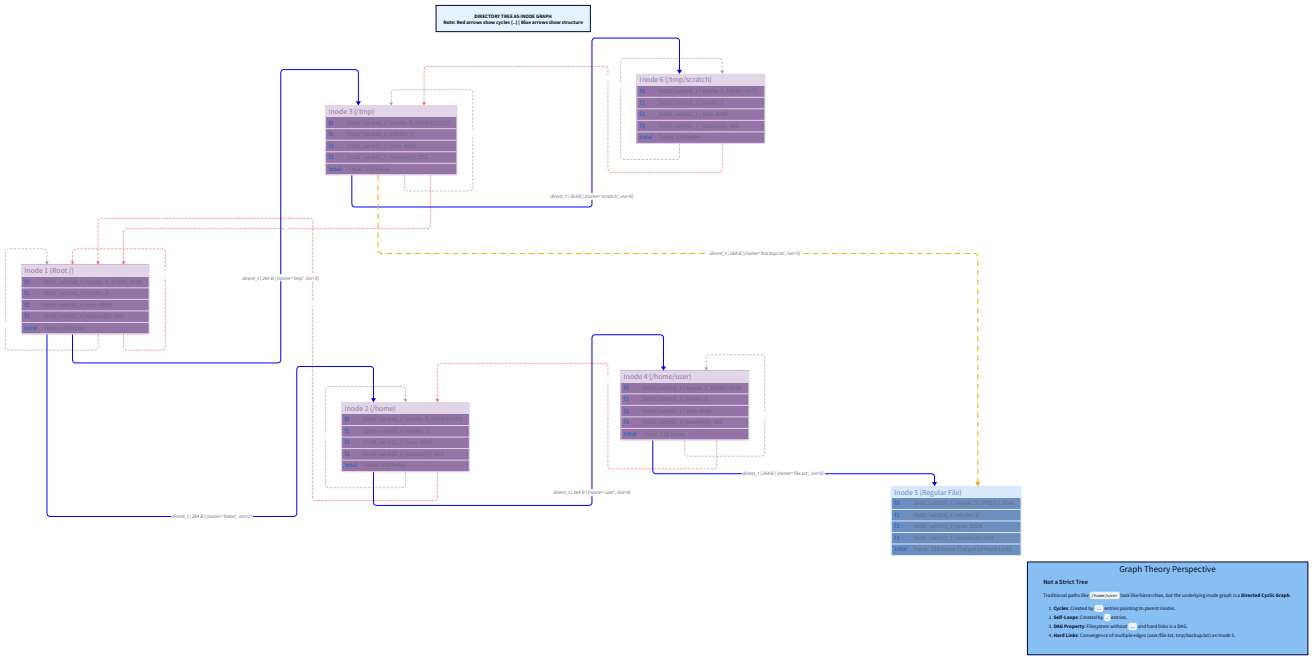
Milestone 3: Directory Operations

The Directory Is a Lie You Have Always Believed

You have been lied to since the first time you opened a file manager. The GUI shows you folders. Nested, hierarchical, expandable folders. You double-click a folder and it opens. You drag files into folders. You see a tree structure — a visual hierarchy where folders contain other folders and files live inside them. The entire metaphor screams: **folders are containers. Files live inside them. The two are fundamentally different things.** Now open a terminal and run this:

```bash
$ ls -la /tmp
total 24
drwxrwxrwt  8 root root 4096 Mar  1 12:00 .
drwxr-xr-x 20 root root 4096 Mar  1 12:00 ..
drwx------  2 user user 4096 Mar  1 12:00 some_dir
-rw-r--r--  1 user user  142 Mar  1 12:00 some_file.txt
```

Notice that `some_dir` is listed with `d` permissions and `some_file.txt` is listed with `-` permissions. The `d` tells you the type. But that `4096` size for `some_dir`? That is exactly one block. One file-sized block. A directory has a *size*. It has an *inode number*. It has timestamps. It has an owner. Every single field you built in Milestone 2's `inode_t` applies to directories. Because **a directory is a file**. Not "kind of like a file." Not "similar to a file." A directory *is* a file, with one difference: its data blocks contain a structured array of records mapping names to inode numbers instead of containing application data. The VFS (Virtual Filesystem Switch — the kernel's abstraction layer over all filesystem types) gives directories the `S_IFDIR` type bit in `mode`, and the kernel interprets that to mean "the data in this file's blocks is a list of directory entries, not raw data." But the storage mechanism — inode pointing to data blocks — is identical. This revelation collapses three separate "things" (files, folders, the path system) into one. And once you understand that a directory is a file, the path resolution algorithm `/home/user/projects/file.txt → inode 42` becomes completely mechanical, blindingly obvious, and satisfying in its recursive simplicity. Let's build it.

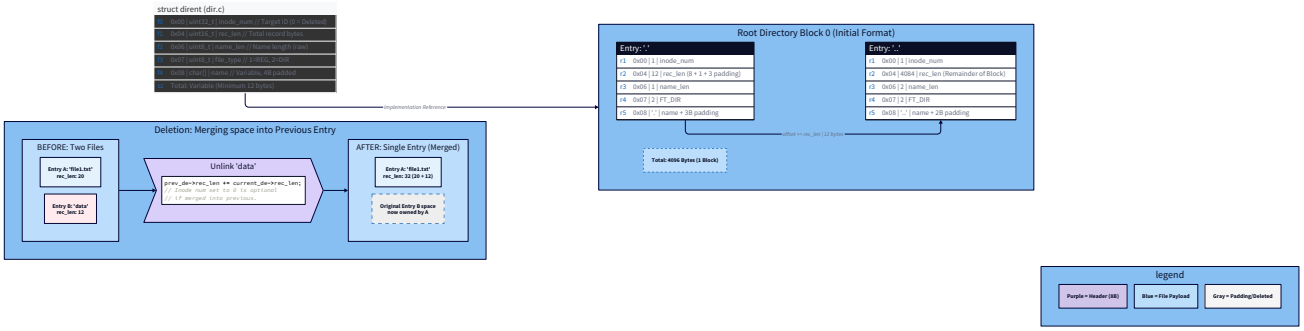


The Fundamental Tension: Hierarchical Names, Flat Blocks

Here is the core hardware and design constraint you are negotiating with: **Humans organize knowledge hierarchically and refer to things by name. Storage hardware is a flat array of numbered blocks with no concept of names, hierarchies, or relationships.** When you type `/usr/bin/gcc`, you are expressing a 3-level hierarchy: root → `usr` → `bin` → `gcc`. The disk stores four separate block regions: one for the root directory's entry list, one for `/usr`'s entry list, one for `/bin`'s entry list, and one for `gcc`'s actual inode and data blocks. The "hierarchy" exists nowhere on disk as a single structure. It is an emergent property of files that point to other files. Every path lookup must be *computed* by walking this chain. There is no shortcut. There is no global hash table. `getattr("/usr/bin/gcc")` requires at minimum:

1. Read root directory's data blocks → find `usr` → get inode 14
2. Read inode 14's data blocks → find `bin` → get inode 81
3. Read inode 81's data blocks → find `gcc` → get inode 4431
4. Read inode 4431 → this is your file That's 4+ disk reads for a trivial path. For `/usr/lib/python3.11/site-packages/numpy/core/_multiarray_umath.cpython-311-x86_64-linux-gnu.so` (a real file path), you'd walk 9 levels. Linux solves this with the **dcache** (directory entry cache) — a kernel-level in-memory hash table mapping `(parent_inode, name) → child_inode`. In your FUSE filesystem, every path lookup must traverse from root unless you implement a dcache. This is the central performance problem of directory-based filesystems, and FUSE exposes it nakedly. The numbers: on a cold cache, each directory lookup requires 1–2 disk reads (inode block + data block). An SSD costs ~100µs per read. A 9-component path = 18 reads = ~1.8ms just for the lookup, before any data is read. The Linux dcache converts this to nanoseconds for hot paths.

The Directory Entry: On-Disk Format



A directory's data blocks contain a sequence of **directory entries** (dirents). Each entry is a variable-length record with a fixed-size header and a variable-length name field. Here is the design we'll use, closely modeled after ext2's format:

```c
/*
 * Directory entry — on-disk format.
 *
 * BYTE LAYOUT:
 *   Offset 0: inode_num  (4 bytes) — 0 means deleted/unused entry
 *   Offset 4: rec_len    (2 bytes) — total length of this record (for skipping)
 *   Offset 6: name_len   (1 byte)  — length of name field (NOT null-terminated length)
 *   Offset 7: file_type  (1 byte)  — type hint: 0=unknown, 1=regular, 2=directory
 *   Offset 8: name       (name_len bytes) — the filename, NOT null-terminated on disk
 *   [padding to align rec_len to a 4-byte boundary]
 *
 * rec_len is CRITICAL: it tells you how many bytes to advance to find the NEXT entry.
 * This allows entries to span variable widths while keeping the scan efficient.
 *
 * The last entry in a block has rec_len extended to fill the remaining space,
 * acting as a sentinel so you know where the block ends.
 */
#define MAX_FILENAME_LEN   255    /* POSIX NAME_MAX */
#define DIRENT_HEADER_SIZE 8      /* inode_num + rec_len + name_len + file_type */
/* File type constants for dirent.file_type */
#define FT_UNKNOWN   0
#define FT_REG_FILE  1
#define FT_DIR       2
#define FT_SYMLINK   7
/*
 * In-memory representation — easier to work with than the raw on-disk format.
 * We serialize to/from the variable-length on-disk format manually.
 */
typedef struct {
    uint32_t inode_num;          /* target inode number (0 = free slot) */
    uint16_t rec_len;            /* total record length including padding */
```

```
    uint8_t  name_len;          /* length of name, not including any null */
    uint8_t  file_type;         /* FT_* constant */
    char     name[MAX_FILENAME_LEN + 1];  /* name, always null-terminated in memory */
} dirent_t;
```

The `rec_len` field deserves careful attention. It is **not** `DIRENT_HEADER_SIZE + name_len`. It is the total number of bytes consumed by this entry in the directory block, which includes:

1. The fixed 8-byte header
2. The name bytes (`name_len`)
3. Padding bytes to align the next entry to a 4-byte boundary
4. For the **last** active entry in a block, `rec_len` is extended to fill the remaining bytes of the block — this entry "owns" all remaining space This variable-length, forward-linked record format means directory scanning is always sequential: read entry at offset 0, advance by `rec_len`, read next entry, repeat until you've consumed `BLOCK_SIZE` bytes.

```c
/*
 * Compute the actual (padded) record length for a given name length.
 * Entries are padded to 4-byte alignment.
 *
 *   name_len=1  → 8 + 1 = 9  → pad to 12
 *   name_len=2  → 8 + 2 = 10 → pad to 12
 *   name_len=3  → 8 + 3 = 11 → pad to 12
 *   name_len=4  → 8 + 4 = 12 → already aligned
 *   name_len=8  → 8 + 8 = 16 → already aligned
 */
static uint16_t dirent_actual_len(uint8_t name_len) {
    uint16_t raw = DIRENT_HEADER_SIZE + name_len;
    return (uint16_t)((raw + 3) & ~3u);   /* round up to next multiple of 4 */
}

/*
 * Write a dirent_t to a raw byte buffer at a given offset.
 * Returns the number of bytes written (rec_len).
 */
static uint16_t dirent_write_to_buf(uint8_t *buf, uint16_t offset,
                                    const dirent_t *de) {
    uint8_t *p = buf + offset;
    memcpy(p + 0, &de->inode_num, 4);
    memcpy(p + 4, &de->rec_len,   2);
    memcpy(p + 6, &de->name_len,  1);
    memcpy(p + 7, &de->file_type, 1);
    memcpy(p + 8, de->name, de->name_len);
    /* zero the padding bytes */
    uint16_t actual = dirent_actual_len(de->name_len);
    if (actual > DIRENT_HEADER_SIZE + de->name_len) {
        memset(p + DIRENT_HEADER_SIZE + de->name_len, 0,
               actual - DIRENT_HEADER_SIZE - de->name_len);
```

```
    }

    return de->rec_len;

}

/*
 * Read a dirent_t from a raw byte buffer at a given offset.
 * Returns the rec_len to advance to the next entry.
 * Returns 0 if the entry is a deleted slot (inode_num == 0 but rec_len != 0).
 */
static uint16_t dirent_read_from_buf(const uint8_t *buf, uint16_t offset,
                                     dirent_t *de) {
    const uint8_t *p = buf + offset;
    memcpy(&de->inode_num, p + 0, 4);
    memcpy(&de->rec_len,   p + 4, 2);
    memcpy(&de->name_len,  p + 6, 1);
    memcpy(&de->file_type, p + 7, 1);
    if (de->rec_len < DIRENT_HEADER_SIZE || de->rec_len > BLOCK_SIZE) {
        return 0;    /* corrupted entry — stop scan */
    }
    if (de->name_len > 0 && de->name_len <= MAX_FILENAME_LEN) {
        memcpy(de->name, p + 8, de->name_len);
    }
    de->name[de->name_len] = '\0';   /* always null-terminate in memory */
    return de->rec_len;
}
```

Why variable-length entries instead of fixed-size? Fixed-size entries are simpler to implement, but they waste enormous space. If you allocate 256 bytes per entry (to hold the max 255-char filename), a directory with 100 files named `a`, `b`, `c` consumes 25KB just for entries that average 9 bytes each. Variable-length entries pack at density proportional to actual name length. The downside: you cannot randomly access entry N without scanning from the start. For directories with hundreds of entries, this linear scan becomes a bottleneck — which is why ext4 introduced `htree` (hash-tree) directories as an optimization (see Knowledge Cascade below).

Scanning a Directory Block

The workhorse of all directory operations is scanning a directory's data blocks for a specific name. Let's implement this as a clean, reusable primitive:

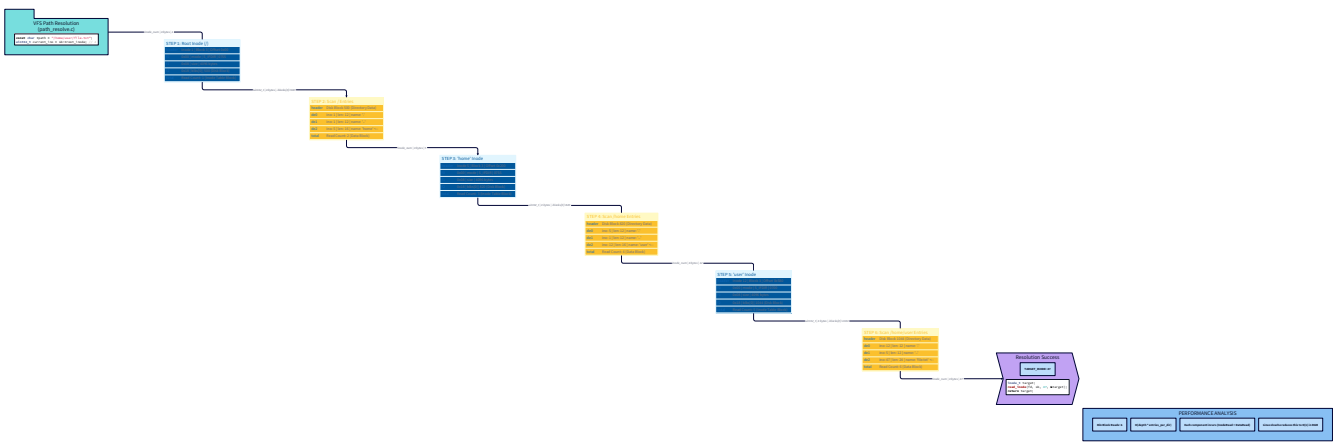
```c
/*
 * Scan all entries in a single directory data block.
 * Calls callback for each valid entry (inode_num != 0).
 *
 * callback return values:
 *   0  → continue scanning
 *   1  → stop scanning, entry found (caller checks de->inode_num)
 *  <0  → stop scanning, error
 *
 * Returns the inode number of the matching entry, 0 if not found, <0 on error.
 */
typedef int (*dirent_callback_t)(const dirent_t *de, void *userdata);
static int scan_dir_block(const uint8_t *block_buf,
                          dirent_callback_t cb, void *userdata) {
    uint16_t offset = 0;
    while (offset + DIRENT_HEADER_SIZE <= BLOCK_SIZE) {
        dirent_t de;
        uint16_t rec_len = dirent_read_from_buf(block_buf, offset, &de);
        if (rec_len == 0) break;     /* corrupted — stop */
        if (de.inode_num != 0) {     /* skip deleted entries */
            int r = cb(&de, userdata);
            if (r != 0) return r;
        }
        offset += rec_len;
    }
    return 0;
}
/* Callback + userdata for name lookup */
typedef struct {
    const char *target_name;
    uint32_t    found_inode;
```

```c
} lookup_ctx_t;

static int lookup_cb(const dirent_t *de, void *userdata) {
    lookup_ctx_t *ctx = (lookup_ctx_t *)userdata;
    if (de->name_len == strlen(ctx->target_name) &&
        memcmp(de->name, ctx->target_name, de->name_len) == 0) {
        ctx->found_inode = de->inode_num;
        return 1;   /* found — stop scan */
    }
    return 0;   /* keep scanning */
}

/*
 * Look up a single name component in a directory inode.
 * Returns the inode number of the matching entry, or 0 if not found.
 * Returns <0 on I/O error.
 */
int dir_lookup(int fd, const superblock_t *sb,
               const inode_t *dir_inode, const char *name) {
    if (!S_ISDIR(dir_inode->mode)) return -ENOTDIR;
    lookup_ctx_t ctx = { .target_name = name, .found_inode = 0 };
    uint8_t buf[BLOCK_SIZE];
    /* Scan each direct block of the directory */
    for (int i = 0; i < N_DIRECT; i++) {
        if (dir_inode->blocks[i] == 0) continue;
        if (read_block(fd, dir_inode->blocks[i], buf) != 0) return -EIO;
        int r = scan_dir_block(buf, lookup_cb, &ctx);
        if (r == 1) return (int)ctx.found_inode;   /* found */
        if (r < 0)  return r;
    }
    /* For milestone 3, we handle directories up to 12 direct blocks (48KB).
     * In practice, that's 48KB / ~12 bytes avg per entry ≈ 4000 entries —
     * sufficient for most use cases. Single-indirect support is straightforward
```

```
     * to add by extending this loop to use inode_get_block() instead. */
    return 0;   /* not found */
}
```

> **Hardware Soul — Directory Scan Performance**: Scanning a directory's data blocks is a sequential memory read of 4KB chunks. Once the first block is in the L1 or L2 cache (64KB and 256KB respectively on typical x86-64), subsequent scans of the same directory data are at memory speed: ~4ns per cache line (64 bytes), covering 8 entries per cache line for short names. A 100-entry directory fits in 2–3 cache lines' worth of data — the entire lookup completes in under 20ns from L1. The bottleneck is the initial disk read if the block is cold: ~100µs on SSD. This 5,000× gap between cached and uncached lookup is exactly what Linux's dcache eliminates.

Path Resolution: Walking the Tree



Path resolution is the algorithm that converts a string like `/home/user/projects/file.txt` into an inode number. It is one of the most frequently executed operations in any OS, and every single call to `open()`, `stat()`, `access()`, `mkdir()`, `rename()`, and most other syscalls triggers it. [[EXPLAIN:vfs-namei-—-the-kernel-path-resolution-engine|VFS namei — how the Linux kernel resolves pathnames through dentry cache and filesystem methods]] The algorithm is beautifully simple:

1. Start at the root inode (if path begins with `/`) or the current working directory inode (if relative path)
2. Tokenize the path by `/` separators, skipping empty components
3. For each component `name`: a. Verify the current inode is a directory (`S_ISDIR(mode)`) b. Scan the directory's data blocks for an entry matching `name` c. Get the inode number from that entry d. Load that inode — it becomes the "current inode" for the next component
4. After the last component, return the inode number The `..` and `.` entries are **real directory entries stored on disk** — not special kernel magic. When you hit `..`, you look it up like any other name, find its inode number, load that inode, and continue. When you hit `.`, you find the same inode you're already at. The tree traversal is entirely implemented by the normal lookup mechanism.

```c
/*
 * Resolve a path to an inode number.
 *
 * path:      absolute path (must start with '/') or relative path
 * root_ino:  inode number of the filesystem root (from sb->root_inode)
 * cwd_ino:   inode number of the current working directory (use root_ino
 *            if you don't track per-process cwd)
 *
 * Returns inode number on success, or a negative errno value.
 */
int path_resolve(int fd, const superblock_t *sb,
                 const char *path, uint32_t root_ino, uint32_t cwd_ino) {
    if (path == NULL || path[0] == '\0') return -EINVAL;
    /* Determine starting inode */
    uint32_t current_ino = (path[0] == '/') ? root_ino : cwd_ino;
    /* Work on a mutable copy of the path for tokenization */
    char path_copy[4096];
    if (strlen(path) >= sizeof(path_copy)) return -ENAMETOOLONG;
    strncpy(path_copy, path, sizeof(path_copy) - 1);
    path_copy[sizeof(path_copy) - 1] = '\0';
    /* Tokenize by '/' and walk each component */
    char *saveptr = NULL;
    char *component = strtok_r(path_copy, "/", &saveptr);
    while (component != NULL) {
        /* Skip empty components (result of consecutive '/' in path) */
        if (component[0] == '\0') {
            component = strtok_r(NULL, "/", &saveptr);
            continue;
        }
        /* Enforce filename length limit */
        if (strlen(component) > MAX_FILENAME_LEN) return -ENAMETOOLONG;
```

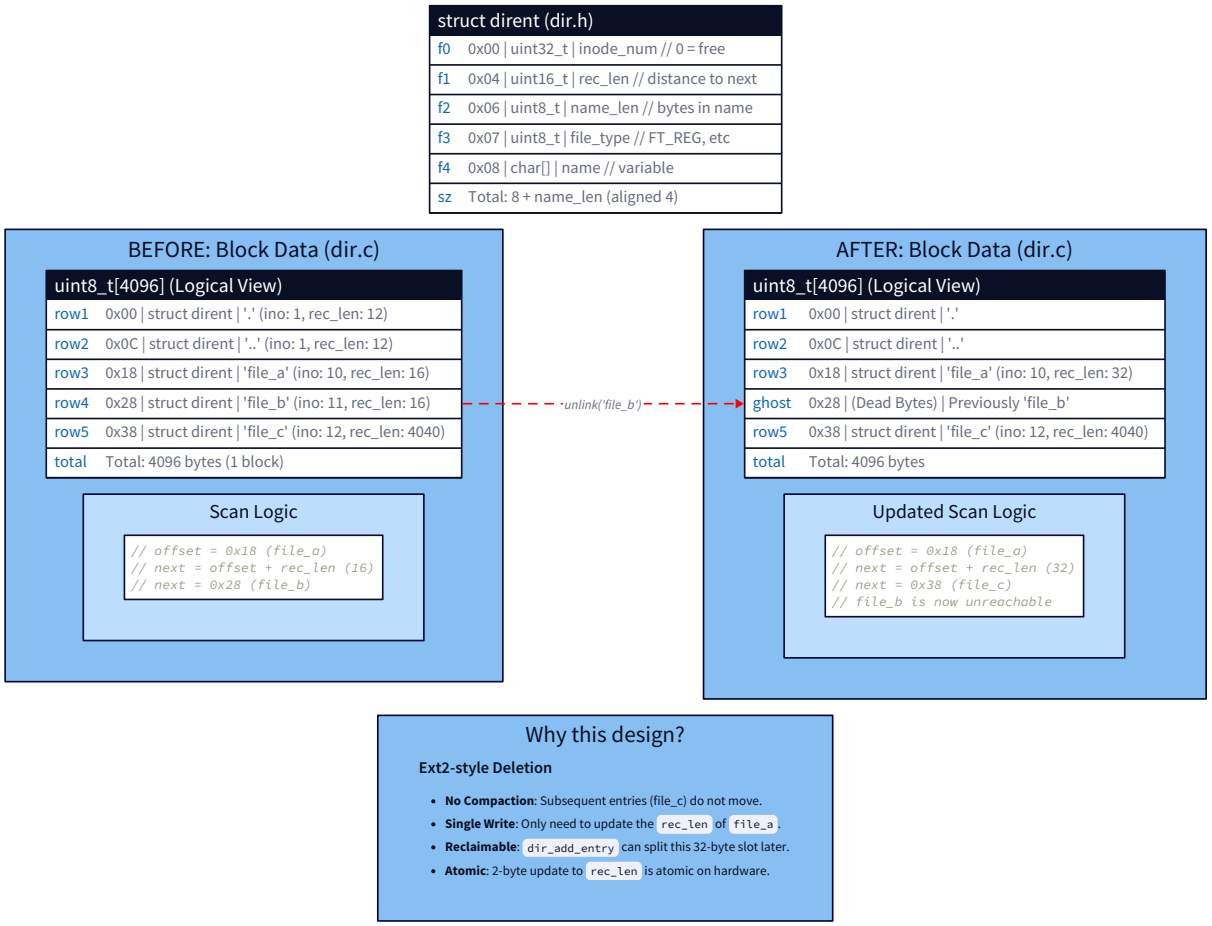
```
        /* Load the current directory inode */
        inode_t dir_inode;
        if (read_inode(fd, sb, current_ino, &dir_inode) != 0) return -EIO;
        /* Must be a directory to descend into */
        if (!S_ISDIR(dir_inode.mode)) return -ENOTDIR;
        /* Look up this component in the directory */
        int child_ino = dir_lookup(fd, sb, &dir_inode, component);
        if (child_ino == 0) return -ENOENT;    /* not found */
        if (child_ino < 0)  return child_ino;  /* I/O error */
        current_ino = (uint32_t)child_ino;
        component = strtok_r(NULL, "/", &saveptr);
    }
    return (int)current_ino;
}
/*
 * Variant: resolve all but the last component, returning the parent inode
 * and the final component name. Used by create, mkdir, unlink, rename.
 *
 * Example: path = "/home/user/new_file.txt"
 *   → parent_ino = inode of "/home/user"
 *   → name_out   = "new_file.txt"
 */
int path_resolve_parent(int fd, const superblock_t *sb,
                        const char *path, uint32_t root_ino, uint32_t cwd_ino,
                        uint32_t *parent_ino_out, char *name_out, size_t name_max) {
    /* Find the last '/' to split parent path from final component */
    const char *last_slash = strrchr(path, '/');
    if (last_slash == NULL) {
        /* Relative path with no slash — parent is cwd */
        *parent_ino_out = cwd_ino;
        if (strlen(path) >= name_max) return -ENAMETOOLONG;
```

```
        strncpy(name_out, path, name_max - 1);
        name_out[name_max - 1] = '\0';
        return 0;
    }
    if (last_slash == path) {
        /* Path like "/filename" — parent is root */
        *parent_ino_out = root_ino;
    } else {
        /* Path like "/home/user/filename" — resolve parent portion */
        char parent_path[4096];
        size_t parent_len = (size_t)(last_slash - path);
        if (parent_len >= sizeof(parent_path)) return -ENAMETOOLONG;
        memcpy(parent_path, path, parent_len);
        parent_path[parent_len] = '\0';
        int p = path_resolve(fd, sb, parent_path, root_ino, cwd_ino);
        if (p < 0) return p;
        *parent_ino_out = (uint32_t)p;
    }
    /* Extract the final component */
    const char *filename = last_slash + 1;
    if (strlen(filename) == 0) return -EINVAL;    /* trailing slash */
    if (strlen(filename) >= name_max) return -ENAMETOOLONG;
    strncpy(name_out, filename, name_max - 1);
    name_out[name_max - 1] = '\0';
    return 0;
}
```

Three-Level View — What Happens When `path_resolve("/usr/bin/gcc")` Runs

Level	What Happens
Application	Calls `open("/usr/bin/gcc", O_RDONLY)` . The path string lives in process memory.
OS / Kernel (Linux)	The VFS `namei` subsystem tokenizes the path and walks the dcache (dentry cache). If `(root_ino, "usr")` is in the dcache, it returns the child dentry without any disk I/O. For a cold cache, it calls your filesystem's `lookup` method for each component. The kernel tracks the current directory per-process via `task_struct.fs->pwd` .
Hardware	For each directory block read: the block device driver issues a read request. The DMA controller transfers 4KB from disk into a kernel page cache page. Your filesystem code then scans that memory — at memory bus speed (~50GB/s), scanning 4KB takes under 100ns.

Adding and Removing Entries: The Mutation API



`add_entry`: Writing a New Name

```c
/*
 * Add a directory entry to a directory inode.
 * Appends the new entry to the first data block with space, or allocates
 * a new block if all existing blocks are full.
 *
 * dir_ino:    inode number of the directory to modify
 * name:       filename to add (must not already exist)
 * target_ino: inode number the new entry points to
 * file_type:  FT_DIR or FT_REG_FILE (stored as a hint in the entry)
 *
 * Returns 0 on success, negative errno on failure.
 *
 * NOTE: Does NOT modify the target inode's nlinks — caller is responsible.
 * This separation keeps add_entry general-purpose.
 */
int dir_add_entry(int fd, superblock_t *sb,
                  uint32_t dir_ino, const char *name,
                  uint32_t target_ino, uint8_t file_type) {
    if (strlen(name) == 0 || strlen(name) > MAX_FILENAME_LEN) return -EINVAL;
    inode_t dir_inode;
    if (read_inode(fd, sb, dir_ino, &dir_inode) != 0) return -EIO;
    if (!S_ISDIR(dir_inode.mode)) return -ENOTDIR;
    /* First pass: check name doesn't already exist */
    if (dir_lookup(fd, sb, &dir_inode, name) > 0) return -EEXIST;
    uint16_t needed = dirent_actual_len((uint8_t)strlen(name));
    uint8_t buf[BLOCK_SIZE];
    /* Search existing blocks for a slot with enough free space */
    for (int i = 0; i < N_DIRECT; i++) {
        uint32_t blk = dir_inode.blocks[i];
        if (blk == 0) continue;
```

```
        if (read_block(fd, blk, buf) != 0) return -EIO;

        /* Scan entries to find the last active entry in this block.
         * We can steal space from its rec_len "tail" if there's room. */
        uint16_t offset = 0;
        while (offset + DIRENT_HEADER_SIZE <= BLOCK_SIZE) {
            dirent_t de;
            uint16_t advance = dirent_read_from_buf(buf, offset, &de);
            if (advance == 0) break;
            uint16_t actual_size;
            if (de.inode_num == 0) {
                /* Deleted entry — its entire rec_len is free */
                actual_size = 0;
            } else {
                actual_size = dirent_actual_len(de.name_len);
            }
            uint16_t free_space = de.rec_len - actual_size;
            if (free_space >= needed) {
                /* Enough space after this entry (or in its deleted slot).
                 * Split: shrink this entry's rec_len to actual_size,
                 * write new entry in the freed space. */
                if (de.inode_num != 0) {
                    /* Shrink the existing entry's rec_len */
                    de.rec_len = actual_size;
                    dirent_write_to_buf(buf, offset, &de);
                }
                /* Write the new entry in the freed space */
                uint16_t new_offset = offset + (de.inode_num != 0 ? actual_size : 0);
                dirent_t new_de = {
                    .inode_num = target_ino,
                    .rec_len   = free_space,
                    .name_len  = (uint8_t)strlen(name),
```

```
                .file_type = file_type,
            };
            memcpy(new_de.name, name, new_de.name_len);
            new_de.name[new_de.name_len] = '\0';
            dirent_write_to_buf(buf, new_offset, &new_de);
            if (write_block(fd, blk, buf) != 0) return -EIO;
            /* Update directory mtime/ctime */
            dir_inode.mtime = dir_inode.ctime = (uint32_t)time(NULL);
            return write_inode(fd, sb, dir_ino, &dir_inode);
        }
        offset += advance;
    }
}
/* No space in existing blocks — allocate a new directory block */
int new_blk = alloc_block(fd, sb);
if (new_blk < 0) return -ENOSPC;
/* Find a free direct block slot in the inode */
int slot = -1;
for (int i = 0; i < N_DIRECT; i++) {
    if (dir_inode.blocks[i] == 0) { slot = i; break; }
}
if (slot < 0) return -ENOSPC;   /* all 12 direct blocks in use */
/* Initialize the new block with the new entry occupying full block */
memset(buf, 0, BLOCK_SIZE);
dirent_t new_de = {
    .inode_num = target_ino,
    .rec_len   = BLOCK_SIZE,     /* owns entire block */
    .name_len  = (uint8_t)strlen(name),
    .file_type = file_type,
};
memcpy(new_de.name, name, new_de.name_len);
```

```
    new_de.name[new_de.name_len] = '\0';

    dirent_write_to_buf(buf, 0, &new_de);

    if (write_block(fd, (uint32_t)new_blk, buf) != 0) return -EIO;

    dir_inode.blocks[slot] = (uint32_t)new_blk;

    dir_inode.size += BLOCK_SIZE;

    dir_inode.mtime = dir_inode.ctime = (uint32_t)time(NULL);

    return write_inode(fd, sb, dir_ino, &dir_inode);

}
```

The entry split strategy — shrinking the last entry's `rec_len` to make room for a new entry in its "tail" space — is exactly how ext2 works. This packs entries densely and avoids frequent block allocations for directories with moderate entry counts.

`remove_entry`: Deleting a Name

Removing a directory entry does **not** physically shift entries in memory or free space immediately. Instead, it uses one of two strategies:

1. **Set `inode_num = 0`**: Mark the entry as deleted. The `rec_len` remains unchanged, so the scanner skips it (it's still in the linked chain for forward scanning, just logically absent). Free space is only reclaimed when `add_entry` detects a deleted slot or empty tail space.
2. **Expand the previous entry's `rec_len`**: Find the preceding entry and extend its `rec_len` to absorb the deleted entry's space. This is cleaner and immediately merges the free space, but requires tracking the "previous entry" during the scan. We'll implement strategy 2 (the ext2 approach) for cleaner space recovery:

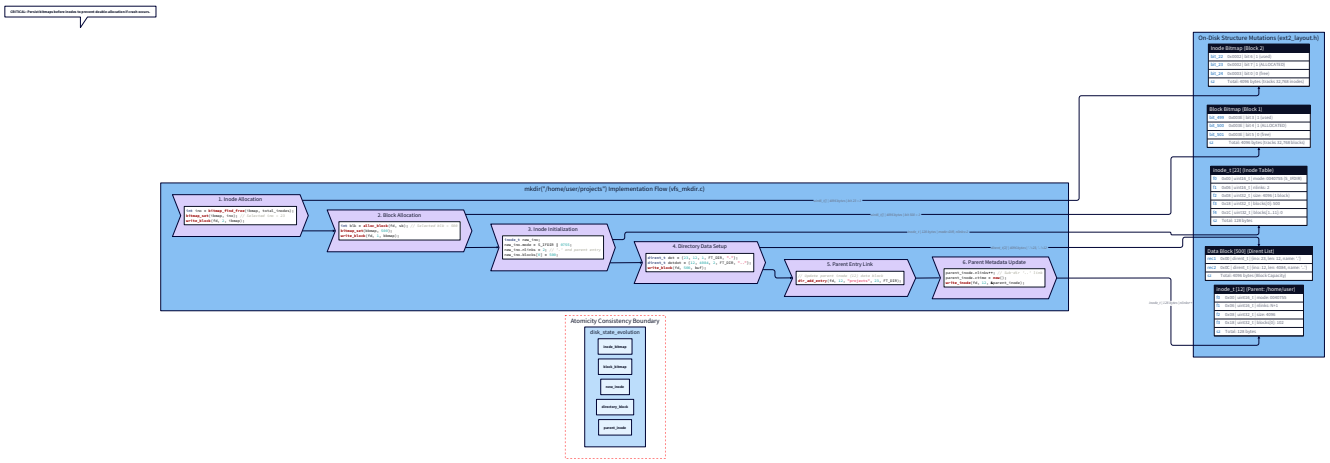
```c
/*
 * Remove a directory entry by name.
 * Decrements the target inode's nlinks and frees it if nlinks reaches 0.
 *
 * Returns 0 on success, -ENOENT if not found, negative errno on error.
 */
int dir_remove_entry(int fd, superblock_t *sb,
                     uint32_t dir_ino, const char *name) {
    inode_t dir_inode;
    if (read_inode(fd, sb, dir_ino, &dir_inode) != 0) return -EIO;
    if (!S_ISDIR(dir_inode.mode)) return -ENOTDIR;
    uint8_t buf[BLOCK_SIZE];
    for (int i = 0; i < N_DIRECT; i++) {
        uint32_t blk = dir_inode.blocks[i];
        if (blk == 0) continue;
        if (read_block(fd, blk, buf) != 0) return -EIO;
        uint16_t offset = 0;
        uint16_t prev_offset = 0;
        bool found = false;
        uint32_t target_ino = 0;
        while (offset + DIRENT_HEADER_SIZE <= BLOCK_SIZE) {
            dirent_t de;
            uint16_t advance = dirent_read_from_buf(buf, offset, &de);
            if (advance == 0) break;
            if (de.inode_num != 0 &&
                de.name_len == strlen(name) &&
                memcmp(de.name, name, de.name_len) == 0) {
                /* Found the entry to remove */
                target_ino = de.inode_num;
                if (offset == 0) {
                    /* First entry in block: just zero out inode_num */
```

```
            /* (Cannot merge backwards — there is no previous entry) */
            uint32_t zero = 0;
            memcpy(buf + offset, &zero, 4);
        } else {
            /* Merge: expand previous entry's rec_len to absorb this one */
            dirent_t prev_de;
            dirent_read_from_buf(buf, prev_offset, &prev_de);
            prev_de.rec_len += de.rec_len;
            dirent_write_to_buf(buf, prev_offset, &prev_de);
        }
        found = true;
        break;
    }
    if (de.inode_num != 0) prev_offset = offset;  /* track last valid entry */
    offset += advance;
}
if (found) {
    if (write_block(fd, blk, buf) != 0) return -EIO;
    /* Update directory mtime/ctime */
    dir_inode.mtime = dir_inode.ctime = (uint32_t)time(NULL);
    if (write_inode(fd, sb, dir_ino, &dir_inode) != 0) return -EIO;
    /* Decrement target inode's nlinks */
    inode_t target_inode;
    if (read_inode(fd, sb, target_ino, &target_inode) != 0) return -EIO;
    if (target_inode.nlinks > 0) target_inode.nlinks--;
    target_inode.ctime = (uint32_t)time(NULL);
    if (target_inode.nlinks == 0) {
        /* Last link removed — free the inode and all its blocks.
         * In a real filesystem, check for open file descriptors first.
         * For now, free immediately. Milestone 5 handles the open fd case. */
        return free_inode(fd, sb, target_ino);
```

```
            }

            return write_inode(fd, sb, target_ino, &target_inode);

        }

    }

    return -ENOENT;

}
```

> **Pitfall — Removing `.` and `..`**: Your `dir_remove_entry` must refuse to remove entries named `.` and `..`. These are structural entries that maintain link counts and tree integrity. Add a guard at the top:
>
> `if (strcmp(name, ".") == 0 || strcmp(name, "..") == 0) return -EINVAL;`

`mkdir`: Creating a Directory



`mkdir` is not just "create an inode and add a directory entry." It involves a precise sequence of state changes that maintain link count invariants. Let's trace through exactly what must happen when you call `mkdir("/home/user/projects")`:

1. Resolve `/home/user` → get parent inode number `parent_ino`
2. Verify `projects` doesn't already exist in parent
3. Allocate a new inode for the new directory
4. Initialize the new directory inode (`mode = S_IFDIR | 0755`, `nlinks = 2`)
5. Allocate a data block for the new directory's initial entries
6. Write `.` (pointing to new inode) and `..` (pointing to parent inode) into that block
7. Write the new directory inode to disk with the data block pointer set
8. Add an entry for `projects` in the parent directory (incrementing parent's `nlinks`)
9. Increment the parent inode's `nlinks` by 1 (because `..` in the new dir points to parent) Step 9 is the one developers forget. The parent's `nlinks` count tracks how many things point *to* it. When you create a child directory, that child's `..` entry points to the parent — so the parent gains one more reference, and its `nlinks` increments.

```
Before mkdir projects:
  /home/user  →  inode 7  →  nlinks = 2 (one for /home/user entry, one for /home/user/. entry)
After mkdir projects:
  /home/user  →  inode 7  →  nlinks = 3 (added: /home/user/projects/.. → inode 7)
```

This is why deeply nested directories cause high `nlinks` on parent directories. And it's why `find -maxdepth N` can use `nlinks` to prune: if a directory's `nlinks` equals 2, it has no subdirectories (only `.` and the parent's entry), so `find` can skip descending.

```c
/*
 * Create a new directory at the given path.
 *
 * parent_ino: inode number of the parent directory
 * name:       name of the new directory (single component, not a path)
 * mode:       permission bits (will be OR'd with S_IFDIR)
 * uid, gid:   owner
 *
 * Returns the new directory's inode number on success, negative errno on failure.
 */
int fs_mkdir(int fd, superblock_t *sb,
             uint32_t parent_ino, const char *name,
             uint16_t mode, uint16_t uid, uint16_t gid) {
    /* Validate */
    if (strlen(name) == 0 || strlen(name) > MAX_FILENAME_LEN) return -EINVAL;
    inode_t parent_inode;
    if (read_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
    if (!S_ISDIR(parent_inode.mode)) return -ENOTDIR;
    /* Check name doesn't already exist */
    if (dir_lookup(fd, sb, &parent_inode, name) > 0) return -EEXIST;
    /* Step 1: Allocate new inode */
    int new_ino = alloc_inode(fd, sb);
    if (new_ino < 0) return -ENOSPC;
    /* Step 2: Allocate a data block for . and .. entries */
    int data_blk = alloc_block(fd, sb);
    if (data_blk < 0) {
        /* Roll back inode allocation — don't leak */
        /* In a journaled filesystem this wouldn't be needed.
         * For now, manually mark inode free in bitmap. */
        uint8_t ibmap[BLOCK_SIZE];
        read_block(fd, INODE_BITMAP_BLOCK, ibmap);
```

```
    bitmap_clear(ibmap, (uint32_t)new_ino - 1);
    write_block(fd, INODE_BITMAP_BLOCK, ibmap);
    sb->free_inodes++;
    return -ENOSPC;
}
/* Step 3: Write . and .. into the data block */
uint8_t dir_data[BLOCK_SIZE];
memset(dir_data, 0, BLOCK_SIZE);
/* '.' entry — points to ourselves */
dirent_t dot = {
    .inode_num = (uint32_t)new_ino,
    .name_len  = 1,
    .file_type = FT_DIR,
};
dot.rec_len = dirent_actual_len(1);
memcpy(dot.name, ".", 1); dot.name[1] = '\0';
dirent_write_to_buf(dir_data, 0, &dot);
/* '..' entry — points to parent; owns rest of block */
dirent_t dotdot = {
    .inode_num = parent_ino,
    .name_len  = 2,
    .file_type = FT_DIR,
};
uint16_t dot_actual = dirent_actual_len(1);
dotdot.rec_len = BLOCK_SIZE - dot_actual;  /* owns remaining space */
memcpy(dotdot.name, "..", 2); dotdot.name[2] = '\0';
dirent_write_to_buf(dir_data, dot_actual, &dotdot);
if (write_block(fd, (uint32_t)data_blk, dir_data) != 0) return -EIO;
/* Step 4: Initialize and write the new directory inode */
inode_t new_inode;
memset(&new_inode, 0, sizeof(new_inode));
```

```
    new_inode.mode    = (uint16_t)(S_IFDIR | (mode & 0777));
    new_inode.uid     = uid;
    new_inode.gid     = gid;
    new_inode.nlinks  = 2;          /* '.' + parent entry pointing to us */
    new_inode.size    = BLOCK_SIZE;
    new_inode.blocks[0] = (uint32_t)data_blk;
    uint32_t now = (uint32_t)time(NULL);
    new_inode.atime = new_inode.mtime = new_inode.ctime = now;
    if (write_inode(fd, sb, (uint32_t)new_ino, &new_inode) != 0) return -EIO;
    /* Step 5: Add entry in parent directory */
    int r = dir_add_entry(fd, sb, parent_ino, name, (uint32_t)new_ino, FT_DIR);
    if (r != 0) return r;
    /* Step 6: Increment parent's nlinks (for the '..' in new dir) */
    if (read_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
    parent_inode.nlinks++;
    parent_inode.ctime = (uint32_t)time(NULL);
    if (write_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
    return new_ino;
}
```

`rmdir`: Removing a Directory

`rmdir` is the inverse of `mkdir`, but with a critical safety check: **a directory can only be removed if it is empty**. "Empty" means it contains exactly two entries: `.` and `..`. Any additional entry — file, subdirectory, or anything else — means `rmdir` must fail with `ENOTEMPTY`. [[EXPLAIN:posix-rmdir-atomicity-—-why-rmdir-cannot-be-interrupted-and-leave-orphaned-inodes|POSIX rmdir atomicity — why rmdir is atomic in the kernel and what happens if it isn't]]

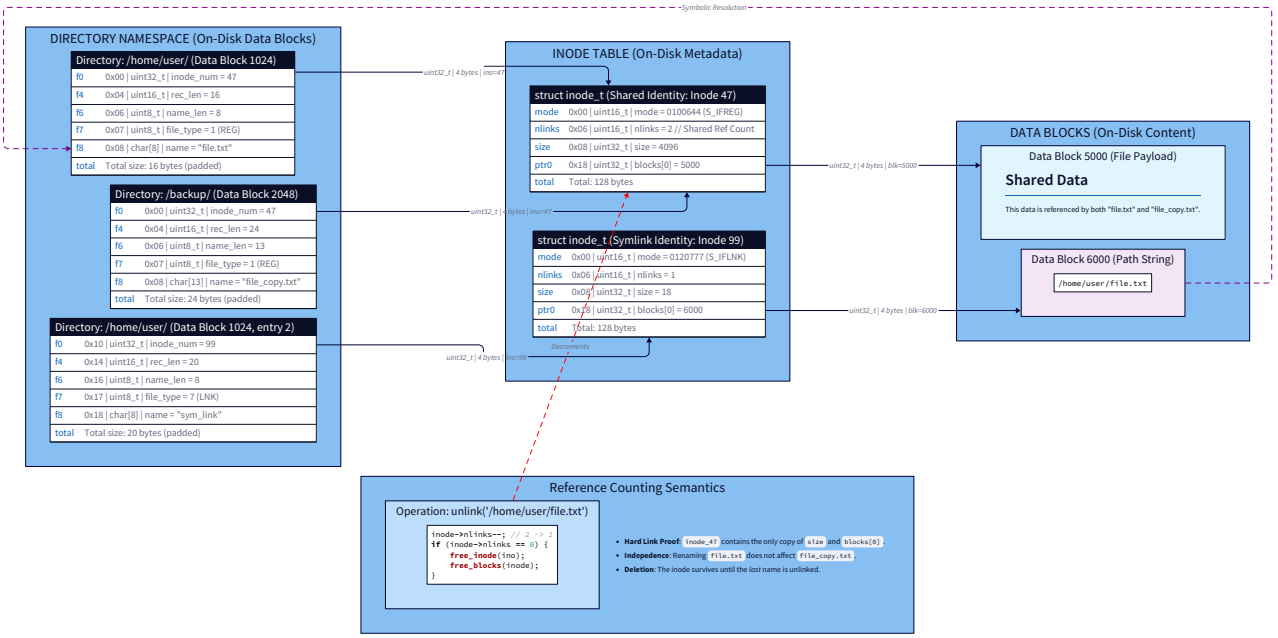
```c
/*
 * Check if a directory is empty (contains only '.' and '..').
 * Returns true if empty, false otherwise.
 */
static bool dir_is_empty(int fd, const inode_t *dir_inode) {
    uint8_t buf[BLOCK_SIZE];
    for (int i = 0; i < N_DIRECT; i++) {
        if (dir_inode->blocks[i] == 0) continue;
        if (read_block(fd, dir_inode->blocks[i], buf) != 0) return false;
        uint16_t offset = 0;
        while (offset + DIRENT_HEADER_SIZE <= BLOCK_SIZE) {
            dirent_t de;
            uint16_t advance = dirent_read_from_buf(buf, offset, &de);
            if (advance == 0) break;
            if (de.inode_num != 0) {
                /* Found a real entry — check if it's '.' or '..' */
                bool is_dot    = (de.name_len == 1 && de.name[0] == '.');
                bool is_dotdot = (de.name_len == 2 && de.name[0] == '.' && de.name[1] == '.');
                if (!is_dot && !is_dotdot) {
                    return false;    /* non-empty: found a real entry */
                }
            }
            offset += advance;
        }
    }
    return true;
}
/*
 * Remove an empty directory.
 *
 * parent_ino: inode number of the parent directory
```

```
 * name:       name of the directory to remove
 *
 * Returns 0 on success, -ENOTEMPTY if directory has entries,
 * -ENOENT if not found, negative errno on error.
 */
int fs_rmdir(int fd, superblock_t *sb,
             uint32_t parent_ino, const char *name) {
    /* Cannot remove '.' or '..' */
    if (strcmp(name, ".") == 0 || strcmp(name, "..") == 0) return -EINVAL;
    /* Resolve the target directory */
    inode_t parent_inode;
    if (read_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
    int target_ino = dir_lookup(fd, sb, &parent_inode, name);
    if (target_ino == 0) return -ENOENT;
    if (target_ino < 0) return target_ino;
    inode_t target_inode;
    if (read_inode(fd, sb, (uint32_t)target_ino, &target_inode) != 0) return -EIO;
    if (!S_ISDIR(target_inode.mode)) return -ENOTDIR;
    /* Safety check: must be empty */
    if (!dir_is_empty(fd, &target_inode)) return -ENOTEMPTY;
    /* Remove the entry from parent (this also decrements target's nlinks via remove_entry) */
    /* But wait — we need to handle nlinks carefully for directories. The target directory
     * has nlinks=2 normally (parent entry + '.' self-reference). We also need to decrement
     * parent's nlinks for the lost '..' reference. Let's do this manually. */
    /* Step 1: Remove the name entry from parent directory.
     * We bypass dir_remove_entry's automatic nlinks decrement and do it manually
     * to also handle the parent's nlinks. */
    /* Find and zero the entry in parent's blocks */
    uint8_t buf[BLOCK_SIZE];
    bool found = false;
    for (int i = 0; i < N_DIRECT && !found; i++) {
```

```
        if (parent_inode.blocks[i] == 0) continue;
        if (read_block(fd, parent_inode.blocks[i], buf) != 0) return -EIO;
        uint16_t offset = 0, prev_offset = 0;
        while (offset + DIRENT_HEADER_SIZE <= BLOCK_SIZE) {
            dirent_t de;
            uint16_t advance = dirent_read_from_buf(buf, offset, &de);
            if (advance == 0) break;
            if (de.inode_num == (uint32_t)target_ino &&
                de.name_len == strlen(name) &&
                memcmp(de.name, name, de.name_len) == 0) {
                /* Merge into previous or zero */
                if (offset == 0) {
                    uint32_t zero = 0;
                    memcpy(buf + offset, &zero, 4);
                } else {
                    dirent_t prev_de;
                    dirent_read_from_buf(buf, prev_offset, &prev_de);
                    prev_de.rec_len += de.rec_len;
                    dirent_write_to_buf(buf, prev_offset, &prev_de);
                }
                write_block(fd, parent_inode.blocks[i], buf);
                found = true;
                break;
            }
            if (de.inode_num != 0) prev_offset = offset;
            offset += advance;
        }
    }
    if (!found) return -ENOENT;
    /* Step 2: Decrement parent's nlinks (lost the '..' reference from removed dir) */
    if (read_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
```

```
    if (parent_inode.nlinks > 0) parent_inode.nlinks--;

    parent_inode.mtime = parent_inode.ctime = (uint32_t)time(NULL);

    if (write_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;

    /* Step 3: Free the target directory inode and its data blocks */

    return free_inode(fd, sb, (uint32_t)target_ino);

}
```

Hard Links: One File, Many Names



The `link()` syscall creates a hard link — a new directory entry pointing to an *existing* inode. No data is copied. No new inode is created. Just one new name that references the same inode, and that inode's `nlinks` counter increments by one.

```c
/*
 * Create a hard link.
 * Adds 'new_name' in 'dir_ino' pointing to the inode at 'target_path'.
 *
 * Rules:
 *   - Cannot hard link a directory (would create cycles, break tree structure)
 *   - target inode must exist
 *   - new_name must not already exist in dir_ino
 *
 * Returns 0 on success, negative errno on failure.
 */
int fs_link(int fd, superblock_t *sb,
            uint32_t target_ino, uint32_t dir_ino, const char *new_name) {
    /* Load target inode */
    inode_t target_inode;
    if (read_inode(fd, sb, target_ino, &target_inode) != 0) return -EIO;
    /* Hard links to directories are forbidden — would create cycles */
    if (S_ISDIR(target_inode.mode)) return -EPERM;
    /* Check nlinks won't overflow (uint16_t max = 65535) */
    if (target_inode.nlinks >= 65535) return -EMLINK;
    /* Add the new directory entry */
    uint8_t file_type = S_ISREG(target_inode.mode) ? FT_REG_FILE : FT_UNKNOWN;
    int r = dir_add_entry(fd, sb, dir_ino, new_name, target_ino, file_type);
    if (r != 0) return r;
    /* Increment target inode's nlinks */
    target_inode.nlinks++;
    target_inode.ctime = (uint32_t)time(NULL);
    return write_inode(fd, sb, target_ino, &target_inode);
}
```

The prohibition on hard-linking directories is a fundamental rule of Unix filesystems. If you could hard-link a directory, you could create:

```
mkdir A
link A A/loop   # A contains an entry 'loop' pointing to A itself
```

Now `A/loop/loop/loop/...` is an infinite descent. `find /`, `du /`, `fsck` — every tool that walks the directory tree would loop forever. The kernel forbids it at the syscall level (`link()` returns `EPERM` for directories). The exception: `mkdir` itself creates `.` and `..`, which *are* directory hard links — but the kernel treats these specially and breaks the cycle by knowing that `..` from root points to root.

`unlink()`: Removing a File

`unlink()` removes a directory entry (just like `remove_entry`). It is the complement to `link()`:

```c
/*
 * Remove a file (not a directory) from a directory.
 * Decrements nlinks; frees inode if nlinks reaches 0.
 *
 * Use fs_rmdir for directories.
 */
int fs_unlink(int fd, superblock_t *sb,
              uint32_t dir_ino, const char *name) {
    inode_t dir_inode;
    if (read_inode(fd, sb, dir_ino, &dir_inode) != 0) return -EIO;
    int target_ino = dir_lookup(fd, sb, &dir_inode, name);
    if (target_ino == 0) return -ENOENT;
    if (target_ino < 0) return target_ino;
    /* Verify it's not a directory */
    inode_t target_inode;
    if (read_inode(fd, sb, (uint32_t)target_ino, &target_inode) != 0) return -EIO;
    if (S_ISDIR(target_inode.mode)) return -EISDIR;   /* use rmdir instead */
    /* dir_remove_entry handles nlinks decrement and inode freeing */
    return dir_remove_entry(fd, sb, dir_ino, name);
}
```

The `.` and `..` Contract

The `.` and `..` entries are real directory entries stored in real data blocks. They are **not** generated on-the-fly by the kernel. Let's verify this claim by looking at what we write during `mkfs` and `mkdir`:

- `.`: inode_num = this directory's own inode number, name = "."
- `..`: inode_num = parent directory's inode number, name = ".." For the root directory specifically, `..` **points to itself** — root's parent is root. This creates a natural base case for path traversal algorithms that follow `..` chains: when you reach a node where `.` and `..` are the same inode, you are at the root.

```c
/* Verify the '.' entry in a directory block is correct */
int verify_dot_entries(int fd, const superblock_t *sb,
                       uint32_t dir_ino, uint32_t expected_parent_ino) {
    inode_t dir_inode;
    if (read_inode(fd, sb, dir_ino, &dir_inode) != 0) return -EIO;
    if (!S_ISDIR(dir_inode.mode)) return -ENOTDIR;
    if (dir_inode.blocks[0] == 0) return -EINVAL;   /* no data block */
    uint8_t buf[BLOCK_SIZE];
    if (read_block(fd, dir_inode.blocks[0], buf) != 0) return -EIO;
    /* First entry must be '.' pointing to self */
    dirent_t dot;
    uint16_t advance = dirent_read_from_buf(buf, 0, &dot);
    if (advance == 0) return -EINVAL;
    if (dot.inode_num != dir_ino) {
        fprintf(stderr, "CORRUPT: '.' entry in inode %u points to %u (expected %u)\n",
                dir_ino, dot.inode_num, dir_ino);
        return -EINVAL;
    }
    if (dot.name_len != 1 || dot.name[0] != '.') {
        fprintf(stderr, "CORRUPT: first entry in inode %u is not '.'\n", dir_ino);
        return -EINVAL;
    }
    /* Second entry must be '..' pointing to parent */
    dirent_t dotdot;
    dirent_read_from_buf(buf, advance, &dotdot);
    if (dotdot.inode_num != expected_parent_ino) {
        fprintf(stderr, "CORRUPT: '..' entry in inode %u points to %u (expected %u)\n",
                dir_ino, dotdot.inode_num, expected_parent_ino);
        return -EINVAL;
    }
    if (dotdot.name_len != 2 ||
```

```
        dotdot.name[0] != '.' || dotdot.name[1] != '.') {
        fprintf(stderr, "CORRUPT: second entry in inode %u is not '..'\n", dir_ino);
        return -EINVAL;
    }
    return 0;   /* valid */
}
```

`readdir`: Listing a Directory

FUSE (Milestone 5) calls `readdir` for every `ls` command. Here is the core implementation:

```c
/*
 * Iterate over all entries in a directory.
 * Calls callback for each non-deleted entry (including '.' and '..').
 *
 * Used by FUSE's readdir callback and by dir_is_empty.
 */
typedef void (*readdir_cb_t)(const char *name, uint32_t inode_num,
                             uint8_t file_type, void *userdata);
int dir_readdir(int fd, const superblock_t *sb,
                uint32_t dir_ino, readdir_cb_t cb, void *userdata) {
    inode_t dir_inode;
    if (read_inode(fd, sb, dir_ino, &dir_inode) != 0) return -EIO;
    if (!S_ISDIR(dir_inode.mode)) return -ENOTDIR;
    uint8_t buf[BLOCK_SIZE];
    for (int i = 0; i < N_DIRECT; i++) {
        if (dir_inode.blocks[i] == 0) continue;
        if (read_block(fd, dir_inode.blocks[i], buf) != 0) return -EIO;
        uint16_t offset = 0;
        while (offset + DIRENT_HEADER_SIZE <= BLOCK_SIZE) {
            dirent_t de;
            uint16_t advance = dirent_read_from_buf(buf, offset, &de);
            if (advance == 0) break;
            if (de.inode_num != 0) {
                cb(de.name, de.inode_num, de.file_type, userdata);
            }
            offset += advance;
        }
    }
    return 0;
}
```

Hardware Soul — `readdir` at Speed: A directory with 1,000 entries across 3 data blocks (three 4KB reads) takes ~300µs on a cold SSD, ~30ns from L1 cache. The sequential access pattern is optimal for hardware prefetchers: the CPU sees a stride-0 access pattern through the 4KB block and prefetches the next cache lines automatically. For warm-cache `ls` performance, the bottleneck is almost never the directory block reads — it's the subsequent `stat()` calls, each of which reads a different inode block (likely random access into the inode table).

Validation: Testing the Directory Layer

```c
void test_directory_layer(int fd, superblock_t *sb) {
    printf("\n=== Directory Layer Tests ===\n");
    uint32_t root_ino = sb->root_inode;
    /* Test 1: mkdir a subdirectory */
    int user_ino = fs_mkdir(fd, sb, root_ino, "user", 0755, 1000, 1000);
    assert(user_ino > 0);
    printf("OK: mkdir 'user' → inode %d\n", user_ino);
    /* Test 2: path resolution */
    int resolved = path_resolve(fd, sb, "/user", root_ino, root_ino);
    assert(resolved == user_ino);
    printf("OK: path_resolve('/user') = %d\n", resolved);
    /* Test 3: nested mkdir */
    int projects_ino = fs_mkdir(fd, sb, (uint32_t)user_ino, "projects",
                                0755, 1000, 1000);
    assert(projects_ino > 0);
    int nested = path_resolve(fd, sb, "/user/projects", root_ino, root_ino);
    assert(nested == projects_ino);
    printf("OK: nested path resolution works\n");
    /* Test 4: verify parent nlinks incremented */
    inode_t user_inode;
    read_inode(fd, sb, (uint32_t)user_ino, &user_inode);
    assert(user_inode.nlinks == 3);  /* '.' + /user entry + /user/projects/.. */
    printf("OK: parent nlinks = %u (correct)\n", user_inode.nlinks);
    /* Test 5: '.' and '..' entries */
    assert(verify_dot_entries(fd, sb, (uint32_t)projects_ino,
                              (uint32_t)user_ino) == 0);
    printf("OK: '.' and '..' entries verified\n");
    /* Test 6: duplicate name rejected */
    assert(fs_mkdir(fd, sb, root_ino, "user", 0755, 0, 0) == -EEXIST);
    printf("OK: duplicate mkdir rejected with EEXIST\n");
```

```c
    /* Test 7: rmdir empty directory */
    int empty_ino = fs_mkdir(fd, sb, (uint32_t)user_ino, "empty",
                             0755, 1000, 1000);
    assert(empty_ino > 0);
    assert(fs_rmdir(fd, sb, (uint32_t)user_ino, "empty") == 0);
    /* Verify inode is now free */
    uint8_t ibmap[BLOCK_SIZE];
    read_block(fd, INODE_BITMAP_BLOCK, ibmap);
    assert(!bitmap_test(ibmap, (uint32_t)empty_ino - 1));
    printf("OK: rmdir empty directory, inode freed\n");
    /* Test 8: rmdir non-empty directory fails */
    assert(fs_rmdir(fd, sb, root_ino, "user") == -ENOTEMPTY);
    printf("OK: rmdir non-empty directory rejected\n");
    /* Test 9: ENOENT for missing path */
    assert(path_resolve(fd, sb, "/nonexistent", root_ino, root_ino) == -ENOENT);
    printf("OK: path_resolve nonexistent returns ENOENT\n");
    /* Test 10: Hard link */
    /* Create a file first (borrow from Milestone 4 — just allocate and init an inode) */
    int file_ino = alloc_inode(fd, sb);
    assert(file_ino > 0);
    inode_t file_inode;
    inode_init_file(&file_inode, 0644, 1000, 1000);
    write_inode(fd, sb, (uint32_t)file_ino, &file_inode);
    dir_add_entry(fd, sb, root_ino, "original", (uint32_t)file_ino, FT_REG_FILE);
    file_inode.nlinks = 1;
    write_inode(fd, sb, (uint32_t)file_ino, &file_inode);
    assert(fs_link(fd, sb, (uint32_t)file_ino, root_ino, "hardlink") == 0);
    read_inode(fd, sb, (uint32_t)file_ino, &file_inode);
    assert(file_inode.nlinks == 2);
    printf("OK: hard link created, nlinks = %u\n", file_inode.nlinks);
    /* Test 11: Unlink one, inode survives */
```

```
    assert(fs_unlink(fd, sb, root_ino, "original") == 0);

    read_inode(fd, sb, (uint32_t)file_ino, &file_inode);

    assert(file_inode.nlinks == 1);

    printf("OK: unlink one link, nlinks = %u (inode survives)\n", file_inode.nlinks);

    /* Test 12: Unlink last link, inode freed */

    assert(fs_unlink(fd, sb, root_ino, "hardlink") == 0);

    read_block(fd, INODE_BITMAP_BLOCK, ibmap);

    assert(!bitmap_test(ibmap, (uint32_t)file_ino - 1));

    printf("OK: last unlink frees inode\n");

    printf("=== All Directory Tests Passed ===\n\n");

}
```

Concurrency Warning: Races in Directory Modification

If multiple threads (as in FUSE with `default_permissions` and multi-threaded mode) concurrently modify the same directory, you have a race condition. Consider:

```
Thread A: dir_lookup("file")    → not found (0)
Thread B: dir_add_entry("file") → creates entry
Thread A: dir_add_entry("file") → creates DUPLICATE entry!
```

The name-uniqueness check in `dir_add_entry` and the lookup are not atomic. In Milestone 5, you will add a per-directory mutex or a global filesystem lock. For now, document this limitation and test only single-threaded:

```C
/*
 * Thread safety: All directory operations on a given directory inode
 * must be serialized. The caller (FUSE callbacks in Milestone 5) is
 * responsible for holding the appropriate lock before calling these functions.
 * A simple approach: one global pthread_mutex_t protecting all FS state.
 * A better approach: per-inode read-write locks (readers can share, writers exclusive).
 */
```

[[EXPLAIN:directory-locking-strategies-—-global-lock-vs-per-inode-vs-hierarchical-locking|Directory locking strategies — global lock vs. per-inode vs. hierarchical locking for concurrent filesystem access]]

Knowledge Cascade: One Directory, Ten Worlds

You have built what appears to be a simple name-to-inode lookup table. But the ideas behind it permeate systems across every domain.

Symbolic Links vs. Hard Links: Now Obvious

With directory entries fully understood, the hard link vs. symbolic link distinction resolves itself completely. **Hard link**: a new directory entry pointing to an *existing inode*. `link("/existing", "/alias")` writes one new `dirent_t` record. One inode, two names. The inode's `nlinks` increments. Delete either name — the inode survives. Delete both — the inode is freed. No indirection, no path lookup at the target, no dangling reference possible. The alias and the original are truly identical in the filesystem's view. **Symbolic link** [[EXPLAIN:symbolic-links-—-files-containing-a-target-path-string-resolved-at-access-time|Symbolic links — files containing a target path string, resolved at every access]]: a new inode whose `mode` is `S_IFLNK` and whose data block contains a *path string*. `symlink("/existing", "/symlink")` creates a new inode (new `nlinks=1`), writes the string `/existing` into its data block, and creates a directory entry pointing to that inode. When the kernel encounters `S_IFLNK` during path resolution, it reads the link's data block, gets the path string, and re-starts path resolution from scratch with that string. The target inode's `nlinks` is NOT incremented — the symlink has no knowledge of the target inode at all, only the target *path*. This means:

- Delete the original: hard links still work (shared inode survives), symlinks break ("dangling symlink")
- Move the original to a different path: hard links still work, symlinks break
- Symlinks can cross filesystem boundaries (the target path is just a string), hard links cannot (inode numbers are filesystem-local)
- Symlinks can point to directories; hard links to directories are forbidden Now you see why: hard links are inode-level operations (increment `nlinks`, add `dirent_t`). Symlinks are inode-creation operations with a path payload.

DNS Resolution: The Same Algorithm

The path resolution algorithm you just implemented — walk a hierarchy component by component, each level stored at a different "authority" — is structurally identical to [[EXPLAIN:dns-resolution-—-recursive-hierarchical-name-lookup-from-root-to-authoritative-server|DNS resolution — recursive hierarchical name lookup]]. `/home/user/file.txt` → resolve:

- `/` (root) → scan for `home` → inode 14
- inode 14 → scan for `user` → inode 82
- inode 82 → scan for `file.txt` → inode 1041 `www.example.com.` → resolve:
- `.` (root nameservers) → query for `com.` → NS record for com. authority
- `com.` authority → query for `example.com.` → NS record for example.com. authority
- `example.com.` authority → query for `www.example.com.` → A record: 93.184.216.34 In both cases: hierarchical namespace, each component stored at a specific authority, walk from root to leaf. The difference is that filesystem lookup is local (disk reads), while DNS is distributed (network queries). The BIND nameserver's zone file format even looks like a directory entry: `(name, class, type, rdata)` where `rdata` is the "inode number" (IP address, or pointer to next authority). This cross-domain connection is not superficial. DNS negative caching (caching NXDOMAIN responses to avoid

re-querying for missing entries) is the exact analogue of the Linux VFS dcache caching ENOENT results. Both systems solve the same "frequent lookups of hierarchical names with high locality" problem.

Container Overlay Filesystems (OverlayFS / Docker)

[[EXPLAIN:overlayfs-—-union-mount-filesystem-that-layers-directories-for-copy-on-write-container-images|OverlayFS — union mount filesystem layering directories for Docker container images]] Docker's layers are a direct consequence of understanding directory entries as data. When you build a Docker image:

```
Layer 1 (base OS):       /bin/bash, /etc/hosts, /lib/...
Layer 2 (your app):      /app/server, /app/config.json
Layer 3 (runtime write): /tmp/cache, /var/log/app.log
```

OverlayFS merges these by stacking directory entry tables. The "upper" layer's entries take precedence. A file deleted in an upper layer is represented by a **whiteout entry** — a special directory entry (in Linux: a character device with major/minor 0,0, or an entry with the `.wh.` prefix) that signals "this name is deleted in this layer." Any lookup that finds a whiteout stops and returns ENOENT, even if a lower layer has an entry with the same name. This is precisely your `inode_num = 0` sentinel for deleted entries — a "whiteout" marker in a specific layer's directory block. The "union mount" view is the result of scanning layers in order and applying precedence rules to the merged list of directory entries. Understanding your `dirent_t` with `inode_num = 0` for deleted slots is understanding Docker's layer isolation model at the filesystem level.

Database Index Lookups: The Linear Scan Problem

You have just implemented linear scan directory lookup: iterate every entry until you find the name. For a directory with 10 entries this is fast. For a directory with 100,000 entries (a common pattern in `/var/spool/mail` or large upload directories), this is catastrophically slow. ext2 uses linear scan. ext4 introduced `htree` directories: when a directory grows beyond a threshold, its entry layout switches from the flat linear format to a **B-tree indexed by the hash of the filename**. The root node of the B-tree lives in the first directory block. Lookups compute `hash(name)`, walk the B-tree to find the leaf block containing entries with that hash, then scan the leaf block (small because hash-identical names are rare). This is **exactly** the difference between a sequential scan and a hash index in a database. A table with no index requires full sequential scan for point lookups. Adding a hash index converts that to O(1). ext4's `dir_index` feature flag marks filesystems where this optimization is active. The performance numbers: linear scan at 100,000 entries in a single directory costs ~~100,000 memory comparisons, ~24 block reads (~~2.4ms from SSD, ~~240ms from spinning disk per lookup). The B-tree index converts this to ~3 block reads per lookup (~~300µs from SSD). At 100 lookups/second, the difference is 240ms vs. 30ms — an 8× throughput improvement.

Git Tree Objects: Directories All The Way Down

[[EXPLAIN:git-objects-—-content-addressable-store-of-blobs-trees-commits-and-tags|Git objects — content-addressable store of blobs (file data), trees (directories), commits, and tags]] Git's internal object model is a direct implementation of the same inode/directory abstraction you just built:

Git Concept	Filesystem Equivalent
Blob object	File data (data blocks)
Tree object	Directory entry list (`dirent_t[]` array)
Commit object	Inode for a snapshot (mode, owner, size → author, date, message; blocks[0] → root tree SHA)
SHA-256 hash	Inode number (unique identifier for the content)
A Git tree object contains entries: `(mode, name, SHA)` — where mode is the file type, name is the filename, and SHA is the content hash of the blob or subtree. This is your `dirent_t` with `(file_type, name, inode_num)` where `inode_num` is replaced by a content hash.	
A Git commit is an inode with a tree pointer (`blocks[0]` → root tree) and metadata (author, timestamp, message). The filesystem hierarchy encoded in Git is navigated exactly like your directory tree: `git show HEAD:src/main.c` resolves `HEAD` → commit object → root tree → `src` subtree → `main.c` blob. Four object lookups, identical to your 4-hop path resolution.	
Understanding this connection means understanding why `git checkout` is fast (it reads blobs from the object store and writes them to the working tree — two filesystem-level operations), why `git status` can be slow in large repos (it must hash every file to compare with the index), and why `git bisect` works without a database (the commit graph is navigable through object references, just like your directory tree).	

Design Decisions

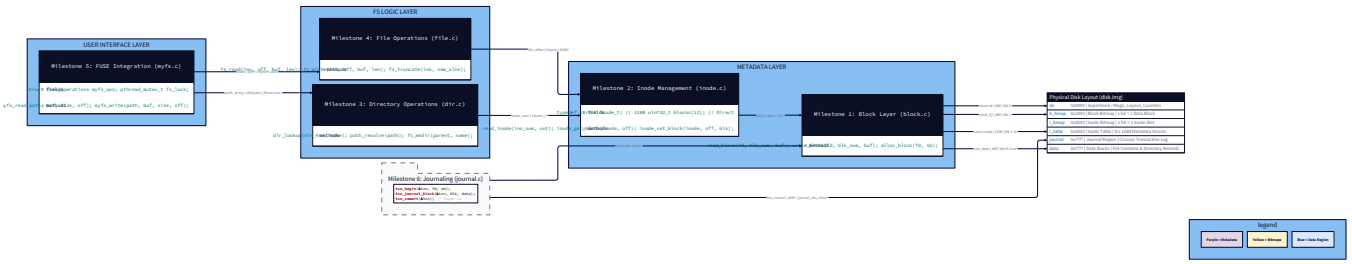
Option	Pros	Cons	Used By
Variable-length dirents with rec_len (chosen ✓)	Space-efficient for short names; proven format; scan-only so cache-friendly	O(n) lookup; scan required to find free space; deletion leaves holes	ext2/3/4, our implementation
Fixed-size dirents (256 bytes each)	O(1) indexed access by entry number; simpler implementation	Wastes 247 bytes for a 1-char filename; common case (short names) pays for worst case	FAT32 (11-byte 8.3 names, fixed), small embedded FSes
Hash-tree directory (htree)	O(1) lookup via filename hash; scales to millions of entries	Complex implementation; hash collisions need linear fallback; not crash-safe without journaling	ext4 (dir_index feature), NTFS, HFS+
B-tree directory index	Self-balancing; O(log n) worst case; handles any distribution	Complex; rebalancing on every insert/delete; overkill for < 10k entries	XFS (B-tree directories by default), ReiserFS
For our purposes, variable-length linear dirents give you the correct behavior and correct on-disk format. The htree optimization is a straightforward engineering addition — the same `dirent_t` format is used at the leaf level, only the lookup mechanism changes.			

Before You Move On: Pitfall Checklist

- ☐ **Name uniqueness on `add_entry`** : `dir_lookup` must be called before every `dir_add_entry`. A duplicate name causes ambiguous lookups — the first matching entry wins in a scan, making one of the two entries permanently unreachable.
- ☐ **Parent nlinks on `mkdir`** : After creating a child directory, increment the parent's `nlinks` by 1. After `rmdir`, decrement it. Every subdirectory represents one `..` reference to the parent. The invariant: `parent.nlinks = 2 + number_of_child_directories`.
- ☐ **Root's `..` points to root**: During `mkfs`, verify that `path_resolve("/..")` returns `root_ino`, not an error. Root is its own parent. This is a base case for all `..` traversal.
- ☐ **Max filename length enforced**: Every function that accepts a name parameter must check `strlen(name) <= MAX_FILENAME_LEN`. A 256-byte name silently overflows `name_len` (which is `uint8_t`, max 255) and writes garbage `rec_len` values that corrupt the directory block scan.
- ☐ **Hard links to directories are forbidden**: `fs_link` must return `-EPERM` for `S_ISDIR` targets. Without this check, you can create filesystem cycles that loop `find` and `du` forever.
- ☐ **`rmdir` only on empty directories**: `dir_is_empty` must be called before any removal. Check that it skips deleted entries (`inode_num == 0`) correctly — a block with only deleted entries should be considered empty.

- [ ] **Bitmap-before-inode write order**: In `mkdir`, write the bitmap marking the new inode and data block as used *before* writing the new inode to the inode table. If you crash between the two writes, a used-but-uninitiated inode (bitmap=1, inode data=garbage) is detectable by `fsck` as an orphan. A valid-inode-pointing-to-a-free-block (inode written, bitmap not yet updated) is a double-allocation bug.
- [ ] **`ctime` updates on all inode modifications**: `dir_add_entry` and `dir_remove_entry` must update `dir_inode.ctime` (and `mtime`). `fs_link` must update `target_inode.ctime`. These timestamps are used by `rsync` for change detection.
- [ ] **`strtok_r` not `strtok`**: Path resolution uses `strtok_r` (reentrant version). `strtok` uses a global static pointer and is not thread-safe. In the FUSE multi-threaded environment (Milestone 5), using `strtok` causes intermittent path resolution corruption that is nearly impossible to debug.

What You've Built and What Comes Next



You have collapsed the illusion of the "folder." A directory is a file. Its data blocks are a linked sequence of `(name, inode_number)` records. The entire filesystem tree emerges from files pointing to other files. Path resolution is a mechanical scan: read a block, find the name, follow the inode number, repeat. The `.` and `..` entries that make `cd ..` work are stored as ordinary bytes in ordinary data blocks. The elegance is complete: every part of the filesystem — the superblock (block 0), the bitmaps (blocks 1–2), the inode table, the journal, and the directory tree — is made from the same substrate: 4KB blocks. The naming system that makes the tree navigable is itself stored *in* the tree. But look at what's missing: you can create directories and add entries, but you cannot yet read or write actual file data. A regular file (`S_IFREG`) whose `blocks[]` array points to data blocks containing bytes is not yet implemented. In Milestone 4, you will build `read(inode, offset, length)` and `write(inode, offset, data)` — the operations that traverse the block pointer tree from Milestone 2 and move data between the application and disk. You will implement `truncate` (shrinking and extending files), handle the read-modify-write problem for partial block writes, and support sparse files where holes read as zeros. The filesystem is almost complete.

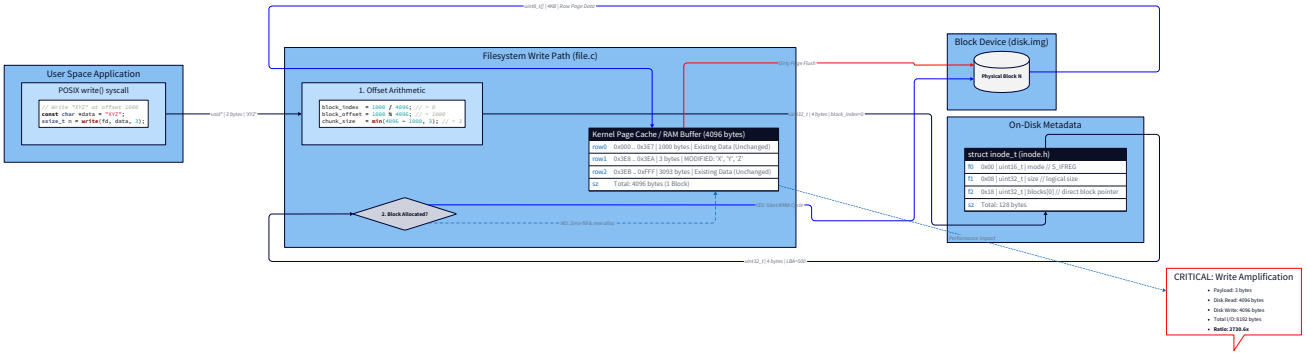
Two milestones ago, you had a flat block array. Now you have a navigable tree of named files. The blocks are starting to look like a filesystem.

Milestone 4: File Read/Write Operations

The Lie Every Developer Believes About File Writes

You have written file I/O code hundreds of times. You have called `write(fd, buf, 100)` and trusted that those 100 bytes landed at the right place on disk. Your mental model is reasonable: files are byte-addressable, persistent arrays. You write to offset 5000, bytes 5000–5099 change. Simple. Now consider what your hardware actually sees when you ask it to change byte 5001. The disk does not know what byte 5001 is. The disk only understands one unit of transfer: a block. Four kilobytes. The disk will not read 1 byte, will not write 1 byte, will not even acknowledge the concept. To change byte 5001 in a file, you must:

1. **Read** the entire 4KB block containing byte 5001 into RAM
2. **Modify** the single byte you care about, in memory
3. **Write** the entire 4KB block back to disk This is a **read-modify-write (RMW) cycle**. And it is not an implementation quirk you can engineer around — it is the fundamental physics of block storage. Every file write, no matter how small, triggers this cycle if the write does not align perfectly to block boundaries. It gets worse. Your file's data does not live at a fixed disk location. It lives somewhere pointed to by the inode's block pointer tree — the structure you built in Milestone 2. So before you can even begin the RMW cycle, you must translate the file offset into a disk block number by traversing that tree. Then perform the RMW. Then update the inode if the file grew. Then update timestamps. What looks like "write 100 bytes" is actually a multi-step dance involving potentially 4–6 disk operations. This is the gap between the POSIX abstraction and the hardware reality. This milestone is where you close that gap — by building the machinery that makes the abstraction work, understanding exactly what it costs, and seeing why the rest of systems programming (databases, SSDs, buffer caches) exists largely to compensate for this fundamental mismatch.



The Fundamental Tension: Byte-Addressable API, Block-Addressable Storage

The POSIX file API is byte-oriented:

- `read(fd, buf, n)` — read exactly `n` bytes from the current position
- `write(fd, buf, n)` — write exactly `n` bytes to the current position
- `lseek(fd, offset, SEEK_SET)` — position to any byte offset The storage hardware is block-oriented:
- Transfer unit: 4,096 bytes (or 512 bytes at the sector level)

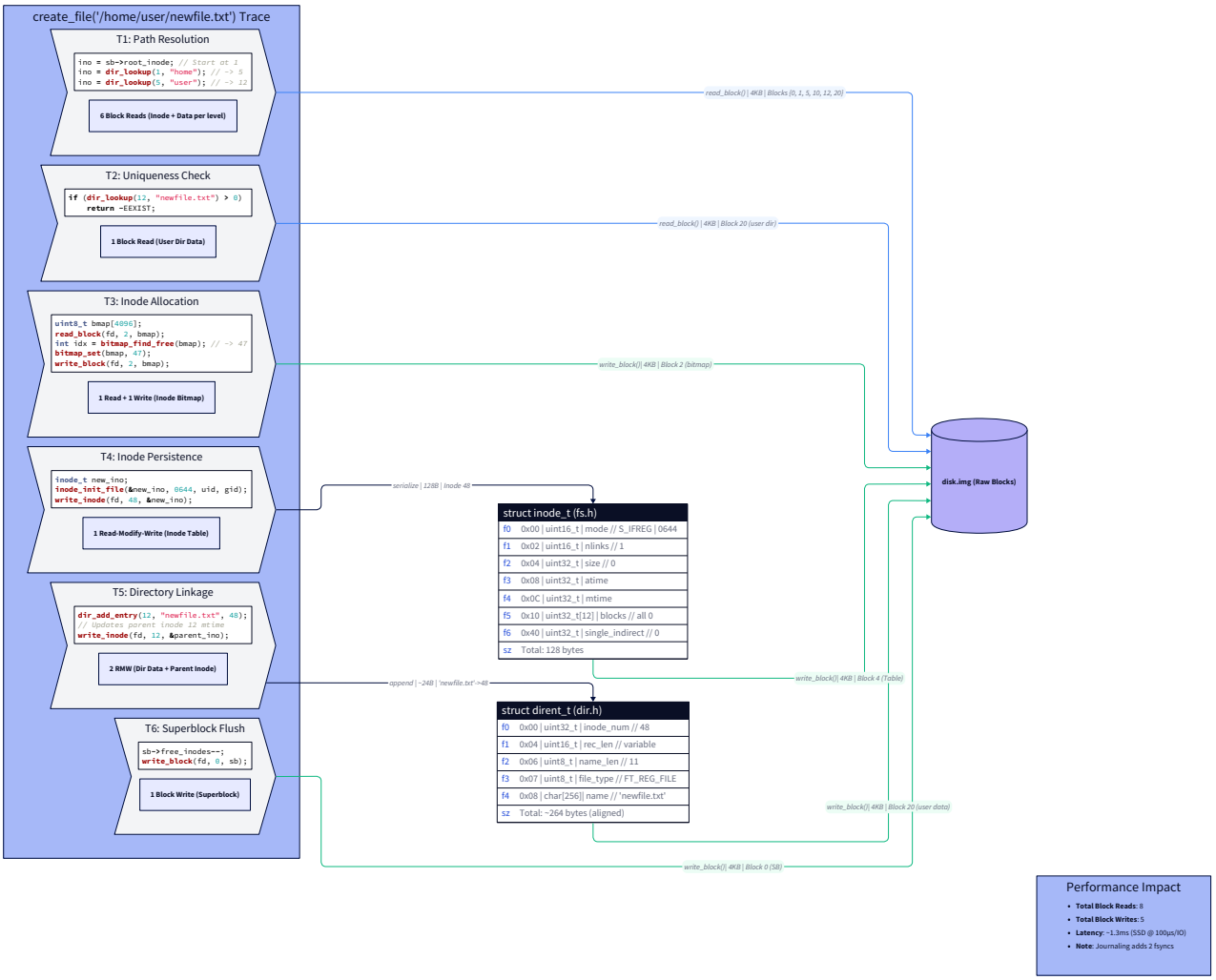
- Operations: read block N, write block N
- No concept of partial block access The filesystem is the translator. It must implement the byte-addressable API entirely on top of block-level primitives. Every operation must be decomposed into:

1. **Which block(s) does this byte range touch?** (offset arithmetic)
2. **Where is each of those blocks on disk?** (pointer tree traversal from Milestone 2)
3. **Which RMW cycles are needed?** (for writes that don't fill entire blocks)
4. **Which new blocks need allocation?** (for writes that extend the file)
5. **What metadata needs updating?** (inode size, timestamps) The concrete numbers driving this tension:

- Writing 1 byte costs: 1 block read (100µs SSD) + 1 block write (100µs SSD) = **200µs minimum**
- A purely sequential 4KB-aligned write costs: 1 block write = **100µs**
- The ratio: a 1-byte random write is 2× more expensive than a 4KB write, even though it transfers 4096× less data This is why database designers lose sleep over random I/O patterns. This is why SSDs use write buffers. This is why the Linux page cache exists. Every optimization in every storage system is ultimately trying to avoid unnecessary RMW cycles.

`create_file`: Bringing a File Into Existence

Before you can read or write a file, it must exist. `create_file` bridges Milestone 3's directory layer and Milestone 2's inode layer into a single coherent operation.



The operation is a composition of primitives you have already built:

1. Verify the parent directory exists and does not already contain `name`
2. Allocate a new inode from the inode bitmap
3. Initialize the inode (regular file, `nlinks = 1`, `size = 0`, all block pointers zero)
4. Write the initialized inode to the inode table on disk
5. Add a directory entry in the parent directory pointing to the new inode
6. Return the new inode number Notice that step 6 returns the inode number, not a file descriptor. Your filesystem layer operates on inode numbers directly. The FUSE integration in Milestone 5 will translate between POSIX file descriptors and inode numbers.

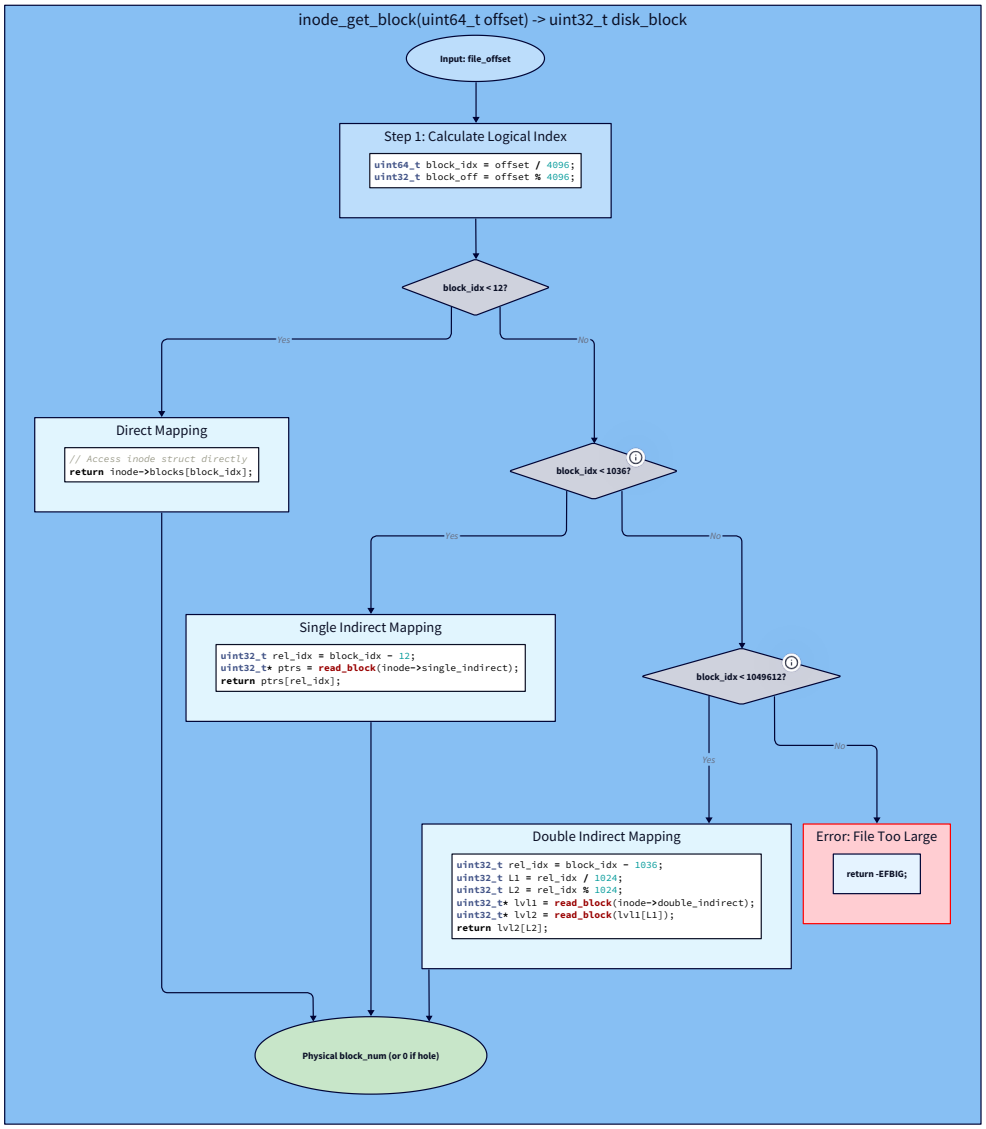
```c
#include <string.h>
#include <time.h>
#include <errno.h>
/*
 * Create a new regular file in a parent directory.
 *
 * fd:         open file descriptor of the disk image
 * sb:         superblock (updated in place for free counts)
 * parent_ino: inode number of the parent directory
 * name:       filename to create (single component, not a full path)
 * mode:       permission bits; will be OR'd with S_IFREG
 * uid, gid:   owner
 *
 * Returns the new file's inode number on success, negative errno on failure.
 */
int fs_create_file(int fd, superblock_t *sb,
                   uint32_t parent_ino, const char *name,
                   uint16_t mode, uint16_t uid, uint16_t gid) {
    /* Validate parent */
    inode_t parent_inode;
    if (read_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
    if (!S_ISDIR(parent_inode.mode)) return -ENOTDIR;
    /* Check name length and uniqueness */
    size_t name_len = strlen(name);
    if (name_len == 0 || name_len > MAX_FILENAME_LEN) return -EINVAL;
    if (dir_lookup(fd, sb, &parent_inode, name) > 0) return -EEXIST;
    /* Allocate new inode */
    int new_ino = alloc_inode(fd, sb);
    if (new_ino < 0) return -ENOSPC;
    /* Initialize inode for a regular empty file */
    inode_t new_inode;
```

```
    memset(&new_inode, 0, sizeof(new_inode));
    new_inode.mode   = (uint16_t)(S_IFREG | (mode & 0777));
    new_inode.uid    = uid;
    new_inode.gid    = gid;
    new_inode.nlinks = 1;
    new_inode.size   = 0;       /* empty file — no data yet */
    uint32_t now = (uint32_t)time(NULL);
    new_inode.atime = new_inode.mtime = new_inode.ctime = now;
    /* All block pointers start as 0 (null) — no blocks allocated yet */
    /* Write inode to disk */
    if (write_inode(fd, sb, (uint32_t)new_ino, &new_inode) != 0) {
        /* Roll back inode allocation */
        uint8_t ibmap[BLOCK_SIZE];
        read_block(fd, INODE_BITMAP_BLOCK, ibmap);
        bitmap_clear(ibmap, (uint32_t)new_ino - 1);
        write_block(fd, INODE_BITMAP_BLOCK, ibmap);
        sb->free_inodes++;
        return -EIO;
    }
    /* Add directory entry in parent */
    int r = dir_add_entry(fd, sb, parent_ino, name,
                          (uint32_t)new_ino, FT_REG_FILE);
    if (r != 0) {
        /* Best-effort cleanup: free the inode */
        free_inode(fd, sb, (uint32_t)new_ino);
        return r;
    }
    return new_ino;
}
```

> **Design note**: You write the inode to disk *before* adding the directory entry. If you crash between the two writes, you have an allocated inode that no directory entry points to — an "orphan." This is detectable and recoverable by `fsck` (it can scan all inodes and verify every allocated inode has at least one directory reference). The reverse — directory entry pointing to an uninitialized inode — is far worse, because code would follow the inode number, read garbage, and potentially corrupt the filesystem. Bitmap-before-inode, inode-before-dirent: this is the safe write ordering without journaling.

Offset Arithmetic: The Two-Level Translation

Before implementing `read` and `write`, you need to deeply understand the offset-to-block translation. This is where bugs hide, where off-by-ones silently corrupt neighboring files, and where the whole system's correctness lives.

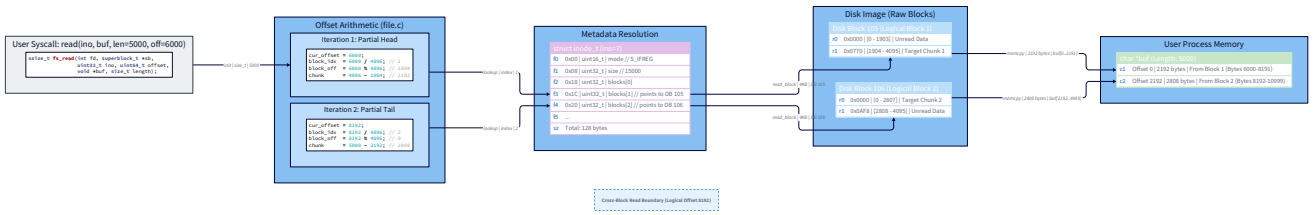


legend	
header	Constants (4KB Blocks)
d	Direct Limit | 12 blocks | 48 KB
s	Single Limit | 1036 blocks | ~4 MB
db	Double Limit | 1,049,612 blocks | ~4 GB

Given a file offset (a byte position within the file), you need three things:

1. **Block index** (`block_index = offset / BLOCK_SIZE`): which logical block of the file, starting at 0
2. **Byte offset within block** (`block_offset = offset % BLOCK_SIZE`): how far into that block
3. **Disk block number**: obtained by calling `inode_get_block(fd, inode, block_index * BLOCK_SIZE, &disk_block_num)` — the function you built in Milestone 2 Let's trace a concrete example. A `read` starting at file offset 6,000 for 5,000 bytes:

```
File offset 6000:
  block_index   = 6000 / 4096 = 1
  block_offset  = 6000 % 4096 = 1904
Bytes [6000 .. 8191] are in block index 1, starting at position 1904
  → 4096 - 1904 = 2192 bytes available in this block
Bytes [8192 .. 10999] are in block index 2, starting at position 0
  → remaining 2808 bytes of the 5000-byte request
So this 5000-byte read requires two block reads.
```



The general algorithm for any multi-block operation:

```
remaining = total_length
current_offset = start_offset
while remaining > 0:
    block_index  = current_offset / BLOCK_SIZE
    block_offset = current_offset % BLOCK_SIZE
    chunk_size   = min(BLOCK_SIZE - block_offset, remaining)
    # get disk block number for block_index
    # do block I/O using chunk_size at block_offset
    current_offset += chunk_size
    remaining      -= chunk_size
```

This loop is the skeleton of both `read` and `write`. The difference is what you do with the block once you have it.

`fs_read`: Reading Data Through the Pointer Tree

Reading is the simpler of the two operations because it is read-only: you traverse the pointer tree, fetch blocks, copy data, update `atime`. No allocation, no RMW cycle. The one nuance: when `inode_get_block` returns 0 (a null pointer — a sparse hole), you must zero-fill the output buffer for that range. No disk I/O. No allocation. Zeros come from CPU registers, not from disk.

```c
/*
 * Read data from a file.
 *
 * fd:       disk image file descriptor
 * sb:       superblock
 * ino:      inode number of the file to read
 * offset:   byte offset within the file to start reading
 * buf:      caller-allocated buffer to receive data
 * length:   number of bytes to read
 *
 * Returns the number of bytes actually read on success (may be less
 * than length if offset + length extends past the end of file),
 * or negative errno on error.
 */
ssize_t fs_read(int fd, superblock_t *sb,
                uint32_t ino, uint64_t offset,
                void *buf, size_t length) {
    inode_t inode;
    if (read_inode(fd, sb, ino, &inode) != 0) return -EIO;
    if (!S_ISREG(inode.mode)) return -EINVAL;
    /* Clamp to end of file — cannot read past EOF */
    if (offset >= inode.size) return 0;
    if (offset + length > inode.size) {
        length = (size_t)(inode.size - offset);
    }
    uint8_t *out = (uint8_t *)buf;
    size_t   remaining = length;
    uint64_t cur_offset = offset;
    uint8_t  block_buf[BLOCK_SIZE];
    while (remaining > 0) {
        /* How many bytes into the current block, and how many to copy from it */
```

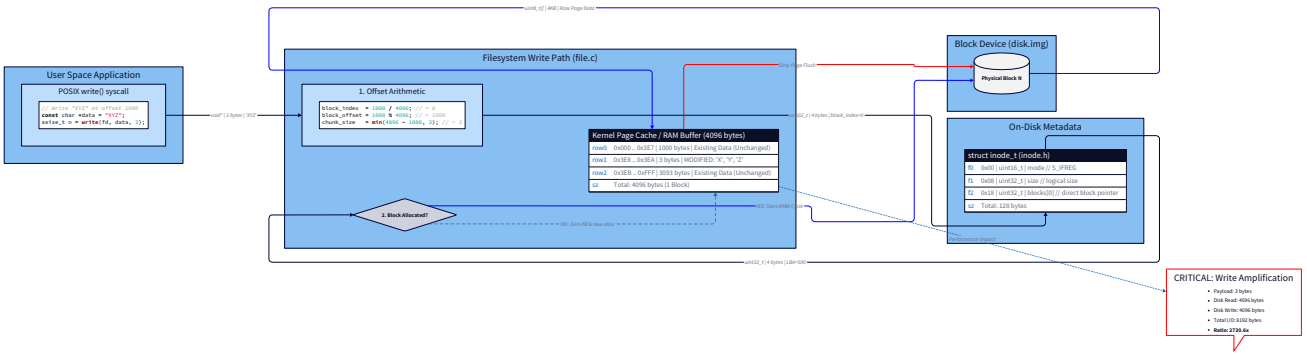
```
    uint32_t block_off  = (uint32_t)(cur_offset % BLOCK_SIZE);
    size_t   chunk      = BLOCK_SIZE - block_off;
    if (chunk > remaining) chunk = remaining;
    /* Translate file offset → disk block number via inode pointer tree */
    uint32_t disk_block;
    int r = inode_get_block(fd, &inode,
                            cur_offset - block_off, &disk_block);
    if (r != 0) return r;
    if (disk_block == 0) {
        /*
         * Sparse hole: this block has never been written.
         * Return zeros — no disk I/O needed.
         * This is the correct POSIX behavior: holes read as zero bytes.
         */
        memset(out, 0, chunk);
    } else {
        /* Read the block from disk */
        if (read_block(fd, disk_block, block_buf) != 0) return -EIO;
        memcpy(out, block_buf + block_off, chunk);
    }
    out        += chunk;
    cur_offset += chunk;
    remaining  -= chunk;
}
/*
 * Update atime.
 * In production, use the 'relatime' heuristic (only update if atime
 * is older than mtime/ctime) to avoid write-per-read amplification.
 * For correctness, update unconditionally here.
 */
inode.atime = (uint32_t)time(NULL);
```

```
    write_inode(fd, sb, ino, &inode);

    return (ssize_t)length;
}
```

> **Hardware Soul — Where Are Those Zero Bytes From?** When `disk_block == 0` and you call `memset(out, 0, chunk)`, this is a write to the user's buffer in RAM. The CPU issues store instructions; modern CPUs can zero ~64 bytes per cycle using SIMD (SSE/AVX `vpxor` + `vmovdqu`). For a 4KB sparse read, this is ~64 CPU cycles — under 30ns. Compare to a real disk read: ~100µs on SSD. Sparse reads are **3,000× faster** than real reads. This is not a quirk; it is a design choice. Holes are free to read, which is why VM disk images work efficiently even when provisioned as 100GB but containing only 2GB of real data.

`fs_write`: The Read-Modify-Write Reality

Writing is where the complexity lives. Three distinct cases must be handled, and they can all occur within a single `write` call: **Case 1: Write completely fills one or more blocks** (offset is block-aligned, length is a multiple of `BLOCK_SIZE`) → No RMW needed. Read zero existing content. Just allocate the block if needed and write full 4KB chunks. **Case 2: Write starts or ends in the middle of a block** (partial first or last block) → RMW required. Read the existing block content, overwrite the portion being written, write the whole block back. **Case 3: Write extends the file past EOF and the partial block at EOF has existing content** → Partial RMW: read the existing partial block (which may be partially written), zero-fill up to `BLOCK_SIZE`, write new data on top, then write back.



The implementation handles all three cases in a single unified loop by always reading the block before writing if the write is partial:

```c
/*
 * Write data to a file.
 *
 * fd:       disk image file descriptor
 * sb:       superblock (updated: free_blocks)
 * ino:      inode number of the file
 * offset:   byte offset within the file to start writing
 * buf:      data to write
 * length:   number of bytes to write
 *
 * Returns the number of bytes written on success (always equals length
 * unless a real error occurs), negative errno on failure.
 *
 * Side effects:
 *   - Allocates new data blocks as needed
 *   - Allocates indirect/double-indirect pointer blocks as needed
 *   - Updates inode.size if writing past current EOF
 *   - Updates inode.mtime and inode.ctime
 */
ssize_t fs_write(int fd, superblock_t *sb,
                 uint32_t ino, uint64_t offset,
                 const void *buf, size_t length) {
    if (length == 0) return 0;
    inode_t inode;
    if (read_inode(fd, sb, ino, &inode) != 0) return -EIO;
    if (!S_ISREG(inode.mode)) return -EINVAL;
    /* Check file size limit (our inode uses uint32_t for size → max ~4GB) */
    if (offset + length > (uint64_t)UINT32_MAX) return -EFBIG;
    const uint8_t *in  = (const uint8_t *)buf;
    size_t   remaining = length;
    uint64_t cur_offset = offset;
```

```
    uint8_t  block_buf[BLOCK_SIZE];

    while (remaining > 0) {

        uint32_t block_off  = (uint32_t)(cur_offset % BLOCK_SIZE);

        size_t   chunk      = BLOCK_SIZE - block_off;

        if (chunk > remaining) chunk = remaining;

        bool partial_write  = (block_off != 0) || (chunk < BLOCK_SIZE);

        /* Get current disk block number for this file offset (may be 0 = hole) */

        uint32_t disk_block;

        int r = inode_get_block(fd, &inode,

                                cur_offset - block_off, &disk_block);

        if (r != 0) return r;

        if (disk_block == 0) {

            /* Block not yet allocated — allocate a new one */

            int new_blk = alloc_block(fd, sb);

            if (new_blk < 0) return -ENOSPC;

            disk_block = (uint32_t)new_blk;

            /*

             * Record the new block number in the inode pointer tree.

             * inode_set_block may allocate indirect/double-indirect blocks

             * if the file offset is in those regions.

             */

            r = inode_set_block(fd, sb, &inode,

                                cur_offset - block_off, disk_block);

            if (r != 0) return r;

            /*

             * Zero-initialize the newly allocated block.

             *

             * WHY: If this is a partial write (we won't fill the entire block),

             * we need the rest of the block to contain zeros, not garbage from

             * whatever previously occupied this disk location.

             *
```

```
            * If the write IS full-block, the memset is overwritten immediately
            * below. Slight inefficiency for full-block writes; correctness
            * for partial writes.
            */
           memset(block_buf, 0, BLOCK_SIZE);
       } else if (partial_write) {
           /*
            * Block exists and this is a partial write: read existing content
            * so we can preserve the bytes we're not overwriting.
            *
            * This is the READ in "read-modify-write".
            * Cost: 1 block read (~100µs SSD) that would not be needed if
            * writes were always block-aligned.
            */
           if (read_block(fd, disk_block, block_buf) != 0) return -EIO;
       }
       /*
        * Copy the new data into the buffer at the correct offset within the block.
        * For full-block writes: block_off == 0, chunk == BLOCK_SIZE.
        * For partial writes: we modify only the [block_off, block_off+chunk) range.
        */
       memcpy(block_buf + block_off, in, chunk);
       /* Write the (possibly modified) block back to disk */
       if (write_block(fd, disk_block, block_buf) != 0) return -EIO;
       in         += chunk;
       cur_offset += chunk;
       remaining  -= chunk;
   }
   /*
    * Update inode metadata.
    * Only extend size — writing within existing content does not shrink the file.
```

```
     */
    if (cur_offset > inode.size) {
        inode.size = (uint32_t)cur_offset;
    }
    uint32_t now = (uint32_t)time(NULL);
    inode.mtime  = now;
    inode.ctime  = now;   /* inode metadata changed (size may have changed) */
    if (write_inode(fd, sb, ino, &inode) != 0) return -EIO;
    return (ssize_t)length;
}
```

Let's trace through the concrete example from the chapter opening: `write(fd, buf, 100)` at offset 5000:

```
offset = 5000, length = 100
Iteration 1:
  block_off = 5000 % 4096 = 904
  chunk     = min(4096 - 904, 100) = min(3192, 100) = 100
  partial_write = true  (block_off != 0)
  → inode_get_block for block index 1
  → if disk_block exists: read_block (THE READ in RMW)
  → memcpy(block_buf + 904, in, 100)  (THE MODIFY)
  → write_block  (THE WRITE)
  → remaining = 0, loop ends
Total disk operations: 1 read + 1 write = 2 block I/Os
Data actually modified: 100 bytes
Data transferred to/from disk: 4096 + 4096 = 8192 bytes
Write amplification factor: 81.92×
```

Eighty-two times more data moved than the actual payload. For a 1-byte write, the amplification is 8192×. This is not a bug; it is the unavoidable cost of block-aligned storage.

> **Hardware Soul — Block Write Ordering on SSD** When `write_block` returns, the data is in the kernel's **page cache** — kernel memory, not on any persistent storage. The kernel may hold it there for seconds or minutes, coalescing multiple writes to the same block before flushing. This is why `write()` is fast in practice despite the RMW overhead: subsequent writes to the same block hit the page cache (memory speed) instead of disk. The page cache converts your RMW cycle from a disk operation into a memory operation. Only when the page is eventually flushed (dirty page writeback, `fsync`, or unmount) does the actual disk I/O occur. For your filesystem running on top of a host kernel, this cache layer is automatic. Your filesystem is inside the cache from the host OS's perspective.

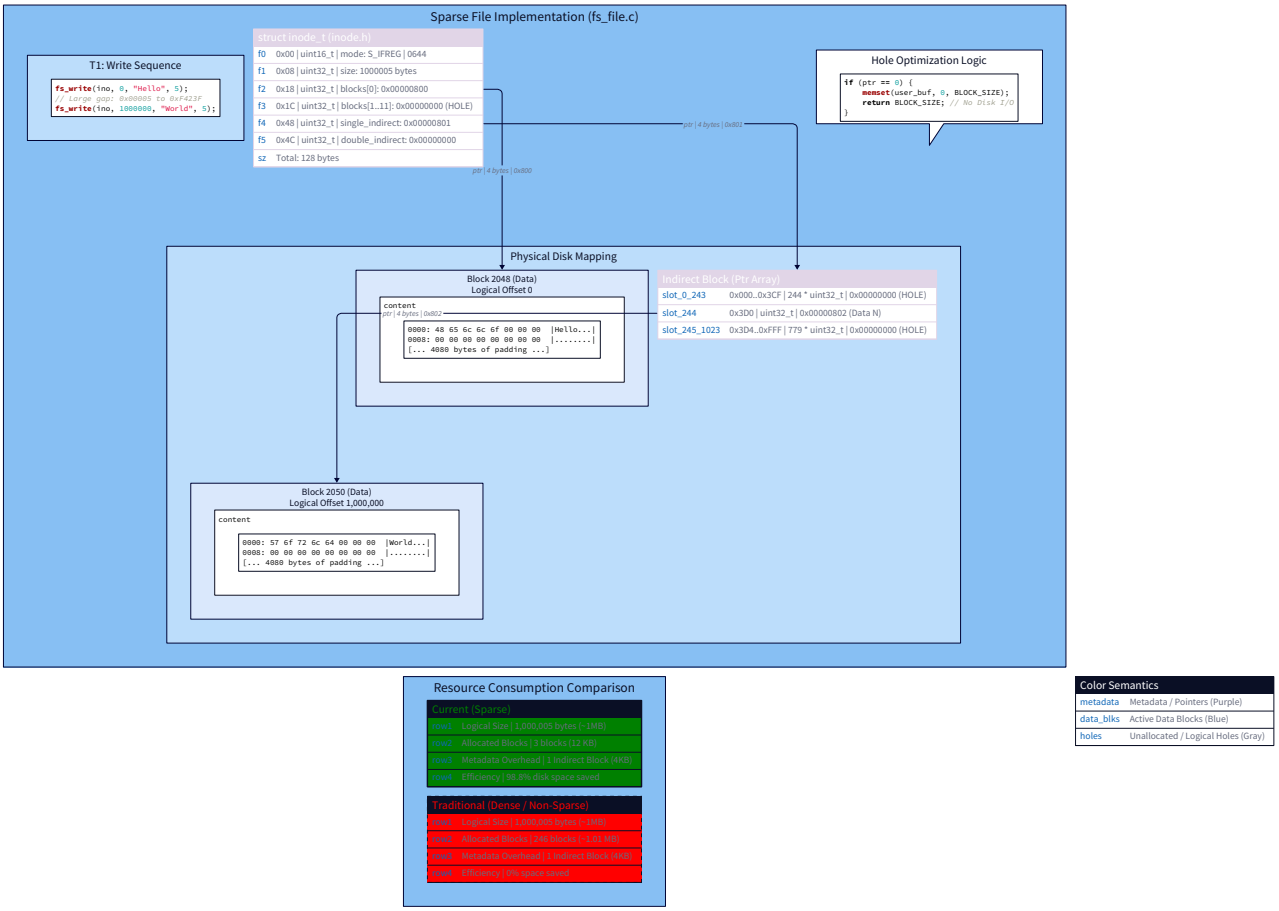
Sparse File Writes: Skipping the Gap

The sparse file case is handled naturally by your write loop, but it deserves explicit attention because the behavior is subtle. Consider: `fs_write(fd, sb, ino, 1000000000, "X", 1)` — writing one byte at offset one billion, into a file that was previously empty.

```
Iteration 1:
  cur_offset  = 1,000,000,000
  block_off   = 1,000,000,000 % 4096 = 3904
  chunk       = 4096 - 3904 = 192... but wait, length=1, so chunk=1
  block_index = 1,000,000,000 / 4096 = 244,140 (in the double-indirect region)
  inode_get_block → returns disk_block = 0 (no block allocated)
  alloc_block → returns new disk block number
  inode_set_block(fd, sb, &inode, 999,996,416, new_block)
    → this allocates the double-indirect block (if not yet allocated)
    → this allocates the single-indirect block within it (if not yet allocated)
    → stores new_block in the correct slot
  memset(block_buf, 0, BLOCK_SIZE)  ← entire block is zeroed
  block_buf[3904] = 'X'             ← our one byte
  write_block(fd, new_block, block_buf)
After loop:
  inode.size = 1,000,000,001
```

The blocks between offset 0 and offset 999,996,416 are **never touched**. They remain as null pointers in the inode's block pointer tree. Every `inode_get_block` call for any offset in that range returns 0, and every `fs_read` call for that range returns zeros from `memset`. The file appears to be 1GB but uses only:

- 1 data block (4KB)
- 1 single-indirect block (4KB)
- 1 double-indirect block (4KB) Total on-disk: 12KB. Logical size: ~1GB. [[EXPLAIN:thin-provisioning-—-storage-allocation-deferred-until-first-write|Thin provisioning — deferred storage allocation where capacity is reserved logically but not physically allocated until written]]



This is not a special mode you enable. Sparse files are the default behavior of null block pointers. The file you create is automatically sparse if you write at high offsets without filling in between.

Append: Writing at the End

Append is a special case of write where the starting offset is always `inode.size` (the current end of file). It is not fundamentally different from a mid-file write, but it has one important optimization opportunity: you never need to RMW the last block if the previous write happened to fill it exactly.

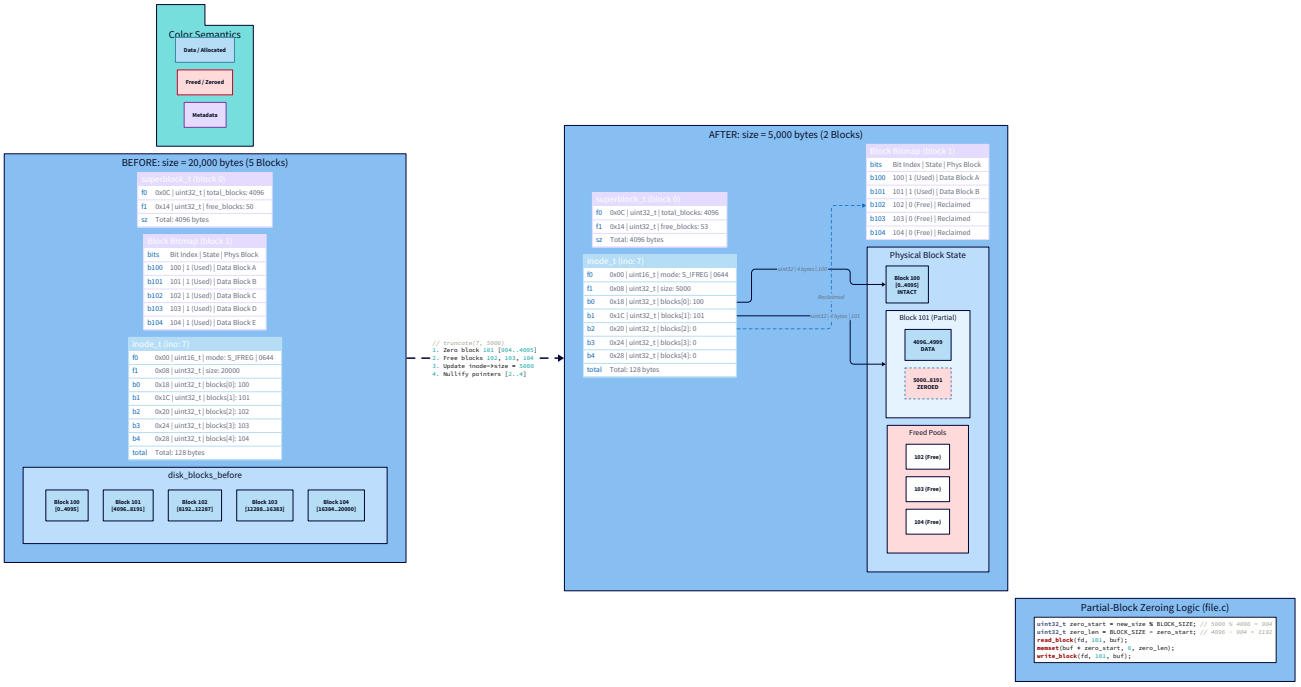
```c
/*
 * Append data to the end of a file.
 * Convenience wrapper around fs_write.
 *
 * Returns number of bytes written, or negative errno.
 */
ssize_t fs_append(int fd, superblock_t *sb,
                  uint32_t ino, const void *buf, size_t length) {
    /* Read current inode to get the file size (= append offset) */
    inode_t inode;
    if (read_inode(fd, sb, ino, &inode) != 0) return -EIO;
    return fs_write(fd, sb, ino, (uint64_t)inode.size, buf, length);
}
```

The interesting case for append: what if the previous write ended at offset 4095 (one byte before a block boundary)? The next append's first byte lands at offset 4096 — a perfect block boundary. `block_off = 4096 % 4096 = 0`. The first iteration has `partial_write = false` if `chunk == BLOCK_SIZE`. No read needed. This is why sequential append-only workloads (log files, write-ahead journals, streaming data) are efficient at the block layer: they naturally generate full-block writes without RMW overhead.

> The lesson: if you control your write pattern, **align writes to block boundaries**. A log that always writes in 4KB chunks avoids every RMW cycle. This is not an accident — it is the same reason database write-ahead logs (WAL), Kafka partitions, and your Milestone 6 journal all use block-sized writes: they are optimizing away the most expensive per-operation cost.

`fs_truncate`: The Inverse of Write

Truncation changes a file's size without moving data. It has two symmetric cases: **Shrink** (`new_size < inode.size`): free all blocks that are now beyond the new end, zero out the partial last block if the new size is not block-aligned, update the inode pointer tree to null out freed entries. **Extend** (`new_size > inode.size`): do nothing to existing data or pointers — the new region is implicitly a sparse hole. Update `inode.size` . Future reads in the extended region return zeros. The extend case is trivially correct with sparse files: since null block pointers already read as zeros, you just need to update `inode.size` . The shrink case requires careful traversal.

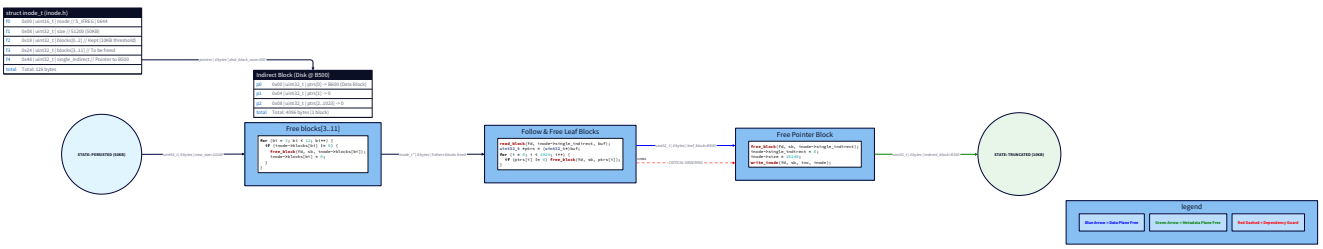


```c
/*
 * Truncate a file to new_size bytes.
 *
 * If new_size > inode.size: extend (hole-filling, no allocation needed).
 * If new_size < inode.size: shrink (free blocks beyond new_size, update pointers).
 * If new_size == inode.size: no-op (but still update ctime).
 *
 * Returns 0 on success, negative errno on failure.
 */
int fs_truncate(int fd, superblock_t *sb,
                uint32_t ino, uint64_t new_size) {
    if (new_size > (uint64_t)UINT32_MAX) return -EFBIG;
    inode_t inode;
    if (read_inode(fd, sb, ino, &inode) != 0) return -EIO;
    if (!S_ISREG(inode.mode)) return -EINVAL;
    if (new_size > inode.size) {
        /*
         * Extend: the new range [inode.size, new_size) is a sparse hole.
         * No block allocation needed. Just update size.
         * Future reads in this range return zeros via the sparse-hole path.
         */
        inode.size  = (uint32_t)new_size;
        inode.mtime = inode.ctime = (uint32_t)time(NULL);
        return write_inode(fd, sb, ino, &inode);
    }
    if (new_size == inode.size) {
        inode.ctime = (uint32_t)time(NULL);
        return write_inode(fd, sb, ino, &inode);
    }
    /*
     * Shrink: new_size < inode.size.
```

```
     *
     * We need to free every block whose file offset is >= new_size.
     * Additionally, if new_size is not block-aligned, we must zero out
     * the tail of the last retained block.
     *
     * Strategy: iterate through block indices from first_free_block_index
     * to the end of the file, free each block, and null its pointer.
     */
    uint32_t new_last_block  = (new_size == 0) ? 0
                                               : (uint32_t)((new_size - 1) / BLOCK_SIZE);
    uint32_t old_last_block  = (inode.size == 0) ? 0
                                                 : (uint32_t)((inode.size - 1) / BLOCK_SIZE);
    /* Step 1: Zero-fill the tail of the last retained block (partial block at new_size) */
    if (new_size > 0 && (new_size % BLOCK_SIZE) != 0) {
        uint32_t disk_block;
        int r = inode_get_block(fd, &inode,
                                (uint64_t)new_last_block * BLOCK_SIZE, &disk_block);
        if (r != 0) return r;
        if (disk_block != 0) {
            uint8_t block_buf[BLOCK_SIZE];
            if (read_block(fd, disk_block, block_buf) != 0) return -EIO;
            uint32_t zero_start = (uint32_t)(new_size % BLOCK_SIZE);
            memset(block_buf + zero_start, 0, BLOCK_SIZE - zero_start);
            if (write_block(fd, disk_block, block_buf) != 0) return -EIO;
        }
    }
    /* Step 2: Free all blocks with index > new_last_block (or all if new_size == 0) */
    uint32_t first_to_free = (new_size == 0) ? 0 : (new_last_block + 1);
    for (uint32_t bi = first_to_free; bi <= old_last_block; bi++) {
        uint32_t disk_block;
        int r = inode_get_block(fd, &inode,
```

```
                                (uint64_t)bi * BLOCK_SIZE, &disk_block);
        if (r != 0) continue;  /* best-effort during truncate */
        if (disk_block != 0) {
            free_block(fd, sb, disk_block);
            /* Null out the pointer in the inode/indirect block */
            inode_set_block(fd, sb, &inode,
                            (uint64_t)bi * BLOCK_SIZE, 0);
        }
    }
    /* Step 3: Free indirect pointer blocks that are now entirely empty.
     *
     * After nulling data block pointers, check if single-indirect and
     * double-indirect blocks themselves are now empty and can be freed.
     */
    truncate_free_indirect_if_empty(fd, sb, &inode, (uint32_t)new_size);
    /* Step 4: Update inode */
    inode.size  = (uint32_t)new_size;
    inode.mtime = inode.ctime = (uint32_t)time(NULL);
    return write_inode(fd, sb, ino, &inode);
}
```

The `truncate_free_indirect_if_empty` function checks whether the single-indirect and double-indirect pointer blocks should themselves be freed. This matters when truncating a large file down to something smaller than 48KB (below the direct-block threshold):



```c
/*
 * After nulling data block pointers during shrink-truncate,
 * check if the indirect pointer blocks themselves are now all-zeros
 * and can be freed.
 */
static void truncate_free_indirect_if_empty(int fd, superblock_t *sb,
                                            inode_t *inode,
                                            uint32_t new_size) {
    uint64_t single_start = (uint64_t)N_DIRECT * BLOCK_SIZE;
    uint64_t double_start = single_start + (uint64_t)PTRS_PER_BLOCK * BLOCK_SIZE;
    /* If new_size is within direct block range and single-indirect exists */
    if ((uint64_t)new_size <= single_start && inode->single_indirect != 0) {
        /* All data pointers in this block should already be nulled.
         * Free the single-indirect block itself. */
        free_block(fd, sb, inode->single_indirect);
        inode->single_indirect = 0;
    }
    /* If new_size is within (direct + single-indirect) range and double-indirect exists */
    if ((uint64_t)new_size <= double_start && inode->double_indirect != 0) {
        /*
         * For full correctness: scan each entry in the double-indirect block,
         * free any remaining single-indirect blocks within it.
         * The loop above already freed their data block pointers;
         * we now free the single-indirect blocks themselves.
         */
        uint8_t buf[BLOCK_SIZE];
        if (read_block(fd, inode->double_indirect, buf) == 0) {
            uint32_t *ptrs = (uint32_t *)buf;
            for (uint32_t i = 0; i < PTRS_PER_BLOCK; i++) {
                if (ptrs[i] != 0) {
                    /* Check if this single-indirect block is now all zeros */
```

```
                    uint8_t si_buf[BLOCK_SIZE];
                    if (read_block(fd, ptrs[i], si_buf) == 0) {
                        bool all_zero = true;
                        for (size_t j = 0; j < BLOCK_SIZE; j++) {
                            if (si_buf[j] != 0) { all_zero = false; break; }
                        }
                        if (all_zero) {
                            free_block(fd, sb, ptrs[i]);
                            ptrs[i] = 0;
                        }
                    }
                }
            }
            write_block(fd, inode->double_indirect, buf);
        }
        /* If new_size is in direct range, free the double-indirect block too */
        if ((uint64_t)new_size <= single_start) {
            free_block(fd, sb, inode->double_indirect);
            inode->double_indirect = 0;
        }
    }
}
```

> **Pitfall — Truncation and Partial Block Zeroing**: When you shrink a file to `new_size = 5000`, the last retained block (block index 1, covering file offsets 4096–8191) must have its bytes from 5000 to 8191 zeroed out. If you skip this step, the bytes from 5000 to 8191 remain on disk as whatever the file previously contained. If the file is later extended back past 5000 (via `truncate` extend), those "ghost bytes" reappear as if they were written, even though the file never logically contained them at those positions. This is a **security vulnerability** in multi-user systems — one user's file data leaks into another user's file after truncation and re-extension. Always zero the tail of the last retained block.

Metadata Correctness: Size, mtime, ctime

Every write and truncate operation must update inode metadata. Getting this right matters for correctness of tools like `rsync`, `make`, and every backup system that uses timestamps and file sizes for change detection. The rules:

Operation	`size`	`mtime`	`ctime`	`atime`
`write` (data changes)	Update if grown	Always update	Always update	Unchanged
`truncate` (size changes)	Always update	Always update	Always update	Unchanged
`read` (data accessed)	No change	No change	No change	Update
`chmod`, `chown` (meta changes)	No change	No change	Always update	Unchanged
A subtle correctness requirement: `ctime` must update whenever **anything** in the inode changes — including `size`, `mtime`, and `nlinks`. The reason: `ctime` is the "inode change time," not just the "data change time." A rename operation (which changes directory entries but not data) must update `ctime` on the affected inode. `mtime` only tracks data changes.				

```c
/*
 * Helper: update inode metadata after a write operation.
 * Call this instead of manually setting fields — ensures consistency.
 */
static void update_write_metadata(inode_t *inode,
                                  uint64_t new_end_offset) {
    uint32_t now = (uint32_t)time(NULL);
    if (new_end_offset > inode->size) {
        inode->size = (uint32_t)new_end_offset;
    }
    inode->mtime = now;
    inode->ctime = now;   /* size and mtime both changed → ctime must change */
}
```

Three-Level View: What Happens When `fs_write` Runs

Let's trace `fs_write(fd, sb, ino=7, offset=8192, buf="Hello", length=5)` — writing "Hello" at a block-aligned offset in a small file.

Level	What Happens
Application (your code)	Calls `fs_write`. Arithmetic: `block_off = 8192 % 4096 = 0`, `chunk = min(4096, 5) = 5`, `partial_write = true` (chunk < BLOCK_SIZE). Calls `inode_get_block(fd, &inode, 8192, &disk_block)`.
Your Filesystem Layer	`inode_get_block` computes `block_index = 2` → within direct range → returns `inode.blocks[2]`. If it's 0: `alloc_block(fd, sb)` reads the bitmap block, finds free bit, sets it, writes bitmap back, returns disk block number. `inode_set_block` stores the new disk block number in `inode.blocks[2]`. `memset(block_buf, 0, 4096)` zeroes new block. `memcpy(block_buf + 0, "Hello", 5)`. `write_block(fd, disk_block, block_buf)` calls `write(fd, buf, 4096)` at the right offset in the image file.
Host OS / Hardware	The host kernel receives the `write` syscall. The disk image file's page in the page cache is found (or faulted in). The 4096 bytes are written to the kernel's page cache page — marking it dirty. The `write` syscall returns immediately. Eventually, the page daemon (or an `fsync`) flushes this dirty page to the actual storage device via block I/O. The device DMA controller transfers the page from kernel memory to disk buffer. Disk confirms write. Page marked clean.
The key observation: your `write_block` returns before the data hits physical storage. The data exists in the host kernel's page cache — safe from process crashes but not from power failures. This is why `fsync(fd)` on your disk image file descriptor matters before claiming a write is "committed."	

Design Decision: When to Flush the Inode

After a `write`, you must update the inode (`size`, `mtime`, `ctime`) and write it to disk. But how often?

Strategy	Write Frequency	Data Safety	Performance
After every write (chosen ✓ for now)	Every `fs_write` call	Good: metadata matches data at all times (within page cache lag)	Poor: 1 inode write per data write = 2× disk I/Os
Batch: flush inode on `close` / `fsync`	Once at close/sync	Poor: crash between write and close loses size update; reads see wrong size	Better: inode writes amortized over many data writes
Write-through cache: dirty inode in memory, flush asynchronously	Asynchronously	With journaling: excellent; without: crash window	Best throughput
For your standalone implementation without a buffer cache, "after every write" is the correct choice — it is simple and correct. The journal in Milestone 6 will group inode updates with data writes into atomic transactions, both improving safety and enabling the batched flush optimization.			
The numbers: on a filesystem with 1,000 sequential 512-byte writes to the same file, the "after every write" strategy generates 1,000 inode writes and 1,000 data block writes (the data writes coalesce since they're in the same 4KB block after the first). The batched strategy generates ~1 inode write and ~1 data write. For sequential small writes, this is a 500× difference in inode write operations. This is one reason why `write` performance degrades sharply for small random writes to the same file.			

Unit Tests: Verifying Round-Trip Correctness

Your acceptance criteria demand round-trip verification. Here is a comprehensive test suite:

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>
/*
 * Full test suite for file read/write operations.
 * Run after mkfs on a fresh disk image.
 * Assumes root_ino is valid and accessible.
 */
void test_file_operations(int fd, superblock_t *sb) {
    printf("\n=== File Read/Write Tests ===\n");
    uint32_t root_ino = sb->root_inode;
    /* ---------------------------------------------------------------- */
    /* Test 1: create_file basic */
    int file_ino = fs_create_file(fd, sb, root_ino, "test.txt",
                                  0644, 1000, 1000);
    assert(file_ino > 0);
    inode_t inode;
    assert(read_inode(fd, sb, (uint32_t)file_ino, &inode) == 0);
    assert(S_ISREG(inode.mode));
    assert(inode.size == 0);
    assert(inode.nlinks == 1);
    printf("OK: create_file → inode %d, size=0\n", file_ino);
    /* ---------------------------------------------------------------- */
    /* Test 2: write and read back — small (sub-block, aligned) */
    const char *msg = "Hello, Filesystem!";
    size_t msg_len = strlen(msg);
    ssize_t written = fs_write(fd, sb, (uint32_t)file_ino,
                               0, msg, msg_len);
    assert(written == (ssize_t)msg_len);
    assert(read_inode(fd, sb, (uint32_t)file_ino, &inode) == 0);
    assert(inode.size == msg_len);
```

```
    char read_buf[64];
    memset(read_buf, 0xAA, sizeof(read_buf));
    ssize_t nread = fs_read(fd, sb, (uint32_t)file_ino,
                            0, read_buf, msg_len);
    assert(nread == (ssize_t)msg_len);
    assert(memcmp(read_buf, msg, msg_len) == 0);
    printf("OK: write + read round-trip (small, aligned)\n");
    /* ---------------------------------------------------------------- */
    /* Test 3: partial write — crosses NO block boundary */
    const char *patch = "World";
    written = fs_write(fd, sb, (uint32_t)file_ino, 7, patch, 5);
    assert(written == 5);
    nread = fs_read(fd, sb, (uint32_t)file_ino, 0, read_buf, msg_len);
    assert(nread == (ssize_t)msg_len);
    assert(memcmp(read_buf, "Hello, World!stem!", msg_len) == 0);
    printf("OK: partial overwrite within single block\n");
    /* ---------------------------------------------------------------- */
    /* Test 4: write crossing a block boundary */
    /* File currently ends at offset msg_len. Write so it crosses block 0→1 */
    /* First, extend to block boundary - 50 bytes */
    uint32_t pre_boundary = BLOCK_SIZE - 50;
    char fill_buf[BLOCK_SIZE];
    memset(fill_buf, 'A', pre_boundary);
    /* Overwrite from offset 0 with BLOCK_SIZE-50 bytes of 'A' */
    /* Then write 100 bytes of 'B' crossing the boundary */
    written = fs_write(fd, sb, (uint32_t)file_ino, 0, fill_buf, pre_boundary);
    assert(written == (ssize_t)pre_boundary);
    char cross_buf[100];
    memset(cross_buf, 'B', 100);
    written = fs_write(fd, sb, (uint32_t)file_ino,
                       pre_boundary, cross_buf, 100);
```

```
    assert(written == 100);

    /* Verify: 50 bytes of 'B' in block 0 tail, 50 bytes of 'B' in block 1 head */

    char verify[100];

    nread = fs_read(fd, sb, (uint32_t)file_ino, pre_boundary, verify, 100);

    assert(nread == 100);

    for (int i = 0; i < 100; i++) {

        assert(verify[i] == 'B');

    }

    /* Verify pre-boundary bytes are still 'A' */

    char check_a[10];

    nread = fs_read(fd, sb, (uint32_t)file_ino,

                    pre_boundary - 10, check_a, 10);

    assert(nread == 10);

    for (int i = 0; i < 10; i++) {

        assert(check_a[i] == 'A');

    }

    printf("OK: write crossing block boundary (block 0 → block 1)\n");

    /* ---------------------------------------------------------------- */

    /* Test 5: sparse file — write at high offset */

    int sparse_ino = fs_create_file(fd, sb, root_ino, "sparse.txt",

                                    0644, 1000, 1000);

    assert(sparse_ino > 0);

    char sparse_data[4] = {'S', 'P', 'A', 'R'};

    uint64_t sparse_offset = (uint64_t)BLOCK_SIZE * 100;  /* well into direct range */

    written = fs_write(fd, sb, (uint32_t)sparse_ino,

                       sparse_offset, sparse_data, 4);

    assert(written == 4);

    /* Read the hole (offset 0) — must be zeros */

    char hole_buf[BLOCK_SIZE];

    nread = fs_read(fd, sb, (uint32_t)sparse_ino, 0, hole_buf, BLOCK_SIZE);

    assert(nread == BLOCK_SIZE);
```

```
    for (int i = 0; i < BLOCK_SIZE; i++) {
        assert(hole_buf[i] == 0);
    }
    /* Read the written data */
    char sparse_read[4];
    nread = fs_read(fd, sb, (uint32_t)sparse_ino,
                    sparse_offset, sparse_read, 4);
    assert(nread == 4);
    assert(memcmp(sparse_read, sparse_data, 4) == 0);
    /* Verify inode size but minimal block allocation */
    assert(read_inode(fd, sb, (uint32_t)sparse_ino, &inode) == 0);
    assert(inode.size == sparse_offset + 4);
    printf("OK: sparse file — hole reads as zeros, data reads correctly\n");
    /* ------------------------------------------------------------------ */
    /* Test 6: truncate shrink — reduce to half */
    int trunc_ino = fs_create_file(fd, sb, root_ino, "trunc.txt",
                                   0644, 1000, 1000);
    assert(trunc_ino > 0);
    char trunc_data[BLOCK_SIZE * 3];
    memset(trunc_data, 'T', sizeof(trunc_data));
    written = fs_write(fd, sb, (uint32_t)trunc_ino,
                       0, trunc_data, sizeof(trunc_data));
    assert(written == (ssize_t)sizeof(trunc_data));
    uint32_t free_before = sb->free_blocks;
    int tr = fs_truncate(fd, sb, (uint32_t)trunc_ino, BLOCK_SIZE);
    assert(tr == 0);
    assert(read_inode(fd, sb, (uint32_t)trunc_ino, &inode) == 0);
    assert(inode.size == BLOCK_SIZE);
    /* Two blocks should have been freed */
    assert(sb->free_blocks == free_before + 2);
    /* Verify truncated data: first block still readable */
```

```
    char trunc_read[BLOCK_SIZE];
    nread = fs_read(fd, sb, (uint32_t)trunc_ino, 0, trunc_read, BLOCK_SIZE);
    assert(nread == BLOCK_SIZE);
    for (int i = 0; i < BLOCK_SIZE; i++) {
        assert(trunc_read[i] == 'T');
    }
    /* Verify reading past new EOF returns 0 */
    nread = fs_read(fd, sb, (uint32_t)trunc_ino, BLOCK_SIZE, trunc_read, 1);
    assert(nread == 0);
    printf("OK: truncate shrink — size updated, blocks freed, data preserved\n");
    /* ---------------------------------------------------------------- */
    /* Test 7: truncate extend (sparse extend) */
    int ext_ino = fs_create_file(fd, sb, root_ino, "ext.txt",
                                 0644, 1000, 1000);
    assert(ext_ino > 0);
    char small_data[10];
    memset(small_data, 'E', 10);
    fs_write(fd, sb, (uint32_t)ext_ino, 0, small_data, 10);
    uint32_t free_before_ext = sb->free_blocks;
    tr = fs_truncate(fd, sb, (uint32_t)ext_ino, BLOCK_SIZE * 5);
    assert(tr == 0);
    assert(read_inode(fd, sb, (uint32_t)ext_ino, &inode) == 0);
    assert(inode.size == BLOCK_SIZE * 5);
    /* No new blocks should have been allocated (extension is sparse) */
    assert(sb->free_blocks == free_before_ext);
    /* Reads in extended region should return zeros */
    char ext_read[64];
    nread = fs_read(fd, sb, (uint32_t)ext_ino, BLOCK_SIZE, ext_read, 64);
    assert(nread == 64);
    for (int i = 0; i < 64; i++) assert(ext_read[i] == 0);
    printf("OK: truncate extend — sparse, no allocation, reads as zeros\n");
```

```c
    /* ---------------------------------------------------------------- */

    /* Test 8: append */

    int app_ino = fs_create_file(fd, sb, root_ino, "append.txt",

                                 0644, 1000, 1000);

    assert(app_ino > 0);

    for (int i = 0; i < 10; i++) {

        char line[32];

        int line_len = snprintf(line, sizeof(line), "Line %d\n", i);

        ssize_t wr = fs_append(fd, sb, (uint32_t)app_ino, line, (size_t)line_len);

        assert(wr == line_len);

    }

    assert(read_inode(fd, sb, (uint32_t)app_ino, &inode) == 0);

    assert(inode.size > 0);

    printf("OK: append — 10 lines written sequentially\n");

    /* ---------------------------------------------------------------- */

    /* Test 9: read past EOF returns truncated data */

    nread = fs_read(fd, sb, (uint32_t)file_ino,

                    inode.size + 1000, read_buf, 100);

    assert(nread == 0);

    printf("OK: read past EOF returns 0\n");

    /* ---------------------------------------------------------------- */

    /* Test 10: mtime/ctime updated after write */

    int ts_ino = fs_create_file(fd, sb, root_ino, "ts.txt",

                                0644, 1000, 1000);

    assert(ts_ino > 0);

    assert(read_inode(fd, sb, (uint32_t)ts_ino, &inode) == 0);

    uint32_t create_mtime = inode.mtime;

    /* Sleep briefly to ensure timestamp difference is detectable */

    /* In a real test, you'd mock time; here we just verify it's non-zero */

    fs_write(fd, sb, (uint32_t)ts_ino, 0, "X", 1);

    assert(read_inode(fd, sb, (uint32_t)ts_ino, &inode) == 0);
```

```
    assert(inode.mtime >= create_mtime);
    assert(inode.ctime >= create_mtime);
    assert(inode.size == 1);
    printf("OK: mtime/ctime updated after write\n");
    /* ---------------------------------------------------------------- */
    /* Test 11: truncate zeroes tail of partial block */
    int zero_ino = fs_create_file(fd, sb, root_ino, "zero.txt",
                                  0644, 1000, 1000);
    assert(zero_ino > 0);
    char full_block[BLOCK_SIZE];
    memset(full_block, 0xFF, BLOCK_SIZE);
    fs_write(fd, sb, (uint32_t)zero_ino, 0, full_block, BLOCK_SIZE);
    /* Truncate to 512 bytes — tail of block should be zeroed */
    fs_truncate(fd, sb, (uint32_t)zero_ino, 512);
    /* Extend back to BLOCK_SIZE — bytes 512..4095 must be zero, not 0xFF */
    fs_truncate(fd, sb, (uint32_t)zero_ino, BLOCK_SIZE);
    char after_extend[BLOCK_SIZE];
    fs_read(fd, sb, (uint32_t)zero_ino, 0, after_extend, BLOCK_SIZE);
    for (int i = 0; i < 512; i++) assert(after_extend[i] == 0xFF);
    for (int i = 512; i < BLOCK_SIZE; i++) assert(after_extend[i] == 0x00);
    printf("OK: truncate zeroes tail of partial block\n");
    printf("=== All File I/O Tests Passed ===\n\n");
}
```

Knowledge Cascade: One Write, Ten Worlds

You have just built the operation that every storage system in existence is trying to optimize. The read-modify-write cycle you implemented is the fundamental bottleneck of block storage. Every major pattern in storage systems engineering is a response to this single fact.

Database Write Amplification and LSM-Trees

[[EXPLAIN:lsm-tree-—-log-structured-merge-tree-batch-random-writes-into-sequential-io|LSM-tree — Log-Structured Merge Tree, converts random writes into sequential I/O by batching in a memtable]] Your `fs_write` at a random offset costs: 1 block read + 1 block write = 2 disk operations per write, regardless of how many bytes you actually changed. For a database updating 8-byte

rows randomly across a 10GB table, every row update triggers this 2-operation cycle for a 4096-byte block — writing ~0.2% of the data you read. This is *write amplification*: you write more to disk than your actual payload requires. The consequence is profound: a spinning disk doing 100 random IOPS can sustain 100 random row updates per second — terrible for any write-heavy workload. LevelDB and RocksDB exist as the answer to this problem. The LSM-tree (Log-Structured Merge Tree) collects all writes in a **memtable** (an in-memory sorted structure), then writes the memtable to disk as a **sequential SSTable** (Sorted String Table) when it fills. Random writes become sequential writes. Your random `fs_write` at arbitrary offsets becomes a sequential append to the SSTable. The "merge" in LSM-Tree is the background process that consolidates multiple SSTables into one, handling the complexity that random access has been deferred to. Understanding your `fs_write` and its RMW cost is understanding *why the LSM-tree is the right data structure for write-heavy workloads*. It is not a clever trick; it is the logical response to a physical constraint you now understand from first principles.

SSD Write Amplification: Compound Costs

Your filesystem's write amplification sits on top of the SSD's own write amplification, and the two multiply. [[EXPLAIN:nand-flash-write-constraints-—-write-in-pages-erase-in-larger-blocks-requiring-flash-translation-layer|NAND flash write constraints — NAND cells write in pages (~~4-16KB) and erase in much larger blocks (~~128-512KB), requiring a Flash Translation Layer]] When your `fs_write` triggers a 4KB block write, the SSD receives a 4KB write request to a logical block address (LBA). Internally, NAND flash cannot overwrite: it must erase before writing. Erase operations work on "erase blocks" of 128KB–1MB. If the SSD's target page is not in an empty erase block, the SSD must:

1. Read the entire erase block (128KB) into an internal buffer
2. Erase the erase block (an expensive ~1ms operation)
3. Write the modified page back plus all the other pages that were in that erase block A 4KB write from your filesystem may cause 128KB of physical NAND writes internally. On top of your filesystem's own amplification (writing 4096 bytes to update 100 bytes), you now have the SSD amplifying that 4096-byte write by 32× internally. A 100-byte application write → 4096-byte filesystem block → 128KB NAND write: 1,280× total amplification. This is why SSDs use large internal caches (DRAM or SLC NAND) to buffer writes, why SSD firmware includes sophisticated write scheduling, why `fstrim` exists (to inform the SSD which blocks are logically free so it can garbage-collect proactively), and why "write endurance" (total bytes written before NAND cell failure) is a key SSD specification. You are not just writing to your filesystem; you are writing to a physical system with its own internal indirection layer.

The Linux Page Cache: Why Your Writes Aren't That Slow

If your `fs_write` caused an actual disk RMW on every call, interactive applications would be unusable. A text editor writing one character at a time to a file would need 200µs per character — 50 characters per second maximum throughput. That's slower than a 1970s teletype. The Linux page cache (also called the buffer cache in older literature) is the mechanism that makes this bearable. Every block your filesystem reads or writes goes through the page cache:

- **Read**: if the 4KB block is in the cache (a "cache hit"), return from memory (~~100ns). If not (a "cache miss"), read from disk~~ (100µs), store in cache, return.
- **Write**: store in cache (mark page "dirty"), return immediately. The kernel writes dirty pages to disk asynchronously in the background (via `pdflush` / `kworker` threads), coalescing multiple writes to the same block into a single disk write. For your `fs_write` calling into a test, the "write" to disk is actually a write to the kernel's page cache, which returns at memory speed. The RMW cycle becomes: read block from page cache (100ns if hot), modify in memory, mark dirty. No disk I/O unless the page cache is under memory pressure or `fsync` is called. Understanding why the page cache exists requires understanding the RMW cycle. The page cache is precisely the buffer that transforms "every write is an RMW" into "every write is a memory operation, with asynchronous disk flushing when convenient." Without the page cache, your text editor, web browser, and every other interactive application that writes small amounts of data would be unusable.

> 🔭 **Deep Dive**: Robert Love's "Linux Kernel Development" (3rd ed.), Chapter 16 covers the page cache and writeback mechanism in detail. The `struct address_space` and `struct page` types are the kernel data structures that implement what you are experiencing as fast writes. For the specific writeback mechanism: Chapter 16.2 "Dirty Page Writeback."

Sparse Files and Cloud Thin Provisioning

Every major cloud provider's block storage product (AWS EBS, Google Persistent Disk, Azure Managed Disks) is implemented using the sparse file principle you just built. When you provision a 1TB EBS volume in AWS, AWS does not immediately allocate 1TB of physical storage. It creates a metadata record saying "this volume is 1TB" and allocates physical storage only as you write to it. This is called **thin provisioning**. A VM disk image file on ext4 or XFS on the hypervisor host is a sparse file. The QCOW2 format (used by QEMU/KVM) implements its own internal sparse allocation, on top of the host filesystem's sparse support, giving two levels of sparseness. A container using an overlay filesystem has its base layer as a sparse-ish collection of files, with only the written-to files consuming real storage in the upper layer. The economic implication: cloud providers can "oversell" storage capacity, knowing that users who provision 1TB typically write far less than 1TB. The system works because sparse writes are free until the data is actually written. When a hypervisor host runs out of physical storage because too many VMs simultaneously started writing heavily, that is a "storage balloon" event — and your `alloc_block` returning `-ENOSPC` is the filesystem-level symptom that applications see.

Copy-on-Write Filesystems: Avoiding RMW Entirely

Your `fs_write` modifies blocks in place: read the existing block, modify bytes, write it back to the same location. This is called **update-in-place** (or write-in-place). ZFS and Btrfs take a radically different approach: **copy-on-write (CoW)**. [[EXPLAIN:copy-on-write-filesystems-—-modified-blocks-written-to-new-locations-with-atomic-pointer-updates|Copy-on-write filesystems — modified blocks written to new locations and pointer updated atomically, never overwriting existing data]] In a CoW filesystem, when you write to a block:

1. Allocate a *new* disk block
2. Write the modified content to the new block
3. Update the inode's block pointer to point to the new block
4. Mark the old block as free The old block is never overwritten. The write goes to a new location. The critical insight: step 3 (updating the inode pointer) is a small, atomic metadata write. If the system crashes between steps 2 and 3, the old block is still intact (the inode still points to it). There is no torn-write state: either the old content is visible (pointer not yet updated) or the new content is visible (pointer updated). The intermediate state — half old, half new — is impossible. This eliminates the crash consistency problem that Milestone 6 solves with journaling. ZFS and Btrfs never need `fsck`. The entire filesystem tree is a tree of immutable blocks with pointer updates propagated to the root. The root block's pointer update is the single atomic commit that makes a write permanent. The downside: CoW creates fragmentation aggressively, since no write ever goes to the same location twice. Solid-state drives handle this better than spinning disks (which prefer sequential access), but even SSDs degrade with severe fragmentation. Btrfs requires periodic `btrfs balance` runs to defragment; ZFS handles it with its pool-level free space management. Understanding your in-place `fs_write` is understanding why CoW is a different tradeoff: you trade simpler crash recovery for higher fragmentation and more complex space management.

The O_DIRECT Flag: Bypassing the Buffer Cache

[[EXPLAIN:direct-i/o-o_direct-—-bypassing-the-kernel-page-cache-for-user-controlled-buffering|Direct I/O (O_DIRECT) — bypassing the kernel page cache,

requiring aligned buffers, used by databases for self-managed caching]] When you open a file with `O_DIRECT`, you instruct the kernel to bypass the page cache entirely. Your reads and writes go directly to the block device. The constraints: buffer addresses and lengths must be block-aligned (typically 512-byte or 4096-byte aligned). Who uses `O_DIRECT`? Databases. MySQL InnoDB, PostgreSQL (with `synchronous_commit = on`), Oracle — all optionally use O_DIRECT to manage their own buffer pools instead of relying on the OS page cache. Why? Because the database knows its access patterns better than the OS. A sequential full table scan should not pollute the page cache with pages that will never be read again, evicting hot index pages. The database's buffer pool implements its own LRU/clock eviction policy tuned to database access patterns. Your `fs_write` calls the kernel's `write()` syscall, which goes through the page cache. If you were building a database storage engine directly on your filesystem (not through a POSIX file), you might want `O_DIRECT` semantics to avoid double-buffering (your database buffer pool + the OS page cache both holding the same data). Understanding block-level I/O is a prerequisite for understanding why `O_DIRECT` exists and when to use it.

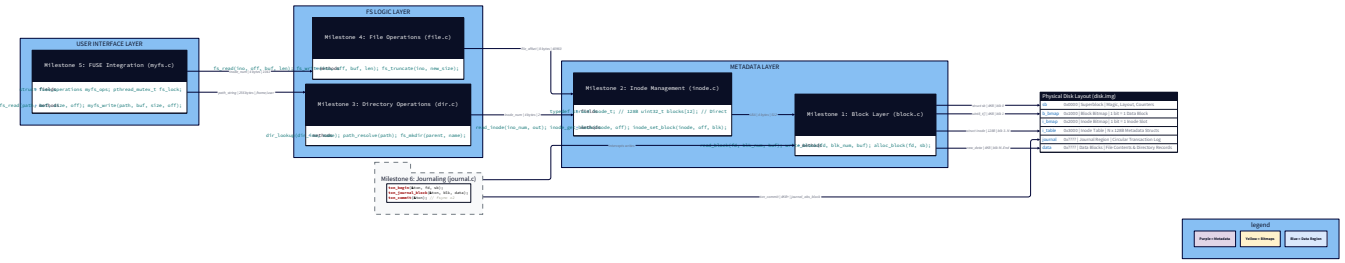
Before You Move On: Pitfall Checklist

- [ ] **Partial block writes zero-fill new blocks**: When `alloc_block` returns a new block and the write is partial, call `memset(block_buf, 0, BLOCK_SIZE)` before writing the new content. A fresh block from the disk image may contain garbage bytes from previous filesystem operations. If you write "Hello" at offset 904 of a new block without zeroing first, bytes 0–903 and 908–4095 contain undefined data from whatever previously occupied that disk location.
- [ ] **RMW only when block exists**: Only call `read_block` in the write path if `disk_block != 0` and the write is partial. If the block was just allocated (`disk_block` was 0 and you called `alloc_block`), skip the read — you already zeroed the buffer with `memset`. Calling `read_block` on a freshly allocated block reads uninitialized disk content, defeating the zero-fill.
- [ ] **Update inode size only on growth**: `inode.size = max(inode.size, offset + length)`. Writing at offset 0 on a 100-byte file with 5 bytes must not shrink the file to 5 bytes. The write path never shrinks the size — only `fs_truncate` does.
- [ ] **Truncate partial-block zeroing**: When truncating to a size that is not block-aligned, zero bytes from `new_size % BLOCK_SIZE` to `BLOCK_SIZE - 1` in the last retained block. Skipping this leaks previous file content into subsequent truncate-extend cycles — a security bug in multi-user systems.
- [ ] **Free indirect blocks during truncate**: When truncating below the single-indirect threshold (48KB), free the single-indirect block itself after nulling its data pointers. When truncating below the double-indirect threshold, free all single-indirect blocks within the double-indirect and then the double-indirect block. Each freed indirect block is 4KB of disk space that `sb->free_blocks` must account for.
- [ ] **Block allocation failure rollback**: If `alloc_block` returns `-ENOSPC` mid-write, you have already written some blocks. The partial write has occurred. Return the bytes actually written (not an error) if any data was written, or `-ENOSPC` if nothing

was written. POSIX specifies that a partial write is legitimate. Do not leave the inode in an inconsistent state (update `size` and timestamps to reflect what was actually written).

- [ ] **`inode_set_block` for newly allocated blocks**: After calling `alloc_block` to get a new disk block number, always call `inode_set_block(fd, sb, &inode, file_offset, disk_block)` to record the pointer before writing data to the block. If you write data first and then crash before recording the pointer, the data is unreachable (leaked block). The order: allocate → record pointer → write data.
- [ ] **`atime` update on reads**: `fs_read` must update `inode.atime` and call `write_inode`. This causes a write for every read — the "atime write amplification" problem. For FUSE (Milestone 5), implement `relatime`: only update `atime` if `atime < mtime`. This reduces the write-per-read to cases where the file has been modified since last access — typically once per read session rather than once per read call.
- [ ] **EOF clamp in `fs_read`**: If `offset >= inode.size`, return 0 immediately (not an error). If `offset + length > inode.size`, clamp `length = inode.size - offset`. Reading past EOF is not an error in POSIX; it returns 0 bytes. An unclamped read would access blocks beyond the file's logical size, potentially reading another file's data if those blocks happen to be allocated.
- [ ] **Block 0 is never a data block**: The `alloc_block` function should never return 0 (the superblock's disk block number). If `disk_block == 0` after `alloc_block`, something is catastrophically wrong. Add an assertion: `assert(disk_block >= sb->data_block_start)` after every `alloc_block` call. Similarly, in `inode_get_block`, a returned `disk_block == 0` means "sparse hole" — not a pointer to the superblock. The convention that 0 is the null/sparse sentinel is what makes this work, and it requires that legitimate data blocks always have numbers ≥ `data_block_start`.

What You've Built and What Comes Next



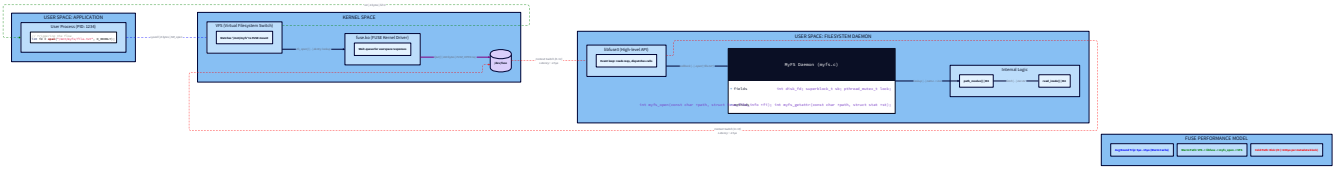
You now have a complete, working filesystem — in the sense that it can create files, name them in directories, store arbitrary byte content in them via a multi-level block pointer tree, retrieve that content by inode number and byte offset, grow and shrink files through write and truncate, and support sparse files where holes cost nothing. But there is a gap between "working filesystem library" and "a filesystem you can actually use." Right now, you can only interact with your filesystem by calling `fs_read`, `fs_write`, `fs_create_file` directly from C code. You cannot use standard Unix tools — `ls`, `cat`, `cp`, `vim`, `grep` — against your filesystem. You cannot mount it. It is a library, not a filesystem in the Unix sense. Milestone 5 closes that gap via FUSE (Filesystem in Userspace). FUSE provides a mechanism for running a filesystem in

userspace (your process) while the kernel routes VFS calls from any application to your callbacks. You will implement a thin translation layer that maps FUSE callback arguments (path strings, POSIX stat structs, file handle integers) to your internal operations (path resolution, `fs_read`, `fs_write`, `inode_t`). After Milestone 5, you will be able to run `mount -t fuse ./myfs /mnt/myfs` and then `cp -r ~/Documents /mnt/myfs/` and have your filesystem actually store those files. The block-level I/O you just built is the foundation. Every FUSE callback for every file operation — `read`, `write`, `create`, `truncate`, `getattr` — will ultimately call the functions in this milestone. The work you have done here is the engine; FUSE is the transmission that connects it to the outside world.

Milestone 5: FUSE Integration

The Wall That Was Never There

You have built a complete filesystem. It can format a raw disk image, allocate blocks and inodes, store file data across a multi-level block pointer tree, create and navigate a directory hierarchy, and read or write arbitrary byte ranges with correct sparse-file semantics. Every function you have written is correct. Every test passes. And yet: if you open a terminal right now and type `ls /mnt/myfs`, nothing happens. Your filesystem does not exist as far as the operating system is concerned. `cat`, `cp`, `grep`, `vim` — none of these tools can see a single byte you have stored. You have built a library, not a filesystem. Most developers hit this point and assume the solution requires kernel programming. Surely, they think, to make the OS route `open("/mnt/myfs/file.txt")` to your code, you need to write a kernel module — modify the VFS layer, compile against kernel headers, deal with `MODULE_LICENSE("GPL")`, and risk a kernel panic every time you make a mistake. The VFS feels like a wall between userspace code and the rest of the OS. That wall does not exist. **FUSE** (Filesystem in USErspace) is a kernel module — already present in every modern Linux system — that acts as a relay. When any process on the system makes a VFS call (`open`, `stat`, `read`, `write`, `mkdir`) against a FUSE-mounted path, the kernel's VFS dispatches that call to the FUSE kernel module, which serializes it into a message, writes it to `/dev/fuse`, and blocks waiting for a response. Your userspace process is reading from that same `/dev/fuse` file descriptor. You receive the request, call your internal filesystem functions, serialize the result, and write it back. The FUSE module receives the response, unpacks it, and returns it to the calling process — which has no idea this round-trip through userspace ever happened. [[EXPLAIN:fuse-architecture-—-how-userspace-filesystems-intercept-vfs-calls|FUSE architecture — how userspace filesystems intercept VFS calls]] The latency cost of this relay is real: approximately 2–5 microseconds per operation, dominated by two context switches (kernel→user for dispatch, user→kernel for response). For metadata-heavy workloads like `find` or `git status`, this adds up. For data-heavy workloads like streaming a video file, the per-byte overhead disappears into the block transfer cost. Production filesystems built on FUSE include GlusterFS, SSHFS, rclone mount, mergerfs, and Google's internal Colossus client. FUSE is not a toy. The real surprise — the one that shapes everything in this milestone — is this: **FUSE calls your code with path strings, not inode numbers**. When `ls /mnt/myfs/home/user` runs, FUSE's `readdir` callback receives the string `"/home/user"`. Before you can call a single one of your internal functions, you must resolve that path to an inode number. The `path_resolve` function you built in Milestone 3 is not just one feature among many — it is the hottest code path in your entire filesystem. Let's wire this up.



The Fundamental Tension: VFS Contract, Userspace Implementation

[[EXPLAIN:vfs-abstraction-layer-—-the-kernel-interface-all-filesystems-must-implement|VFS abstraction layer — the kernel's unified interface that ext4, XFS, NFS, and your filesystem all implement]] The kernel's VFS (Virtual Filesystem Switch) defines a contract: every filesystem, whether it is ext4 built into the kernel, NFS communicating over the network, or your FUSE implementation running as a userspace process, must implement a specific set of operations. In the kernel, these are function pointers grouped into three C structures:

- `struct inode_operations`: operations on inodes (`lookup`, `create`, `mkdir`, `unlink`, `rename`, `getattr`, `chmod`, `utimes`)
- `struct file_operations`: operations on open files (`open`, `read`, `write`, `release`, `readdir`)
- `struct super_operations`: operations on the filesystem itself (`statfs`, `sync_fs`, `put_super`) The FUSE kernel module implements all of these by forwarding them over the `/dev/fuse` channel to your process. `libfuse` — the C library you link against — handles the low-level protocol: reading requests from the file descriptor, dispatching them to function pointers you register, and writing responses back. Your job is to fill in those function pointers with implementations that call your Milestone 1–4 code. The architectural tension is this: **The VFS contract is inode-based. FUSE delivers path-based requests. Your internal functions are inode-based. Every FUSE callback must perform path resolution to bridge the gap.** For a filesystem mounted at `/mnt/myfs`, when a process calls `stat("/mnt/myfs/home/user/file.txt")`:

1. The kernel VFS resolves the mount point, sees it's FUSE-backed
2. FUSE kernel module packages the request: opcode=`GETATTR`, path=`"/home/user/file.txt"`
3. Your process receives it, calls `path_resolve(fd, sb, "/home/user/file.txt", root_ino, root_ino)`
4. That traverses three directory blocks, returns inode 1041
5. You call `read_inode(fd, sb, 1041, &inode)`, convert to a POSIX `struct stat`, and return it
6. FUSE kernel module delivers the `stat` result to the calling process Step 3–4 is 3+ disk reads on a cold cache. For a process that calls `stat` on 10,000 files (like `find` or `git status`), that is 30,000+ cold disk reads just for path resolution. This is why the Linux VFS maintains a **dcache** (directory entry cache): a kernel-level hash table mapping `(parent_inode_number, filename)` to child inode number. FUSE can optionally tell the kernel to cache lookup results, dramatically reducing path resolution costs. We will implement this with entry timeouts. The numbers matter here:

- Cold path resolution (3 disk reads × 100µs per SSD read) = ~300µs per lookup
- With dcache hit: ~1µs (pure kernel memory access)
- With FUSE entry timeout cache: after first lookup, subsequent lookups return cached result for the configured timeout duration This 300× difference between cached and uncached lookup explains why `ls` on a freshly mounted FUSE filesystem feels slow, while the second `ls` feels instant.

Installing libfuse and Project Structure

Before writing a single callback, set up your build environment. On Linux, libfuse 3.x is the current major version and the one we will use:

```bash
# Debian/Ubuntu
sudo apt-get install libfuse3-dev fuse3

# Fedora/RHEL
sudo dnf install fuse3-devel

# Verify
pkg-config --modversion fuse3
```

Your project now has two main entry points: `mkfs.c` (from Milestone 1) and `myfs.c` (the FUSE daemon). A sensible Makefile:

```makefile
CC      = gcc
CFLAGS  = -Wall -Wextra -g -O2 $(shell pkg-config --cflags fuse3)
LDFLAGS = $(shell pkg-config --libs fuse3)
SRCS_COMMON = block.c bitmap.c inode.c dir.c file.c
SRCS_MKFS   = mkfs.c $(SRCS_COMMON)
SRCS_MYFS   = myfs.c $(SRCS_COMMON)
all: mkfs myfs
mkfs: $(SRCS_MKFS)
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)
myfs: $(SRCS_MYFS)
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)
clean:
        rm -f mkfs myfs *.o
```

Notice `FUSE_USE_VERSION` — this macro tells libfuse which API version you target. Always set it before including the FUSE header:

```c
/*
 * myfs.c — FUSE integration layer.
 *
 * MUST define FUSE_USE_VERSION before including fuse.h.
 * 31 targets the libfuse 3.x "high-level" API.
 */
#define FUSE_USE_VERSION 31
#include <fuse.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>
#include "fs.h"   /* your superblock_t, inode_t, all Milestone 1-4 functions */
```

Global State: The Filesystem Context

Your FUSE daemon is a long-running process that holds all filesystem state. The state must be accessible from every FUSE callback, which libfuse delivers on different threads. Package everything into a single context struct:

```c
/*
 * Global filesystem context.
 * Initialized during mount; accessed by all FUSE callbacks.
 *
 * Thread safety: All filesystem operations acquire fs_lock before
 * modifying any shared state. A coarse-grained global mutex is correct
 * for initial implementation; Milestone 6 will introduce per-transaction
 * locking via the journal.
 *
 * MEMORY LAYOUT:
 *   fs_ctx:                  ~512 bytes (struct + sb fields)
 *   fs_ctx.sb:               4096 bytes (superblock, kept in memory)
 *   Lock overhead (pthread): 40 bytes typical
 */
typedef struct {
    int         fd;             /* file descriptor of the disk image */
    superblock_t sb;            /* in-memory copy of the superblock */
    pthread_mutex_t lock;       /* global filesystem mutex */
    char        image_path[256]; /* path to disk image, for fsync on unmount */
} fs_ctx_t;

static fs_ctx_t g_fs;

/*
 * RAII-style lock helpers.
 * Use these in every callback — never call pthread_mutex_lock directly.
 */
static inline void fs_lock(void)   { pthread_mutex_lock(&g_fs.lock); }
static inline void fs_unlock(void) { pthread_mutex_unlock(&g_fs.lock); }
```

Hardware Soul — Cache Line Contention on the Global Lock The `g_fs` struct is accessed by multiple threads simultaneously. The `pthread_mutex_t` at offset `8 + 4096 = 4104` bytes into the struct — this is on a different cache line (64 bytes) than the `fd` and `sb` fields. This matters: when one thread holds the lock and another thread spins waiting, the spinning thread repeatedly reads the mutex state, keeping that cache line hot. If the mutex shared a cache line with `fd` or `sb`, every lock acquisition would invalidate the `fd` cache line in the holder's CPU, causing false sharing. The struct layout above avoids this naturally because `sb` is 4096 bytes and pushes `lock` onto its own cache line. The `fuse_get_context()` function provides per-call information (UID, GID, PID of the calling process). You can use it inside callbacks to implement permission checking:

```c
/* Get the context inside a FUSE callback */

struct fuse_context *ctx = fuse_get_context();

uid_t caller_uid = ctx->uid;

gid_t caller_gid = ctx->gid;
```

The Path Translation Problem: Your Central Design

Every FUSE high-level API callback receives a path string. Your internal functions operate on inode numbers. The translation is `path_resolve` from Milestone 3. But calling `path_resolve` naively in every callback carries a cost: each component lookup is a directory block read. We need a helper that cleanly bridges this gap and handles the common case (getting an inode for a path, then doing something with it):

```c
/*
 * Resolve a path to an inode number.
 * Returns inode number (> 0) on success, negative errno on failure.
 *
 * This is the hottest function in the FUSE layer. Every callback
 * starts here. Profile first if optimizing.
 *
 * Thread safety: Caller must hold fs_lock.
 */
static int resolve_path(const char *path) {
    return path_resolve(g_fs.fd, &g_fs.sb, path,
                        g_fs.sb.root_inode, g_fs.sb.root_inode);
}
/*
 * Resolve a path and read its inode into *out.
 * Returns 0 on success, negative errno on failure.
 *
 * Thread safety: Caller must hold fs_lock.
 */
static int path_to_inode(const char *path, uint32_t *ino_out, inode_t *inode_out) {
    int ino = resolve_path(path);
    if (ino < 0) return ino;
    if (read_inode(g_fs.fd, &g_fs.sb, (uint32_t)ino, inode_out) != 0)
        return -EIO;
    *ino_out = (uint32_t)ino;
    return 0;
}
```

For operations that need both a parent directory and a leaf name (create, mkdir, unlink, rmdir, rename), we need path decomposition:

```c
/*
 * Split "/home/user/file.txt" into parent_path="/home/user" and name="file.txt".
 * Returns 0 on success, -EINVAL if path has no parent (root).
 *
 * Writes into caller-supplied buffers. parent_buf must be PATH_MAX bytes.
 * name_out points into path (no allocation needed) — do not modify.
 *
 * Thread safety: No shared state; safe to call without lock.
 */
static int split_path(const char *path,
                      char *parent_buf, size_t parent_buf_size,
                      const char **name_out) {
    const char *last_slash = strrchr(path, '/');
    if (last_slash == NULL) return -EINVAL;
    *name_out = last_slash + 1;
    if (**name_out == '\0') return -EINVAL;   /* trailing slash */
    size_t parent_len = (size_t)(last_slash - path);
    if (parent_len == 0) {
        /* parent is root: path = "/filename" */
        if (parent_buf_size < 2) return -ENAMETOOLONG;
        parent_buf[0] = '/';
        parent_buf[1] = '\0';
    } else {
        if (parent_len >= parent_buf_size) return -ENAMETOOLONG;
        memcpy(parent_buf, path, parent_len);
        parent_buf[parent_len] = '\0';
    }
    return 0;
}
```

Mapping Inodes to `struct stat`: The `inode_to_stat` Bridge

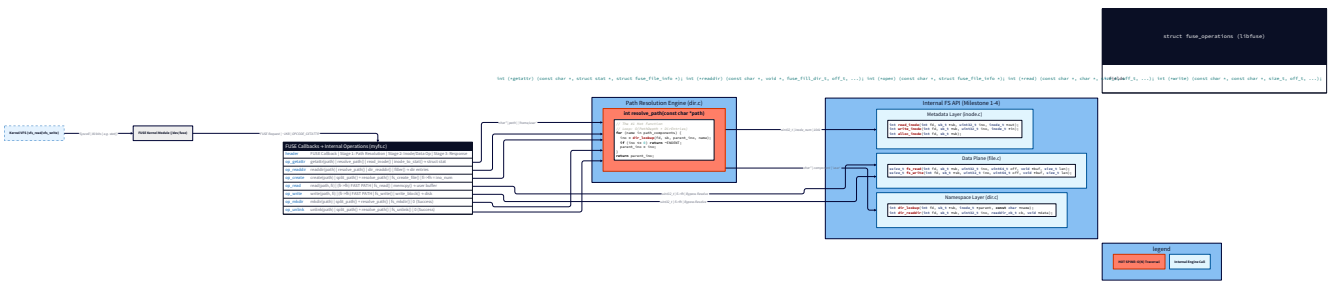
Every FUSE callback that returns file metadata — `getattr`, `lookup`, `readdir` — must fill in a POSIX `struct stat`. This struct is what `ls -la`, `stat`, and `fstat` return to applications. Getting it right is critical: `ls` calls `getattr` on every path, and if `getattr` returns wrong values, nothing works.

```c
/*
 * Convert an in-memory inode_t to a POSIX struct stat.
 *
 * struct stat field mapping:
 *   st_ino     ← inode number
 *   st_mode    ← inode.mode (already in POSIX format: S_IFDIR | 0755, etc.)
 *   st_nlink   ← inode.nlinks
 *   st_uid     ← inode.uid
 *   st_gid     ← inode.gid
 *   st_size    ← inode.size (logical file size in bytes)
 *   st_blocks  ← computed: disk blocks used × 512-byte units (POSIX convention)
 *   st_blksize ← 4096 (preferred I/O block size hint to applications)
 *   st_atime   ← inode.atime
 *   st_mtime   ← inode.mtime
 *   st_ctime   ← inode.ctime
 *
 * Note: st_blocks is in 512-byte units by POSIX convention, even though
 * our actual block size is 4096. Multiply logical block count by 8
 * (4096/512 = 8) to get the correct st_blocks value.
 */
static void inode_to_stat(const inode_t *inode, uint32_t ino_num,
                          struct stat *st) {
    memset(st, 0, sizeof(*st));
    st->st_ino     = ino_num;
    st->st_mode    = inode->mode;
    st->st_nlink   = inode->nlinks;
    st->st_uid     = inode->uid;
    st->st_gid     = inode->gid;
    st->st_size    = inode->size;
    st->st_blksize = BLOCK_SIZE;
    /*
```

```
     * Compute number of 512-byte blocks actually allocated on disk.
     * For sparse files: count only non-null direct block pointers.
     * For a full count including indirect blocks, we'd need to traverse
     * the pointer tree. For simplicity, approximate from size.
     */
    st->st_blocks  = ((off_t)inode->size + 511) / 512;
    /* Timestamps: struct stat uses struct timespec for nanosecond precision.
     * Our inode stores only second-precision Unix timestamps.
     * The .tv_nsec field is 0 — acceptable for our implementation. */
    st->st_atim.tv_sec  = inode->atime;
    st->st_atim.tv_nsec = 0;
    st->st_mtim.tv_sec  = inode->mtime;
    st->st_mtim.tv_nsec = 0;
    st->st_ctim.tv_sec  = inode->ctime;
    st->st_ctim.tv_nsec = 0;
}
```

> **The `getattr` Pitfall**: `getattr` is the first callback FUSE invokes for virtually every operation. Before `ls` can list directory contents, it calls `getattr` on the directory itself. Before `open` can open a file, the kernel calls `getattr`. Before `cp` can copy, `getattr` on both source and destination. If your `getattr` implementation is wrong or slow, nothing else in the system works. Test it first, test it exhaustively, test it before anything else.

Implementing the Core Callbacks



`getattr` — The Most Called Callback

```c
/*
 * getattr — Get file attributes.
 *
 * Called by: stat(), lstat(), fstat(), open(), ls, find, cp, mv — everything.
 * This is the hottest callback in the system.
 *
 * FUSE high-level API signature:
 *   path:   absolute path within the mount (e.g., "/home/user/file.txt")
 *   st:     struct stat to fill in
 *   fi:     file info (may be NULL for path-based calls; non-NULL for fd-based)
 *
 * Returns 0 on success, negative errno on failure.
 */
static int myfs_getattr(const char *path, struct stat *st,
                        struct fuse_file_info *fi) {
    (void)fi;   /* unused in our implementation */
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) {
        fs_unlock();
        return r;   /* -ENOENT, -EIO, etc. */
    }
    inode_to_stat(&inode, ino, st);
    fs_unlock();
    return 0;
}
```

`readdir` — Directory Listing

`readdir` is called when any program lists directory contents: `ls`, `find`, `opendir`/`readdir` loops, `glob`. libfuse's high-level API provides a `filler` function that you call for each entry:

```c
/*
 * readdir — List directory contents.
 *
 * Called by: ls, find, opendir/readdir, glob expansion.
 *
 * For each entry, call filler(buf, name, stat_or_NULL, offset, fill_flags).
 * Returning stat is optional (NULL means FUSE will call getattr separately).
 * Providing stat avoids an extra getattr round-trip — do it for performance.
 *
 * We always output '.' and '..' first, as POSIX requires.
 */
typedef struct {
    void *buf;
    fuse_fill_dir_t filler;
    int error;
    int fd;
    superblock_t *sb;
} readdir_ctx_t;
static void readdir_emit(const char *name, uint32_t inode_num,
                         uint8_t file_type, void *userdata) {
    readdir_ctx_t *ctx = (readdir_ctx_t *)userdata;
    if (ctx->error != 0) return;  /* short-circuit on previous error */
    struct stat st;
    memset(&st, 0, sizeof(st));
    /* Read the inode to fill stat — costs a disk read per entry.
     * For large directories this adds up; a future optimization would
     * cache inode reads here. */
    inode_t inode;
    if (read_inode(ctx->fd, ctx->sb, inode_num, &inode) == 0) {
        inode_to_stat(&inode, inode_num, &st);
    } else {
```

```
        /* Populate minimal stat from directory entry type hint */
        st.st_ino = inode_num;
        st.st_mode = (file_type == FT_DIR) ? S_IFDIR | 0755 : S_IFREG | 0644;
    }
    int r = ctx->filler(ctx->buf, name, &st, 0, FUSE_FILL_DIR_PLUS);
    if (r != 0) {
        ctx->error = -ENOMEM;  /* filler returns non-zero when buffer is full */
    }
}
static int myfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                        off_t offset, struct fuse_file_info *fi,
                        enum fuse_readdir_flags flags) {
    (void)offset; (void)fi; (void)flags;
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) { fs_unlock(); return r; }
    if (!S_ISDIR(inode.mode)) { fs_unlock(); return -ENOTDIR; }
    readdir_ctx_t ctx = {
        .buf    = buf,
        .filler = filler,
        .error  = 0,
        .fd     = g_fs.fd,
        .sb     = &g_fs.sb,
    };
    /* dir_readdir calls readdir_emit for each entry (including '.' and '..') */
    dir_readdir(g_fs.fd, &g_fs.sb, ino, readdir_emit, &ctx);
    fs_unlock();
    return ctx.error;
}
```

`open` and `release` — File Handle Lifecycle

In libfuse's high-level API, `open` does not need to allocate OS-level resources — libfuse handles that. Your `open` implementation validates that the file exists and is accessible. The `fi->fh` field (file handle) lets you store a per-open-file opaque value — we store the inode number there for fast access in subsequent `read`/`write` calls:

```c
/*
 * open — Open a file.
 *
 * Called when any process calls open(2), fopen(3), etc.
 * fi->flags contains the open flags (O_RDONLY, O_WRONLY, O_RDWR, O_TRUNC, etc.)
 *
 * We validate existence and permissions here.
 * We store the inode number in fi->fh to avoid re-resolving the path on every read/write.
 */
static int myfs_open(const char *path, struct fuse_file_info *fi) {
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) { fs_unlock(); return r; }
    if (S_ISDIR(inode.mode)) { fs_unlock(); return -EISDIR; }
    /* Handle O_TRUNC flag: truncate file to zero on open */
    if (fi->flags & O_TRUNC) {
        r = fs_truncate(g_fs.fd, &g_fs.sb, ino, 0);
        if (r != 0) { fs_unlock(); return r; }
    }
    /* Store inode number in file handle for fast access later */
    fi->fh = ino;
    /*
     * Set direct_io flag to bypass FUSE's page cache for this file.
     * Without this, FUSE caches file data in the kernel's page cache,
     * which can serve stale reads if another process modifies the file
     * directly. For our implementation, disable kernel-side page caching
     * to ensure reads always go through our fs_read() function.
     *
     * Trade-off: disabling the kernel page cache means every read causes
```

```
     * a call to our read callback. For read-heavy workloads, enabling the
     * cache (removing this line) dramatically improves performance.
     */
    fi->direct_io = 1;
    fs_unlock();
    return 0;
}
/*
 * release — Called when the last reference to an open file is closed.
 *
 * Note: "release" in FUSE means "the last open file descriptor is closed."
 * This is NOT called for every close(2) — only when the kernel's reference
 * count for this file drops to zero. Multiple opens of the same file by the
 * same process may result in only one release.
 *
 * Here: flush any pending writes, clean up per-file state.
 */
static int myfs_release(const char *path, struct fuse_file_info *fi) {
    (void)path;
    fs_lock();
    /* fsync the disk image to ensure all writes are durable.
     * In production, you would do this selectively (only if file was written).
     * fi->fh holds the inode number we stored in open(). */
    fsync(g_fs.fd);
    (void)fi;
    fs_unlock();
    return 0;
}
```

> **`open` vs `create` in FUSE**: When a process calls `open("path", O_CREAT | O_WRONLY, 0644)`, FUSE will call the `create` callback if you have registered it, or fall back to calling `mknod` followed by `open` if you haven't. You MUST register `create` — if you omit it, `echo "text" > file` will fail because the shell uses `O_CREAT`. The `mknod` fallback does not work correctly for regular files.

`create` — The O_CREAT Path

```c
/*
 * create — Create and open a file atomically.
 *
 * Called when: open(O_CREAT | O_WRONLY), open(O_CREAT | O_RDWR),
 *              creat(2), fopen("w"), echo > file
 *
 * Must create the file AND open it. Store inode number in fi->fh.
 * If the file already exists and O_EXCL is set, return -EEXIST.
 */
static int myfs_create(const char *path, mode_t mode,
                       struct fuse_file_info *fi) {
    char parent_path[PATH_MAX];
    const char *name;
    int r = split_path(path, parent_path, sizeof(parent_path), &name);
    if (r < 0) return r;
    fs_lock();
    /* Resolve parent directory */
    int parent_ino = resolve_path(parent_path);
    if (parent_ino < 0) { fs_unlock(); return parent_ino; }
    /* If O_EXCL: fail if file already exists */
    inode_t parent_inode;
    read_inode(g_fs.fd, &g_fs.sb, (uint32_t)parent_ino, &parent_inode);
    int existing = dir_lookup(g_fs.fd, &g_fs.sb, &parent_inode, name);
    if (existing > 0) {
        if (fi->flags & O_EXCL) { fs_unlock(); return -EEXIST; }
        /* File exists, O_EXCL not set: open existing file */
        fi->fh = (uint64_t)existing;
        if (fi->flags & O_TRUNC)
            fs_truncate(g_fs.fd, &g_fs.sb, (uint32_t)existing, 0);
        fs_unlock();
```

```
        return 0;
    }
    /* Get caller identity for ownership */
    struct fuse_context *ctx = fuse_get_context();
    /* Create the new file */
    int new_ino = fs_create_file(g_fs.fd, &g_fs.sb,
                                 (uint32_t)parent_ino, name,
                                 (uint16_t)(mode & 0777),
                                 (uint16_t)ctx->uid,
                                 (uint16_t)ctx->gid);
    if (new_ino < 0) { fs_unlock(); return new_ino; }
    fi->fh       = (uint64_t)new_ino;
    fi->direct_io = 1;
    /* Persist superblock (free counts changed) */
    uint8_t sb_buf[BLOCK_SIZE];
    memset(sb_buf, 0, BLOCK_SIZE);
    memcpy(sb_buf, &g_fs.sb, sizeof(g_fs.sb));
    write_block(g_fs.fd, 0, sb_buf);
    fs_unlock();
    return 0;
}
```

`read` and `write` — Data Plane Callbacks

These are the most performance-critical callbacks. FUSE delivers `read` requests with a pre-allocated buffer; `write` requests with the data to write. Both use the inode number stored in `fi->fh` from `open` / `create`:

```c
/*
 * read — Read data from an open file.
 *
 * size: maximum bytes to return (FUSE may request up to 128KB at once)
 * offset: byte offset within the file
 * Returns number of bytes actually read (may be less than size at EOF).
 */
static int myfs_read(const char *path, char *buf, size_t size,
                     off_t offset, struct fuse_file_info *fi) {
    (void)path;   /* fi->fh has the inode number from open() */
    if (offset < 0) return -EINVAL;
    fs_lock();
    ssize_t nread = fs_read(g_fs.fd, &g_fs.sb,
                            (uint32_t)fi->fh,
                            (uint64_t)offset, buf, size);
    fs_unlock();
    return (int)nread;
}
/*
 * write — Write data to an open file.
 *
 * Returns number of bytes written (should always equal size — partial
 * writes indicate errors; FUSE will retry). Return negative errno on error.
 */
static int myfs_write(const char *path, const char *buf, size_t size,
                      off_t offset, struct fuse_file_info *fi) {
    (void)path;
    if (offset < 0) return -EINVAL;
    fs_lock();
    ssize_t nwritten = fs_write(g_fs.fd, &g_fs.sb,
                                (uint32_t)fi->fh,
```

```
                                (uint64_t)offset, buf, size);
    if (nwritten > 0) {
        /* Persist superblock (free block count may have changed) */
        uint8_t sb_buf[BLOCK_SIZE];
        memset(sb_buf, 0, BLOCK_SIZE);
        memcpy(sb_buf, &g_fs.sb, sizeof(g_fs.sb));
        write_block(g_fs.fd, 0, sb_buf);
    }
    fs_unlock();
    return (int)nwritten;
}
```

`mkdir` and `rmdir`

```c
static int myfs_mkdir(const char *path, mode_t mode) {
    char parent_path[PATH_MAX];
    const char *name;
    int r = split_path(path, parent_path, sizeof(parent_path), &name);
    if (r < 0) return r;
    fs_lock();
    int parent_ino = resolve_path(parent_path);
    if (parent_ino < 0) { fs_unlock(); return parent_ino; }
    struct fuse_context *ctx = fuse_get_context();
    int new_ino = fs_mkdir(g_fs.fd, &g_fs.sb,
                           (uint32_t)parent_ino, name,
                           (uint16_t)(mode & 0777),
                           (uint16_t)ctx->uid,
                           (uint16_t)ctx->gid);
    if (new_ino < 0) { fs_unlock(); return new_ino; }
    /* Persist superblock */
    uint8_t sb_buf[BLOCK_SIZE];
    memset(sb_buf, 0, BLOCK_SIZE);
    memcpy(sb_buf, &g_fs.sb, sizeof(g_fs.sb));
    write_block(g_fs.fd, 0, sb_buf);
    fs_unlock();
    return 0;
}
static int myfs_rmdir(const char *path) {
    char parent_path[PATH_MAX];
    const char *name;
    int r = split_path(path, parent_path, sizeof(parent_path), &name);
    if (r < 0) return r;
    fs_lock();
    int parent_ino = resolve_path(parent_path);
```

```
    if (parent_ino < 0) { fs_unlock(); return parent_ino; }

    r = fs_rmdir(g_fs.fd, &g_fs.sb, (uint32_t)parent_ino, name);

    fs_unlock();

    return r;

}
```

`unlink` — File Deletion

```c
static int myfs_unlink(const char *path) {

    char parent_path[PATH_MAX];

    const char *name;

    int r = split_path(path, parent_path, sizeof(parent_path), &name);

    if (r < 0) return r;

    fs_lock();

    int parent_ino = resolve_path(parent_path);

    if (parent_ino < 0) { fs_unlock(); return parent_ino; }

    r = fs_unlink(g_fs.fd, &g_fs.sb, (uint32_t)parent_ino, name);

    if (r == 0) {

        uint8_t sb_buf[BLOCK_SIZE];

        memset(sb_buf, 0, BLOCK_SIZE);

        memcpy(sb_buf, &g_fs.sb, sizeof(g_fs.sb));

        write_block(g_fs.fd, 0, sb_buf);

    }

    fs_unlock();

    return r;

}
```

`truncate` — Resize from Path

```c
static int myfs_truncate(const char *path, off_t new_size,
                         struct fuse_file_info *fi) {
    if (new_size < 0) return -EINVAL;
    fs_lock();
    uint32_t ino;
    int r;
    if (fi != NULL) {
        /* fd-based truncate: fi->fh has inode number */
        ino = (uint32_t)fi->fh;
        r = 0;
    } else {
        /* path-based truncate */
        inode_t inode;
        r = path_to_inode(path, &ino, &inode);
    }
    if (r == 0)
        r = fs_truncate(g_fs.fd, &g_fs.sb, ino, (uint64_t)new_size);
    fs_unlock();
    return r;
}
```

`chmod` and `utimens` — Metadata Changes

```c
static int myfs_chmod(const char *path, mode_t mode,
                      struct fuse_file_info *fi) {
    (void)fi;
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) { fs_unlock(); return r; }
    /* Replace permission bits; preserve file type bits */
    inode.mode = (inode.mode & S_IFMT) | (mode & 07777);
    inode.ctime = (uint32_t)time(NULL);
    r = write_inode(g_fs.fd, &g_fs.sb, ino, &inode);
    fs_unlock();
    return r;
}
static int myfs_utimens(const char *path, const struct timespec ts[2],
                        struct fuse_file_info *fi) {
    (void)fi;
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) { fs_unlock(); return r; }
    /* ts[0] = atime, ts[1] = mtime */
    if (ts[0].tv_nsec != UTIME_OMIT)
        inode.atime = (uint32_t)ts[0].tv_sec;
    if (ts[1].tv_nsec != UTIME_OMIT)
        inode.mtime = (uint32_t)ts[1].tv_sec;
    inode.ctime = (uint32_t)time(NULL);
    r = write_inode(g_fs.fd, &g_fs.sb, ino, &inode);
```

```
    fs_unlock();

    return r;

}
```

`rename`: The Hardest Operation

`rename` is the most complex single operation in any filesystem. At the POSIX level, `rename(old, new)` must be **atomic**: any process that observes the filesystem either sees the old name or the new name, never a state where neither exists. This is a crash-consistency requirement even before Milestone 6's journal. The implementation must handle four distinct cases:

1. **Simple rename in the same directory**: `rename("/home/user/a", "/home/user/b")`
2. **Move to different directory**: `rename("/home/user/a", "/tmp/a")`
3. **Replace an existing file**: `rename("/tmp/new", "/tmp/old")` — `old` must be atomically replaced
4. **Replace with a directory**: the target must be an empty directory

```c
/*
 * rename — Move/rename a file or directory.
 *
 * POSIX guarantees: if new_path already exists, it is atomically replaced.
 * Without journaling, we approximate atomicity by careful write ordering:
 *   1. Write new directory entry pointing to the inode
 *   2. Remove old directory entry
 * If we crash between 1 and 2, the file has two names — detectable by fsck.
 * If we crash after 2 but before 1... well, ordering prevents this.
 *
 * flags: RENAME_NOREPLACE (fail if new_path exists), RENAME_EXCHANGE (swap),
 *        etc. We implement only the basic rename (flags == 0).
 */
static int myfs_rename(const char *old_path, const char *new_path,
                       unsigned int flags) {
    if (flags != 0) return -EINVAL;   /* RENAME_NOREPLACE etc. not implemented */
    char old_parent[PATH_MAX], new_parent[PATH_MAX];
    const char *old_name, *new_name;
    int r;
    r = split_path(old_path, old_parent, sizeof(old_parent), &old_name);
    if (r < 0) return r;
    r = split_path(new_path, new_parent, sizeof(new_parent), &new_name);
    if (r < 0) return r;
    fs_lock();
    int old_parent_ino = resolve_path(old_parent);
    if (old_parent_ino < 0) { fs_unlock(); return old_parent_ino; }
    int new_parent_ino = resolve_path(new_parent);
    if (new_parent_ino < 0) { fs_unlock(); return new_parent_ino; }
    /* Find the inode we're moving */
    inode_t old_parent_inode;
    read_inode(g_fs.fd, &g_fs.sb, (uint32_t)old_parent_ino, &old_parent_inode);
```

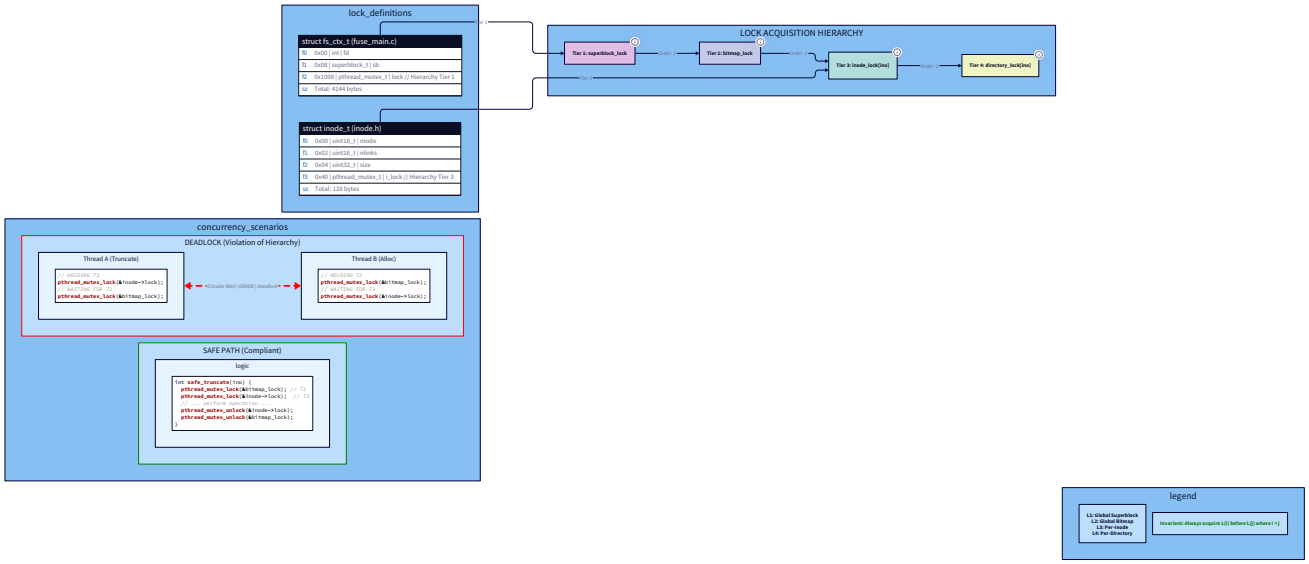
```
    int src_ino = dir_lookup(g_fs.fd, &g_fs.sb, &old_parent_inode, old_name);
    if (src_ino <= 0) { fs_unlock(); return src_ino < 0 ? src_ino : -ENOENT; }
    inode_t src_inode;
    read_inode(g_fs.fd, &g_fs.sb, (uint32_t)src_ino, &src_inode);
    /* Check if new_path already exists */
    inode_t new_parent_inode;
    read_inode(g_fs.fd, &g_fs.sb, (uint32_t)new_parent_ino, &new_parent_inode);
    int dst_ino = dir_lookup(g_fs.fd, &g_fs.sb, &new_parent_inode, new_name);
    if (dst_ino > 0) {
        /* new_path exists: must atomically replace it */
        inode_t dst_inode;
        read_inode(g_fs.fd, &g_fs.sb, (uint32_t)dst_ino, &dst_inode);
        if (S_ISDIR(src_inode.mode) && !S_ISDIR(dst_inode.mode))
            { fs_unlock(); return -ENOTDIR; }
        if (!S_ISDIR(src_inode.mode) && S_ISDIR(dst_inode.mode))
            { fs_unlock(); return -EISDIR; }
        if (S_ISDIR(dst_inode.mode) && !dir_is_empty(g_fs.fd, &dst_inode))
            { fs_unlock(); return -ENOTEMPTY; }
        /* Remove the existing destination entry (freeing dst if nlinks → 0) */
        if (S_ISDIR(dst_inode.mode))
            r = fs_rmdir(g_fs.fd, &g_fs.sb, (uint32_t)new_parent_ino, new_name);
        else
            r = fs_unlink(g_fs.fd, &g_fs.sb, (uint32_t)new_parent_ino, new_name);
        if (r < 0) { fs_unlock(); return r; }
        /* Reload new_parent_inode as it may have changed */
        read_inode(g_fs.fd, &g_fs.sb, (uint32_t)new_parent_ino, &new_parent_inode);
    }
    /* Step 1: Add new entry in destination directory */
    uint8_t ft = S_ISDIR(src_inode.mode) ? FT_DIR : FT_REG_FILE;
    r = dir_add_entry(g_fs.fd, &g_fs.sb,
                      (uint32_t)new_parent_ino, new_name,
```

```
                    (uint32_t)src_ino, ft);
    if (r < 0) { fs_unlock(); return r; }
    /* If moving a directory, update its '..' entry to point to new parent */
    if (S_ISDIR(src_inode.mode) &&
        (uint32_t)old_parent_ino != (uint32_t)new_parent_ino) {
        /* Read src directory's data block, update '..' inode_num */
        uint8_t dir_buf[BLOCK_SIZE];
        if (src_inode.blocks[0] != 0 &&
            read_block(g_fs.fd, src_inode.blocks[0], dir_buf) == 0) {
            /* '..' is the second entry; find and update it */
            dirent_t dot;
            uint16_t dot_len = dirent_read_from_buf(dir_buf, 0, &dot);
            if (dot_len > 0) {
                dirent_t dotdot;
                dirent_read_from_buf(dir_buf, dot_len, &dotdot);
                dotdot.inode_num = (uint32_t)new_parent_ino;
                dirent_write_to_buf(dir_buf, dot_len, &dotdot);
                write_block(g_fs.fd, src_inode.blocks[0], dir_buf);
            }
        }
        /* Old parent loses one '..' reference; new parent gains one */
        inode_t op_inode, np_inode;
        read_inode(g_fs.fd, &g_fs.sb, (uint32_t)old_parent_ino, &op_inode);
        read_inode(g_fs.fd, &g_fs.sb, (uint32_t)new_parent_ino, &np_inode);
        if (op_inode.nlinks > 0) op_inode.nlinks--;
        np_inode.nlinks++;
        op_inode.ctime = np_inode.ctime = (uint32_t)time(NULL);
        write_inode(g_fs.fd, &g_fs.sb, (uint32_t)old_parent_ino, &op_inode);
        write_inode(g_fs.fd, &g_fs.sb, (uint32_t)new_parent_ino, &np_inode);
    }
    /* Step 2: Remove old entry from source directory */
```

```
    /* Use a lower-level removal that does NOT decrement src_ino's nlinks */
    /* (we're not deleting the file, just moving the directory entry) */
    /* Temporarily increment nlinks to prevent free_inode on remove */
    src_inode.nlinks++;
    write_inode(g_fs.fd, &g_fs.sb, (uint32_t)src_ino, &src_inode);
    r = dir_remove_entry(g_fs.fd, &g_fs.sb, (uint32_t)old_parent_ino, old_name);
    if (r == 0) {
        /* dir_remove_entry decremented nlinks; restore to correct value */
        read_inode(g_fs.fd, &g_fs.sb, (uint32_t)src_ino, &src_inode);
        /* nlinks is correct now: dir_remove_entry decremented by 1, which undoes our +1 */
        src_inode.ctime = (uint32_t)time(NULL);
        write_inode(g_fs.fd, &g_fs.sb, (uint32_t)src_ino, &src_inode);
    }
    /* Persist superblock */
    uint8_t sb_buf[BLOCK_SIZE];
    memset(sb_buf, 0, BLOCK_SIZE);
    memcpy(sb_buf, &g_fs.sb, sizeof(g_fs.sb));
    write_block(g_fs.fd, 0, sb_buf);
    fs_unlock();
    return r;
}
```

> **Rename Atomicity Without a Journal**: The write ordering above (add new entry → remove old entry) guarantees that if you crash mid-rename, the file is accessible under at least one name. The worst case is the file having two names temporarily — a "hard link" that `fsck` can detect. Milestone 6's journal will make rename truly atomic: both operations are in one transaction, applied together or not at all.

Concurrency: The Locking Model



FUSE by default dispatches callbacks on multiple threads simultaneously. The `default_permissions` option handles POSIX permission checking in the kernel before your callback is invoked. You still need to protect your shared state. [[EXPLAIN:pthread-mutex-vs-rwlock-—-exclusive-vs-shared-locking-tradeoffs|pthread_mutex_t vs. pthread_rwlock_t — when to use exclusive locking vs. reader-writer locks]] The three-tier locking model for a production FUSE filesystem:

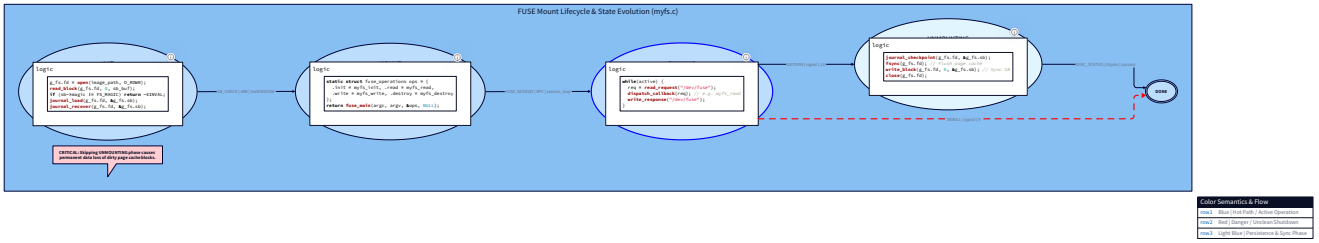
Level	Scope	Mechanism	When to use
Global mutex (chosen ✓)	All filesystem state	`pthread_mutex_t`	Correct, simple, sufficient for most workloads
Per-inode rwlock	Single inode's data + metadata	`pthread_rwlock_t`	High-concurrency reads; complex to implement correctly
Lock-free bitmaps	Block/inode allocation	Atomic bit operations	Highest performance; requires careful memory ordering
For this milestone, the global mutex is correct and appropriate. The single lock serializes all filesystem operations, making your code trivially safe for concurrent access. The performance cost: concurrent `read` calls from two processes must take turns, even though they could theoretically proceed in parallel.			
The global lock is standard for FUSE implementations. Even the Linux ext4 filesystem uses per-filesystem locks for many operations. The key is that the lock is held only during your callback execution — disk I/O inside the callback holds the lock, which means a slow disk blocks other filesystem operations. A future optimization would use asynchronous I/O and release the lock while waiting for disk, but this requires a fundamentally different architecture.			
Register a single-threaded FUSE with:			

```c
static struct fuse_operations myfs_ops = { ... };

/* In main(), when calling fuse_main: add "-s" to argv to force single-threaded mode */

/* Or explicitly use fuse_session_loop_mt() vs fuse_session_loop() */
```

For the initial implementation, single-threaded mode is recommended to eliminate concurrency bugs while you get the callbacks right. Enable multi-threading once all callbacks are working:

```c
/* Force single-threaded FUSE for initial development */

/* Add "-s" to fuse args to disable multithreading */
```

The Mount/Unmount Lifecycle



Two callbacks control the filesystem's lifecycle: `init` (called once when FUSE is ready to serve requests) and `destroy` (called once when the filesystem unmounts). These are your constructors and destructors:

```c
/*
 * init — Called when FUSE is ready to handle requests.
 * conn: connection info (protocol version, capabilities).
 * cfg:  FUSE configuration options (caching behavior, timeouts).
 *
 * Returns a pointer passed to all subsequent callbacks as userdata.
 * (We use global state instead, so return NULL.)
 */
static void *myfs_init(struct fuse_conn_info *conn,
                       struct fuse_config *cfg) {
    (void)conn;
    /*
     * entry_timeout: how long the kernel caches "name → inode" lookups.
     * Setting to 0 disables caching — correctness first.
     * Setting to 1.0 (1 second) dramatically speeds up repeated path lookups.
     * Trade-off: stale cache if another process modifies the filesystem
     * directly (not through FUSE). For our single-client model, 1.0 is safe.
     */
    cfg->entry_timeout = 1.0;
    cfg->attr_timeout  = 1.0;  /* how long to cache stat results */
    cfg->negative_timeout = 0.5;  /* how long to cache ENOENT results */
    cfg->use_ino = 1;   /* use our inode numbers in readdir and getattr */
    return NULL;
}
/*
 * destroy — Called when FUSE unmounts.
 * Flush all pending I/O to disk. Release all resources.
 * After this returns, the disk image must be in a consistent state.
 */
static void myfs_destroy(void *private_data) {
    (void)private_data;
```

```
    fs_lock();
    /*
     * Flush all dirty kernel page cache pages for our disk image to disk.
     * Without fsync here, recently written data may still be in the kernel's
     * page cache and lost if the system shuts down after unmount.
     */
    fsync(g_fs.fd);
    /* Persist the current superblock (free counts) */
    uint8_t sb_buf[BLOCK_SIZE];
    memset(sb_buf, 0, BLOCK_SIZE);
    memcpy(sb_buf, &g_fs.sb, sizeof(g_fs.sb));
    write_block(g_fs.fd, 0, sb_buf);
    /* Final fsync after superblock write */
    fsync(g_fs.fd);
    close(g_fs.fd);
    pthread_mutex_destroy(&g_fs.lock);
    fs_unlock();
    /* Note: do not call fs_unlock() after destroying the mutex */
}
```

> **Pitfall — Not fsyncing on Unmount**: When you unmount a FUSE filesystem, the kernel calls your `destroy` callback. If you do not call `fsync(g_fs.fd)` before `close`, any data written by your callbacks but not yet flushed by the host kernel's writeback daemon is lost. The sequence is: your `myfs_write` → host kernel page cache (dirty) → eventual writeback → disk. The writeback may not have happened by the time `destroy` is called. `fsync` forces the flush. This is especially important for the superblock — always write and fsync the superblock last, after all data blocks.

`statfs` — Disk Space Reporting

`df` and `df -h` call `statfs` to report available disk space. Without it, standard tools cannot report how much space your filesystem has:

```c
static int myfs_statfs(const char *path, struct statvfs *stbuf) {

    (void)path;

    fs_lock();

    memset(stbuf, 0, sizeof(*stbuf));

    stbuf->f_bsize   = BLOCK_SIZE;              /* filesystem block size */

    stbuf->f_frsize  = BLOCK_SIZE;              /* fragment size (same) */

    stbuf->f_blocks  = g_fs.sb.total_blocks - g_fs.sb.data_block_start;

    stbuf->f_bfree   = g_fs.sb.free_blocks;

    stbuf->f_bavail  = g_fs.sb.free_blocks;     /* no reserved blocks */

    stbuf->f_files   = g_fs.sb.total_inodes;

    stbuf->f_ffree   = g_fs.sb.free_inodes;

    stbuf->f_favail  = g_fs.sb.free_inodes;

    stbuf->f_namemax = MAX_FILENAME_LEN;

    fs_unlock();

    return 0;

}
```

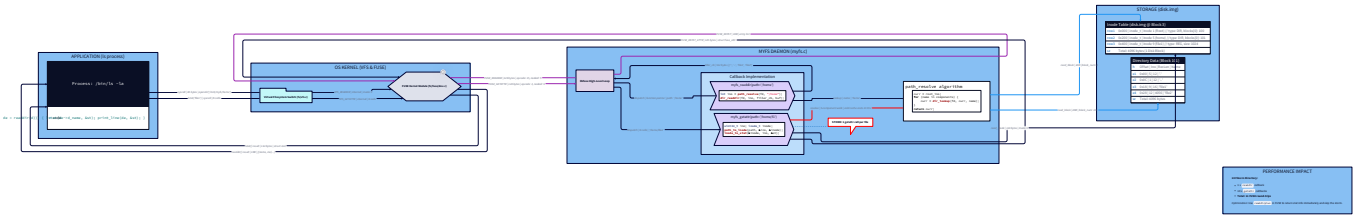
Wiring Everything Together: `main` and Registration

```c
/*
 * FUSE operations table.
 * Every NULL field uses FUSE's default behavior.
 * Explicitly NULL-ify operations you haven't implemented — don't leave
 * function pointers uninitialized (they point to garbage in C).
 */
static struct fuse_operations myfs_ops = {
    .init       = myfs_init,
    .destroy    = myfs_destroy,
    .getattr    = myfs_getattr,
    .readdir    = myfs_readdir,
    .open       = myfs_open,
    .create     = myfs_create,
    .read       = myfs_read,
    .write      = myfs_write,
    .release    = myfs_release,
    .mkdir      = myfs_mkdir,
    .rmdir      = myfs_rmdir,
    .unlink     = myfs_unlink,
    .rename     = myfs_rename,
    .truncate   = myfs_truncate,
    .chmod      = myfs_chmod,
    .utimens    = myfs_utimens,
    .statfs     = myfs_statfs,
    /* Not implemented in this milestone: */
    .link       = NULL,   /* hard links: call fs_link from Milestone 3 */
    .symlink    = NULL,   /* symbolic links: future extension */
    .readlink   = NULL,
    .mknod      = NULL,   /* we use create instead */
    .chown      = NULL,   /* future: update uid/gid */
```

```c
    .flush      = NULL,
    .fsync      = NULL,
};

/*
 * main — Mount the filesystem.
 *
 * Usage: ./myfs <disk_image> <mountpoint> [fuse options]
 * Example: ./myfs disk.img /mnt/myfs -f -d   (-f foreground, -d debug)
 *          ./myfs disk.img /mnt/myfs -s       (-s single-threaded)
 *
 * We consume the first argument (disk_image) ourselves; the rest
 * are passed to fuse_main.
 */
int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <disk_image> <mountpoint> [fuse options]\n",
                argv[0]);
        return 1;
    }
    const char *image_path = argv[1];
    /* Remove disk_image from argv so fuse_main sees only its own args */
    /* argv[0] = program name, argv[1] = image_path, argv[2..] = fuse args */
    argc--;
    argv[1] = argv[0];   /* shift: argv[0]=program, argv[1]=mountpoint, ... */
    argv++;
    /* Open the disk image */
    g_fs.fd = open(image_path, O_RDWR);
    if (g_fs.fd < 0) {
        perror("open disk image");
        return 1;
    }
```

```
    /* Read and validate the superblock */
    uint8_t sb_buf[BLOCK_SIZE];
    if (read_block(g_fs.fd, 0, sb_buf) != 0) {
        fprintf(stderr, "Failed to read superblock\n");
        close(g_fs.fd);
        return 1;
    }
    memcpy(&g_fs.sb, sb_buf, sizeof(g_fs.sb));
    if (g_fs.sb.magic != FS_MAGIC) {
        fprintf(stderr, "Invalid filesystem magic: 0x%08X (expected 0x%08X)\n",
                g_fs.sb.magic, FS_MAGIC);
        close(g_fs.fd);
        return 1;
    }
    /* Initialize the global mutex */
    if (pthread_mutex_init(&g_fs.lock, NULL) != 0) {
        perror("pthread_mutex_init");
        close(g_fs.fd);
        return 1;
    }
    strncpy(g_fs.image_path, image_path, sizeof(g_fs.image_path) - 1);
    printf("Mounting %s\n", image_path);
    printf("  Total blocks: %u, Free blocks: %u\n",
           g_fs.sb.total_blocks, g_fs.sb.free_blocks);
    printf("  Total inodes: %u, Free inodes: %u\n",
           g_fs.sb.total_inodes, g_fs.sb.free_inodes);
    /* Hand off to FUSE */
    return fuse_main(argc, argv, &myfs_ops, NULL);
}
```

The `ls -la` Trace: What Actually Happens



Let's trace exactly what happens when you run `ls -la /mnt/myfs/home` against your mounted filesystem. This is the Three-Level View in full operation:

Step	POSIX Syscall	FUSE Callback	Your Code
1	`stat("/mnt/myfs/home")`	`getattr("/home")`	`path_resolve` → dir_lookup in root → read inode 14 → `inode_to_stat`
2	`opendir("/mnt/myfs/home")`	`opendir` (handled by libfuse)	—
3	`readdir(dir)` → first batch	`readdir("/home")`	`dir_readdir` → scan blocks → `filler(".", ...)`, `filler("user", ...)`
4	For each entry name: `lstat("/mnt/myfs/home/user")`	`getattr("/home/user")`	`path_resolve("/home/user")` → 2 dir lookups → read inode → `inode_to_stat`
5	Repeat step 4 for every entry	—	—
The critical insight: `ls -la` calls `getattr` **once per entry in the directory**, plus once for the directory itself. A directory with 100 entries generates 101 `getattr` calls. Each `getattr` call, cold, costs 3+ disk reads for path resolution. With the `attr_timeout` cache set to 1.0 second, the second `ls` on the same directory costs only ~1µs per entry (all cache hits).			

Hardware Soul — The Context Switch Cost Each FUSE callback involves two context switches: kernel→user (FUSE module writes request to /dev/fuse) and user→kernel (your process writes response back). A context switch on modern x86-64 costs ~2,000–5,000 CPU cycles, or ~1–2.5µs at 2GHz. For a `getattr` that hits the inode cache (100ns disk read from page cache), the context switch dominates: the round-trip is ~3µs, of which 2µs is context-switch overhead and only 0.1µs is actual work. This is why FUSE adds latency per-operation rather than per-byte: the overhead is constant per call, not proportional to data size.

Building and Mounting: Your First Real Mount

```bash
# Step 1: Format a 16MB disk image with 1024 inodes
./mkfs disk.img 4096 1024

# Step 2: Create a mount point
mkdir -p /tmp/myfs_mount

# Step 3: Mount (foreground mode with debug output)
./myfs disk.img /tmp/myfs_mount -f -d -s

# In another terminal:

# Step 4: Test with standard Unix tools
ls -la /tmp/myfs_mount/
mkdir /tmp/myfs_mount/testdir
echo "Hello, Filesystem!" > /tmp/myfs_mount/testdir/hello.txt
cat /tmp/myfs_mount/testdir/hello.txt
cp /etc/hosts /tmp/myfs_mount/hosts_copy
ls -la /tmp/myfs_mount/

# Step 5: Unmount
fusermount3 -u /tmp/myfs_mount

# Step 6: Verify persistence — remount and check data survived
./myfs disk.img /tmp/myfs_mount -f -s &
cat /tmp/myfs_mount/testdir/hello.txt  # Must print "Hello, Filesystem!"
fusermount3 -u /tmp/myfs_mount
```

The `-f` flag runs FUSE in the foreground (so you see debug output and can Ctrl+C to unmount). The `-d` flag enables verbose FUSE debugging showing every callback invocation. The `-s` flag enforces single-threaded mode — essential while debugging. If anything fails, FUSE's debug output shows exactly which callback returned which error. Common failures and their causes:

Symptom	Callback	Likely Cause
`ls: cannot access '/mnt': Transport endpoint is not connected`	—	Previous mount crashed; run `fusermount3 -u /mnt` first
`ls` shows nothing, no error	`readdir`	`filler` not called for `.` and `..`; or `getattr` returning wrong mode
`cat` returns empty	`read`	`fs_read` returning 0 before EOF; check offset clamping
`echo > file` succeeds but data lost on remount	`write` / `destroy`	Not fsyncing disk image in `destroy`
`cp` into mount returns `ENOSPC`	`write` / `create`	`alloc_block` returning -1; check bitmap and free block count
`mkdir` succeeds but `ls` shows wrong permissions	`getattr`	`inode_to_stat` not copying `st_mode` correctly

Integration Test Suite

Write a shell script that exercises every FUSE callback through real Unix tool invocations:

```bash
#!/bin/bash

# test_fuse.sh — Integration test for FUSE-mounted filesystem

set -euo pipefail

DISK="./test_disk.img"

MOUNT="./test_mount"

FS="./myfs"

MKFS="./mkfs"

cleanup() {

    fusermount3 -u "$MOUNT" 2>/dev/null || true

    rm -f "$DISK"

    rmdir "$MOUNT" 2>/dev/null || true

}

trap cleanup EXIT

# Setup

mkdir -p "$MOUNT"

$MKFS "$DISK" 4096 1024

$FS "$DISK" "$MOUNT" -s &

FUSE_PID=$!

sleep 0.5   # Wait for FUSE to be ready

echo "=== FUSE Integration Tests ==="

# Test 1: Root directory accessible

ls "$MOUNT" > /dev/null

echo "OK: root directory accessible"

# Test 2: mkdir and ls

mkdir "$MOUNT/testdir"

ls -la "$MOUNT" | grep "testdir" > /dev/null

echo "OK: mkdir creates directory visible to ls"

# Test 3: File creation and write

echo "Hello, Filesystem!" > "$MOUNT/testdir/hello.txt"

echo "OK: echo > file works"

# Test 4: File read
```

```
CONTENT=$(cat "$MOUNT/testdir/hello.txt")

[ "$CONTENT" = "Hello, Filesystem!" ]

echo "OK: cat reads correct content"

# Test 5: File permissions

chmod 0600 "$MOUNT/testdir/hello.txt"

PERMS=$(stat -c "%a" "$MOUNT/testdir/hello.txt")

[ "$PERMS" = "600" ]

echo "OK: chmod works"

# Test 6: cp into mount

cp /etc/hostname "$MOUNT/hostname_copy"

diff /etc/hostname "$MOUNT/hostname_copy"

echo "OK: cp works, content matches"

# Test 7: Large file (multi-block)

dd if=/dev/urandom of=/tmp/test_large bs=4096 count=100 2>/dev/null

cp /tmp/test_large "$MOUNT/large_file"

diff /tmp/test_large "$MOUNT/large_file"

echo "OK: large file (400KB) round-trip correct"

rm /tmp/test_large

# Test 8: rename

mv "$MOUNT/testdir/hello.txt" "$MOUNT/testdir/renamed.txt"

[ ! -e "$MOUNT/testdir/hello.txt" ]

CONTENT=$(cat "$MOUNT/testdir/renamed.txt")

[ "$CONTENT" = "Hello, Filesystem!" ]

echo "OK: rename works"

# Test 9: rmdir

mkdir "$MOUNT/emptydir"

rmdir "$MOUNT/emptydir"

[ ! -e "$MOUNT/emptydir" ]

echo "OK: rmdir works"

# Test 10: unlink

rm "$MOUNT/testdir/renamed.txt"
```

```
[ ! -e "$MOUNT/testdir/renamed.txt" ]

echo "OK: rm (unlink) works"

# Test 11: df reports reasonable values

df -h "$MOUNT" | grep -v Filesystem > /dev/null

echo "OK: statfs/df works"

# Test 12: Unmount and remount — data must persist

fusermount3 -u "$MOUNT"

wait $FUSE_PID 2>/dev/null || true

$FS "$DISK" "$MOUNT" -s &

FUSE_PID=$!

sleep 0.5

HOSTNAME_CONTENT=$(cat "$MOUNT/hostname_copy")

[ -n "$HOSTNAME_CONTENT" ]

echo "OK: data persists across unmount/remount"

fusermount3 -u "$MOUNT"

wait $FUSE_PID 2>/dev/null || true

echo "=== All Integration Tests Passed ==="
```

Three-Level View: One `open()` Syscall

Let's trace `open("/mnt/myfs/home/user/file.txt", O_RDONLY)` from application through hardware:

Level	What Happens
Application	Calls `open(2)` syscall. CPU switches to kernel mode via syscall interrupt.
OS / Kernel VFS	`do_sys_open()` → `file_open_name()` → `path_openat()`. VFS walks the mount table: `/mnt/myfs` is a FUSE mount. Hands off to the FUSE kernel module. FUSE module serializes: opcode=`LOOKUP` for each path component, queued to `/dev/fuse`. Blocks waiting.
Your Process	libfuse reads from `/dev/fuse`. Dispatches to `getattr("/home/user/file.txt")`. Your `myfs_getattr` calls `path_resolve` → 3 directory block reads → inode read. Returns `struct stat`. libfuse serializes response, writes to `/dev/fuse`.
Kernel again	FUSE module receives response. VFS gets the file's inode attributes. Constructs a `struct file` kernel object. Returns the file descriptor integer to the calling process.
Hardware	For cold-cache directory reads: 4 × 100µs SSD reads = 400µs. Page cache fill: DMA transfer from SSD to kernel memory. For warm-cache: 4 × 100ns = 400ns.
The total latency for a warm-cache `open` through FUSE: ~3–5µs (two context switches) + ~400ns (cache hits) ≈ **4µs**. For comparison, `open` on a native ext4 file: ~1–2µs (one kernel-level path walk, no context switch). The FUSE overhead is 2–4× for latency-sensitive operations.	

Knowledge Cascade: One FUSE Mount, Ten Worlds

VFS as Universal Contract: ext4, XFS, NFS, Your Filesystem

The VFS layer you have just implemented against (via FUSE) is the same interface that every Linux filesystem implements. ext4, XFS, tmpfs, procfs, sysfs, and NFS all register `inode_operations` and `file_operations` function pointers with the VFS. FUSE simply adds a forwarding layer that sends those calls to userspace instead of handling them in-kernel. This means: everything you built in this milestone — the callback signatures, the error codes, the `struct stat` semantics, the `rename` atomicity requirement — reflects the VFS contract. When you read the ext4 source code (`fs/ext4/file.c`, `fs/ext4/dir.c`), you will see the same function signatures: `ext4_file_read_iter`, `ext4_readdir`, `ext4_create`, `ext4_rename`. The difference is that ext4's implementations directly manipulate kernel data structures while yours calls across a process boundary. The interface is identical. This knowledge unlocks reading kernel filesystem source code with clarity. The VFS is not a mystery; it is a well-defined interface that you have now implemented. [[EXPLAIN:linux-vfs-inode-operations-struct-—-the-kernel-interface-every-filesystem-implements|Linux VFS inode_operations struct — the complete list of function pointers every kernel filesystem must implement]]

Microkernel Architecture Inside a Monolithic Kernel

FUSE is a microkernel pattern embedded inside Linux's monolithic kernel. In a pure microkernel (QNX, Minix, L4), all device drivers, filesystems, and services run as separate userspace processes communicating via message passing. The kernel provides only IPC, scheduling, and memory management. Everything else — including the filesystem — is a userspace server.

Linux chose the monolithic design for performance: fewer context switches, shared address space, faster function calls. But FUSE grafts the microkernel pattern onto it for filesystems specifically, trading the 2–5µs per-operation overhead for:

- **Crash safety**: a bug in your filesystem crashes your process, not the kernel. No kernel panic, no system crash, no reboot required.
- **Language freedom**: FUSE filesystems can be written in Python, Go, Rust, Java — anything with a `/dev/fuse` binding. sshfs was originally written in C but go-fuse enables Go implementations.
- **Development velocity**: modify your filesystem, kill the old process, mount again. No kernel module recompilation, no `insmod`/`rmmod`, no risk of hanging the kernel during development. This is the exact trade-off that separates QNX (used in automotive safety systems — a crash cannot take down the OS) from early Linux (maximum performance — filesystem bugs can kernel-panic). FUSE gives Linux the microkernel safety property for filesystem development specifically.

gRPC and IPC: FUSE is an RPC System

[[EXPLAIN:rpc-protocol-—-remote-procedure-call-serialization-and-dispatch-over-a-channel|RPC protocol — serializing a function call with arguments into bytes, sending over a channel, and deserializing the response]] The `/dev/fuse` protocol is an RPC system. When the kernel dispatches `getattr("/home/user/file.txt")`, it serializes the request into a binary message:

```
[unique_id: uint64] [opcode: uint32=FUSE_GETATTR] [len: uint32] [nodeid: uint64] [path: bytes]
```

Your process reads this message, dispatches based on `opcode`, executes the handler, and serializes the response:

```
[unique_id: uint64 (matches request)] [error: int32=0] [attr: struct fuse_attr]
```

This is structurally identical to gRPC: a client (kernel) serializes a request with a unique ID, sends it over a channel (/dev/fuse ↔ network socket), a server (your process) deserializes and handles it, returns a response with the matching ID. The FUSE protocol uses binary encoding (like Protocol Buffers in gRPC); the flow control (one outstanding request per "session") is simpler than gRPC's multiplexing, but the pattern is identical. Understanding FUSE as an RPC system means understanding why latency-sensitive FUSE operations have a fixed overhead: two message serializations and two context switches per call, regardless of data size. This is the same overhead that makes gRPC unsuitable for high-frequency, low-payload calls (where Protocol Buffers serialization overhead dominates over per-byte cost).

Thread Safety as a First-Class Design Concern

Your global `pthread_mutex_t` is your first production-grade use of concurrent locking in this project. The patterns you applied here — acquire-before-access, release-before-return, covering all code paths including error paths — are the same patterns used in every concurrent system. The classic pitfall you avoided: returning early from a callback without releasing the lock:

```c
/* BUG: lock never released if path_to_inode returns early */
static int myfs_getattr_BUGGY(const char *path, struct stat *st, ...) {
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) return r;   /* ← DEADLOCK: lock never released */
    /* ... */
    fs_unlock();
    return 0;
}
/* CORRECT: always release before return */
static int myfs_getattr_CORRECT(const char *path, struct stat *st, ...) {
    fs_lock();
    uint32_t ino;
    inode_t inode;
    int r = path_to_inode(path, &ino, &inode);
    if (r < 0) { fs_unlock(); return r; }   /* ← unlock on error path */
    /* ... */
    fs_unlock();
    return 0;
}
```

This pattern — "unlock on every exit path" — is what C++ RAII (`std::lock_guard`), Rust's `MutexGuard`, and Go's `defer mu.Unlock()` automate. In C, you must discipline yourself to write `fs_unlock()` before every `return`. This is a real reason why systems programming in C is error-prone: the compiler does not enforce lock pairing. The alternative you should know about for future optimization: **per-inode reader-writer locks**. Reads from different files can proceed concurrently (multiple readers share the rwlock), but a write to any file still needs exclusive access to that file's inode. The global mutex forces all operations to serialize, even two concurrent reads of completely unrelated files. A `pthread_rwlock_t` per inode eliminates this unnecessary serialization. The complexity cost: lock ordering (always acquire parent-directory lock before child-inode lock), deadlock detection, and the overhead of N mutexes instead of 1.

Path Resolution as the Performance Bottleneck

Every FUSE callback starts with a path-to-inode translation. For a filesystem mounted at `/mnt/myfs`, a 5-component path like `/home/user/projects/src/main.c` requires 5 directory lookups, each requiring:

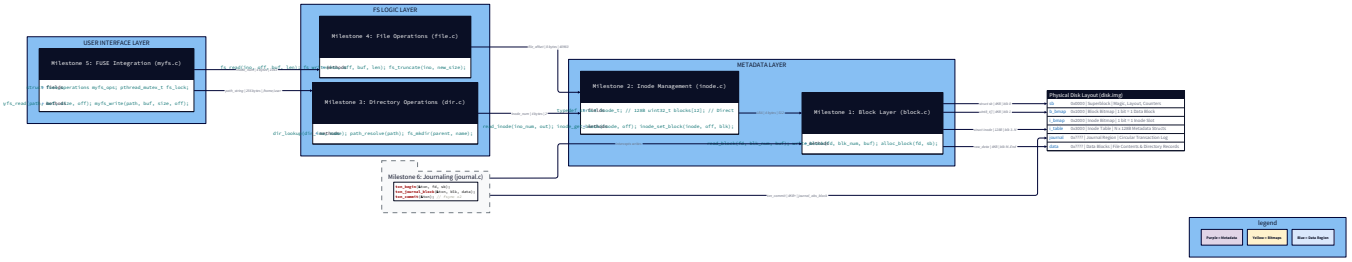
1. Read the inode for the current directory (1 block read)

2. Read the data block(s) of that directory and scan entries (1+ block reads) That is 10+ disk reads per path resolution, cold. With the page cache warm, it drops to 10+ memory reads (~1µs total). With the FUSE `entry_timeout` cache set to 1.0 second, the kernel caches the translation and your callback is not invoked at all for subsequent lookups within the timeout window. This is why profiling always points to `path_resolve` as the hottest function. The optimization path is:
3. **FUSE entry/attr timeouts** (already implemented above): free, effective for repeated accesses
4. **In-process path cache**: a hash map `(path_string → inode_number)` in your process memory, consulted before calling `path_resolve`. Invalidate on any write operation.
5. **Inode-number-based API**: libfuse's "low-level" API works with inode numbers directly, similar to how the kernel VFS works internally. This requires implementing `lookup` (returns inode number for a name in a parent) instead of path-based `getattr`. The kernel caches these lookup results in the dcache and passes inode numbers in subsequent calls, eliminating repeated path walks. The low-level FUSE API (`fuse_lowlevel.h`) is the production path for performance-critical filesystems. It mirrors the kernel's VFS interface exactly: every call carries a parent inode number and a name, not a full path. GlusterFS, the Ceph FUSE client, and most high-performance FUSE filesystems use the low-level API. The high-level API (what we implemented) is simpler to write but sacrifices this optimization.

Before You Move On: Pitfall Checklist

- [ ] **`getattr` on root works**: Run `ls /mnt/myfs` with no files. It must work. `getattr("/")` must return the root inode's `struct stat` with `st_mode = S_IFDIR | 0755` and `st_nlink >= 2`. If this fails, nothing else will.
- [ ] **`create` registered, not `mknod`**: Shell redirection (`echo > file`) calls `open(O_CREAT)` which FUSE dispatches to `create`. Register `myfs_ops.create`. If it's NULL, file creation silently fails.
- [ ] **`readdir` emits `.` and `..`**: Some tools (Python's `os.listdir`, `find`) depend on `.` and `..` being in the readdir output. Your `dir_readdir` emits them because they are real directory entries — verify with `ls -a`.
- [ ] **`fi->fh` stores inode number in `open` / `create`**: Your `read` and `write` callbacks use `(uint32_t)fi->fh` as the inode number. Verify that `fi->fh` is set correctly in `open` and `create`. A zero value means you forgot to set it and `read`/`write` will operate on inode 0 (invalid).
- [ ] **`fsync` in `destroy`**: Unmounting without `fsync(g_fs.fd)` can lose recently written data. Test by writing a file, unmounting immediately, and remounting — data must be present.
- [ ] **Lock on every code path**: Every callback that accesses `g_fs` must call `fs_lock()` at entry and `fs_unlock()` before every `return`. Run with `-s` (single-threaded) first to verify correctness, then remove `-s` and stress test with concurrent `cp` commands to verify thread safety.
- [ ] **Magic number validation in `main`**: Verify `g_fs.sb.magic == FS_MAGIC` before mounting. A typo in the image path that points to a random file would otherwise mount garbage as a filesystem, causing confusing crashes in every callback.
- [ ] **`rename` handles same-directory rename**: `rename("/a", "/b")` has `old_parent == new_parent`. Your implementation must handle this case without double-locking or corrupting the parent directory.
- [ ] **FUSE options `-f -s` for development**: Always develop with `-f` (foreground) so Ctrl+C unmounts cleanly, and `-s` (single-threaded) to eliminate concurrency issues while debugging. Add `-d` for verbose callback tracing.
- [ ] **`fusermount3 -u` on crashed mount**: If your process crashes without unmounting, the mount point is left in a broken state. Running `./myfs disk.img /mnt -f` again fails with "Transport endpoint is not connected." Always `fusermount3 -u /mnt` before remounting after a crash.

What You've Built and What Comes Next



You have crossed the final boundary between "library" and "operating system component." Your filesystem is now a real mount point. Standard Unix tools — `ls`, `cat`, `cp`, `find`, `grep`, `vim` — work against it without modification, without knowing your code exists. The VFS contract is satisfied. The FUSE relay is wired. The entire stack from user application through kernel VFS through FUSE through your C code through block-level I/O runs seamlessly. But there is a crack in the foundation, and you have known about it since Milestone 1. Every write to your filesystem is a race condition. Not a concurrency race — a time race against power failure. Consider: `echo "critical data" > /mnt/myfs/important.txt` calls your `create` callback, which writes the block bitmap, the inode, and the directory entry in three separate disk operations. If power fails between the bitmap write and the inode write, you have a leaked block that the bitmap marks as used but no inode references. If power fails between the inode write and the directory entry write, you have an orphaned inode that no path can reach. If power fails during the `rename` you just implemented — after adding the new entry but before removing the old one — you have a file with two names. In all these cases, after a reboot, your filesystem is in an inconsistent state. The data blocks may be intact, but the metadata — bitmaps, inode table, directory entries — is incoherent. `fsck` can repair some of these by brute-force scanning all blocks and reconstructing consistent metadata, but that takes minutes on a large filesystem and sometimes loses data. Milestone 6 solves this definitively with write-ahead journaling. Every multi-step operation is wrapped in a transaction: write all changes to a sequential journal region first, write a commit record atomically, then apply the changes to the primary data structures. On crash: scan the journal, replay all committed transactions, discard incomplete ones. The filesystem is always consistent after recovery — not just "recoverable," but guaranteed consistent by the

structure of the write protocol. The filesystem is almost production-ready. One milestone remains.

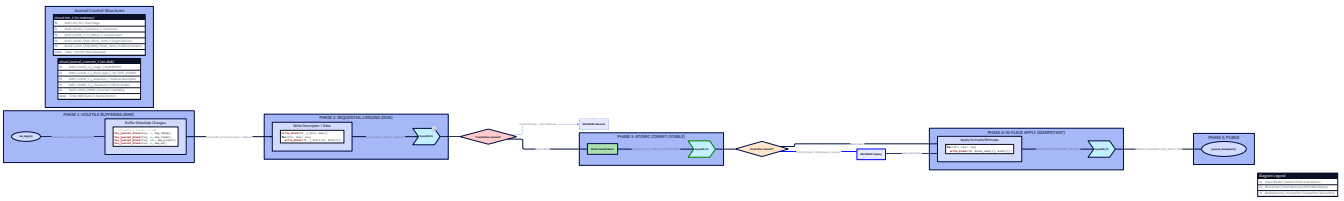
Milestone 6: Write-Ahead Journaling and Crash Recovery

The Lie Your Code Has Been Telling

Every filesystem operation you have written since Milestone 1 has contained a silent assumption so pervasive you probably never noticed it: that writes to the disk image happen in exactly the order your code issues them. You call `write_block(fd, BLOCK_BITMAP_BLOCK, bitmap)` and then `write_inode(fd, sb, new_ino, &inode)`. You feel safe because the bitmap is written before the inode — if you crash between the two, at worst you have a leaked block (bitmap says "used," no inode points to it), and that is recoverable. You have been relying on write ordering as your crash consistency strategy. That assumption is false. And the falseness is not subtle. Here is the chain of reordering that can happen between your `write_block` call and the bits reaching magnetic media or NAND cells:

1. **The host kernel's page cache**: When your process calls `write(fd, buf, 4096)` via `write_block`, the kernel copies the data into a page cache page and marks it dirty. The actual flush happens asynchronously via the kernel's writeback daemon, which decides order and timing based on I/O scheduler heuristics — not your call order.
2. **The disk's write buffer**: Modern HDDs and SSDs have DRAM write buffers of 64MB to 1GB. The disk accepts writes from the OS and reorders them internally to optimize head seek (HDD) or program/erase sequencing (SSD). Unless you issue a barrier or `fsync`, the disk's buffer is unconstrained.
3. **The disk controller's NCQ (Native Command Queuing)**: HDDs queue up to 32 commands and reorder them by rotational position. Your bitmap write, queued before your inode write, may execute second because the inode block happens to be at a more favorable rotational position when the command is dispatched. The result: you call `write_block(bitmap)` then `write_block(inode)`. The disk writes the inode first. You suffer a power failure. On recovery, the inode exists on disk pointing to a block that the bitmap marks as free — potentially double-allocated to a new file. **Data corruption. Not a leaked block. Actual data from two files occupying the same disk block.** [[EXPLAIN:write-ahead-logging-—-the-principle-of-logging-intent-before-action|Write-ahead logging — the principle of logging intent before action]] The only mechanism that actually enforces ordering is `fsync`. When you call `fsync(fd)`, the kernel flushes all dirty pages for that file descriptor to the device and waits for the device to confirm that the data is durable — not just in the device's buffer, but confirmed written to stable storage. After `fsync` returns, you have a guarantee: everything written to `fd` before the call is on disk in the order the flushes were issued. Write-ahead logging (WAL) is the design pattern that weaponizes this guarantee. The protocol is simple and absolute:
4. Write all changes to a **sequential journal region** on disk
5. Call `fsync` — all journal writes are now durable and ordered
6. Write a **commit record** to the journal indicating this transaction is complete
7. Call `fsync` again — the commit record is now durable
8. Write the actual changes to their permanent locations (inode table, bitmaps, directory blocks)
9. Call `fsync` — the main data structures now reflect the committed state If power fails at any point before step 4, the commit record does not exist. On recovery, the partial journal entries are discarded. The main data structures are unchanged — perfectly consistent, as if the operation never happened. If power fails between step 4 and step 6, the commit record exists. On recovery, the journal entries are replayed against the main data structures. The operation is completed. The filesystem is consistent. The journal is not an optimization. It is the fundamental mechanism that makes crash consistency possible without a full filesystem scan. Without it, the only recovery option is `fsck` — which must read every inode, every bitmap, and every

directory block on the entire filesystem to reconstruct consistent state. On a 10TB disk at 500MB/s sequential read speed, that is six hours of downtime every time someone trips over a power cable.



The Fundamental Tension: Atomicity on Non-Atomic Hardware

Here is the core hardware constraint you are negotiating with: **A filesystem update requires multiple disk writes. Each disk write is independently atomic at the sector level (512 bytes or 4KB). No multi-block operation is atomic at the hardware level.** Creating a file requires at minimum: one bitmap write (inode bitmap), one inode table write, one directory block write, and potentially a bitmap write for the data block. Four separate 4KB writes. The hardware can guarantee that each individual 4KB sector write either completes or does not — it will not write half a sector. But there is no hardware primitive for "write these four blocks atomically." The software solution — WAL — synthesizes atomicity from sequential writes plus a commit record. The key insight is that a single 4KB sector write *is* hardware-atomic. If the commit record fits in one sector (and journal commit records are small — a few dozen bytes), the commit write is atomic. Either the commit record is present on disk or it is not. There is no half-written commit record. The commit record becomes the single atomic boundary that separates "committed" from "not committed," and the journal structure ensures that a missing commit record always means "safe to discard."

Three Crash Scenarios: Journal Recovery Logic

Operation: `mkdir("/test")` — updates Inode Bitmap (blk 2) and Inode Table (blk 5)

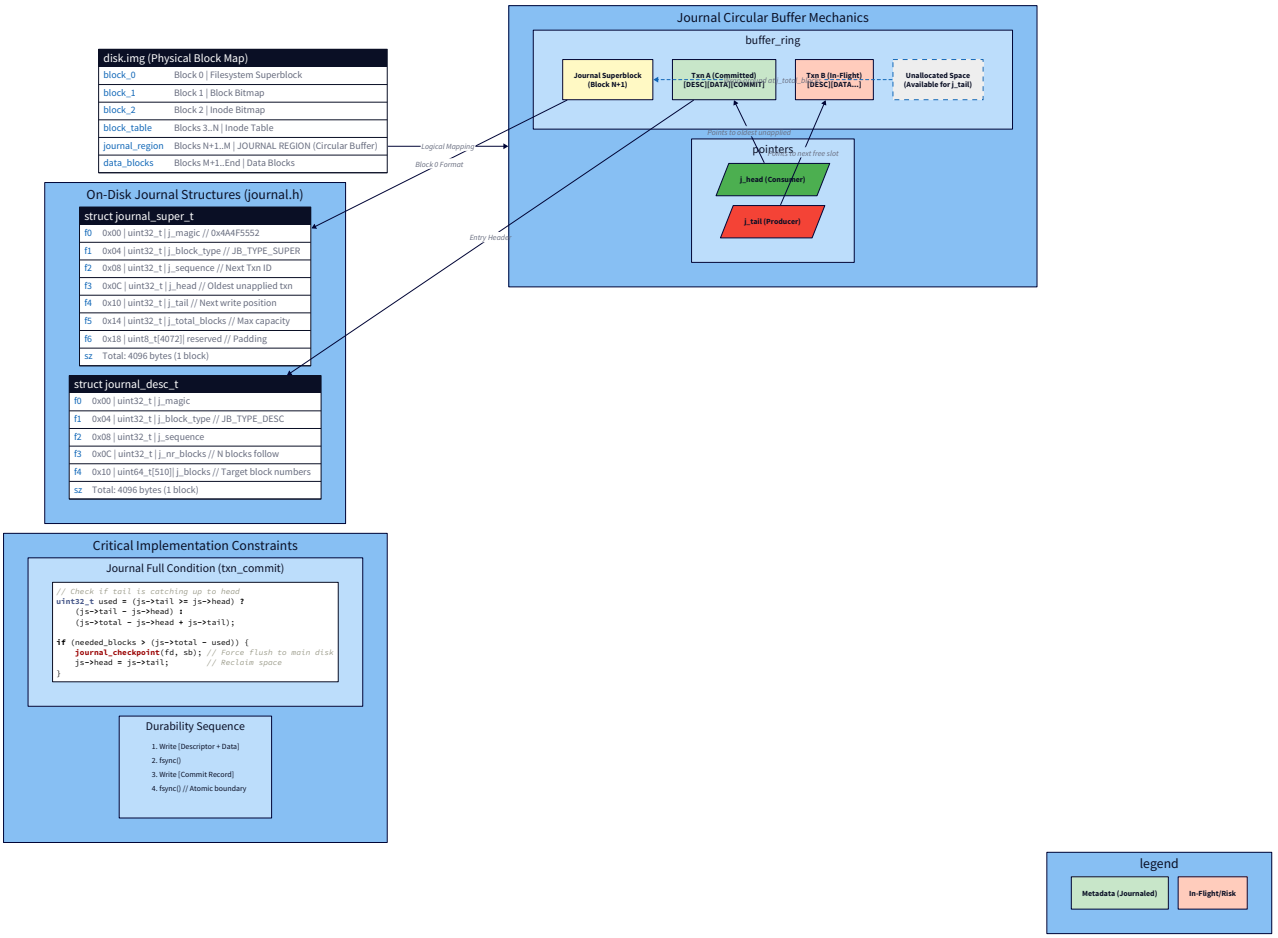
Status Legend	
s1	Red Border | Dangerous Path / Abort
s2	Green Border | Safe Replay / Success
s3	Yellow Fill | Inconsistent / Fixable

The numbers that drive the design:

- Journal write of 4KB: ~100µs on SSD (same as any block write)
- `fsync` call: ~100µs–1ms on SSD depending on device and host OS
- Journal entry overhead: 2× `fsync` per transaction minimum (journal flush + commit flush)
- Benefit: O(journal_size) recovery instead of O(filesystem_size) recovery
- Typical journal size: 4MB–128MB (1,024–32,768 blocks at 4KB)
- Recovery time from 4MB journal: ~40ms (400 block reads at SSD speed)
- Recovery time from full fsck on 10TB disk: hours

Journal Region: On-Disk Layout

The journal occupies a contiguous region of blocks configured in the superblock. From Milestone 1, you already reserved `sb->journal_start` and `sb->journal_blocks` for this purpose. Now you give that region a structure.



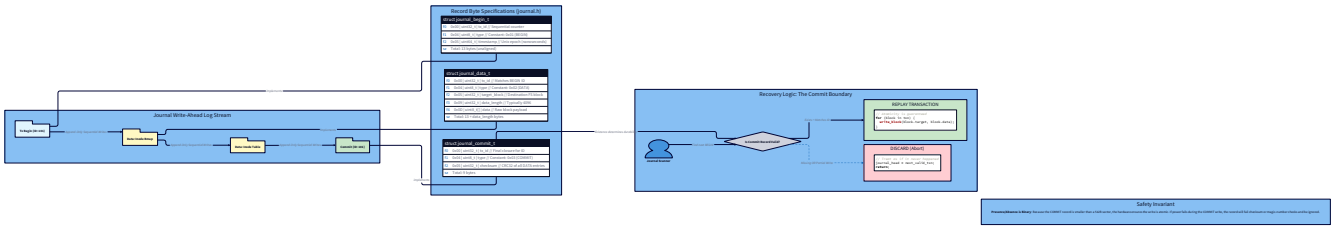
The journal region has three kinds of content:

1. **Journal superblock** (first block of the journal region): stores the journal's head, tail, and sequence number. This is the journal's own metadata — distinct from the filesystem superblock.
2. **Transaction entries** (the body): a circular buffer of journal blocks, each either a metadata block copy or a descriptor block describing what follows.
3. **Commit blocks**: special single-block entries that mark transaction boundaries. We implement a simplified but correct journal with these on-disk structures:

```c
#include <stdint.h>

#include <string.h>

#define JOURNAL_MAGIC       0x4A4F5552    /* "JOUR" in ASCII */

#define JOURNAL_BLOCK_SIZE  BLOCK_SIZE    /* 4096 bytes */

/* Journal block types */

#define JB_TYPE_SUPER    0x01    /* journal superblock */

#define JB_TYPE_DESC     0x02    /* descriptor: describes following data blocks */

#define JB_TYPE_COMMIT   0x03    /* commit record: transaction is complete */

#define JB_TYPE_REVOKE   0x04    /* revoke: this block should not be replayed */

/*

 * Journal Superblock — lives at journal_start (first block of journal region).

 *

 * BYTE LAYOUT:

 *   Offset  0: j_magic          (4 bytes)

 *   Offset  4: j_block_type     (4 bytes) — always JB_TYPE_SUPER

 *   Offset  8: j_sequence       (4 bytes) — next transaction sequence number

 *   Offset 12: j_head           (4 bytes) — first block with live data

 *   Offset 16: j_tail           (4 bytes) — first free block (circular)

 *   Offset 20: j_total_blocks   (4 bytes) — total blocks in journal region (excluding superblock)

 *   Offset 24: reserved         (4072 bytes)

 *   Total: 4096 bytes

 */

typedef struct __attribute__((packed)) {

    uint32_t j_magic;           /* JOURNAL_MAGIC */

    uint32_t j_block_type;      /* JB_TYPE_SUPER */

    uint32_t j_sequence;        /* monotonically increasing transaction counter */

    uint32_t j_head;            /* offset of oldest committed, unapplied transaction */

    uint32_t j_tail;            /* offset where next write goes */

    uint32_t j_total_blocks;    /* usable journal blocks (journal_blocks - 1) */

    uint8_t  reserved[BLOCK_SIZE - 24];

} journal_super_t;
```

```c
_Static_assert(sizeof(journal_super_t) == BLOCK_SIZE,
               "journal_super_t must be exactly 4096 bytes");

/*
 * Journal Descriptor Block — precedes a set of data blocks in the journal.
 *
 * A descriptor block describes the next N data blocks in the journal:
 * which filesystem block each one belongs to, and how many blocks follow.
 * After the descriptor block come exactly j_nr_blocks raw copies of metadata.
 *
 * BYTE LAYOUT:
 *   Offset  0: j_magic          (4 bytes)
 *   Offset  4: j_block_type     (4 bytes) — JB_TYPE_DESC
 *   Offset  8: j_sequence       (4 bytes) — transaction this belongs to
 *   Offset 12: j_nr_blocks      (4 bytes) — number of data blocks that follow
 *   Offset 16: j_blocks[0..N]   (8 bytes each) — target filesystem block numbers
 *              (max N = (BLOCK_SIZE - 16) / 8 = 510 blocks per descriptor)
 *   Total: 4096 bytes
 */
#define MAX_BLOCKS_PER_DESC  ((BLOCK_SIZE - 16) / sizeof(uint64_t))
typedef struct __attribute__((packed)) {
    uint32_t j_magic;
    uint32_t j_block_type;      /* JB_TYPE_DESC */
    uint32_t j_sequence;
    uint32_t j_nr_blocks;       /* how many data blocks follow this descriptor */
    uint64_t j_blocks[MAX_BLOCKS_PER_DESC]; /* target block numbers (0-indexed from j_blocks[0]) */
} journal_desc_t;
_Static_assert(sizeof(journal_desc_t) == BLOCK_SIZE,
               "journal_desc_t must be exactly 4096 bytes");

/*
 * Commit Block — marks the end of a transaction.
 *
```

```
 * A transaction is committed if and only if a commit block with a matching
 * sequence number exists in the journal after its descriptor and data blocks.
 * The commit block is written LAST, after fsync of the preceding data.
 * It is small enough to fit in a single atomic sector write.
 *
 * BYTE LAYOUT:
 *   Offset  0: j_magic          (4 bytes)
 *   Offset  4: j_block_type     (4 bytes) — JB_TYPE_COMMIT
 *   Offset  8: j_sequence       (4 bytes) — must match the descriptor
 *   Offset 12: j_checksum       (4 bytes) — CRC32 of all journaled block data
 *   Offset 16: reserved         (4080 bytes)
 *   Total: 4096 bytes
 */
typedef struct __attribute__((packed)) {
    uint32_t j_magic;
    uint32_t j_block_type;      /* JB_TYPE_COMMIT */
    uint32_t j_sequence;
    uint32_t j_checksum;        /* simple sum of all journaled bytes */
    uint8_t  reserved[BLOCK_SIZE - 16];
} journal_commit_t;
_Static_assert(sizeof(journal_commit_t) == BLOCK_SIZE,
               "journal_commit_t must be exactly 4096 bytes");
```



The journal operates as a **circular buffer** of blocks. The `j_head` and `j_tail` pointers — offsets into the journal region, not absolute disk block numbers — define the live region:

- `j_tail`: where the next write goes (producer end)
- `j_head`: the oldest live entry (consumer end, advanced during checkpointing)
- When `j_tail` wraps around to `j_head`, the journal is full — block until checkpoint The absolute disk block number for journal offset `N` is:

```c
uint32_t journal_abs_block(const superblock_t *sb, uint32_t journal_offset) {
    /* Offset 0 is the journal superblock; data starts at offset 1 */
    return sb->journal_start + 1 + (journal_offset % sb->j_total_blocks);
}
```

The Transaction API

Every filesystem operation that modifies metadata must be wrapped in a transaction. The transaction API has four calls:

```c
/*
 * In-memory transaction state.
 * A transaction accumulates block changes before committing them atomically.
 * Maximum transaction size: limited by journal capacity and descriptor block size.
 */
#define MAX_TXNBLOCKS   64   /* max blocks in one transaction */
typedef struct {
    int          fd;             /* disk image fd */
    superblock_t *sb;            /* filesystem superblock */
    journal_super_t *jsuper;     /* journal superblock (in-memory, flushed on commit) */
    uint32_t     sequence;       /* this transaction's sequence number */
    /* Accumulated dirty blocks */
    uint32_t     nr_blocks;      /* count of blocks pending */
    uint32_t     block_nums[MAX_TXNBLOCKS];     /* filesystem block numbers */
    uint8_t      block_data[MAX_TXNBLOCKS][BLOCK_SIZE]; /* block contents */
} txn_t;
/* Journal superblock in memory — loaded at mount, persisted at checkpoint */
static journal_super_t g_jsuper;
/*
 * Load the journal superblock from disk.
 * Called once during filesystem mount.
 */
int journal_load(int fd, const superblock_t *sb) {
    uint8_t buf[BLOCK_SIZE];
    if (read_block(fd, sb->journal_start, buf) != 0) return -EIO;
    memcpy(&g_jsuper, buf, sizeof(g_jsuper));
    if (g_jsuper.j_magic != JOURNAL_MAGIC) {
        /* First mount after mkfs: initialize the journal superblock */
        memset(&g_jsuper, 0, sizeof(g_jsuper));
        g_jsuper.j_magic        = JOURNAL_MAGIC;
        g_jsuper.j_block_type   = JB_TYPE_SUPER;
```

```
        g_jsuper.j_sequence     = 1;
        g_jsuper.j_head         = 0;
        g_jsuper.j_tail         = 0;
        g_jsuper.j_total_blocks = sb->journal_blocks - 1;
        return journal_persist_super(fd, sb);
    }
    return 0;
}
/*
 * Write the journal superblock to disk.
 */
int journal_persist_super(int fd, const superblock_t *sb) {
    uint8_t buf[BLOCK_SIZE];
    memset(buf, 0, BLOCK_SIZE);
    memcpy(buf, &g_jsuper, sizeof(g_jsuper));
    return write_block(fd, sb->journal_start, buf);
}
```

`txn_begin`: Opening a Transaction

```c
/*
 * Begin a new transaction.
 *
 * Initializes the in-memory transaction state. No disk I/O yet.
 * The transaction accumulates changes until txn_commit() is called.
 *
 * Every filesystem operation that modifies metadata must begin with this call.
 * NEVER modify inode table, bitmaps, or directory blocks without an active transaction.
 */
void txn_begin(txn_t *txn, int fd, superblock_t *sb) {
    memset(txn, 0, sizeof(*txn));
    txn->fd       = fd;
    txn->sb       = sb;
    txn->jsuper   = &g_jsuper;
    txn->sequence = g_jsuper.j_sequence;
    txn->nr_blocks = 0;
}
```

`txn_journal_block`: Recording a Change

Instead of calling `write_block` directly, every metadata write goes through `txn_journal_block`. This records the block's new content in the transaction but does not write to the block's final destination yet:

```c
/*
 * Record a block change in the current transaction.
 *
 * block_num: the FILESYSTEM block number that will eventually be updated.
 * data:      the new 4KB content for that block.
 *
 * This does NOT write to block_num on disk yet. The actual write happens
 * in txn_commit() after the journal entry is made durable.
 *
 * Returns 0 on success, -ENOSPC if the transaction is full.
 *
 * NOTE: If block_num is already in the transaction's pending list,
 * we update its entry (this is the "revoke" optimization for blocks
 * modified multiple times within one transaction).
 */
int txn_journal_block(txn_t *txn, uint32_t block_num, const void *data) {
    /* Check if this block is already in the transaction */
    for (uint32_t i = 0; i < txn->nr_blocks; i++) {
        if (txn->block_nums[i] == block_num) {
            /* Update existing entry — no need for duplicate journal writes */
            memcpy(txn->block_data[i], data, BLOCK_SIZE);
            return 0;
        }
    }
    if (txn->nr_blocks >= MAX_TXNBLOCKS) return -ENOSPC;
    txn->block_nums[txn->nr_blocks] = block_num;
    memcpy(txn->block_data[txn->nr_blocks], data, BLOCK_SIZE);
    txn->nr_blocks++;
    return 0;
}
```

`txn_commit`: The Critical Path

This is the most important function in the entire journaling system. Every design decision here traces back to the crash consistency guarantee:

```c
/*
 * Commit a transaction.
 *
 * Protocol (each step must complete before the next begins):
 *   1. Write descriptor block to journal (describes which blocks follow)
 *   2. Write all journaled block copies to journal
 *   3. fsync the disk image → journal entries are durable
 *   4. Write commit block to journal
 *   5. fsync again → commit record is durable (POINT OF NO RETURN)
 *   6. Write each block to its actual filesystem location
 *   7. fsync → main data structures updated
 *   8. Advance journal tail; update sequence number
 *
 * After step 5: transaction is committed. Crash recovery WILL replay it.
 * Before step 5: transaction is not committed. Crash recovery WILL NOT replay it.
 *
 * Returns 0 on success, negative errno on failure.
 */
int txn_commit(txn_t *txn) {
    if (txn->nr_blocks == 0) return 0;   /* empty transaction — nothing to do */
    superblock_t *sb    = txn->sb;
    journal_super_t *js = txn->jsuper;
    int fd              = txn->fd;
    /* Check journal has enough space for: 1 descriptor + nr_blocks data + 1 commit */
    uint32_t blocks_needed = 1 + txn->nr_blocks + 1;
    uint32_t journal_used  = (js->j_tail >= js->j_head)
                           ? (js->j_tail - js->j_head)
                           : (js->j_total_blocks - js->j_head + js->j_tail);
    uint32_t journal_free  = js->j_total_blocks - journal_used;
    if (blocks_needed > journal_free) {
        /*
```

```
         * Journal is full. In production: block until checkpoint completes.
         * For our implementation: force a checkpoint inline.
         * This is the "journal full" condition that can block writers.
         */
        int r = journal_checkpoint(fd, sb);
        if (r != 0) return r;
        /* Recompute after checkpoint */
        journal_used  = 0;   /* checkpoint cleared the journal */
        journal_free  = js->j_total_blocks;
        if (blocks_needed > journal_free) return -ENOSPC;
    }
    /* --- STEP 1: Write descriptor block to journal --- */
    uint8_t desc_buf[BLOCK_SIZE];
    memset(desc_buf, 0, BLOCK_SIZE);
    journal_desc_t *desc = (journal_desc_t *)desc_buf;
    desc->j_magic      = JOURNAL_MAGIC;
    desc->j_block_type = JB_TYPE_DESC;
    desc->j_sequence   = txn->sequence;
    desc->j_nr_blocks  = txn->nr_blocks;
    for (uint32_t i = 0; i < txn->nr_blocks; i++) {
        desc->j_blocks[i] = txn->block_nums[i];
    }
    uint32_t journal_pos = js->j_tail;
    uint32_t desc_abs    = journal_abs_block(sb, journal_pos);
    if (write_block(fd, desc_abs, desc_buf) != 0) return -EIO;
    journal_pos = (journal_pos + 1) % js->j_total_blocks;
    /* --- STEP 2: Write each block's data copy to journal --- */
    uint32_t checksum = 0;
    for (uint32_t i = 0; i < txn->nr_blocks; i++) {
        uint32_t data_abs = journal_abs_block(sb, journal_pos);
        if (write_block(fd, data_abs, txn->block_data[i]) != 0) return -EIO;
```

```
    /* Accumulate checksum for commit block */
    const uint32_t *words = (const uint32_t *)txn->block_data[i];
    for (int j = 0; j < (int)(BLOCK_SIZE / sizeof(uint32_t)); j++) {
        checksum += words[j];
    }
    journal_pos = (journal_pos + 1) % js->j_total_blocks;
}
/* --- STEP 3: fsync — all journal writes durable --- */
/*
 * This fsync is MANDATORY and NON-NEGOTIABLE.
 *
 * Without it, the OS may reorder the journal data writes relative to
 * the commit block write. The commit block could reach disk before the
 * descriptor or data blocks, creating a committed-but-unreplayable
 * transaction. The fsync establishes a memory barrier at the hardware level:
 * everything before this call is on stable storage before anything after
 * this call begins writing.
 */
if (fsync(fd) != 0) return -EIO;
/* --- STEP 4: Write commit block --- */
uint8_t commit_buf[BLOCK_SIZE];
memset(commit_buf, 0, BLOCK_SIZE);
journal_commit_t *commit = (journal_commit_t *)commit_buf;
commit->j_magic      = JOURNAL_MAGIC;
commit->j_block_type = JB_TYPE_COMMIT;
commit->j_sequence   = txn->sequence;
commit->j_checksum   = checksum;
uint32_t commit_abs  = journal_abs_block(sb, journal_pos);
if (write_block(fd, commit_abs, commit_buf) != 0) return -EIO;
journal_pos = (journal_pos + 1) % js->j_total_blocks;
/* --- STEP 5: fsync — commit record is durable (POINT OF NO RETURN) --- */
```

```
    /*
     * After this fsync returns, the transaction is committed.
     * If power fails any time after this point, recovery WILL replay this
     * transaction. The commit block on stable storage is the guarantee.
     *
     * If power fails before this fsync completes (including during the fsync),
     * the commit block is not durable. Recovery will not find it. The
     * transaction is treated as aborted. The main data structures are unchanged.
     */
    if (fsync(fd) != 0) return -EIO;
    /* --- STEP 6: Write blocks to their actual filesystem locations --- */
    /*
     * Only now do we update the actual inode table, bitmaps, and directory blocks.
     * The order here does not matter for crash consistency — if we crash during
     * step 6, the journal commit record guarantees recovery will complete the writes.
     * We write in the order they appear in the transaction for simplicity.
     */
    for (uint32_t i = 0; i < txn->nr_blocks; i++) {
        if (write_block(fd, txn->block_nums[i], txn->block_data[i]) != 0) {
            /*
             * This write failed, but the journal is committed.
             * Recovery will re-apply this transaction on next mount.
             * Safe to return error — the filesystem remains consistent.
             */
            return -EIO;
        }
    }
    /* --- STEP 7: fsync — main data structures updated --- */
    /*
     * This fsync is advisory for crash consistency (recovery handles the
     * case where it doesn't happen), but it improves performance by
```

```
     * allowing the journal to be checkpointed sooner: we know the
     * in-place writes completed, so the journal entries are redundant.
     */
    fsync(fd);   /* best-effort; ignore return value */
    /* --- STEP 8: Advance journal tail and sequence number --- */
    js->j_tail     = journal_pos;
    js->j_sequence = txn->sequence + 1;
    journal_persist_super(fd, sb);
    return 0;
}
```

Hardware Soul — Why Two `fsync` Calls? The first `fsync` (step 3) flushes the journal data pages from the kernel's page cache to the disk's write buffer, and instructs the disk to flush its write buffer to stable media. On SSDs, this typically takes 100–500µs depending on the write buffer state. The second `fsync` (step 5) does the same for the commit block. Why two? Because the commit block's durability is the single binary condition that separates "committed" from "not committed." Without the first `fsync`, the disk might write the commit block before the data blocks (internal reordering). With the first `fsync`, we know: the data blocks are on stable media, then the commit block is written. The ordering is established at the hardware level. On an SSD with power-loss protection (enterprise SSDs, NVMe with capacitors), the first `fsync` cost can be eliminated using disk-level barriers — but that requires more complex device capability detection. Two `fsync` calls is the conservative, universally correct approach.

The `txn_journal_block` Wrapper Idiom

Every metadata write in your codebase — every call to `write_inode`, `write_block` for bitmaps, every directory entry modification — must be converted to use `txn_journal_block`. Here is the pattern:

```c
/*
 * Journaled wrapper for write_inode.
 * Instead of writing to the inode table immediately, record the
 * intended change in the active transaction.
 *
 * Caller must have an active txn_t and must call txn_commit() at the
 * end of the operation.
 */
int txn_write_inode(txn_t *txn, uint32_t inode_num, const inode_t *inode) {
    superblock_t *sb = txn->sb;
    if (inode_num == 0 || inode_num > sb->total_inodes) return -EINVAL;
    uint32_t idx           = inode_num - 1;
    uint32_t block_num     = sb->inode_table_start + idx / INODES_PER_BLOCK;
    uint32_t slot_in_block = idx % INODES_PER_BLOCK;
    /* Read the current inode block content */
    uint8_t buf[BLOCK_SIZE];
    int r = read_block(txn->fd, block_num, buf);
    if (r != 0) return r;
    /* Apply the change in memory */
    memcpy(buf + slot_in_block * INODE_SIZE, inode, INODE_SIZE);
    /* Record in transaction — no disk write yet */
    return txn_journal_block(txn, block_num, buf);
}
/*
 * Journaled wrapper for block bitmap update.
 * Records the bitmap change in the transaction.
 */
int txn_write_block_bitmap(txn_t *txn, uint8_t *bitmap) {
    return txn_journal_block(txn, BLOCK_BITMAP_BLOCK, bitmap);
}
/*
```

```
 * Journaled wrapper for inode bitmap update.
 */
int txn_write_inode_bitmap(txn_t *txn, uint8_t *bitmap) {
    return txn_journal_block(txn, INODE_BITMAP_BLOCK, bitmap);
}
```

Here is what `fs_create_file` looks like when converted to use journaling:

```c
/*
 * Journaled version of fs_create_file.
 * Wraps all metadata changes in a single atomic transaction.
 *
 * If ANY step fails before txn_commit(), the transaction is abandoned.
 * No partial changes will appear in the filesystem.
 * If the process crashes after txn_commit() but before the in-place
 * writes complete, recovery will replay the transaction on next mount.
 */
int fs_create_file_journaled(int fd, superblock_t *sb,
                             uint32_t parent_ino, const char *name,
                             uint16_t mode, uint16_t uid, uint16_t gid) {
    txn_t txn;
    txn_begin(&txn, fd, sb);
    /* --- 1. Validate parent --- */
    inode_t parent_inode;
    if (read_inode(fd, sb, parent_ino, &parent_inode) != 0) return -EIO;
    if (!S_ISDIR(parent_inode.mode)) return -ENOTDIR;
    if (dir_lookup(fd, sb, &parent_inode, name) > 0) return -EEXIST;
    /* --- 2. Allocate inode (in-memory only for now) --- */
    /* Read inode bitmap, find free slot, mark used */
    uint8_t ibmap[BLOCK_SIZE];
    if (read_block(fd, INODE_BITMAP_BLOCK, ibmap) != 0) return -EIO;
    int new_ino = bitmap_find_free(ibmap, sb->total_inodes);
    if (new_ino < 0) return -ENOSPC;
    bitmap_set(ibmap, (uint32_t)new_ino);
    /* Journal the bitmap change — not written to disk yet */
    txn_write_inode_bitmap(&txn, ibmap);
    uint32_t new_ino_num = (uint32_t)new_ino + 1;  /* 1-based */
    /* --- 3. Initialize new inode --- */
    inode_t new_inode;
```

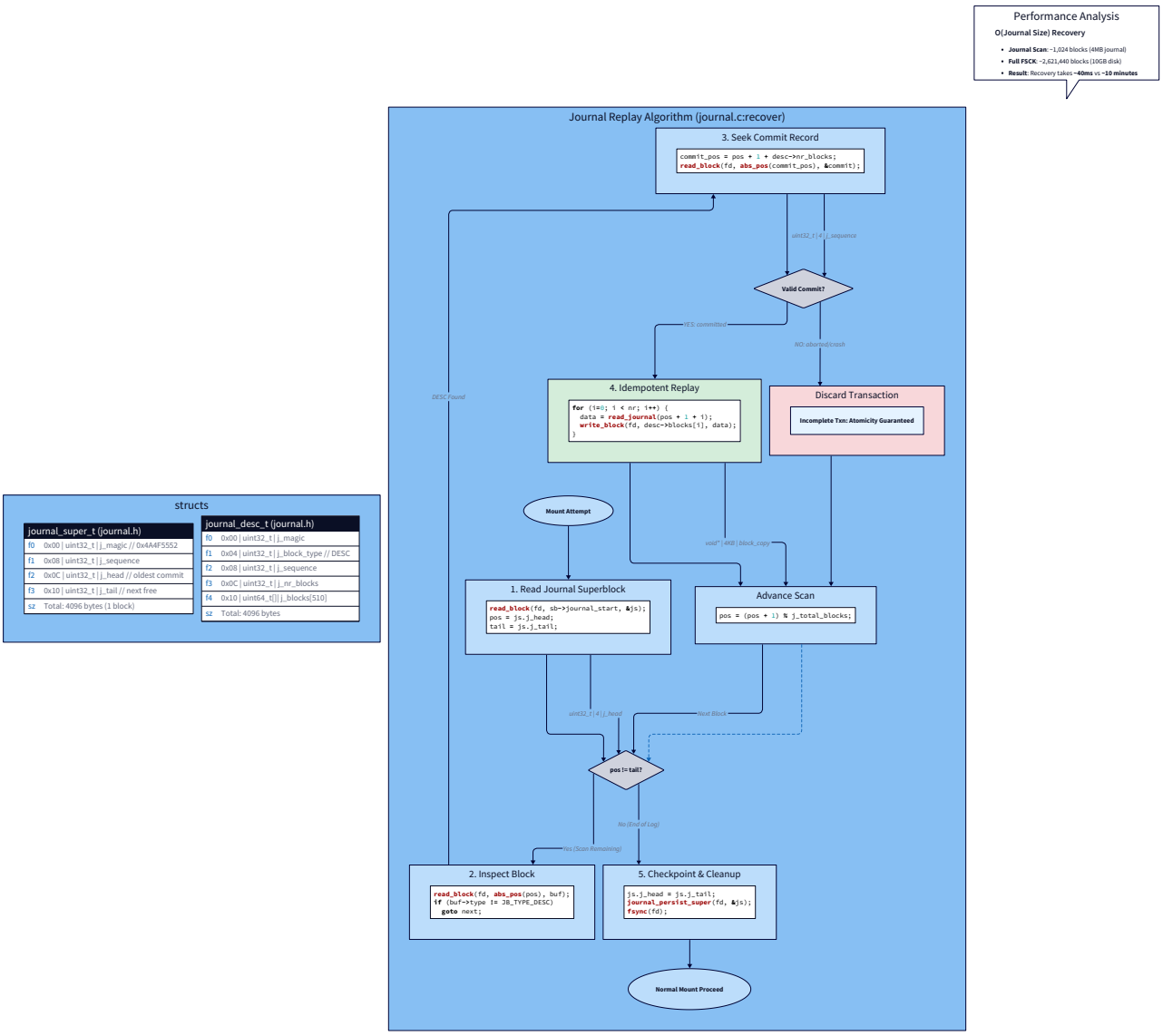
```
    memset(&new_inode, 0, sizeof(new_inode));
    new_inode.mode   = (uint16_t)(S_IFREG | (mode & 0777));
    new_inode.uid    = uid;
    new_inode.gid    = gid;
    new_inode.nlinks = 1;
    new_inode.size   = 0;
    uint32_t now = (uint32_t)time(NULL);
    new_inode.atime = new_inode.mtime = new_inode.ctime = now;
    /* Journal the new inode */
    txn_write_inode(&txn, new_ino_num, &new_inode);
    /* --- 4. Add directory entry --- */
    /*
     * dir_add_entry modifies the parent directory's data blocks.
     * We need to capture the modified block and journal it.
     * In a full implementation, dir_add_entry would take a txn_t parameter.
     * For clarity, we show the pattern inline here.
     */
    /* ... (see full implementation with txn-aware dir_add_entry below) ... */
    /* --- 5. Update parent inode mtime/ctime --- */
    parent_inode.mtime = parent_inode.ctime = now;
    txn_write_inode(&txn, parent_ino, &parent_inode);
    /* --- 6. Update superblock free counts --- */
    sb->free_inodes--;
    /* Journal the superblock block (block 0) */
    uint8_t sb_buf[BLOCK_SIZE];
    memset(sb_buf, 0, BLOCK_SIZE);
    memcpy(sb_buf, sb, sizeof(*sb));
    txn_journal_block(&txn, 0, sb_buf);
    /* --- 7. COMMIT: write journal + fsync twice + write in place + fsync --- */
    int r = txn_commit(&txn);
    if (r != 0) {
```

```
        /* Commit failed. Rollback the in-memory sb free count. */
        sb->free_inodes++;
        return r;
    }
    return (int)new_ino_num;
}
```

The key structural change: every operation now builds up a set of block changes in `txn`, then commits them all atomically. If any step fails before `txn_commit`, the transaction is simply abandoned — `txn` goes out of scope, nothing was written to disk, the filesystem is unchanged.

Journal Recovery: Replaying on Mount

[[EXPLAIN:write-ahead-log-replay-—-scanning-the-log-and-re-applying-committed-operations|Write-ahead log replay — scanning committed journal entries and re-applying operations to primary data structures]] Recovery happens at mount time, before the filesystem is made available for use. The algorithm:




```
1. Read journal superblock → get j_head and j_tail
2. Scan from j_head toward j_tail:
   a. Read block at current position
   b. Check j_magic — if invalid, stop (journal is empty past here)
   c. If JB_TYPE_DESC: remember the descriptor (block list + sequence)
   d. Read the following data blocks (one per entry in descriptor)
   e. Look for JB_TYPE_COMMIT with matching j_sequence
   f. If commit found: REPLAY — write each data block to its target
   g. If commit NOT found before hitting another descriptor or invalid block:
      DISCARD — skip to next descriptor or stop
3. After all committed transactions replayed: checkpoint (mark journal clean)
```

```c
/*
 * Replay the journal on mount.
 *
 * Scans all journal blocks from j_head to j_tail.
 * Applies committed transactions. Discards incomplete transactions.
 *
 * After this function returns, the filesystem is in a consistent state
 * matching the most recent committed transaction.
 *
 * Returns 0 on success (including "no recovery needed"), negative errno on error.
 */
int journal_recover(int fd, superblock_t *sb) {
    journal_super_t *js = &g_jsuper;
    if (js->j_head == js->j_tail) {
        /* Journal is empty — nothing to replay */
        printf("journal_recover: journal is clean, no replay needed\n");
        return 0;
    }
    printf("journal_recover: replaying journal (head=%u, tail=%u)\n",
           js->j_head, js->j_tail);
    uint32_t pos = js->j_head;
    int replayed = 0;
    while (pos != js->j_tail) {
        uint32_t abs_block = journal_abs_block(sb, pos);
        uint8_t buf[BLOCK_SIZE];
        if (read_block(fd, abs_block, buf) != 0) {
            fprintf(stderr, "journal_recover: I/O error at journal offset %u\n", pos);
            return -EIO;
        }
        /* Examine the block type */
        uint32_t *magic = (uint32_t *)buf;
```

```
        uint32_t *btype = (uint32_t *)(buf + 4);
        if (*magic != JOURNAL_MAGIC) {
            /* No more valid journal entries */
            printf("journal_recover: end of valid entries at offset %u\n", pos);
            break;
        }
        if (*btype != JB_TYPE_DESC) {
            /* Not a descriptor — could be a stale commit or other block; skip */
            pos = (pos + 1) % js->j_total_blocks;
            continue;
        }
        /* Found a descriptor block */
        journal_desc_t *desc = (journal_desc_t *)buf;
        uint32_t seq          = desc->j_sequence;
        uint32_t nr_blocks    = desc->j_nr_blocks;
        if (nr_blocks > MAX_BLOCKS_PER_DESC || nr_blocks == 0) {
            fprintf(stderr, "journal_recover: corrupt descriptor (nr_blocks=%u)\n",
                    nr_blocks);
            break;
        }
        /* Read the data blocks that follow the descriptor */
        uint32_t data_pos = (pos + 1) % js->j_total_blocks;
        uint8_t data_bufs[MAX_TXNBLOCKS][BLOCK_SIZE];
        uint32_t checksum_computed = 0;
        for (uint32_t i = 0; i < nr_blocks; i++) {
            uint32_t data_abs = journal_abs_block(sb, data_pos);
            if (read_block(fd, data_abs, data_bufs[i]) != 0) {
                fprintf(stderr, "journal_recover: I/O error reading data block %u\n", i);
                return -EIO;
            }
            const uint32_t *words = (const uint32_t *)data_bufs[i];
```

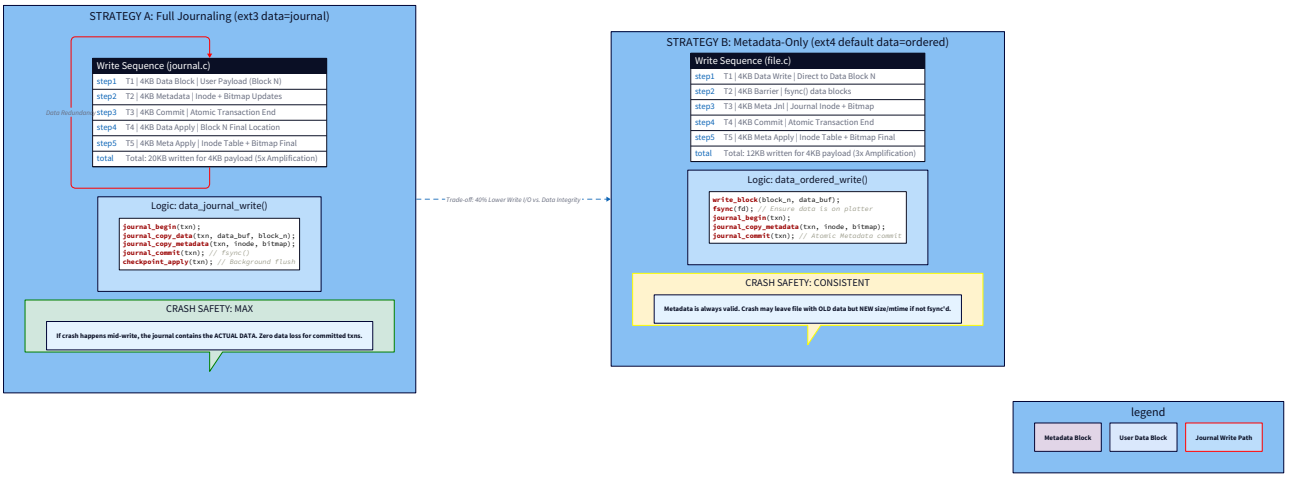
```
        for (int j = 0; j < (int)(BLOCK_SIZE / sizeof(uint32_t)); j++) {
            checksum_computed += words[j];
        }
        data_pos = (data_pos + 1) % js->j_total_blocks;
    }
    /* Look for the commit block at data_pos */
    uint32_t commit_abs = journal_abs_block(sb, data_pos);
    uint8_t commit_buf[BLOCK_SIZE];
    if (read_block(fd, commit_abs, commit_buf) != 0) {
        fprintf(stderr, "journal_recover: I/O error reading commit block\n");
        return -EIO;
    }
    journal_commit_t *commit = (journal_commit_t *)commit_buf;
    if (commit->j_magic      != JOURNAL_MAGIC ||
        commit->j_block_type != JB_TYPE_COMMIT ||
        commit->j_sequence   != seq) {
        /*
         * No valid commit block for this transaction.
         * The transaction was incomplete at crash time — discard it.
         *
         * This is the correct behavior: a missing commit record means
         * the transaction never durably completed. We act as if the
         * operation never started. The filesystem metadata below is
         * unchanged from the last committed transaction.
         */
        printf("journal_recover: incomplete txn seq=%u discarded\n", seq);
        pos = (pos + 1) % js->j_total_blocks;
        continue;
    }
    /*
     * Validate checksum.
```

```
         * A checksum mismatch means either the data blocks are corrupt
         * or the commit block belongs to a different transaction (false match).
         */
        if (commit->j_checksum != checksum_computed) {
            fprintf(stderr, "journal_recover: checksum mismatch in txn seq=%u "
                    "(expected 0x%08X, got 0x%08X) — discarding\n",
                    seq, commit->j_checksum, checksum_computed);
            pos = data_pos + 1;
            continue;
        }
        /*
         * REPLAY: write each journaled block to its filesystem location.
         *
         * Idempotency: if we crash DURING replay and replay is run again,
         * these writes are safe to apply multiple times. Each write is
         * exactly the same data written to the same block. Replaying
         * a transaction twice produces the same result as replaying it once.
         * This is the idempotency guarantee that makes WAL replay correct.
         */
        printf("journal_recover: replaying committed txn seq=%u (%u blocks)\n",
               seq, nr_blocks);
        for (uint32_t i = 0; i < nr_blocks; i++) {
            uint32_t target_block = (uint32_t)desc->j_blocks[i];
            if (write_block(fd, target_block, data_bufs[i]) != 0) {
                fprintf(stderr, "journal_recover: failed to write block %u\n",
                        target_block);
                return -EIO;
            }
        }
        replayed++;
        /* Advance past this transaction's descriptor + data + commit */
```

```
        pos = (data_pos + 1) % js->j_total_blocks;
    }
    /* Flush replayed writes to disk */
    if (replayed > 0) {
        if (fsync(fd) != 0) return -EIO;
        printf("journal_recover: replayed %d transactions\n", replayed);
    }
    /* Checkpoint: mark journal as clean */
    return journal_checkpoint(fd, sb);
}
```

Metadata-Only Journaling

[[EXPLAIN:journaling-modes-—-metadata-only-vs-full-data-journaling-tradeoff|Journaling modes — the tradeoff between metadata-only journaling (ext3 default) and full data journaling (higher safety, higher write amplification)]] Your journal records metadata changes: inode table blocks, bitmap blocks, directory data blocks. It does **not** journal file data blocks. This is the **metadata journaling** mode (called "ordered" mode in ext3/ext4 terminology). The distinction matters enormously for performance:



Mode	What's journaled	Write amplification	Crash behavior
Metadata only (chosen ✓)	Inodes, bitmaps, directories	2–3× for metadata, 1× for data	Data written before crash may be lost; metadata is always consistent
Full journaling	Metadata + file data	2× for all writes	All committed writes survive crash
Writeback	Metadata only, no ordering	~2× for metadata	Metadata consistent; data blocks may contain garbage
Metadata-only journaling with ordered data writes (the safe default) means: before journaling the inode that points to a data block, ensure the data block has been written to disk. The protocol for a file write operation is:			

1. Write file data blocks to their final locations (no journal)
2. `fsync(fd)` — data blocks are durable
3. Begin transaction
4. Journal inode update (new size, new block pointers)
5. Commit transaction
6. `fsync(fd)` — inode update is durable If you crash between steps 1 and 5: the data exists on disk but the inode does not point to it. From the filesystem's perspective, the write never happened. The file has its old size and old content. Data is lost but the filesystem is consistent. If you crash after step 5: the inode points to the written data. The write is committed. Full durability.

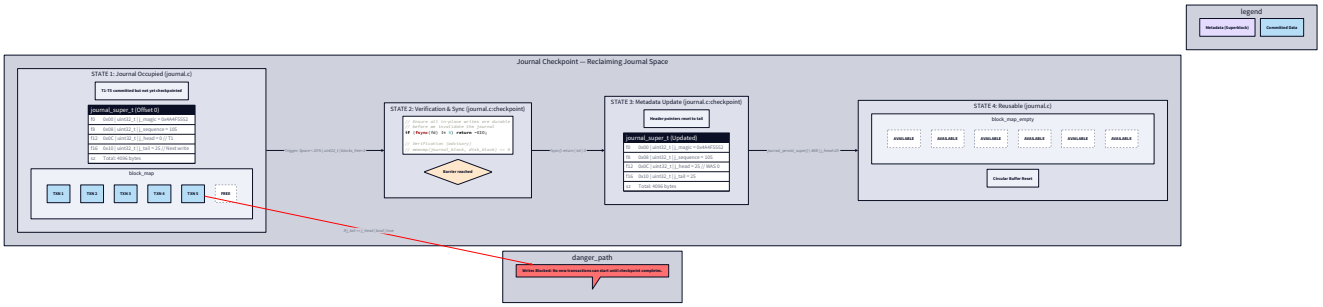
```c
/*
 * Journaled file write — ordered metadata journaling mode.
 *
 * Protocol:
 *   1. Write data blocks to final locations (not journaled)
 *   2. fsync — data is durable
 *   3. Journal inode update (new size + any new block pointer changes)
 *   4. Commit journal transaction
 *
 * This ensures: if crash between 1 and 3, old inode is intact (old file content).
 * If crash after 4, inode reflects new content.
 */
int fs_write_journaled(int fd, superblock_t *sb,
                       uint32_t ino, uint64_t offset,
                       const void *buf, size_t length) {
    /* Step 1 & 2: Write data blocks directly (not journaled) */
    ssize_t written = fs_write(fd, sb, ino, offset, buf, length);
    if (written < 0) return (int)written;
    /* fsync data before journaling the metadata */
    if (fsync(fd) != 0) return -EIO;
    /* Step 3 & 4: Journal the inode metadata update */
    txn_t txn;
    txn_begin(&txn, fd, sb);
    inode_t inode;
    if (read_inode(fd, sb, ino, &inode) != 0) return -EIO;
    /* The inode was already updated in-place by fs_write above.
     * Re-read and journal the current (updated) inode block. */
    uint32_t idx       = ino - 1;
    uint32_t block_num = sb->inode_table_start + idx / INODES_PER_BLOCK;
    uint8_t  iblock[BLOCK_SIZE];
    if (read_block(fd, block_num, iblock) != 0) return -EIO;
```

```
    txn_journal_block(&txn, block_num, iblock);

    return txn_commit(&txn);

}
```

> **Why Not Journal Data Too?** Full journaling doubles the write load for all file writes — every byte written to a file is written twice: once to the journal, once to the final location. For a filesystem serving a 1Gbps network storage workload, full journaling would halve effective write throughput and double the SSD write endurance consumption. Metadata-only journaling adds overhead only for metadata operations (which are small and infrequent relative to data writes), while keeping data writes on the fast path. ext4's default mode is "ordered" — metadata journaled, data written to disk before the journal transaction commits. This is what you implemented above.

Journal Checkpointing

A checkpoint marks journal entries as no longer needed — their changes have been applied to the main data structures. Without checkpointing, the circular journal buffer fills up and all filesystem operations block.



```c
/*
 * Checkpoint the journal.
 *
 * A checkpoint declares: all committed transactions have been applied
 * to the main data structures. The journal can now be reused from the
 * beginning (j_head reset to j_tail, effectively clearing the journal).
 *
 * When to checkpoint:
 *   - When the journal is getting full (< 20% free)
 *   - On unmount (ensure all changes are in main structures)
 *   - Periodically by a background thread (every N seconds)
 *
 * After checkpoint, recovery from a crash replays no transactions —
 * the journal is empty. This is the desired post-checkpoint state.
 *
 * Returns 0 on success, negative errno on failure.
 */
int journal_checkpoint(int fd, superblock_t *sb) {
    journal_super_t *js = &g_jsuper;
    /*
     * All committed transactions have already been applied to the main
     * data structures by txn_commit() step 6. The journal entries are
     * now redundant. We just need to fsync to ensure the in-place writes
     * are durable, then reset the journal head to the tail.
     */
    /* Ensure all in-place writes from prior transactions are on disk */
    if (fsync(fd) != 0) return -EIO;
    /*
     * Reset journal: head catches up to tail.
     * The journal is now empty — no transactions to replay.
     * On next crash recovery, nothing will be replayed.
```

```
     */
    js->j_head = js->j_tail;
    /*
     * Persist the updated journal superblock.
     * This write must be durable before we return — otherwise a crash
     * immediately after checkpoint could leave j_head pointing to the
     * wrong location, causing spurious transaction replay.
     */
    int r = journal_persist_super(fd, sb);
    if (r != 0) return r;
    if (fsync(fd) != 0) return -EIO;
    printf("journal_checkpoint: journal cleared (sequence=%u)\n",
           js->j_sequence);
    return 0;
}
```

Idempotency: The Safety Net for Replay

[[EXPLAIN:idempotent-operations-—-safe-to-apply-multiple-times-without-changing-the-result|Idempotent operations — an operation where applying it once or N times produces the same result, required for safe retry and replay semantics]] A critical property of WAL replay is that it may apply a transaction multiple times. Consider: the system crashes during replay itself (a "crash during recovery"). When the system reboots again, it replays the same journal entries again. Replaying a committed transaction twice must produce the same result as replaying it once. Your journal writes are idempotent because each journal entry records the **final state** of a block, not a delta. Writing "inode 7's block pointer 0 is now block 2500" twice leaves the inode in exactly the same state as writing it once. This is fundamentally different from a delta log like "increment free_blocks by 1" — replaying that twice would corrupt the free count. The same idempotency requirement appears throughout distributed systems:

```c
/*
 * Verify idempotency in your journaled operations.
 * This test replays a transaction twice and checks that the result
 * is the same as replaying it once.
 */
void test_idempotent_replay(int fd, superblock_t *sb) {
    /* Create a file, capture the transaction */
    int ino = fs_create_file_journaled(fd, sb, sb->root_inode,
                                       "idempotent_test", 0644, 0, 0);
    assert(ino > 0);
    /* Read the committed inode state */
    inode_t inode_after_create;
    read_inode(fd, sb, (uint32_t)ino, &inode_after_create);
    /* Simulate replay: write the same blocks again (as recovery would) */
    /* In a real test, we would parse the journal and re-apply the blocks */
    /* For simplicity: write the inode block to itself twice */
    uint32_t idx       = (uint32_t)ino - 1;
    uint32_t block_num = sb->inode_table_start + idx / INODES_PER_BLOCK;
    uint8_t buf[BLOCK_SIZE];
    read_block(fd, block_num, buf);
    /* First replay */
    write_block(fd, block_num, buf);
    inode_t after_replay1;
    read_inode(fd, sb, (uint32_t)ino, &after_replay1);
    /* Second replay — same data, same target */
    write_block(fd, block_num, buf);
    inode_t after_replay2;
    read_inode(fd, sb, (uint32_t)ino, &after_replay2);
    /* All three must be identical */
    assert(memcmp(&inode_after_create, &after_replay1, sizeof(inode_t)) == 0);
    assert(memcmp(&after_replay1, &after_replay2, sizeof(inode_t)) == 0);
```

```
    printf("OK: replay is idempotent\n");
}
```

Crash Simulation Test

The acceptance criteria require a crash simulation test. You cannot test crash recovery without simulating a crash. The approach: send `SIGKILL` to the filesystem process mid-operation, then remount and verify consistency.

```bash
#!/bin/bash

# crash_test.sh — Verify crash recovery correctness

set -euo pipefail

DISK="./crash_test.img"

MOUNT="./crash_mount"

FS="./myfs"

MKFS="./mkfs"

cleanup() {

    fusermount3 -u "$MOUNT" 2>/dev/null || true

    kill "$FUSE_PID" 2>/dev/null || true

    rm -f "$DISK"

    rmdir "$MOUNT" 2>/dev/null || true

}

trap cleanup EXIT

mkdir -p "$MOUNT"

$MKFS "$DISK" 8192 2048

echo "=== Crash Recovery Test ==="

# Phase 1: Mount and create files

$FS "$DISK" "$MOUNT" -s &

FUSE_PID=$!

sleep 0.5

# Create a directory and several files

mkdir "$MOUNT/important_dir"

echo "critical data 1" > "$MOUNT/important_dir/file1.txt"

echo "critical data 2" > "$MOUNT/important_dir/file2.txt"

# Start a background operation that will be interrupted

(

    for i in $(seq 1 100); do

        echo "data $i" >> "$MOUNT/important_dir/growing_file.txt"

        mkdir -p "$MOUNT/dir_$i" 2>/dev/null || true

    done
```

```
) &

BG_OP_PID=$!

# Let the background operation run briefly, then simulate power failure

sleep 0.1

# CRASH: send SIGKILL (unclean shutdown — no destroy callback)

kill -9 "$FUSE_PID" 2>/dev/null || true

kill -9 "$BG_OP_PID" 2>/dev/null || true

wait "$FUSE_PID" 2>/dev/null || true

wait "$BG_OP_PID" 2>/dev/null || true

# Unmount the broken FUSE endpoint

fusermount3 -u "$MOUNT" 2>/dev/null || true

echo "Simulated crash. Remounting for recovery..."

sleep 0.5

# Phase 2: Mount again — journal_recover() runs automatically

$FS "$DISK" "$MOUNT" -s &

FUSE_PID=$!

sleep 1.0

echo "Recovery complete. Verifying filesystem consistency..."

# Check: files that were fully committed before crash must be intact

CONTENT1=$(cat "$MOUNT/important_dir/file1.txt" 2>/dev/null || echo "MISSING")

CONTENT2=$(cat "$MOUNT/important_dir/file2.txt" 2>/dev/null || echo "MISSING")

[ "$CONTENT1" = "critical data 1" ] && echo "OK: file1.txt intact" \

    || echo "FAIL: file1.txt content wrong: '$CONTENT1'"

[ "$CONTENT2" = "critical data 2" ] && echo "OK: file2.txt intact" \

    || echo "FAIL: file2.txt content wrong: '$CONTENT2'"

# Check: no orphaned inodes (all allocated inodes have at least one directory reference)

# This would require a custom fsck tool; for now, verify basic mount works

ls "$MOUNT" > /dev/null && echo "OK: root directory accessible after recovery"

ls "$MOUNT/important_dir" > /dev/null && echo "OK: important_dir accessible after recovery"

# Check: df reports sane numbers (no negative free block counts)

df "$MOUNT" | grep -v Filesystem | awk '{
```

```
    if ($4 < 0) { print "FAIL: negative free blocks"; exit 1; }
    else { print "OK: block counts are sane"; }
}'
fusermount3 -u "$MOUNT"
wait "$FUSE_PID" 2>/dev/null || true
echo "=== Crash Recovery Test Complete ==="
```

Integrating the Journal Into Mount/Unmount

The journal requires two additions to your FUSE lifecycle from Milestone 5:

```c
/*
 * Updated myfs_init: run journal recovery before serving requests.
 */
static void *myfs_init(struct fuse_conn_info *conn,
                       struct fuse_config *cfg) {
    (void)conn;
    cfg->entry_timeout    = 1.0;
    cfg->attr_timeout     = 1.0;
    cfg->negative_timeout = 0.5;
    cfg->use_ino          = 1;
    /* Load journal superblock */
    if (journal_load(g_fs.fd, &g_fs.sb) != 0) {
        fprintf(stderr, "FATAL: failed to load journal superblock\n");
        /* In production: abort mount. For development: continue with warning. */
    }
    /* Replay committed transactions from any prior crash */
    if (journal_recover(g_fs.fd, &g_fs.sb) != 0) {
        fprintf(stderr, "WARNING: journal recovery encountered errors\n");
        /* Continue mounting — partial recovery is better than no mount */
    }
    return NULL;
}
/*
 * Updated myfs_destroy: checkpoint journal before unmount.
 */
static void myfs_destroy(void *private_data) {
    (void)private_data;
    fs_lock();
    /* Checkpoint: ensure all committed transactions are in main structures */
    journal_checkpoint(g_fs.fd, &g_fs.sb);
    /* Final fsync to ensure checkpoint is durable */
```

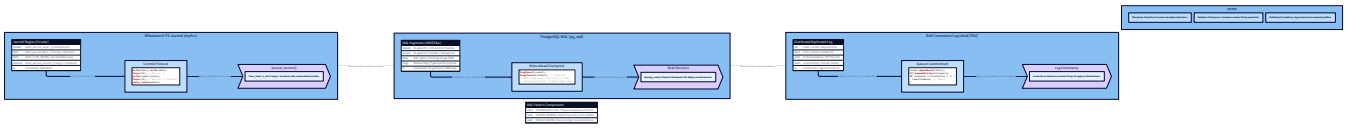
```
    fsync(g_fs.fd);

    close(g_fs.fd);

    pthread_mutex_destroy(&g_fs.lock);

    /* Note: fs_unlock() NOT called after mutex destroy */

}
```

Three-Level View: What Happens During `txn_commit`

Let's trace `txn_commit(&txn)` for a two-block transaction (one inode update, one bitmap update):

Level	What Happens
Application (your code)	Calls `write_block(fd, desc_abs, desc_buf)` — writes 4KB descriptor. Calls `write_block(fd, data1_abs, inode_block)` — writes 4KB inode copy. Calls `write_block(fd, data2_abs, bitmap_block)` — writes 4KB bitmap copy. Calls `fsync(fd)` . Calls `write_block(fd, commit_abs, commit_buf)` . Calls `fsync(fd)` . Then writes inode block and bitmap block to actual locations. Final `fsync` .
OS / Kernel	First three `write_block` calls: data goes into kernel page cache (dirty pages). `fsync(fd)` : kernel submits all dirty pages for `fd` to the block layer in order, issues a FLUSH CACHE command to the disk. Waits for disk to confirm. Returns. Second set of `write_block` calls: page cache again. Second `fsync` : same sequence. Total: 5 dirty-page writes to page cache + 2 FLUSH CACHE commands to disk + 2 disk acknowledgments.
Hardware	First `fsync` : disk receives write commands for descriptor + data1 + data2 blocks. Disk schedules them (possibly reorders for efficiency within this batch). Disk flushes its write buffer to NAND (SSD) or commits rotational writes to platter (HDD). Disk sends ACK. The FLUSH CACHE command is the barrier: everything before it is durable before anything after it starts. Second `fsync` : same for commit block. The commit block is now on stable media — the transaction is committed.
The critical asymmetry: within each `fsync` group, the disk may reorder writes freely. But no write in the second group (commit) can reach stable storage before the first `fsync` completes. That ordering at the `fsync` boundary is the entire mechanism.	

Knowledge Cascade: One Journal, Ten Domains



Database WAL: The Identical Pattern

You just built the same crash recovery mechanism that every production database uses. The structural identity is exact: **PostgreSQL WAL**: PostgreSQL writes changes to WAL segments before applying them to heap pages. Each WAL record has a Log Sequence Number (LSN), analogous to your `j_sequence`. A "checkpoint" in PostgreSQL writes all dirty buffer pool pages to disk and records the checkpoint LSN — analogous to your `journal_checkpoint`. Recovery after a crash replays WAL from the last checkpoint LSN. The `pg_wal` directory in every PostgreSQL data directory is your journal region. **SQLite WAL mode**: SQLite's WAL file contains uncommitted changes. Readers see the database file (clean) plus any committed WAL entries. Writers append to the WAL. A checkpoint copies WAL entries to the main database file. The WAL `shm` (shared memory) file tracks which WAL frames are committed — analogous to your journal superblock's head/tail pointers. **MySQL InnoDB redo log**: InnoDB's `ib_logfile0` and `ib_logfile1` form a circular log (exactly like your journal region with wrap-around). The "log sequence number" is a monotonic 64-bit counter. Crash recovery reads from the last checkpoint LSN to the current write position and replays all redo log entries. The "fuzzy checkpoint" mechanism flushes dirty pages asynchronously and advances the checkpoint LSN — your `journal_checkpoint` in graduated form. The concepts are identical at every level: transaction, sequence number, commit record, head pointer, tail pointer, replay, checkpoint. You have built the foundation that makes these systems' crash consistency possible.

Redis AOF: Durability vs. Performance

Redis Append-Only File (AOF) is a write-ahead log for an in-memory database. Every write command (`SET` , `LPUSH` , `ZADD`) is appended to the AOF file before the in-memory data structure is modified. On restart, Redis replays the AOF from the beginning to reconstruct state. The AOF has three durability modes: `always` (fsync after every command — maximum durability, lowest performance), `everysec` (fsync once per second — at most one second of data loss on crash), and `no` (let the OS flush — maximum performance, OS controls durability). These map directly to the tradeoff your journal exposes:

- `always` = your protocol with `fsync` after every transaction
- `everysec` = batch transactions, fsync once per second (amortize fsync cost across many operations)
- `no` = skip fsync (fast but loses data on crash) The `everysec` mode trades one second of potential data loss for dramatically higher throughput — relevant whenever the cost of the `fsync` dominates transaction latency. For your filesystem, you could implement a similar optimization: buffer multiple transactions in the journal and fsync once per 100ms, accepting that a crash loses at most 100ms of committed transactions.

Raft and Distributed WAL

[[EXPLAIN:raft-consensus-algorithm-—-replicated-state-machine-using-a-replicated-log|Raft consensus algorithm — achieving crash-consistent agreement across multiple nodes by replicating a write-ahead log]] Single-node WAL is the foundation for understanding distributed consensus. The Raft consensus algorithm — used in etcd (Kubernetes' database), CockroachDB, TiKV, and Consul — is fundamentally a distributed WAL with the additional constraint that the commit record requires a quorum of nodes to acknowledge receipt. In Raft:

- Your `txn_begin` = Raft leader receives a client request
- Your `txn_journal_block` = Raft leader appends the log entry (with index and term)
- Your first `fsync` = the leader flushes the log to its local disk AND sends AppendEntries RPC to followers

- The "majority acknowledgment" = equivalent to your commit `fsync` — the transaction cannot be lost because a majority of nodes have it durably
- Your `apply` (writing blocks to final locations) = Raft's "apply to state machine" (updating key-value store, etc.) The crash recovery protocol in Raft mirrors your journal recovery: a new leader scans committed log entries and re-applies any that may not have reached the state machine. The idempotency requirement is identical: applying the same log entry twice must produce the same state as applying it once. Understanding your single-node journal makes Raft's log replication intuitive rather than mysterious. The "distributed" part adds leader election and quorum logic; the "WAL" part is exactly what you built.

`fsync` and the Durability API

Understanding the journal explains every `fsync`-related performance complaint you will ever encounter. [[EXPLAIN:fsync-vs-fdatasync-—-the-difference-between-flushing-metadata-and-data|fsync vs. fdatasync — fsync flushes both data and metadata (file size, modification time); fdatasync flushes only data, skipping metadata unless required for correct reads]] The classic complaint: "why is my database write throughput only 1,000 writes/second on an SSD rated for 100,000 IOPS?" The answer: the database calls `fsync` (or `fdatasync`) after every committed transaction to ensure durability. Each `fsync` requires a FLUSH CACHE command to the SSD, which takes 100µs–1ms depending on the device and its write buffer state. At 1ms per fsync, maximum throughput is 1,000 fsyncs/second. The SSD's rated IOPS don't matter — the bottleneck is the barrier cost, not the raw write speed. Solutions used in production:

- **Group commit**: buffer multiple transactions, fsync once. PostgreSQL's `synchronous_commit = on` with multiple concurrent writers naturally groups their commits into one WAL flush.
- **Write-ahead log on separate device**: put the WAL on a dedicated SSD with power-loss protection capacitors (enterprise SSDs). These devices guarantee that all data in the write buffer survives a power failure, so `fsync` returns immediately after the buffer accepts the write.
- **Battery-backed write cache**: RAID controllers with BBU (Battery Backup Unit) can acknowledge `fsync` as soon as data enters the cache — the battery ensures the cache survives power loss long enough to flush to disk.
- **Ordered writes without fsync** (dangerous): some applications skip `fsync` entirely and rely on the OS page cache writeback ordering being "good enough." This works until it doesn't — any crash during a multi-block operation can corrupt the data structure. This is what your journal was invented to prevent.

Copy-on-Write: Eliminating Journaling's Complexity

Understanding why journaling is complex motivates understanding why ZFS and Btrfs chose a completely different architecture. In a copy-on-write (CoW) filesystem, a write to a block never modifies the existing block. Instead:

1. Allocate a new block
2. Write new content to the new block
3. Update the parent pointer (in the inode or B-tree node) to point to the new block
4. The old block is now unreferenced — free it The atomic operation is step 3: a single pointer update. If the system crashes before the pointer update, the old block is still intact (the old pointer still points to it). If it crashes after, the new pointer is valid. The intermediate state — "new block written but pointer not yet updated" — is always safe: the new block is unreferenced, and the old block is still the live version. CoW extends this: updating the inode requires writing a new inode block (step 3 for the parent of the block pointer). Updating the inode block requires updating the inode table block. This cascades all the way to the filesystem's root block — called the "tree root" or "uberblock" in ZFS. The final atomic write is the uberblock update: a single 4KB write that atomically commits the entire set of changes. If the uberblock write succeeds, the entire transaction is committed. If it fails, the old uberblock is still valid. The price: **every write fragments the filesystem**, because writes never go to the same location twice. Over time, a CoW filesystem's data becomes highly fragmented. ZFS handles this with its pool-level allocation, spread across many disks. Btrfs requires periodic `btrfs balance` runs to defragment. The CoW design trades journaling complexity for fragmentation management complexity — a different tradeoff, not a free lunch. Now you

understand WHY both approaches exist: journaling gives you the performance of in-place writes at the cost of maintaining a separate write path (the journal); CoW eliminates the journal at the cost of fragmentation and more complex space management. Neither is universally superior.

fsck: The Pre-Journal Alternative

Before journaling, the only recovery mechanism was `fsck` (filesystem check). On an unclean mount (power failure, crash), `fsck` would scan every inode in the inode table, every directory block, and every bitmap block to verify consistency. It would detect:

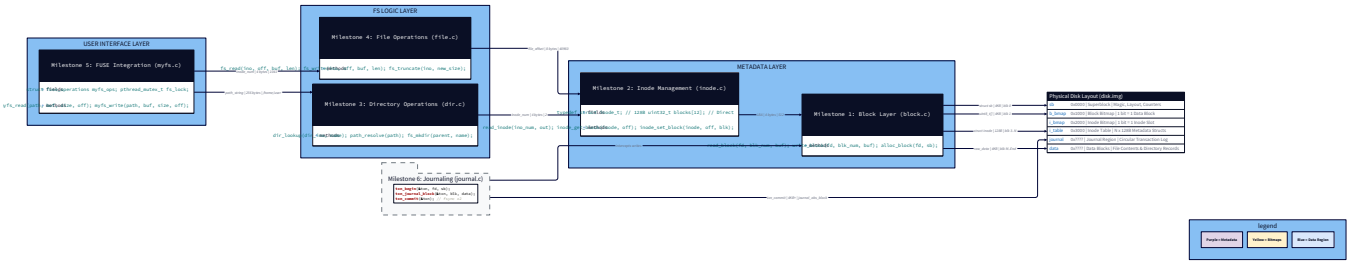
- Inode with nlinks > 0 but no directory entry referencing it (orphaned inode) → move to `lost+found`
- Block marked used in bitmap but not referenced by any inode (leaked block) → mark free
- Block referenced by an inode but marked free in bitmap (double allocation) → dangerous — usually requires user intervention
- Directory entry referencing an invalid inode number → remove the entry The time complexity of `fsck` is O(filesystem size) — it must read every metadata block. On a 10TB filesystem at 500MB/s: 20TB of metadata reads ÷ 500MB/s = ~11 hours minimum. In practice, because metadata is scattered (bitmaps are sequential but inode table and directory blocks are random-access), the actual time is much longer on spinning disks due to seek overhead. Journaling reduces recovery to O(journal size). The journal is typically 64MB–1GB, readable in under a second on modern SSDs. This is the quantitative argument for journaling: not correctness (fsck achieves correctness eventually), but the difference between 10ms recovery and 10 hours recovery. Your `journal_recover` function is the alternative to running `fsck` after every unclean mount. The guarantee: after `journal_recover` completes, the filesystem is in exactly the state it would be in if the committed transactions had been applied cleanly. No scanning, no heuristics, no `lost+found`.

Before You Move On: Pitfall Checklist

- [ ] **Two `fsync` calls per commit, in order**: First `fsync` after journal data writes (before commit block). Second `fsync` after commit block write. Remove either one and the crash consistency guarantee is broken. Verify by adding a `sleep(1)` between the first and second `fsync` and simulating a crash — you should see the transaction treated as aborted on recovery.
- [ ] **Commit record in one atomic write**: The commit block must be written in a single `write_block` call (one 4KB block write). A commit record that spans multiple block writes is not atomic — a crash mid-write produces a partial commit record that your recovery code must detect and discard. Your `journal_commit_t` is 4KB and written atomically. Verify: the magic number check at the start of the block is your signal that the block is a valid, complete commit record.
- [ ] **Idempotent operations — no delta records**: Journal entries record the final state of each block, not deltas ("increment by 1"). Verify: write two identical `txn_journal_block` calls for the same block with the same data. The second call should update the in-transaction copy (deduplication). Replaying the same transaction twice must produce the same filesystem state. Add a test that calls `journal_recover` twice on the same committed journal and checks that the second recovery produces no changes.
- [ ] **Circular journal wrap-around**: The `journal_abs_block` function must correctly wrap around. Test: format a filesystem with `journal_blocks = 16` (very small journal). Write 20 transactions. Verify that the journal pointer wraps and older transactions are correctly checkpointed before being overwritten. An off-by-one in the wrap-around calculation causes either early wrap (losing live journal entries) or failure to wrap (writing past journal bounds into data blocks).
- [ ] **Journal recovery before accepting FUSE requests**: `journal_recover` must complete before `myfs_init` returns. If FUSE starts serving requests while recovery is in progress, operations may observe partially replayed state. The `myfs_init` function is called before FUSE starts the event loop — recovery there is safe and correct.

- [ ] **Checkpoint before journal full**: If the journal fills (no space for a new transaction), your `txn_commit` must either block or force an immediate checkpoint. An undetected full-journal condition causes `txn_commit` to overwrite live journal entries (unrecovered committed transactions) with new transaction data. Add a `blocks_needed > journal_free` check in `txn_commit` and trigger `journal_checkpoint` proactively.
- [ ] **Metadata-only: `fsync` data before journaling inode**: In ordered metadata journaling mode, always write and fsync the file's data blocks BEFORE journaling the inode that points to them. If the inode is journaled first and you crash before the data write, recovery replays the inode update but the data block contains garbage from a previous file. The ordering: data write → data fsync → journal inode update → journal commit.
- [ ] **Do not journal data blocks**: Block bitmap updates and inode table blocks are metadata — journal them. File data blocks (blocks in the data region pointed to by inodes) are not metadata — do not journal them. Journaling data doubles write amplification. Verify: in your transaction's `block_nums` array, no entry should have a value ≥ `sb->data_block_start` unless it is a directory data block (which IS metadata).
- [ ] **Checksum validation in recovery**: Your recovery code validates the commit block's checksum against the data blocks. A corrupted journal entry (partial write of a data block due to torn write) should be detected by checksum mismatch and discarded. Verify: flip a single bit in a journaled data block before writing the commit record, then run recovery and confirm the transaction is discarded rather than replayed with corrupt data.
- [ ] **Superblock update in transactions**: Any operation that modifies `sb->free_blocks` or `sb->free_inodes` must journal the superblock block (block 0) as part of that transaction. An uncommitted superblock update that survives a crash will show incorrect free counts, causing `ENOSPC` errors when space is actually available or allocating blocks that the free count says don't exist.

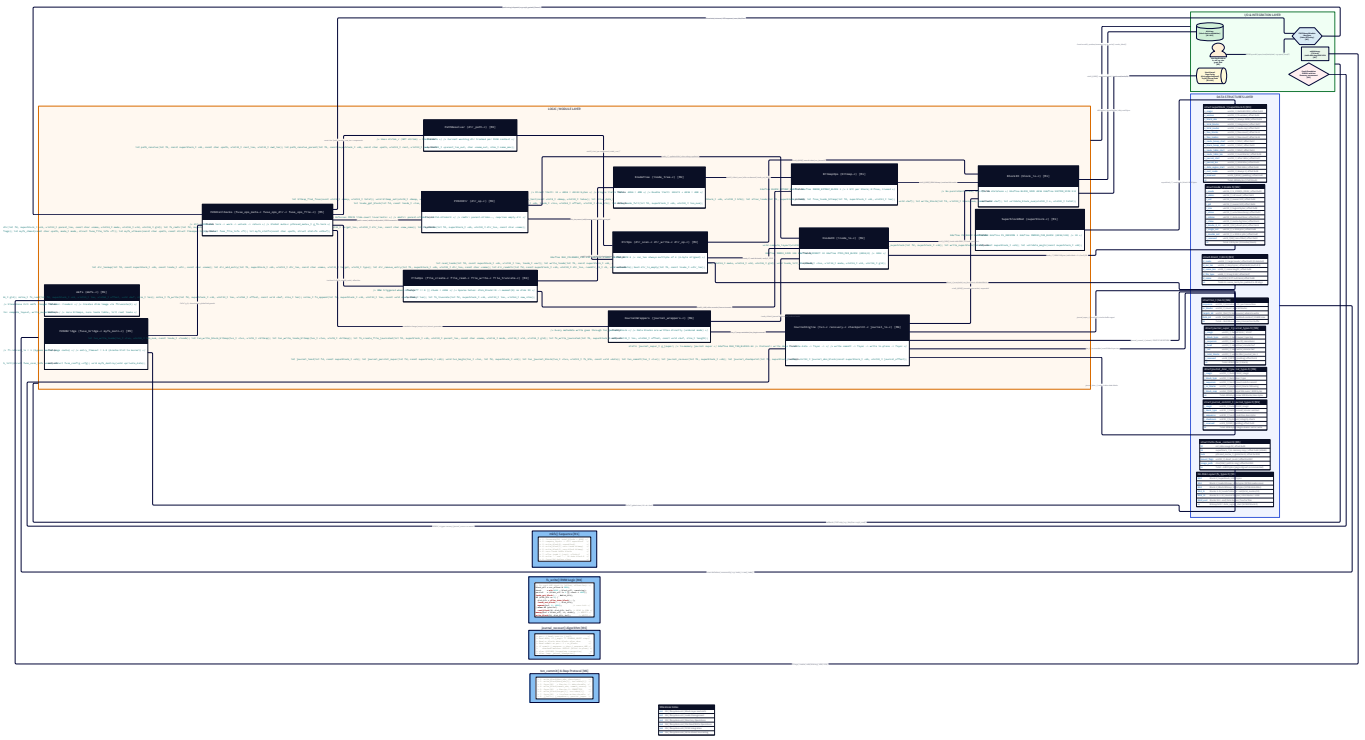
What You've Built and What It Means



You have completed the filesystem. Not a toy — a crash-consistent, journaled filesystem that can be trusted with real data. The chain is complete: raw block device abstraction, bitmap-based allocation, inode metadata with multi-level indirection, directory entries and path resolution, file read/write with sparse file support, FUSE integration for real OS mounting, and now write-ahead journaling for crash consistency. Step back and look at what you understand now that you did not before this milestone. You understand why databases call `fsync`. You understand why skipping `fsync` loses data. You understand why PostgreSQL has WAL segments, why SQLite has a WAL file, why MySQL has redo logs — they are all solving the same problem you just solved: making multi-step metadata updates

atomic despite non-atomic hardware, despite OS write reordering, despite disk write buffer reordering. The journal is not a database invention. It is a universal pattern for crash consistency. You understand why ZFS never needs `fsck`. You understand why ext4's `dir_index` feature requires a journal-aware implementation (the B-tree restructuring touches multiple blocks that must be atomic). You understand why NTFS has a `$LogFile`. You understand why every serious storage system — from SQLite to Kafka to etcd — maintains a sequential write-ahead log as its first and most fundamental data structure. The disk does not know you exist. But now you know how to make it remember exactly what you committed — and nothing more.

System Overview



TDD

A ground-up inode-based filesystem built in six layered milestones, each milestone a self-contained module with explicit hardware constraints, byte-level memory layouts, and crash-consistency guarantees. Every module is designed so an engineer can open it, read the diagrams, and begin implementation without ambiguity. The architecture negotiates the fundamental tension between byte-addressable software abstractions and block-addressable hardware reality at every layer.

Module Technical Design Specification: Block Layer and mkfs (filesystem-m1)

1. Module Charter

The **Block Layer and mkfs** module serves as the foundational substrate for the entire filesystem. Its primary responsibility is to provide a reliable block-level I/O abstraction over a flat file (disk image) and to establish the physical on-disk layout required for higher-level metadata structures.

Scope:

- Implements `read_block` and `write_block` primitives with strict 4KB alignment.
- Defines the `superblock_t` structure and its exact byte-level serialization.
- Implements bit-level allocation tracking (bitmaps) for data blocks and inodes.
- Provides a formatting tool (`mkfs`) to initialize a disk image with a valid root directory.

Out of Scope:

- Multi-level inode pointer traversal (Milestone 2).
- Directory entry management beyond the initial root creation (Milestone 3).
- Write-ahead journaling logic (Milestone 6), though space is reserved.
- In-memory buffer caching (all I/O is direct to/from the host OS page cache).

Invariants:

- Every disk operation must occur in units of exactly 4096 bytes.
- Block 0 is always the Superblock; it can never be allocated for data.
- The block bitmap only tracks blocks in the "Data Region"; bit 0 of the bitmap corresponds to the first block of the data region.
- All on-disk structures must be little-endian and packed (no compiler padding).

---

2. File Structure

The implementation follows a strict creation order to ensure dependencies are met:

1. `fs_types.h` : Constant definitions (Magic numbers, block sizes, region offsets).
2. `block_io.h / block_io.c` : Low-level `lseek/read/write` wrappers.
3. `bitmap.h / bitmap.c` : Bit-level manipulation and search logic.
4. `superblock.h / superblock.c` : Layout calculation and serialization.
5. `mkfs.c` : The standalone CLI tool for image initialization.
6. `test_m1.c` : Verification suite for the block layer.

---

3. Complete Data Model

3.1. Superblock (superblock_t)

The Superblock is the "Birth Certificate" of the filesystem. It resides at byte offset 0.

Offset	Field	Type	Description
0x00	`s_magic`	`uint32_t`	Magic Number (0xDEADC0DE).
0x04	`s_version`	`uint32_t`	Filesystem version (default 1).
0x08	`s_block_size`	`uint32_t`	Always 4096. Used for validation.
0x0C	`s_total_blocks`	`uint32_t`	Total size of the image in blocks.
0x10	`s_total_inodes`	`uint32_t`	Total capacity of the inode table.
0x14	`s_free_blocks`	`uint32_t`	Current count of unallocated data blocks.
0x18	`s_free_inodes`	`uint32_t`	Current count of unallocated inodes.
0x1C	`s_inode_bmap_start`	`uint32_t`	Block index for Inode Bitmap (usually 1).
0x20	`s_block_bmap_start`	`uint32_t`	Block index for Block Bitmap (usually 2).
0x24	`s_inode_table_start`	`uint32_t`	First block of the Inode Table.
0x28	`s_inode_table_len`	`uint32_t`	Number of blocks in the Inode Table.
0x2C	`s_journal_start`	`uint32_t`	First block of the Journal region.
0x30	`s_journal_len`	`uint32_t`	Number of blocks reserved for Journal.
0x34	`s_data_region_start`	`uint32_t`	First block available for file data.
0x38	`s_root_inode`	`uint32_t`	Inode number of the root directory (usually 1).
0x3C	`s_reserved`	`uint8_t[4036]`	Padding to reach exactly 4096 bytes.

3.2. Inode Stub (inode_t)

A minimal definition required for `mkfs` to initialize the root directory. Full implementation occurs in M2.

```c
#define INODE_SIZE 128

#define N_DIRECT 12


typedef struct __attribute__((packed)) {

    uint16_t i_mode;          // 0x00: File type + permissions

    uint16_t i_nlinks;        // 0x02: Hard link count

    uint32_t i_size;          // 0x04: File size in bytes

    uint32_t i_atime;         // 0x08: Access time

    uint32_t i_mtime;         // 0x0C: Modification time

    uint32_t i_ctime;         // 0x10: Metadata change time

    uint32_t i_blocks[N_DIRECT]; // 0x14: Direct block pointers

    uint32_t i_single_ind;    // 0x44: Single indirect pointer

    uint32_t i_double_ind;    // 0x48: Double indirect pointer

    uint8_t  i_pad[52];       // Padding to 128 bytes

} inode_t;
```

4. Interface Contracts

4.1. Block I/O Layer

```c
/**
 * Reads a 4KB block from the disk image.
 * @param fd Open file descriptor to the image.
 * @param block_num Logical block index (0 to s_total_blocks - 1).
 * @param buf Destination buffer (must be at least 4096 bytes).
 * @return 0 on success, -EIO on short read, -EINVAL on bounds error.
 */
int read_block(int fd, uint32_t block_num, void *buf);

/**
 * Writes a 4KB block to the disk image.
 * @param fd Open file descriptor.
 * @param block_num Logical block index.
 * @param buf Source buffer.
 * @return 0 on success, -EIO on short write.
 */
int write_block(int fd, uint32_t block_num, const void *buf);
```

4.2. Bitmap Allocator

```c
/**
 * Allocates a data block from the free block bitmap.
 * @return The absolute disk block number, or -ENOSPC if full.
 * @note Triggers a read-modify-write on the bitmap block.
 */
int alloc_data_block(int fd, superblock_t *sb);

/**
 * Frees a data block and updates the bitmap.
 * @param block_num Absolute disk block address.
 */
int free_data_block(int fd, superblock_t *sb, uint32_t block_num);
```

---

5. Algorithm Specification

5.1. On-Disk Layout Computation

During `mkfs`, the regions must be calculated based on user-provided counts.

1. **Block 0**: Superblock.
2. **Block 1**: Inode Bitmap. Size: 1 block (supports up to 32,768 inodes).
3. **Block 2**: Block Bitmap. Size: 1 block (supports up to 32,768 data blocks).
4. **Inode Table**: Starts at Block 3.
   - `table_blocks = (total_inodes * INODE_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE`.
5. **Journal Region**: Starts at `3 + table_blocks`.
   - Default size: 1024 blocks (4MB).
6. **Data Region**: Starts at `journal_start + journal_len`.
   - Total usable data blocks = `total_blocks - data_region_start`.

5.2. Bitmap Search (`bitmap_find_free`)

To minimize CPU cycles, the search should be byte-aligned before checking bits.

1. Iterate through the 4096 bytes of the bitmap buffer.
2. If `byte[i] == 0xFF`, skip (all 8 bits in this byte are used).
3. If `byte[i] < 0xFF`, find the first zero bit using `__builtin_ctz(~byte[i])` or a manual loop.
4. Calculate `index = i * 8 + bit_position`.
5. Ensure `index < tracked_items_limit`.
6. Return `index`.

5.3. Root Directory Initialization

`mkfs` must create a valid `/` to make the disk mountable.

1. Allocate Inode 1 (Bit 0 in Inode Bitmap).
2. Allocate Data Block 0 (Bit 0 in Block Bitmap).
3. Initialize `inode_t` for root: `mode = S_IFDIR | 0755`, `nlinks = 2`, `size = 4096`, `blocks[0] = s_data_region_start`.
4. Write two directory entries to the first data block: `.` pointing to Inode 1, `..` pointing to Inode 1.
5. Update Superblock `free_blocks` and `free_inodes`.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`EINVAL`	`read_block`	Reject operation; do not seek.	Yes (API return)
`EIO`	`write_block`	Image file might be read-only or disk full.	Yes
`ENOSPC`	`alloc_data_block`	Fail file creation or write.	Yes
Magic Mismatch	`main/mount`	Refuse to mount; protect data.	Yes
Short Write	`write_block`	Attempt `fsync`; if still short, return `EIO`.	Yes

---

7. Implementation Sequence with Checkpoints

Phase 1: Block I/O Bedrock (2 Hours)

Implement `read_block`, `write_block`, and `validate_block_num`. Create a small test utility that opens a file, writes "DEADBEEF" to block 5, and reads it back. **Checkpoint:** `block_test disk.img` successfully writes/reads block 5.

Phase 2: Serialization & Layout (3 Hours)

Define `superblock_t` with `__attribute__((packed))`. Implement `compute_layout`. Write a small tool to print where regions would live for a 100MB disk. **Checkpoint:** `layout_calc 100MB` prints `Data Region starts at block 1059`.

Phase 3: The Allocator (3 Hours)

Implement `bitmap_find_free` and the `alloc/free` wrappers. **Checkpoint:** `test_alloc` allocates 10 blocks, verifies bits are set in the bitmap block, frees 5, and verifies bits are cleared.

Phase 4: mkfs (5 Hours)

Combine all logic into `mkfs.c`. It must:

1. Create the file.
2. Write the Superblock.
3. Zero out the bitmaps and inode table.

4. Manually inject the Root Inode and its first Data Block. **Checkpoint:** `hexdump -C disk.img` shows `DE AD C0 DE` at offset 0 and the string `.` at the start of the data region.

---

8. Test Specification

8.1. Happy Path: Formatting

- **Input**: `mkfs disk.img 4096 1024`
- **Assertion**: File size is exactly 16,777,216 bytes. Superblock fields match calculated layout. Root inode (Inode 1) has `nlinks == 2`.

8.2. Edge Case: Boundary Allocation

- **Input**: Allocate blocks until `alloc_data_block` returns `-ENOSPC`.
- **Assertion**: Number of successful allocations must exactly equal `s_total_blocks - s_data_region_start`.

8.3. Failure Case: Invalid Mount

- **Input**: Point a mock mount function to a text file.
- **Assertion**: Function returns `-EINVAL` because magic number `0xDEADC0DE` is not found at offset 0.

---

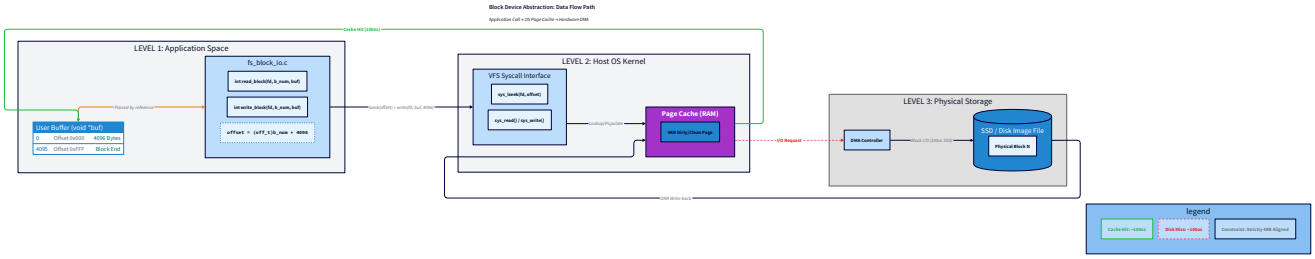
9. Performance Targets

Operation	Target	Measurement
Block Write Latency	< 200µs (SSD)	`clock_gettime` around `write_block`
Bitmap Scan	< 500ns	Benchmark `bitmap_find_free` with 32k bits set
mkfs Throughput	> 100 MB/s	Format a 1GB image; time the process

---

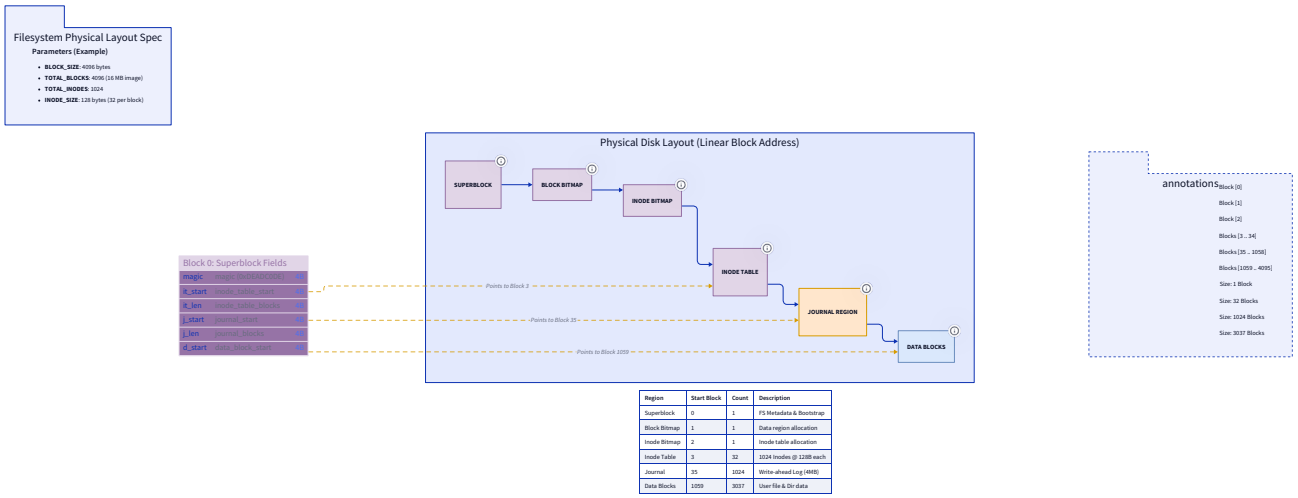
10. Hardware Soul (Cache & Alignment)

1. **Cache Line Alignment**: The `superblock_t` is 4096 bytes. When read into memory, it spans exactly 64 cache lines (assuming 64B lines). The "hot" fields (magic, free counts) are at the start of the first cache line.
2. **TLB Impact**: Because our block size matches the x86-64 page size (4KB), every `read_block` into a page-aligned buffer minimizes TLB misses during the `memcpy` or syscall transition.
3. **Read-Modify-Write**: `alloc_data_block` fetches one 4KB block (the bitmap). This block resides in the host OS Page Cache. Subsequent allocations within the same 32,768-block range will result in Page Cache hits, avoiding physical disk I/O until the kernel flushes dirty pages.



Logical to Physical Mapping in mkfs

```
[Disk Image File]
|
|-- [Block 0: Superblock] (4KB)
|-- [Block 1: Inode Bitmap] (4KB)
|-- [Block 2: Block Bitmap] (4KB)
|-- [Block 3 to N: Inode Table] (Variable)
|-- [Block N+1 to M: Journal] (Default 4MB)
|-- [Block M+1 to End: Data Region] (Usable blocks)
```



Bitmap Bit Manipulation

```
Byte Index = bit_index / 8
Bit Mask   = 1 << (bit_index % 8)

[00001011]  <- Byte 0
 ^   ^  ^
 |   |  bit 0 (Used)
 |   bit 3 (Used)
 bit 7 (Free)
```

Module Technical Design Specification: Inode Management (filesystem-m2)

1. Module Charter

The **Inode Management** module is the structural heart of the filesystem. It defines the identity and data-mapping capability of every file and directory in the system. While the Block Layer (M1) provides raw containers, this module provides the hierarchical pointer machinery required to transform those containers into variable-sized, addressable files.

Scope:

- Definition and serialization of the 128-byte `inode_t` structure.
- Implementation of the multi-level block pointer tree (Direct, Single-Indirect, Double-Indirect).
- Logic for logical-to-physical block translation (`inode_get_block`).
- On-demand allocation of indirect blocks during file growth (`inode_set_block`).
- Atomic-style inode allocation from bitmaps and recursive, bottom-up deallocation of entire pointer trees.
- Enforcement of sparse file semantics where unallocated regions (holes) are represented by null pointers.

Out of Scope:

- Directory entry parsing or path-to-inode resolution (Milestone 3).
- Moving actual file data bytes (Milestone 4).
- Cache management or buffer pooling.

Invariants:

- Inode numbers are 1-based; Inode 0 is strictly reserved as a NULL/Invalid sentinel.
- A block pointer value of `0` always represents a sparse hole; it must never point to the Superblock (Block 0).
- Recursive deallocation must free data blocks before freeing the indirect blocks that point to them to avoid orphaning.
- Any modification to an inode's pointer tree requires an update to the inode's `i_ctime` .

---

2. File Structure

Implementation follows the recursive complexity of the pointer tree:

1. `inode.h` : Definition of `inode_t` , constants for limits, and function prototypes.
2. `inode_io.c` : Serialization logic (`read_inode` , `write_inode`) and table offset math.
3. `inode_alloc.c` : Bitmap integration for `alloc_inode` and basic `free_inode` .
4. `inode_tree.c` : The core traversal engine (`inode_get_block` , `inode_set_block`).
5. `inode_free_recursive.c` : Depth-first traversal for full tree reclamation.
6. `test_m2.c` : Comprehensive test suite verifying 4GB file addressing and sparse holes.

---

3. Complete Data Model

3.1. Inode Structure (inode_t)

Each inode occupies exactly 128 bytes. There are 32 inodes per 4KB block.

Offset	Field	Type	Description
0x00	`i_mode`	`uint16_t`	File type (High 4 bits) + Permissions (Low 12 bits).
0x02	`i_nlinks`	`uint16_t`	Reference count (Hard links). Inode is freed when this is 0.
0x04	`i_uid`	`uint16_t`	Owner User ID.
0x06	`i_gid`	`uint16_t`	Owner Group ID.
0x08	`i_size`	`uint32_t`	Logical file size in bytes (max ~4GB).
0x0C	`i_atime`	`uint32_t`	Access time (Unix timestamp).
0x10	`i_mtime`	`uint32_t`	Data modification time.
0x14	`i_ctime`	`uint32_t`	Inode metadata change time.
0x18	`i_blocks[12]`	`uint32_t[12]`	Direct block pointers (Addresses first 48KB).
0x48	`i_single_ind`	`uint32_t`	Pointer to a block of 1024 pointers (+4MB).
0x4C	`i_double_ind`	`uint32_t`	Pointer to a block of 1024 single-indirect pointers (+4GB).
0x50	`i_reserved`	`uint8_t[48]`	Zero-filled padding to 128 bytes.

Hardware Soul (Alignment):

- **Packed:** The struct uses `__attribute__((packed))` to ensure field offsets are identical across architectures.
- **Cache Line:** Two `inode_t` structs (256B) fit exactly into four 64-byte L1 cache lines.
- **RMW Cycle:** Updating one inode requires reading 4096 bytes (32 inodes), modifying 128, and writing 4096.

3.2. Indirect Block Format

An indirect block is simply a raw 4KB block treated as an array of block addresses.

```c
#define PTRS_PER_BLOCK (BLOCK_SIZE / sizeof(uint32_t)) // 1024 pointers

typedef uint32_t indirect_block_t[PTRS_PER_BLOCK];
```

4. Interface Contracts

4.1. Lifecycle Management

```c
/**
 * Allocate a new inode from the inode bitmap.
 * @return 1-based inode number, or -ENOSPC.
 */
int alloc_inode(int fd, superblock_t *sb);

/**
 * Free an inode and recursively reclaim all associated data and indirect blocks.
 * @param inode_num 1-based index.
 */
int free_inode(int fd, superblock_t *sb, uint32_t inode_num);

/**
 * Initialize an inode struct with default values based on type.
 */
void inode_init(inode_t *inode, uint16_t mode, uint16_t uid, uint16_t gid);
```

4.2. Block Translation Engine

```c
/**
 * Translates a logical file offset to an absolute disk block address.
 * @param offset Byte offset in file.
 * @param out_block Absolute disk block number. 0 indicates a sparse hole.
 * @return 0 on success, -EFBIG if offset exceeds 4GB limit.
 */
int inode_get_block(int fd, const inode_t *inode, uint64_t offset, uint32_t *out_block);

/**
 * Maps a logical file offset to a specific disk block.
 * Allocates indirect blocks if necessary to reach the required depth.
 * @param disk_block Absolute block address to store.
 */
int inode_set_block(int fd, superblock_t *sb, inode_t *inode, uint64_t offset, uint32_t disk_block);
```

---

5. Algorithm Specification

5.1. The Traversal Logic (`inode_get_block`)

The translation follows a branching decision based on the logical block index `bi = offset / 4096`.

1. **Direct (0 <= bi < 12)**:
   - Return `inode->i_blocks[bi]`.
2. **Single Indirect (12 <= bi < 1024 + 12)**:
   - If `inode->i_single_ind == 0`, return `0` (Hole).
   - Read block `inode->i_single_ind`.
   - Return `indirect_block[bi - 12]`.
3. **Double Indirect (1024+12 <= bi < 1024^2 + 1024 + 12)**:
   - If `inode->i_double_ind == 0`, return `0`.
   - Read `inode->i_double_ind` into `buf_outer`.
   - `outer_idx = (bi - 1036) / 1024`.
   - `inner_idx = (bi - 1036) % 1024`.
   - If `buf_outer[outer_idx] == 0`, return `0`.
   - Read `buf_outer[outer_idx]` into `buf_inner`.
   - Return `buf_inner[inner_idx]`.

5.2. Recursive Deallocation (`free_inode`)

To prevent leaks, deallocation must follow a Post-Order Traversal (Bottom-Up).

1. **Level 0**: Loop `i_blocks[0..11]`. If non-zero, `free_data_block`.
2. **Level 1**: If `i_single_ind != 0`:
   - Read block. Loop all 1024 pointers. If non-zero, `free_data_block`.
   - `free_data_block(i_single_ind)`.
3. **Level 2**: If `i_double_ind != 0`:
   - Read `i_double_ind` (Outer).
   - For each `p_outer` in Outer:
     - If `p_outer != 0`:
       - Read `p_outer` (Inner).
       - For each `p_inner` in Inner: if `p_inner != 0`, `free_data_block`.
       - `free_data_block(p_outer)`.
   - `free_data_block(i_double_ind)`.
4. Clear bits in Inode Bitmap and decrement `s_free_inodes`.

5.3. Sparse Allocation (`inode_set_block`)

When writing to a previously unallocated offset, the filesystem must "fill" the path to that block.

1. Calculate target depth (Direct, Single, or Double).
2. If the required indirect block pointer is `0`:
   - `new_block = alloc_data_block(fd, sb)`.
   - Zero out `new_block` (crucial to avoid garbage pointers).
   - Update parent pointer to `new_block`.
3. Descend to next level and repeat until the final data pointer slot is reached.
4. Write the `disk_block` value into the leaf slot.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`EINVAL`	`read_inode`	Validate number range (1 to `total_inodes`).	Yes
`EFBIG`	`inode_get_block`	Prevent seeking/writing beyond ~4GB.	Yes
`ENOSPC`	`inode_set_block`	Allocation of indirect block failed. Fail the write.	Yes
`EIO`	`write_inode`	Device error during RMW cycle.	Yes
Corruption	`inode_get_block`	If a pointer > `total_blocks`, return `EIO` and log.	Yes

---

7. Implementation Sequence with Checkpoints

Phase 1: Serialization & Inode Table Math (3 Hours)

Implement `read_inode` and `write_inode`.

- **Offset Math:** `byte_offset = (sb->s_inode_table_start * 4096) + ((ino-1) * 128)`.
- **Read-Modify-Write:** Ensure `write_inode` reads the 4KB block, modifies the 128B slot, and writes back. **Checkpoint:** Using a hex editor, manually change Inode 5's `i_mode` and verify `read_inode` sees the change.

Phase 2: Inode Lifecycle (3 Hours)

Implement `alloc_inode`, `free_inode` (stubbed for direct blocks), and `inode_init`. **Checkpoint:** Allocate 100 inodes, verify bitmap bits are set, then free them and verify bits are cleared.

Phase 3: The Translation Engine (4 Hours)

Implement `inode_get_block` and `inode_set_block` for Direct and Single-Indirect regions.

- **Math Check:** 48KB boundary is the transition point. **Checkpoint:** Write a block at offset 64KB (Single Indirect), read it back, and verify the physical block address is stored in the indirect block.

Phase 4: Recursive Reclamation (4 Hours)

Implement the full depth-first `free_inode` including double-indirection. **Checkpoint:** Create a 10MB file (Double Indirect). Record `s_free_blocks`. Delete file. Verify `s_free_blocks` returns exactly to the previous value.

---

8. Test Specification

8.1. Sparse File Integrity

1. Create new inode.
2. Call `inode_get_block` at offset 1GB.
3. **Assertion:** `out_block == 0`.
4. Call `inode_set_block` at offset 1GB with block 5000.
5. **Assertion:** `inode->i_double_ind != 0`. `inode_get_block` at offset 500MB still returns 0.

8.2. Address Boundary Transitions

1. Set block at offset 49,151 (Last byte of Direct 11).
2. Set block at offset 49,152 (First byte of Single Indirect).
3. **Assertion:** Block 11 is in `i_blocks[11]`. Block 12 is in the first slot of `i_single_ind`.

8.3. Recursive Cleanup (Leak Detection)

1. Write a file that uses exactly 1 double-indirect block, 2 single-indirect blocks, and 12 direct blocks.
2. Calculate total metadata blocks: 1 + 2 = 3 blocks.
3. Total blocks used = 3 + data blocks.
4. Call `free_inode`.

5. **Assertion:** Superblock `s_free_blocks` increases by exactly `3 + data_count`.

---

9. Performance Targets

Operation	Target	Measurement
Inode Translation (Direct)	O(1) Memory Access	No `read_block` calls.
Inode Translation (Single)	1 Disk I/O	Exactly one `read_block`.
Inode Translation (Double)	2 Disk I/Os	Exactly two `read_block` calls.
Inode Table Density	32 inodes/block	`_Static_assert(sizeof(inode_t) == 128)`.

---

10. Concurrency Specification

While the project is currently single-threaded, the design requires:

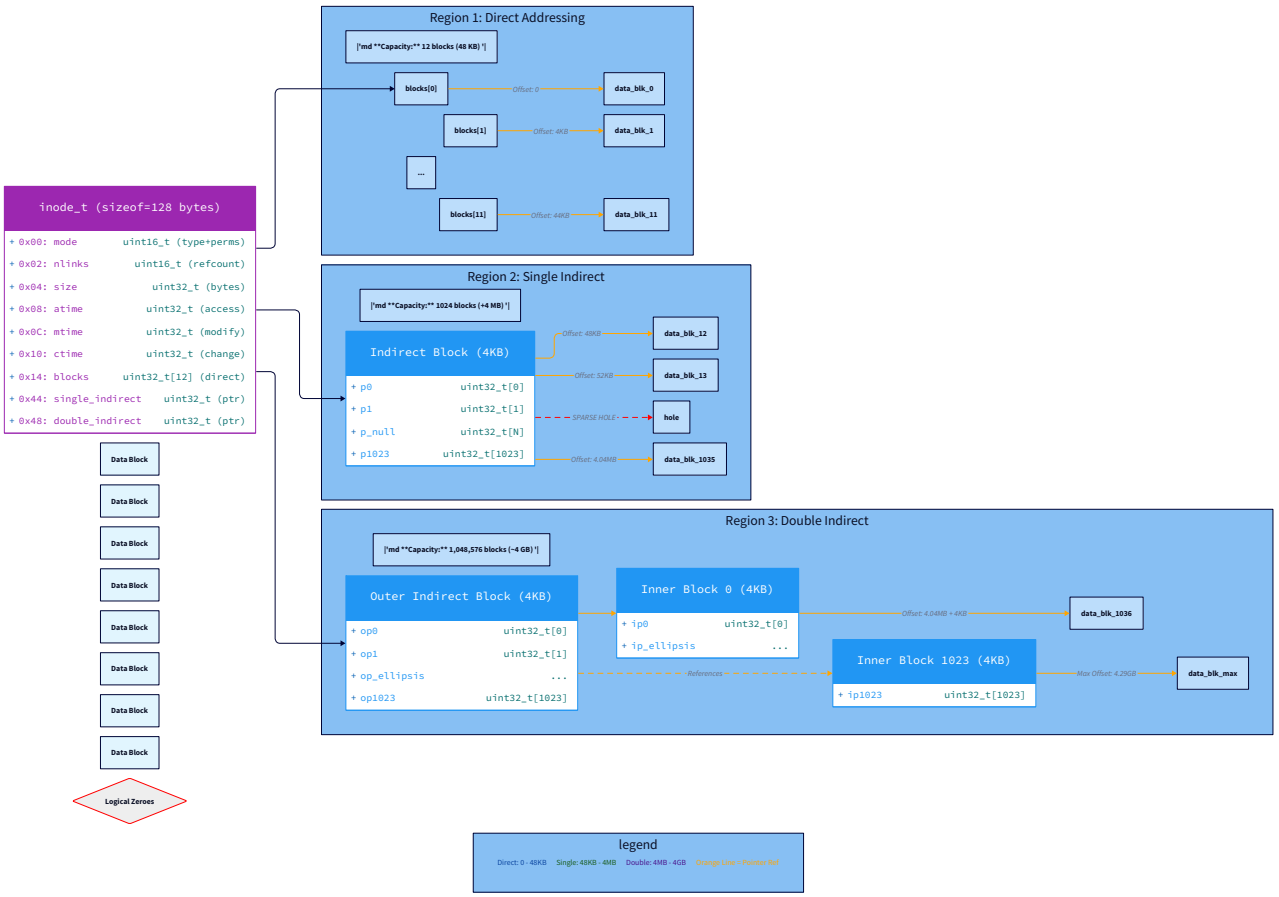
1. **Serialization:** Inodes must be locked before `inode_set_block` begins, as it modifies internal pointers and may trigger block allocations that update the Superblock.
2. **Order of Operations:** To avoid corruption on crash, the **Indirect Block** must be zeroed and written to disk **before** the parent inode pointer is updated and written.

64B CACHE LINE BOUNDARY

struct inode_t (sizeof=128 bytes)		
0x00	mode	uint16_t (2B)
0x02	uid	uint16_t (2B)
0x04	gid	uint16_t (2B)
0x06	nlinks	uint16_t (2B)
0x08	size	uint32_t (4B)
0x0C	atime	uint32_t (4B)
0x10	mtime	uint32_t (4B)
0x14	ctime	uint32_t (4B)
0x18	i_blocks[0..9]	uint32_t[10] (40B)
0x40	i_blocks[10..11]	uint32_t[2] (8B)
0x48	single_indirect	uint32_t (4B)
0x4C	double_indirect	uint32_t (4B)
0x50	reserved	uint8_t[48] (48B)

Inode Block Pointer Tree

```text
inode_t (128B)
|
|-- [0..11] Direct Pointers -------> [Data Block] (4KB)
|
|-- [12] Single Indirect ----------> [Pointer Block] (4KB)
|                                    |-- [0..1023] --------> [Data Block]
|
|-- [13] Double Indirect ----------> [Outer Pointer Block] (4KB)
                                     |-- [0..1023]
                                         |
                                         --> [Inner Pointer Block] (4KB)
                                             |-- [0..1023] --> [Data Block]
```



Memory Layout of Inode Table Block

```
Disk Block (4096 Bytes)
+----------------+----------------+----------------+-----+----------------+
| Inode 1 (128B) | Inode 2 (128B) | Inode 3 (128B) | ... | Inode 32 (128) |
+----------------+----------------+----------------+-----+----------------+
^                ^                ^
0x00             0x80             0x100
```

Module Technical Design Specification: Directory Operations (filesystem-m3)

1. Module Charter

The **Directory Operations** module transforms the flat, numeric inode space into a human-navigable hierarchical tree. It implements the "Directory is a special file" abstraction, where data blocks contain variable-length directory entry records mapping UTF-8 strings to inode numbers.

Scope:

- Definition of the variable-length on-disk `dirent` format and the `rec_len` skip-logic.
- Implementation of linear directory scanning for name lookups and entry additions.
- Path resolution engine supporting absolute and relative paths, including `.` and `..` traversal.

- Atomic-style directory mutations (`mkdir` , `rmdir` , `link` , `unlink`) with strict adherence to POSIX link-count invariants.
- Support for directory entry merging during deletion to prevent fragmentation within blocks.

Out of Scope:

- Higher-level FUSE callback wrappers (Milestone 5).
- Directory indexing optimizations like HTrees or B-Trees (linear scan only).
- Multi-threaded locking (deferred to the global FUSE lock).

Invariants:

- Every directory MUST contain `.` (self) and `..` (parent) as its first two entries.
- The `rec_len` of the last entry in a directory block MUST extend to the end of the 4KB block.
- Directory entries must never span across two physical blocks.
- Hard links to directories are strictly forbidden to prevent filesystem cycles.
- Root directory's `..` entry points to itself (Inode 1).

---

2. File Structure

Implementation follows a dependency-driven sequence:

1. `dir.h` : Structure definitions for on-disk and in-memory directory entries.
2. `dir_rec.c` : Low-level record manipulation (padding, length calculation, buffer serialization).
3. `dir_scan.c` : Linear scanning logic (`dir_lookup` , `dir_readdir`).
4. `dir_path.c` : Recursive path resolution and component splitting using `strtok_r` .
5. `dir_write.c` : Mutation logic (`dir_add_entry` with splitting, `dir_remove_entry` with merging).
6. `dir_op.c` : High-level POSIX-style operations (`fs_mkdir` , `fs_rmdir` , `fs_link` , `fs_unlink`).
7. `test_m3.c` : Integration tests for tree navigation and link-count correctness.

---

3. Complete Data Model

3.1. On-Disk Directory Entry (Physical Layout)

Entries are variable-length. The header is exactly 8 bytes. The name is NOT null-terminated on disk to save space; `name_len` provides the boundary.

Offset	Field	Type	Description
0x00	`d_inode`	`uint32_t`	Target Inode Number. 0 = Deleted/Empty Slot.
0x04	`d_rec_len`	`uint16_t`	Distance to the next record header in bytes.
0x06	`d_name_len`	`uint8_t`	Actual length of the filename string.
0x07	`d_file_type`	`uint8_t`	Hint: 1=Reg, 2=Dir, 7=Symlink.
0x08	`d_name[...]`	`char[]`	Filename string (1 to 255 bytes).
Ends	`padding`	`uint8_t[]`	0-3 bytes to ensure the NEXT record starts on a 4-byte boundary.

Hardware Soul (Alignment & Scanning):

- **4-Byte Alignment:** Every `d_rec_len` is a multiple of 4. This ensures `uint32_t` fields in the next header are naturally aligned, preventing unaligned access penalties on sensitive architectures.
- **Skip-Logic:** To find "FileB" in a block, the CPU reads the header of "FileA", adds `d_rec_len` to the current pointer, and jumps directly to "FileB".

3.2. In-Memory Representation (dirent_t)

Used for passing data between internal functions.

```c
#define MAX_FILENAME_LEN 255

#define DIRENT_HEADER_SIZE 8


typedef struct {

    uint32_t inode_num;

    uint16_t rec_len;

    uint8_t  name_len;

    uint8_t  file_type;

    char     name[MAX_FILENAME_LEN + 1]; // Null-terminated for C string compatibility

} dirent_t;
```

---

4. Interface Contracts

4.1. Lookup and Resolution

```c
/**
 * Resolves a full path (e.g., "/usr/bin/gcc") to an inode number.
 * @param path Null-terminated path string.
 * @return Inode number > 0, or negative errno.
 */
int path_resolve(int fd, superblock_t *sb, const char *path);

/**
 * Resolves the parent directory of a path.
 * @param path Input path.
 * @param out_parent_ino Set to parent inode number.
 * @param out_name Buffer for the final component name (e.g., "gcc").
 * @return 0 on success.
 */
int path_resolve_parent(int fd, superblock_t *sb, const char *path, uint32_t *out_parent_ino, char *out_name);
```

4.2. Directory Mutation

```c
/**
 * Adds a name->inode mapping to a directory.
 * @return 0 or -EEXIST if name found.
 */
int dir_add_entry(int fd, superblock_t *sb, uint32_t dir_ino, const char *name, uint32_t target_ino, uint8_t type);

/**
 * Removes a name mapping. Merges the record's space into the previous entry.
 * @return 0 or -ENOENT.
 */
int dir_remove_entry(int fd, superblock_t *sb, uint32_t dir_ino, const char *name);
```

5. Algorithm Specification

5.1. Directory Entry Packing (`dir_add_entry`)

When adding "NewFile" (Needed space = `actual_len(7)` = 12 bytes):

1. **Iterate** existing directory blocks.
2. For each block, **scan** records using `rec_len`.
3. For each record, calculate its `min_required_len = align4(8 + name_len)`.
4. **Find Free Space:** `available_gap = current_rec_len - min_required_len`.
5. If `available_gap >= needed_space`:
   - **Split:** Set `current_record->rec_len = min_required_len`.
   - **Insert:** Write new record at `offset + min_required_len` with `rec_len = available_gap`.
   - Return success.
6. If no gap found in any block, allocate a new data block via `alloc_data_block`, initialize it with one record where `rec_len = 4096`.

5.2. Path Resolution Engine (`path_resolve`)

1. **Start:** `current_ino = sb->s_root_inode` if path starts with `/`, else `cwd`.
2. **Tokenize:** Use `strtok_r` with `/` as delimiter.
3. **Loop components:**
   - Call `read_inode(current_ino)`.
   - Verify `S_ISDIR(mode)`. Return `-ENOTDIR` if not.
   - Call `dir_lookup(current_ino, component_name)`.
   - If returns 0, return `-ENOENT`.
   - `current_ino = result`.
4. **End:** Return `current_ino`.

5.3. Link Count Invariants (`fs_mkdir` / `fs_rmdir`)

Creating a directory `D` inside parent `P`:

1. `P->i_nlinks` increases by 1 (because `D/..` points to `P`).
2. `D->i_nlinks` is initialized to 2 (`D/.` and `P/D`).

Removing directory `D`:

1. Check if `D` is empty (scan data blocks, verify only `.` and `..` exist).
2. `P->i_nlinks` decreases by 1.
3. `D->i_nlinks` drops to 0; triggers `free_inode`.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`ENOTEMPTY`	`fs_rmdir`	Abort removal if entries > 2 found.	Yes
`EEXIST`	`dir_add_entry`	Return error; prevent duplicate names.	Yes
`EPERM`	`fs_link`	If target is `S_IFDIR`, reject to prevent cycles.	Yes
`ENAMETOOLONG`	`path_resolve`	Reject if component > 255 bytes.	Yes
`ELOOP`	`path_resolve`	(Advanced) Count components; reject if > 1024.	Yes
`EIO`	`dir_scan`	If `rec_len == 0` or overlaps, stop and return error.	Yes

7. Implementation Sequence with Checkpoints

Phase 1: Serialization Primitives (3 Hours)

Implement `dirent_actual_len` and buffer serializers. Ensure 4-byte padding logic is perfect. **Checkpoint:** Write a "FileA" entry followed by "FileB". Verify "FileB" starts at an address divisible by 4.

Phase 2: Scanning (3 Hours)

Implement `dir_lookup`. It must iterate through `inode->i_blocks` (using M2 logic) and walk the `rec_len` chain. **Checkpoint:** Manually create a directory block in `mkfs`. Use `dir_lookup` to find the inode of `..`.

Phase 3: Path Resolution (4 Hours)

Implement `path_resolve` and `split_path`. **Checkpoint:** Resolve `/` (returns 1). Resolve `/./.` (returns 1). Resolve `/nonexistent` (returns -ENOENT).

Phase 4: Mutation (5 Hours)

Implement `dir_add_entry` and `dir_remove_entry`. Focus on the `rec_len` manipulation. **Checkpoint:** Add 3 files. Remove the middle file. Verify the first file's `rec_len` now covers the gap where the middle file was.

Phase 5: POSIX Operations (4 Hours)

Implement `fs_mkdir`, `fs_rmdir`, `fs_link`. **Checkpoint:** `mkdir("/test")`, verify `root->i_nlinks == 3`. `rmdir("/test")`, verify `root->i_nlinks == 2`.

8. Test Specification

8.1. Hierarchical Persistence

1. `fs_mkdir(root, "a")`.
2. `fs_mkdir(a_ino, "b")`.
3. `path_resolve("/a/b")`.

4. **Assertion:** Returns `b_ino`. `b_ino->i_blocks[0]` contains `..` pointing to `a_ino`.

8.2. Empty Directory Constraint

1. `fs_mkdir(root, "dir")`.
2. `fs_create_file(dir_ino, "file")`.
3. `fs_rmdir(root, "dir")`.
4. **Assertion:** Returns `-ENOTEMPTY`. File still exists.

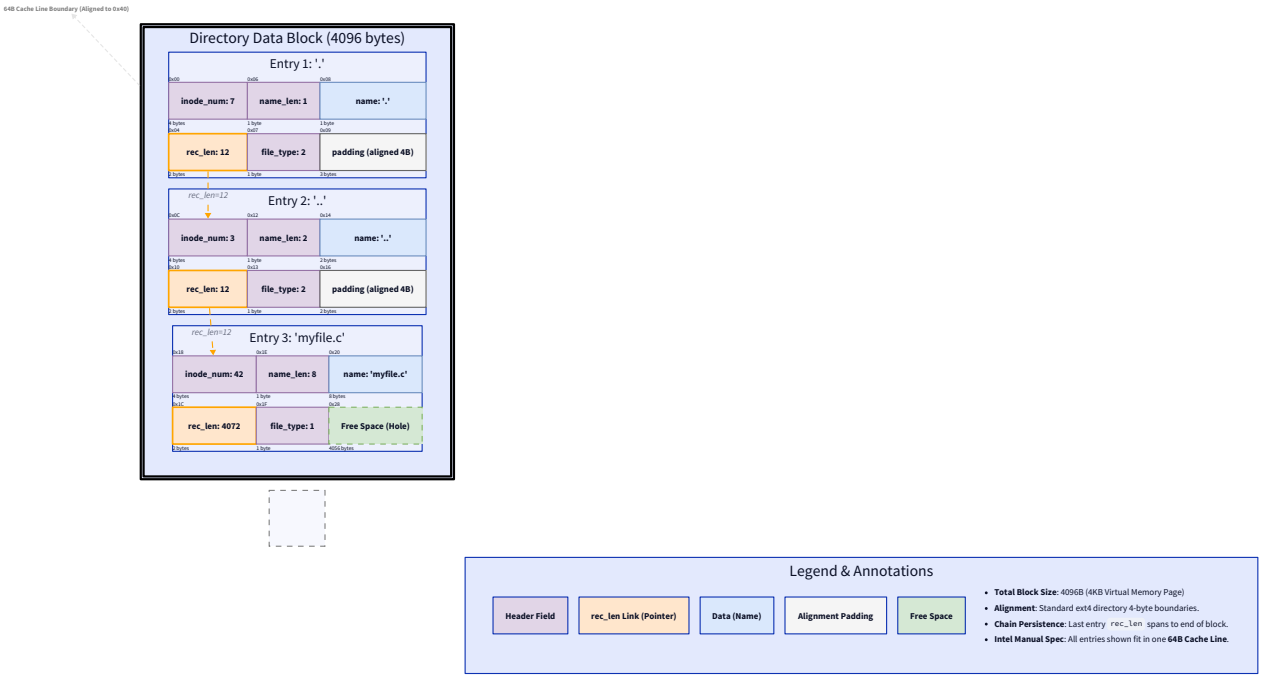
8.3. Hard Link Integrity

1. `fs_link(file_ino, root, "alias")`.
2. **Assertion:** `file_ino->i_nlinks == 2`.
3. `fs_unlink(root, "file")`.
4. **Assertion:** `file_ino->i_nlinks == 1`. Inode is NOT freed. Data is still readable via "alias".

---

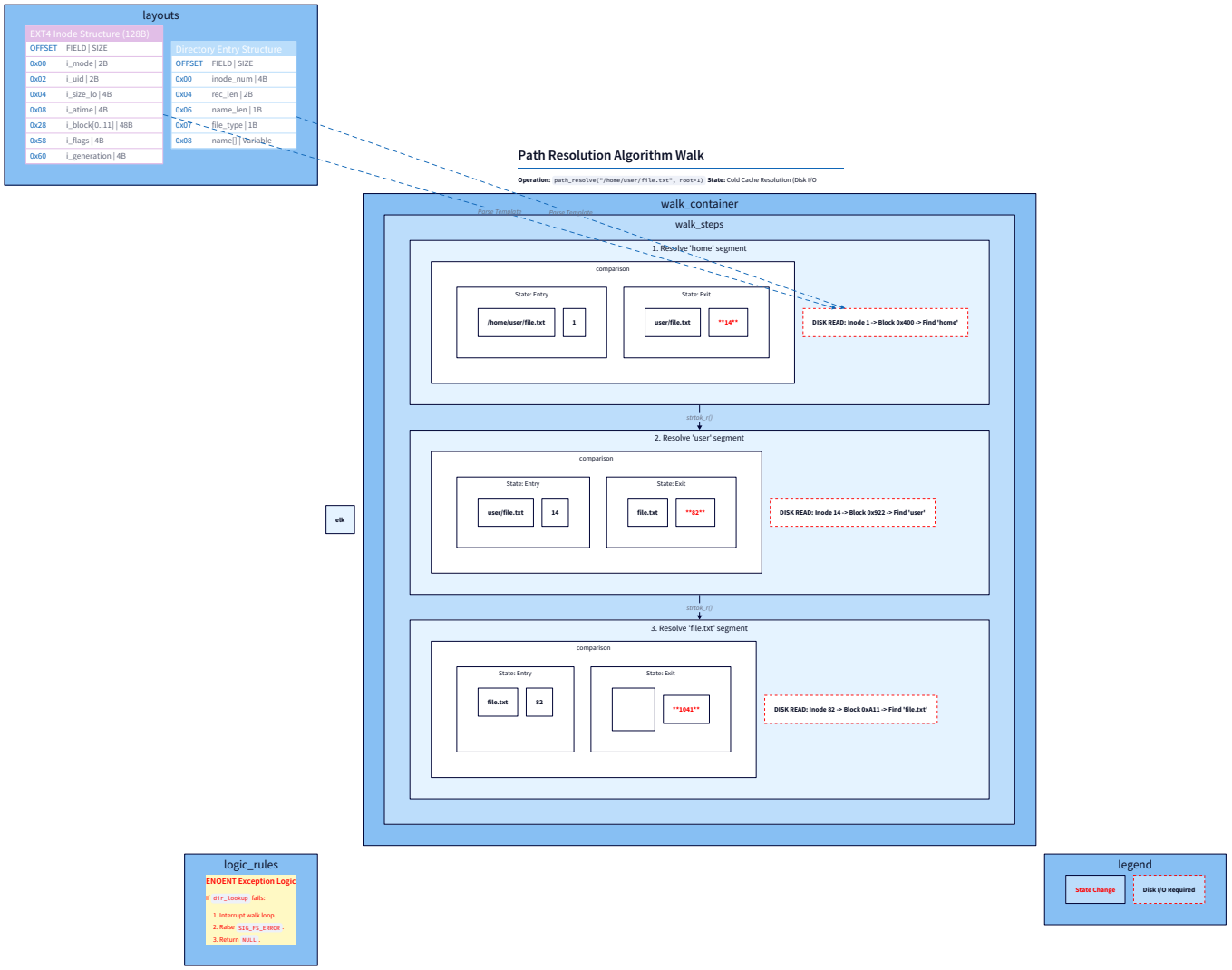
9. Performance Targets

Operation	Target	Measurement
Name Lookup	O(N) entries	Benchmark 1000 entries in one block.
Path Resolve	O(M) components	Resolve 10-level deep path.
Add Entry	< 2 Disk I/Os	Usually 1 read (find space) + 1 write.
Memory Overhead	8B per entry	Excludes name string.



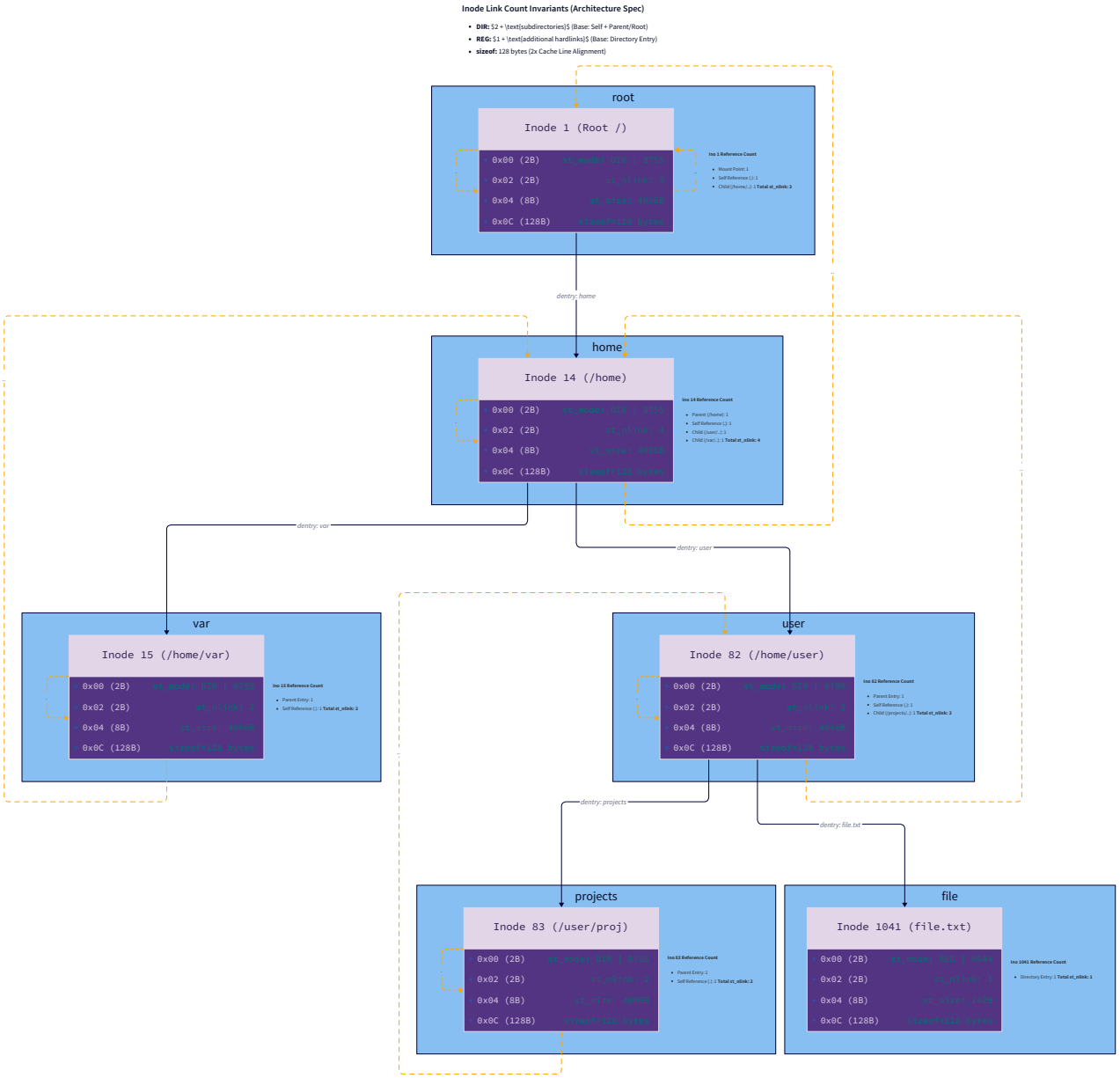
On-Disk Directory Block Layout

```text
Block Offset: 0
[ Inode: 1 | RecLen: 12 | NameLen: 1 | Type: 2 | Name: "."  | Pad: 3 ]
Block Offset: 12
[ Inode: 1 | RecLen: 12 | NameLen: 2 | Type: 2 | Name: ".." | Pad: 2 ]
Block Offset: 24
[ Inode: 5 | RecLen: 4072| NameLen: 4 | Type: 1 | Name: "file"| Pad: 0 ]
                              ^
                              |--- RecLen of last entry fills the block
```



Directory Entry Removal (Merging)

```text
BEFORE:
[ Entry A (RecLen: 12) ] [ Entry B (RecLen: 16) ] [ Entry C (RecLen: 4068) ]

AFTER REMOVE(B):
[ Entry A (RecLen: 28) ]                          [ Entry C (RecLen: 4068) ]
          ^------- A's RecLen increased (12 + 16)
```



Path Resolution Flow

```
Path: "/home/user/doc.txt"

1. Lookup "home" in Inode 1 (Root) -> Inode 10
2. Lookup "user" in Inode 10       -> Inode 45
3. Lookup "doc.txt" in Inode 45    -> Inode 102
4. Return 102
```

TEXT

Module Technical Design Specification: File Read/Write Operations (filesystem-m4)

1. Module Charter

The **File Read/Write Operations** module implements the data plane of the filesystem. It bridges the gap between byte-addressable application requests and the block-addressable hardware reality. This module is responsible for the life cycle of regular file data, including creation, sequential and random access, and resizing.

Scope:

- **Creation:** Orchestrating inode allocation and directory entry insertion for new files.
- **Read Logic:** Implementing byte-to-block mapping, handling sparse "holes" via zero-filling, and enforcing End-Of-File (EOF) boundaries.
- **Write Logic:** Implementing the Read-Modify-Write (RMW) cycle for partial-block updates and on-demand block allocation for file growth.
- **Truncation:** Reclaiming disk space during shrink operations (including recursive indirect block cleanup) and implementing "thin" extension via sparse holes.
- **Metadata Sync:** Ensuring `i_size`, `i_atime`, `i_mtime`, and `i_ctime` are updated according to POSIX semantics.

Out of Scope:

- FUSE kernel-to-user relay (Milestone 5).
- Atomic crash consistency via journaling (Milestone 6).
- Permission/ACL enforcement (deferred to the VFS layer).

Invariants:

- `fs_read` never returns more than `inode.i_size - offset` bytes.
- A sparse hole (disk block pointer = 0) must return exactly zeroed bytes without performing disk I/O.
- `fs_write` at an offset > `inode.i_size` automatically creates a sparse hole between the old EOF and the new write start.
- `fs_truncate` (shrink) must zero-fill the byte range from `new_size` to the next 4KB boundary in the last retained block to prevent data leakage.

---

2. File Structure

Implementation proceeds from basic creation to complex mutation:

1. `file_ops.h`: Function prototypes and offset arithmetic macros.
2. `file_create.c`: High-level file creation logic.
3. `file_read.c`: The read path and sparse hole handling.
4. `file_write.c`: The write path, RMW logic, and allocation-on-demand.
5. `file_truncate.c`: Shrink/Extend logic and recursive block freeing.
6. `test_m4.c`: Comprehensive I/O suite (Round-trip verification, sparse tests, 4GB boundary tests).

---

3. Complete Data Model

This module relies on the `inode_t` structure defined in Milestone 2. Every operation navigates this structure's block pointer tree.

3.1. Inode Metadata (inode_t) Reference Table

Offset	Field	Type	Description
0x00	`i_mode`	`uint16_t`	Must have `S_IFREG` (0100000) for these ops.
0x04	`i_size`	`uint32_t`	Logical size. Clamps `fs_read` ; updated by `fs_write` .
0x18	`i_blocks[12]`	`uint32_t[12]`	Direct pointers for offsets 0 to 49,151.
0x48	`i_single_ind`	`uint32_t`	Pointer to 1024 addresses (49,152 to 4,243,455).
0x4C	`i_double_ind`	`uint32_t`	Pointer to 1024^2 addresses (up to ~4.29GB).

3.2. Internal Operation Context

Operations decompose a requested byte range `[offset, offset + length)` into a sequence of block-sized "chunks".

Variable	Definition	Formula
`block_index`	0-based logical block	`offset / 4096`
`block_off`	Starting byte in block	`offset % 4096`
`chunk`	Bytes to process in this block	`min(4096 - block_off, remaining_len)`

4. Interface Contracts

4.1. File Lifecycle

```c
/**
 * Creates a regular file.
 * @param parent_ino Inode of the directory to hold the file.
 * @param name Filename (max 255 chars).
 * @param mode Permissions (e.g., 0644).
 * @return New 1-based inode number, or negative errno (EEXIST, ENOSPC).
 */
int fs_create_file(int fd, superblock_t *sb, uint32_t parent_ino,
                   const char *name, uint16_t mode, uint16_t uid, uint16_t gid);
```

4.2. The Data Plane

```c
/**
 * Reads data from a regular file.
 * @param buf Destination buffer.
 * @return Actual bytes read. Returns 0 if offset >= i_size.
 * @note Clamps length to i_size. Zero-fills sparse holes.
 */
ssize_t fs_read(int fd, superblock_t *sb, uint32_t ino,
                uint64_t offset, void *buf, size_t length);

/**
 * Writes data to a regular file.
 * @return Bytes written (always 'length' on success).
 * @note Triggers RMW for partial blocks. Allocates blocks on growth.
 */
ssize_t fs_write(int fd, superblock_t *sb, uint32_t ino,
                 uint64_t offset, const void *buf, size_t length);

/**
 * Convenience wrapper for writing at current i_size.
 */
ssize_t fs_append(int fd, superblock_t *sb, uint32_t ino,
                  const void *buf, size_t length);
```

4.3. Resizing

```c
/**
 * Resizes a file to new_size.
 * @note Shrink: frees blocks > new_size and zeroes the last block's tail.
 * @note Extend: updates i_size, creating a sparse hole.
 */
int fs_truncate(int fd, superblock_t *sb, uint32_t ino, uint64_t new_size);
```

5. Algorithm Specification

5.1. `fs_read`: The Zero-Fill Loop

1. **Read Inode:** Load `inode_t` for `ino`.
2. **Validate:** Return `-EINVAL` if not `S_IFREG`.
3. **Clamp:** If `offset >= i_size`, return `0`. If `offset + length > i_size`, set `length = i_size - offset`.
4. **Loop:** While `remaining > 0`:
   - Calculate `block_index`, `block_off`, and `chunk`.
   - Call `inode_get_block(block_index * 4096)` (from M2).
   - **Case Hole (`disk_block == 0`):** `memset(buf + progress, 0, chunk)`. Skip Disk I/O.
   - **Case Data (`disk_block != 0`):** `read_block(disk_block, temp_buf)`. `memcpy(buf + progress, temp_buf + block_off, chunk)`.
   - Update `progress`, `remaining`, `offset`.
5. **Metadata:** Update `i_atime = now()`. Write Inode.

5.2. `fs_write`: The Read-Modify-Write (RMW) Engine

1. **Read Inode:** Load `inode_t`.
2. **Capacity Check:** If `offset + length > 4GB`, return `-EFBIG`.
3. **Loop:** While `remaining > 0`:
   - Calculate `block_index`, `block_off`, and `chunk`.
   - Determine if RMW is needed: `bool partial = (block_off != 0 || chunk < 4096)`.
   - `inode_get_block` → `disk_block`.
   - **Case New Block (`disk_block == 0`):**
     - `disk_block = alloc_data_block()`.
     - `inode_set_block(..., disk_block)`.
     - `memset(temp_buf, 0, 4096)`. (Ensures hole-alignment).
   - **Case Existing Block (`disk_block != 0`):**
     - If `partial`, `read_block(disk_block, temp_buf)`.
   - **Modify:** `memcpy(temp_buf + block_off, buf + progress, chunk)`.
   - **Write:** `write_block(disk_block, temp_buf)`.
   - Update `progress`, `remaining`, `offset`.
4. **Metadata:** If `offset > i_size`, `i_size = offset`. Set `i_mtime = i_ctime = now()`. Write Inode.

5.3. `fs_truncate`: Shrink with Data Leak Protection

1. **Read Inode.**
2. **Extend Case (`new_size > i_size`):** Update `i_size`. Write Inode. Done.
3. **Shrink Case (`new_size < i_size`):**
   - **Zero Tail:** Find the block index containing `new_size`. If `new_size % 4096 != 0`:
     - `inode_get_block` → `disk_block`.
     - If `disk_block != 0`, `read_block`, `memset` bytes from `new_size % 4096` to 4095, `write_block`.
   - **Free Blocks:** Iterate `bi` from `(new_size + 4095) / 4096` to `(old_size + 4095) / 4096`.

        - Call `inode_get_block(bi * 4096)`.
        - If `disk_block != 0`, `free_data_block(disk_block)`.
        - `inode_set_block(..., 0)`. (Null the pointer).
    - **Cleanup:** Call `truncate_free_indirect_if_empty` (M2 logic) to reclaim indirect pointer blocks.
4. **Metadata:** `i_size = new_size`, `i_mtime = i_ctime = now()`.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`ENOSPC`	`fs_write`	Stop write; return partial bytes written or error.	Yes
`EFBIG`	`fs_write`	Reject write before allocation.	Yes
`EINVAL`	`fs_read`	Ensure inode is regular file (`S_ISREG`).	Yes
`EIO`	`read_block`	Return I/O error; filesystem may be inconsistent.	Yes
Data Leak	`fs_truncate`	Zeroing the tail of the partial block prevents reading old data.	No (Internal)

---

7. Implementation Sequence with Checkpoints

Phase 1: Creation & Math (2 Hours)

Implement `fs_create_file`. Use M2 `alloc_inode` and M3 `dir_add_entry`. Implement internal offset-to-block helpers. **Checkpoint:** `mkfs` a disk, call `fs_create_file("/test.txt")`. Verify `ls` shows the file.

Phase 2: The Read Path (3 Hours)

Implement `fs_read`. Ensure it handles offsets correctly. Add `memset(0)` logic for sparse holes. **Checkpoint:** Manually set a block pointer in an inode to 0. Read 4KB at that offset. Verify buffer is all zeros and `read_block` was never called.

Phase 3: The Write Path (4 Hours)

Implement `fs_write`. Implement the RMW cycle logic. **Checkpoint:** Write "Hello" at offset 0. Read back. Write "World" at offset 6. Read back. Verify "Hello World" is present (proves RMW preserved the space between them).

Phase 4: Truncate & Append (4 Hours)

Implement `fs_truncate` (shrink/extend) and `fs_append`. **Checkpoint:** Create 10MB file. Truncate to 1MB. Verify `s_free_blocks` increases by ~2304 blocks.

---

8. Test Specification

8.1. Partial Block RMW Verification

- **Test:** Write "A" at offset 4095 (end of block 0) and "B" at 4096 (start of block 1).

- **Assertion:** `fs_read` returns "AB" across the 4KB boundary. Block 0 and Block 1 must have been allocated.

8.2. Sparse Hole Expansion

- **Test:** Write "X" at offset 0. Write "Y" at offset 1,000,000.
- **Assertion:** `fs_read` at offset 500,000 returns 0s. `i_size` is 1,000,001. `s_free_blocks` indicates only 2 data blocks (plus indirects) were used.

8.3. Data Leakage (Security)

- **Test:** Write 4KB of `0xFF` to a file. Truncate to 10 bytes. Truncate back to 4KB (Extend).
- **Assertion:** Reading bytes 11–4095 returns `0x00`, NOT `0xFF`.

---

9. Performance Targets

Operation	Target	Measurement
Sequential Read	> 200 MB/s	DD from mount to /dev/null
Aligned Write	1 Disk I/O / 4KB	No `read_block` calls in write path
Random 1-byte Write	2 Disk I/Os	1 Read + 1 Write (RMW)
Sparse Read	< 50ns / block	`clock_gettime` for 4KB hole

---

10. Hardware Soul (Cache and Alignment)

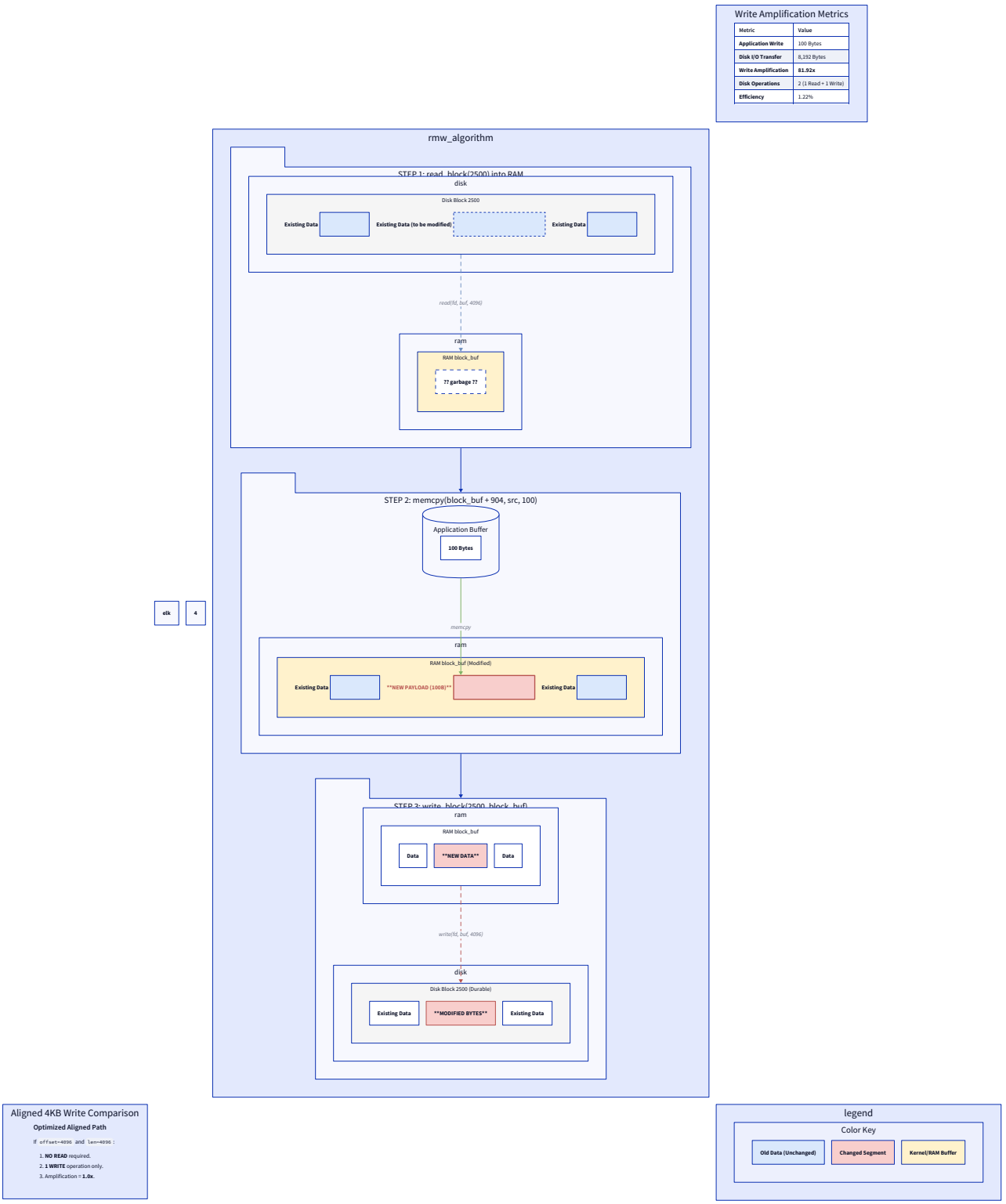
10.1. Write Amplification Analysis

Every write that is not 4096-byte aligned and 4096-byte sized triggers a Read-Modify-Write.

- **Example:** `write(fd, "a", 1)`
- **Cost:**
  1. `read_block(n)`: 4096 bytes into RAM.
  2. `memcpy`: 1 byte.
  3. `write_block(n)`: 4096 bytes to Disk.
- **Amplification:** 8192 / 1 = 8192x.
- **Mitigation:** Applications should buffer I/O to 4KB boundaries (e.g., `setvbuf` in C).

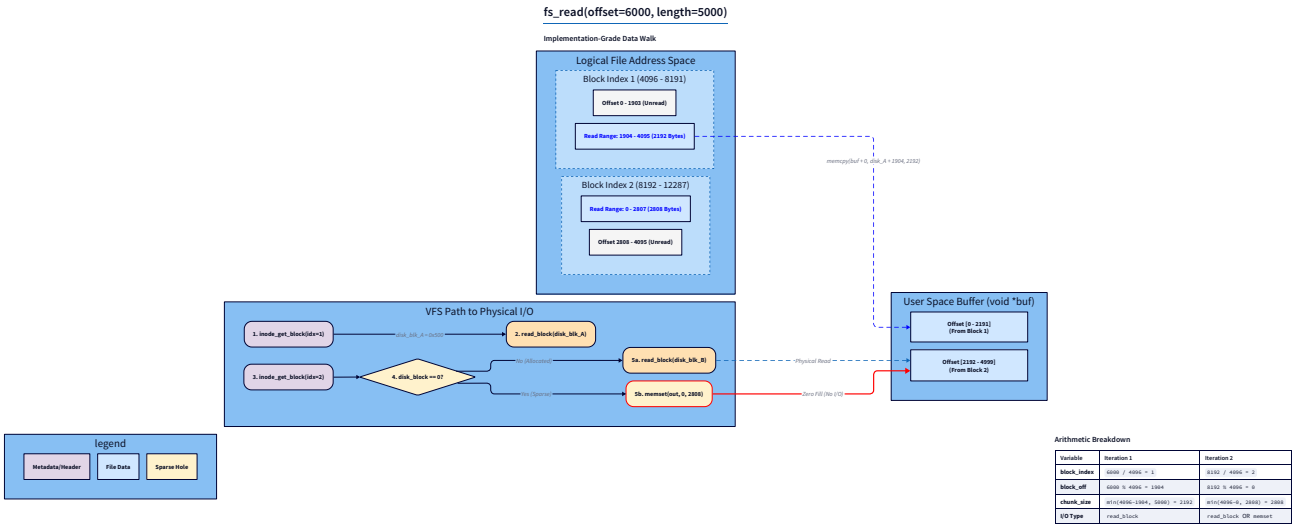
10.2. TLB and Page Alignment

Our `block_buf` is 4096 bytes. To optimize performance, the buffer passed to `read(2)/write(2)` of the disk image should be **page-aligned** (using `posix_memalign`). This ensures that the kernel's Direct Memory Access (DMA) can map the buffer directly to physical disk sectors without an intermediate copy (Zero-Copy I/O).



Read-Modify-Write (RMW) Decision Flow

```
Write Request (Offset, Length)
|
|-- Is Offset % 4096 == 0 AND Length >= 4096?
|   |-- YES: Full Block Write
|   |   |-- No existing block? Allocate & Write.
|   |   |-- Existing block? Overwrite directly.
|   |
|   |-- NO: Partial Block Write (RMW)
|       |-- No existing block? Allocate & Zero-Fill.
|       |-- Existing block? READ block into buffer.
|       |-- MODIFY buffer with user data.
|       |-- WRITE buffer back to disk.
```



File Offset Arithmetic

```
File Offset: 10,000
Block Size:  4,096

Block Index: 10,000 / 4,096 = 2  (0-indexed)
Block Off:   10,000 % 4,096 = 1,808
Remaining:   10,000 to End
Chunk:       min(4,096 - 1,808, total_len) = 2,288
```

Module Technical Design Specification: FUSE Integration (filesystem-m5)

1. Module Charter

The **FUSE Integration** module is the definitive bridge between the internal block-and-inode machinery and the Linux Virtual Filesystem (VFS). It provides the high-level interface that allows the operating system to treat a raw disk image as a mountable directory.

Scope:

- **Translation:** Converting VFS path-based requests into internal inode-based operations using the Milestone 3 resolution engine.
- **State Management:** Maintaining the global filesystem context (`fs_ctx_t`), including the disk image file descriptor, the in-memory superblock, and a coarse-grained serialization lock.
- **Resource Mapping:** Mapping internal `inode_t` fields to POSIX `struct stat` and `struct statvfs` records for system-wide compatibility.
- **Lifecycle:** Orchestrating the mount-time initialization (including recovery hooks) and the unmount-time cleanup (ensuring metadata and data durability via `fsync`).
- **Optimization:** Leveraging FUSE entry and attribute timeouts to mitigate the performance penalties of path resolution over the FUSE kernel-user boundary.

Out of Scope:

- In-kernel filesystem logic (this is strictly a userspace daemon).
- Low-level FUSE protocol handling (delegated to `libfuse3`).
- Fine-grained locking or lock-free data structures (single global mutex).

Invariants:

- Every callback must acquire the global `fs_lock` before accessing the `fs_ctx_t` or calling internal FS functions.
- The `fi->fh` (file handle) must be populated with the inode number during `open` / `create` to bypass path resolution in `read` / `write` .
- All callbacks must return negative errno values on failure (e.g., `-ENOENT`) as per the FUSE specification.
- No callback may return to the kernel without releasing the global lock, regardless of the error path.

---

2. File Structure

Implementation follows a strict integration sequence, moving from global state to specific VFS operations:

1. `myfs_main.c` : Entry point, argument parsing, and `fuse_main` invocation.
2. `fuse_context.h` : Definition of `FsCtx` and global lock macros.
3. `fuse_bridge.c` : Conversion logic (`inode_to_stat` , `path_to_inode`).
4. `fuse_ops_meta.c` : Implementation of `getattr` , `statfs` , `chmod` , `utimens` .
5. `fuse_ops_dir.c` : Implementation of `readdir` , `mkdir` , `rmdir` , `rename` , `unlink` .
6. `fuse_ops_file.c` : Implementation of `open` , `create` , `read` , `write` , `truncate` , `release` .
7. `Makefile` : Integration with `pkg-config --cflags --libs fuse3` .

---

3. Complete Data Model

3.1. Global Filesystem Context (FsCtx)

This structure resides in the heap/global segment of the FUSE daemon. It is the single source of truth for the mounted instance.

```c
typedef struct {
    int fd;                   // 0x00: Disk image file descriptor
    superblock_t sb;          // 0x04: In-memory copy of the superblock (4096 bytes)
    pthread_mutex_t lock;     // 0x1004: Global coarse-grained lock (usually 40 bytes)
    uint32_t mount_flags;     // 0x102C: Internal flags (direct_io, etc.)
    char image_path[256];     // 0x1030: Path to the .img file for persistence
} FsCtx;                      // Total Size: ~4400 bytes
```

Hardware Soul (Cache Line Analysis):

- **Superblock Separation:** The `sb` field (4KB) acts as a massive buffer that physically separates the `fd` and the `lock`. On modern CPUs with 64B cache lines, this ensures that frequent lock acquisitions (which dirty the cache line containing the mutex) do not cause false-sharing with the `fd` or other configuration flags.
- **Alignment:** `FsCtx` should be page-aligned (`posix_memalign`) to ensure the `sb` field is aligned with the host's page cache, facilitating zero-copy reads from the image file into the context.

3.2. FUSE File Handle (fi->fh)

We utilize the 64-bit `fh` field in `struct fuse_file_info` to cache the inode number. This is critical for performance.

Field	Value	Why?
`fi->fh`	`uint64_t(inode_num)`	Avoids calling `path_resolve` in every `read()` and `write()` call.
`fi->direct_io`	`1`	Bypasses the FUSE kernel page cache. Ensures correctness for the single-node implementation.
`fi->keep_cache`	`0`	Discard host-side page cache on close to prevent stale data.

---

4. Interface Contracts

4.1. The Path-to-Inode Bridge

```c
/**
 * Resolves a path to an inode number and fetches the inode.
 * @param path Absolute path from FUSE.
 * @param out_ino Target inode number.
 * @param out_inode Pointer to a stack-allocated inode_t to fill.
 * @return 0 on success, -ENOENT or -EIO on failure.
 * @note MUST be called while holding g_fs.lock.
 */
static int bridge_path_to_inode(const char *path, uint32_t *out_ino, inode_t *out_inode);

/**
 * Maps inode_t fields to struct stat.
 * @param st Pointer to the FUSE-provided stat structure.
 */
static void bridge_inode_to_stat(uint32_t ino_num, const inode_t *inode, struct stat *st);
```

4.2. Core FUSE Callbacks (The VFS Contract)

```c
// Metadata
static int myfs_getattr(const char *path, struct stat *st, struct fuse_file_info *fi);
static int myfs_statfs(const char *path, struct statvfs *stbuf);

// Directories
static int myfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                        off_t offset, struct fuse_file_info *fi, enum fuse_readdir_flags flags);
static int myfs_mkdir(const char *path, mode_t mode);

// Files
static int myfs_create(const char *path, mode_t mode, struct fuse_file_info *fi);
static int myfs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi);
static int myfs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info
*fi);
static int myfs_truncate(const char *path, off_t size, struct fuse_file_info *fi);
```

---

5. Algorithm Specification

5.1. The `getattr` Hot-Path

`getattr` is called by the kernel for almost every operation.

1. **Lock:** `pthread_mutex_lock(&g_fs.lock)`.
2. **Resolve:** Call `path_resolve(path)`.
3. **Handle Error:** If `-ENOENT`, unlock and return `-ENOENT` immediately.
4. **Read:** Call `read_inode` for the resolved inode number.
5. **Convert:** Call `bridge_inode_to_stat`.
   - `st->st_blocks` = `(inode->size + 511) / 512` (POSIX expects 512B blocks).
   - `st->st_blksize` = `4096`.
6. **Unlock:** `pthread_mutex_unlock(&g_fs.lock)`.
7. **Return:** `0`.

5.2. `readdir` Filler Loop

1. Resolve `path` to `dir_ino`.
2. Read `dir_ino` and verify `S_ISDIR`.
3. Define a local context for the readdir callback.
4. Invoke `dir_readdir(dir_ino, callback_fn)`.

5. Inside `callback_fn`:
    - Call `filler(buf, name, &st, 0, 0)`.
    - If `filler` returns non-zero (buffer full), abort the scan and return `0` (FUSE will resume).

5.3. Atomic Rename

1. Identify `src_parent_path` and `src_name` from `old_path`.
2. Identify `dst_parent_path` and `dst_name` from `new_path`.
3. **Locking:** Acquire `g_fs.lock`.
4. **Validation:**
    - Resolve `src_ino`.
    - If `new_path` exists, resolve `dst_ino` and check if `S_ISDIR` match.
    - If `dst_ino` is a directory, verify it is empty.
5. **Mutation:**
    - `dir_add_entry(dst_parent, dst_name, src_ino)`.
    - `dir_remove_entry(src_parent, src_name)`.
    - Update `ctime` for `src_ino` and both parents.
6. **Unlock.**

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`ENOENT`	`path_resolve`	Standard return to VFS; shell prints "No such file".	Yes
`ENOTEMPTY`	`fs_rmdir`	Directory has entries; reject removal.	Yes
`EBADF`	`myfs_read`	`fi->fh` is 0 or invalid.	Yes
`EACCES`	FUSE Kernel	Kernel checks `st_mode` returned by `getattr`.	Yes
`ENOMEM`	`filler`	FUSE buffer is full; current readdir batch complete.	No
`EIO`	`read_block`	Disk image corrupted or unreadable.	Yes

---

7. Implementation Sequence with Checkpoints

Phase 1: The Context and Mount (2 Hours)

Implement `main` and the FUSE `init` / `destroy` callbacks.

- **init:** Open the disk image, read the superblock into `g_fs.sb`, initialize the mutex.
- **destroy:** `fsync(g_fs.fd)`, write the superblock back to block 0, close `fd`. **Checkpoint:** `./myfs disk.img /mnt/test` runs and waits. `mount | grep myfs` shows the mount point.

Phase 2: Metadata and Navigation (3 Hours)

Implement `bridge_path_to_inode`, `bridge_inode_to_stat`, and `getattr`. **Checkpoint:** `ls /mnt/test` does not return "Permission Denied" or "Invalid Argument" (though it may be empty).

Phase 3: Directory Contents (3 Hours)

Implement `readdir` (using Milestone 3 primitives). **Checkpoint:** `ls -la /mnt/test` correctly displays `.` and `..` with correct inode numbers (Inode 1).

Phase 4: File Data (4 Hours)

Implement `create`, `open`, `read`, and `write`.

- **Crucial:** In `open`, set `fi->fh = ino_num`.
- **Crucial:** In `read`, use `(uint32_t)fi->fh`. **Checkpoint:** `echo "hello" > /mnt/test/file.txt` followed by `cat /mnt/test/file.txt` works.

Phase 5: The Full POSIX Set (4 Hours)

Implement `mkdir`, `rmdir`, `unlink`, `truncate`, `rename`. **Checkpoint:** `mkdir`, `mv`, `rm` all behave exactly like a native ext4 directory.

---

8. Test Specification

8.1. Round-Trip Persistence

1. Mount `disk.img`.
2. `mkdir /mnt/test/data`.
3. `echo "bytes" > /mnt/test/data/file.bin`.
4. Unmount (`fusermount3 -u`).
5. Remount.
6. **Assertion:** `cat /mnt/test/data/file.bin` returns "bytes".

8.2. Concurrent Access (Stress)

1. In terminal A: `while true; do ls -R /mnt/test; done`.
2. In terminal B: `while true; do dd if=/dev/urandom of=/mnt/test/stress bs=4k count=10; done`.
3. **Assertion:** No kernel panics, no "Transport endpoint not connected", no deadlocks.

8.3. Sparse File Verification via Tools

1. `truncate -s 1G /mnt/test/huge`.
2. `ls -l /mnt/test/huge` shows 1GB.
3. `du -h /mnt/test/huge` shows 0.
4. **Assertion:** Filesystem correctly reports logical vs physical size via `getattr`.

---

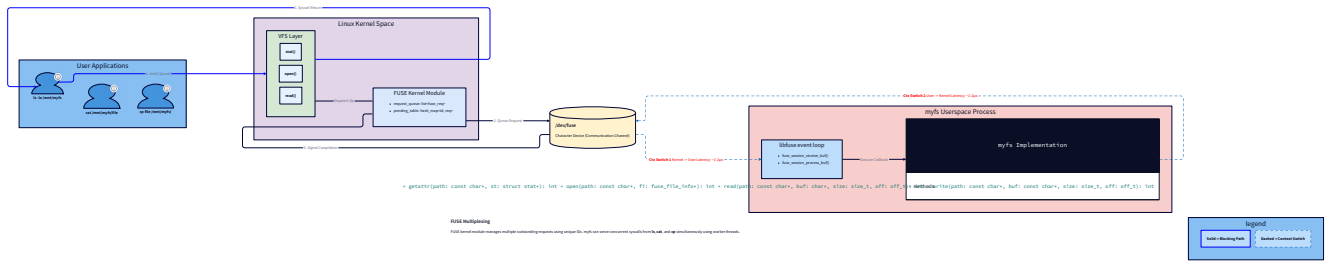
9. Performance Targets

Operation	Target	Measurement
`getattr` Latency	< 5ms (Cold)	`time ls /mnt/test`
`read` Throughput	> 80% of Raw Image Read	`dd` from mount vs `dd` from image
Context Switch	< 10µs per call	`strace -c` total time / call count
Unmount Time	< 1s	`time fusermount3 -u`

10. Concurrency Specification (Global Lock)

The global mutex implementation follows a "Leaf-Only Locking" strategy:

1. **Acquisition:** Every callback acquires `g_fs.lock` at the first line of execution.
2. **Recursion:** Internal functions (like `path_resolve`) **MUST NOT** acquire the lock themselves, as the mutex is non-recursive.
3. **I/O:** The lock is held during `read_block/write_block`. While this serializes I/O, it ensures that the Superblock and Bitmaps remain consistent across threads without complex fine-grained coordination.
4. **Release:** The lock is released only at the `return` statement of the FUSE callback.



FUSE to Internal FS Call Flow

```
Application: open("/mnt/myfs/doc.txt")
      |
      V (System Call)
Linux Kernel VFS
      |
      V (FUSE Kernel Module)
/dev/fuse <--- (Writes Request: OPEN, path="/doc.txt")
      |
      V (Context Switch)
libfuse (Reads /dev/fuse)
      |
      V (Calls myfs_open)
+--------------------------------------+
| myfs_open("/doc.txt")                |
| 1. lock(g_fs.lock)                   |
| 2. ino = path_resolve("/doc.txt")    |
| 3. fi->fh = ino                      |
| 4. unlock(g_fs.lock)                 |
| 5. return 0                          |
+--------------------------------------+
```

TEXT

FUSE High-Level Callback to Low-Level Operation Vector (sizeof=44 entries)

FUSE Callback Interface	open(path, flags)	write(path, size, offset)	fs_rmdir(p_ino, name)
Concurrency Control	fs_lock() ▼ ... fs_unlock() ▲	fs_lock() ▼ ... fs_unlock() ▲	unlink(path)
Path/Translation Layer	resolve_path(path) → ino	**`ino = (uint32_t)fi->fh`**	fs_lock() ▼ ... fs_unlock() ▲
Internal FS Core Operations	1. `read_inode(ino)` 2. `if O_TRUNC: fs_truncate(ino, 0)` 3. `fi->fh = ino` (Cache)	fs_write(ino, offset, buf, size)	split_path(path) → parent, name resolve_path(parent) → p_ino
getattr(path, stat)	create(path, mode)	mkdir(path, mode)	fs_unlink(p_ino, name)
fs_lock() ▼ ... fs_unlock() ▲	fs_lock() ▼ ... fs_unlock() ▲	fs_lock() ▼ ... fs_unlock() ▲	rename(old, new)
resolve_path(path) → ino	split_path(path) → parent, name resolve_path(parent) → p_ino	split_path(path) → parent, name resolve_path(parent) → p_ino	fs_lock() ▼ ... fs_unlock() ▲
read_inode(ino) + inode_to_stat()	1. `fs_create_file(p_ino, name, mode)` 2. `fi->fh = new_ino`	fs_mkdir(p_ino, name, mode)	`split_path` x2 `resolve_path` (parents) x2 1. `dir_add_entry(new_p, new_name, src_ino)` 2. `dir_remove_entry(old_p, old_name)`
readdir(path, filler)	read(path, size, offset)	rmdir(path)	truncate(path, size)
fs_lock() ▼ ... fs_unlock() ▲	fs_lock() ▼ ... fs_unlock() ▲	fs_lock() ▼ ... fs_unlock() ▲	fs_lock() ▼ ... fs_unlock() ▲
resolve_path(path) → ino	**`ino = (uint32_t)fi->fh`**	split_path(path) → parent, name resolve_path(parent) → p_ino	if fi: fi->fh else: resolve_path(path)
dir_readdir(ino) → filler(name, stat)	fs_read(ino, offset, buf, size)		fs_truncate(ino, size)

legend

▼ Lock Acquire ▲ Lock Release fh = file handle cache

Stat Bridge Mapping

```
inode_t (Internal)              struct stat (POSIX)
+-----------------+             +-------------------+
| i_mode          | ----------> | st_mode           |
| i_nlinks        | ----------> | st_nlink          |
| i_uid / i_gid   | ----------> | st_uid / st_gid   |
| i_size          | ----------> | st_size           |
| i_atime         | ----------> | st_atim.tv_sec    |
| i_mtime         | ----------> | st_mtim.tv_sec    |
| (ino_num)       | ----------> | st_ino            |
+-----------------+             +-------------------+
```

Module Technical Design Specification: Write-Ahead Journaling and Crash Recovery (filesystem-m6)

1. Module Charter

The **Write-Ahead Journaling and Crash Recovery** module is the final and most critical layer of the filesystem, providing a mathematical guarantee of metadata consistency in the event of power failure or system crash. It implements a sequential, circular write-ahead log (WAL) that captures intended metadata changes before they are applied to the primary on-disk structures.

Scope:

- **Atomicity:** Wrapping multi-block metadata updates (e.g., `mkdir` which touches a bitmap, two inodes, and a directory block) into single, atomic transactions.
- **Durability:** Implementing the double-`fsync` protocol to ensure journal entries and commit records are persisted to physical media in the correct order.
- **Recovery:** Automated scanning and replaying of the journal at mount-time to restore the filesystem to its last known consistent state.
- **Checkpointing:** Managing the circular buffer life cycle, including flushing journaled changes to their final locations and advancing the journal head.
- **Integration:** Converting all existing mutation logic (from M2, M3, and M4) to use a transaction-based API rather than direct block writes.

Out of Scope:

- **Data Journaling:** Only metadata (bitmaps, inodes, directory blocks) is journaled. File data blocks are written using "ordered" mode (data-first, then metadata-journal).
- **Distributed Consensus:** This is a single-node consistency mechanism.
- **Checksum Offloading:** Checksums are computed in software, not via hardware CRC32 instructions (for portability).

Invariants:

- No metadata block may be written to its permanent location until its corresponding transaction's **Commit Record** is durable on disk.
- Journal replay must be **idempotent**: replaying the same transaction multiple times must produce the same result as a single replay.
- The `j_head` must never be overtaken by `j_tail`; the filesystem must block and checkpoint if the journal is full.
- All journaled blocks must be exactly 4096 bytes and aligned to 4KB boundaries.

2. File Structure

Implementation follows a transition from data definitions to transaction logic, followed by recovery and integration.

1. `journal_types.h`: On-disk structure definitions for the journal superblock, descriptors, and commit blocks.
2. `journal_io.c`: Low-level journal region addressing and circular buffer math (`journal_abs_block`).
3. `txn.h / txn.c`: The Transaction API (`txn_begin`, `txn_journal_block`, `txn_commit`).
4. `recovery.c`: The crash recovery engine (`journal_recover`) and idempotency checks.
5. `checkpoint.c`: Journal clearing logic (`journal_checkpoint`) and background flush coordination.
6. `journal_wrappers.c`: Journaled versions of `write_inode`, `bitmap_set`, etc.
7. `test_m6_crash.c`: A simulation suite that kills the process mid-transaction to verify recovery.

---

3. Complete Data Model

3.1. Journal Superblock (`journal_super_t`)

Resides at absolute block `sb->s_journal_start`. It tracks the live region of the circular log.

Offset	Field	Type	Description
0x00	`j_magic`	`uint32_t`	`0x4A4F5552` ("JOUR").
0x04	`j_block_type`	`uint32_t`	Always `0x01` (Superblock).
0x08	`j_sequence`	`uint32_t`	Monotonic ID of the next transaction to be written.
0x0C	`j_head`	`uint32_t`	Index of the oldest unapplied transaction in the journal.
0x10	`j_tail`	`uint32_t`	Index where the next transaction will be written.
0x14	`j_total_blocks`	`uint32_t`	Number of usable blocks in the journal (usually `s_journal_len - 1`).
0x18	`j_reserved`	`uint8_t[4072]`	Padding to 4096 bytes.

3.2. Descriptor Block (`journal_desc_t`)

Describes the "payload" of a transaction. It lists which filesystem blocks are being updated.

Offset	Field	Type	Description
0x00	`j_magic`	`uint32_t`	`0x4A4F5552`.
0x04	`j_block_type`	`uint32_t`	Always `0x02` (Descriptor).
0x08	`j_sequence`	`uint32_t`	Transaction sequence ID.
0x0C	`j_nr_blocks`	`uint32_t`	Number of data blocks immediately following this descriptor.
0x10	`j_block_map[500]`	`uint64_t[500]`	The target block numbers in the primary FS for each journaled block.

3.3. Commit Block (`journal_commit_t`)

The "Atomic Sentinel". Its presence on disk signifies a successful transaction.

Offset	Field	Type	Description
0x00	`j_magic`	`uint32_t`	`0x4A4F5552`.
0x04	`j_block_type`	`uint32_t`	Always `0x03` (Commit).
0x08	`j_sequence`	`uint32_t`	Must match the corresponding Descriptor sequence.
0x0C	`j_checksum`	`uint32_t`	Cumulative sum of all bytes in the transaction's data blocks.
0x10	`j_reserved`	`uint8_t[4080]`	Padding to 4096 bytes.

3.4. In-Memory Transaction (`txn_t`)

Accumulates changes before flushing.

```c
#define MAX_TXN_BLOCKS 64


typedef struct {

    uint32_t sequence;               // Sequence number for this txn

    uint32_t nr_blocks;             // Current count of dirty metadata blocks

    uint32_t targets[MAX_TXN_BLOCKS]; // Original FS block numbers

    uint8_t  data[MAX_TXN_BLOCKS][4096]; // New content for these blocks

} txn_t;
```

4. Interface Contracts

4.1. Transaction Life Cycle

```c
/**
 * Starts a new metadata transaction.
 * @param txn Pointer to stack-allocated txn_t.
 */
void txn_begin(txn_t *txn);

/**
 * Stages a metadata block for journaling.
 * If the block is already in the txn, it is overwritten (deduplication).
 * @return 0 on success, -ENOSPC if txn is full (64 blocks).
 */
int txn_journal_block(txn_t *txn, uint32_t fs_block_num, const void *data);

/**
 * Executes the 8-step commit protocol.
 * @return 0 on success. If returns < 0, the FS must remain in its PRE-txn state.
 */
int txn_commit(txn_t *txn);
```

4.2. Recovery & Maintenance

```c
/**
 * Scans the journal and replays all committed transactions.
 * MUST be called before FUSE starts.
 */
int journal_recover(int fd, superblock_t *sb);

/**
 * Flushes the circular buffer and resets j_head to j_tail.
 */
int journal_checkpoint(int fd, superblock_t *sb);
```

5. Algorithm Specification

5.1. The `txn_commit` Protocol (8-Step)

To ensure durability, the commit follows a strict ordering of writes and barriers:

1. **Prepare Descriptor:** Populate `journal_desc_t` with `nr_blocks` and `targets`.
2. **Write Descriptor:** `write_block(abs_journal_tail++, desc)`.
3. **Write Data:** For each block in `txn->data`, `write_block(abs_journal_tail++, data)`.
4. **Barrier 1:** `fsync(fd)`. Ensures journal data is on disk before the commit record exists.
5. **Write Commit:** `write_block(abs_journal_tail++, commit_record)`.
6. **Barrier 2 (The Point of No Return):** `fsync(fd)`. Transaction is now durable.
7. **Apply In-Place:** Iterate through `txn->targets` and write `txn->data` to the primary FS region.
8. **Update Super:** Update `g_jsuper.j_tail`, `g_jsuper.j_sequence++`, and `write_block(sb->j_start, jsuper)`.

5.2. `journal_recover` (Replay Logic)

On mount, the system performs a linear scan of the journal region.

1. Load `journal_super_t`.
2. Set `current_pos = j_head`.
3. **While** `current_pos != j_tail`:
   - Read `desc = read_block(current_pos)`.
   - If `desc->j_type != JB_TYPE_DESC`, advance `current_pos` and continue.
   - Locate the **Commit Block** at `current_pos + 1 + desc->j_nr_blocks`.
   - **If** Commit Block exists AND `commit->j_sequence == desc->j_sequence` AND Checksum matches:
     - **REPLAY:** For `i` in `0..desc->j_nr_blocks`:
       - `data = read_block(current_pos + 1 + i)`.
       - `write_block(desc->j_block_map[i], data)`.
     - Advance `current_pos` past the commit block.
   - **Else**:
     - **DISCARD:** The transaction was not committed. Stop recovery.
4. Call `journal_checkpoint` to clear the log.

5.3. Ordered Data Write (Metadata-Only Mode)

When a file is written (`fs_write`), we must ensure the data blocks land on disk *before* the metadata (inode size/pointers) that references them is journaled.

1. **Write File Data:** Use `write_block` directly on the data region blocks.
2. **Barrier:** `fsync(fd)`.
3. **Start Transaction:** `txn_begin`.
4. **Journal Inode:** `txn_journal_block(inode_table_block)`.
5. **Commit:** `txn_commit`. *Why?* If we crash between 2 and 5, the data is on disk but the inode size is old. No corruption. If we crash between 4 and 5, the data is there, but the transaction isn't committed. Still no corruption.

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`ENOSPC` (Journal)	`txn_commit`	Invoke `journal_checkpoint` immediately to free space.	No (Delayed)
Checksum Fail	`journal_recover`	Discard the transaction; assume it was a partial write during crash.	No
`EIO` (Barrier 1)	`txn_commit`	Fail the transaction. Data has not been applied in-place.	Yes
`EIO` (Barrier 2)	`txn_commit`	FATAL. The filesystem state is now ambiguous.	Yes
Wrap-around	`journal_abs_block`	`index % j_total_blocks`. Ensure block 0 of region is skipped.	No

7. Implementation Sequence with Checkpoints

Phase 1: Journal Basics (3 Hours)

Define all `packed` structures. Implement `journal_abs_block` for circular addressing. Implement `journal_load` and `journal_persist_super`. **Checkpoint:** `test_journal_layout` formats a journal and verifies the superblock is written correctly at the start of the journal region.

Phase 2: The Transaction Engine (5 Hours)

Implement `txn_begin`, `txn_journal_block` (with deduplication logic), and the 8-step `txn_commit`. **Checkpoint:** Call `txn_commit` on a 5-block update. Use `hexdump` on the `.img` file to verify the Descriptor, Data, and Commit blocks appear in sequence in the journal region.

Phase 3: Recovery & Checkpointing (6 Hours)

Implement `journal_recover` and `journal_checkpoint`.

- **Idempotency Test:** Write a transaction, call `journal_recover` twice, and verify the filesystem state is identical after both calls. **Checkpoint:** Manually write a committed transaction to the journal but do *not* apply it in-place. Run `journal_recover`. Verify the primary FS blocks now match the journaled data.

Phase 4: Integration (4 Hours)

Convert `write_inode`, `bitmap_set`, and directory mutations to use `txn_t`. **Checkpoint:** `mkdir("/test")` now results in multiple blocks appearing in the journal before the directory is visible in `ls`.

Phase 5: The Crash Simulation (4 Hours)

Write a test that kills the FUSE process using `SIGKILL` during the middle of `txn_commit` (between step 3 and 5). **Checkpoint:** Mount the image again. Verify that the "interrupted" operation either fully happened or did not happen at all, but the filesystem remains mountable and `fsck` passes.

8. Test Specification

8.1. Atomic `rename` Verification

1. `rename("/A", "/B")`.
2. Internally, this involves `dir_remove(A)` and `dir_add(B)`.
3. Inject a crash between these two internal steps.
4. **Assertion:** After recovery, either the file is at `/A` OR at `/B`. It must NEVER be at both, and it must NEVER be at neither.

8.2. Journal Wrap-around

1. Set journal size to 10 blocks.
2. Write 5 transactions of 3 blocks each.
3. **Assertion:** The system correctly triggers checkpoints when `j_tail` approaches `j_head`, and the journal continues to function using circular logic.

8.3. Idempotent Replay

1. Commit a transaction that sets Inode 5's size to 100 bytes.
2. Manually run `journal_recover` twice.
3. **Assertion:** Inode 5 size is 100 bytes. The second replay did not "increment" or corrupt the value.

---

9. Performance Targets

Operation	Target	Measurement
`txn_commit` (Metadata)	< 2ms (SSD)	Two `fsync` barriers + sequential writes.
Recovery (4MB Journal)	< 50ms	Time from `journal_load` to `journal_checkpoint`.
Write Amplification	~2.5x	Ratio of (Journal Bytes + FS Bytes) / (User Bytes) for metadata.
Throughput (Ordered)	> 90% of M4	Data blocks bypass the journal; metadata is small.

---

10. Hardware Soul (Atomic Writes)

10.1. The Sector Atomicity Assumption

Our design relies on the fact that a **single sector write (512B or 4KB)** is atomic at the hardware level. The `journal_commit_t` block is the lynchpin. Because we write it in a single block operation (4KB), the hardware ensures it is either fully written or not written at all. We never have to handle a "half-written" commit record.

10.2. Barrier Analysis

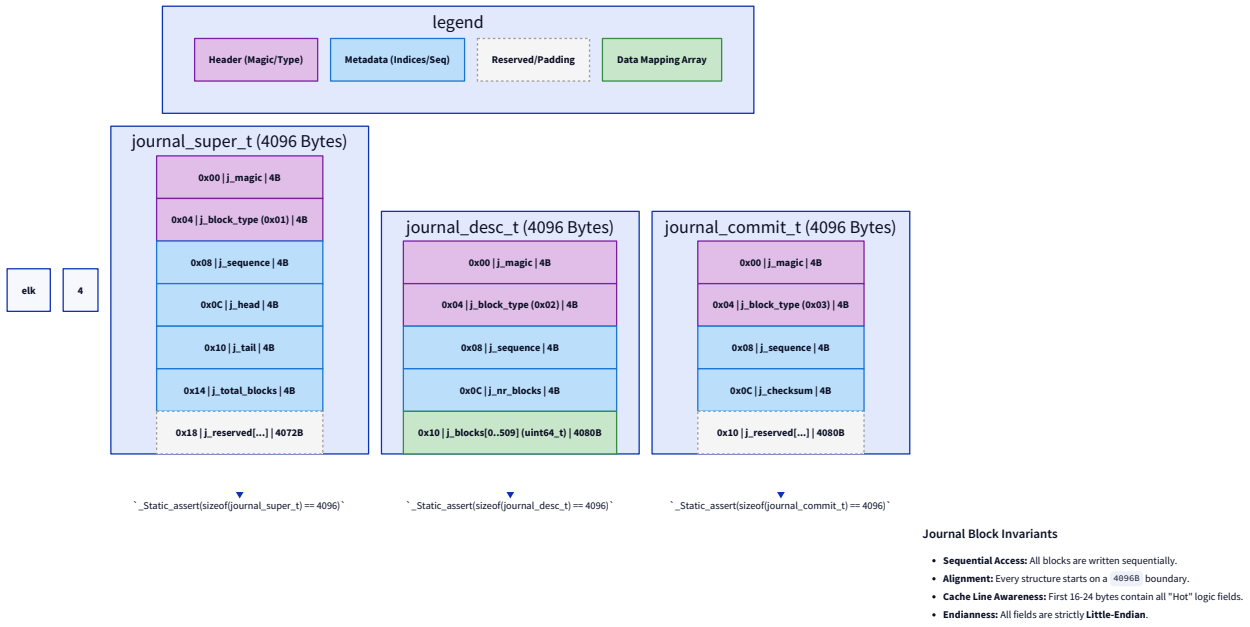
- **Barrier 1:** Protects against the disk controller's Out-of-Order execution. It ensures the "intent" (data blocks) is physically on the platters/NAND before the "permission" (commit record) is granted.

- **Barrier 2:** Protects against the "Volatile Write Cache" (DRAM on the disk). It ensures the commit record is persistent, making the transaction a permanent part of history before we begin the "messy" in-place updates.

{{DIAGRAM:tdd-diag-32}}

Circular Journal Buffer States

```
[S][D][B1][B2][C][.][.][.][.][.]
 ^  ^           ^  ^
 |  j_head      |  j_tail (Next write)
 j_super        j_tail - 1 (Last Commit)

Wrap-around:
[.][.][.][S][D][B1][B2][C][D][B1]
 ^        ^                 ^
 j_tail   j_super           j_head
```



The Transaction Write Pipeline

```
1. txn_journal_block(B5) -> Mem Buffer
2. txn_journal_block(B10) -> Mem Buffer
3. txn_commit()
   |-- Write Desc (B5, B10) to Journal
   |-- Write Data (B5_new, B10_new) to Journal
   |-- fsync() <------------------------------ Barrier 1
   |-- Write Commit to Journal
   |-- fsync() <------------------------------ Barrier 2 (Durability Point)
   |-- Write B5_new to Inode Table
   |-- Write B10_new to Inode Bitmap
   |-- Update Journal Super (Advance Tail)
```

Project Structure: Filesystem Implementation

Directory Tree

```text
filesystem-root/
├── include/                 # Header Definitions
│   ├── fs_types.h          # Global constants and magic (M1)
│   ├── block_io.h          # Block read/write prototypes (M1)
│   ├── bitmap.h            # Bitmap manipulation macros (M1)
│   ├── superblock.h        # Superblock structure (M1)
│   ├── inode.h             # Inode structure & limits (M2)
│   ├── dir.h               # Dirent structures (M3)
│   ├── file_ops.h          # File I/O prototypes (M4)
│   ├── fuse_context.h      # Global FsCtx & locking (M5)
│   ├── journal_types.h     # Journaling block types (M6)
│   └── txn.h               # Transaction API (M6)
├── src/                     # Source Implementation
│   ├── core/               # Layer 1: Block Foundation
│   │   ├── block_io.c      # lseek/read/write wrappers (M1)
│   │   ├── bitmap.c        # Bit-level search logic (M1)
│   │   └── superblock.c    # Layout & serialization (M1)
│   ├── inode/              # Layer 2: Metadata Management
│   │   ├── inode_io.c      # Inode RMW cycles (M2)
│   │   ├── inode_alloc.c   # Inode bitmap integration (M2)
│   │   ├── inode_tree.c    # Indirect pointer traversal (M2)
│   │   └── inode_free_recursive.c # Depth-first reclamation (M2)
│   ├── dir/                # Layer 3: Directory Hierarchy
│   │   ├── dir_rec.c       # Record padding/length math (M3)
│   │   ├── dir_scan.c      # Linear entry scanning (M3)
│   │   ├── dir_path.c      # Recursive path resolution (M3)
│   │   ├── dir_write.c     # Entry splitting/merging (M3)
│   │   └── dir_op.c        # mkdir/rmdir/link logic (M3)
│   ├── file/               # Layer 4: File Data Plane
│   │   ├── file_create.c   # File-specific allocation (M4)
│   │   ├── file_read.c     # Sparse hole handling (M4)
│   │   ├── file_write.c    # RMW data engine (M4)
│   │   └── file_truncate.c # Shrink/Extend space recovery (M4)
│   ├── fuse/               # Layer 5: OS Integration
│   │   ├── myfs_main.c     # libfuse entry point (M5)
│   │   ├── fuse_bridge.c   # path-to-inode translation (M5)
│   │   ├── fuse_ops_meta.c # getattr/statfs/chmod (M5)
│   │   ├── fuse_ops_dir.c  # readdir/mkdir/unlink (M5)
│   │   └── fuse_ops_file.c # open/read/write/release (M5)
│   └── journal/            # Layer 6: Consistency
│       ├── journal_io.c    # Circular buffer math (M6)
│       ├── txn.c           # 8-step commit protocol (M6)
│       ├── recovery.c      # Mount-time log replay (M6)
│       ├── checkpoint.c    # Log flushing/truncation (M6)
│       └── journal_wrappers.c # Transactional metadata APIs (M6)
├── tools/                   # Standalone Utilities
│   └── mkfs.c              # Disk image formatter (M1)
├── tests/                   # Verification Suites
│   ├── test_m1.c           # Block layer verification
│   ├── test_m2.c           # Pointer tree integrity
│   ├── test_m3.c           # Directory/Link invariants
│   ├── test_m4.c           # Sparse I/O & RMW tests
│   └── test_m6_crash.c     # SIGKILL consistency tests
├── build/                   # Compiled binaries (Artifact)
├── Makefile                 # Pkg-config & build system (M5)
├── README.md               # Project Documentation
├── disk.img                # Generated 4KB-block image (Artifact)
└── mnt/                     # FUSE mount point (Artifact)
```

Creation Order

1. **Block Layer (M1)**: Establish `block_io.c` and `mkfs.c`. Ensure you can format a 16MB file and read the magic number back via `test_m1.c`.
2. **Inode Engine (M2)**: Build the pointer tree. Crucial: Test `inode_get_block` with high offsets to verify single/double indirection.
3. **Namespace (M3)**: Implement `dir_path.c` and `dir_add_entry`. Verify that `path_resolve` can find the root inode `.` and `..`.
4. **Data Plane (M4)**: Implement `file_write.c`. Verify that partial writes trigger the Read-Modify-Write cycle and don't corrupt the rest of the block.
5. **FUSE Bridge (M5)**: Connect your C functions to `myfs_main.c`. Use the `-f -d -s` flags to watch calls flow from the OS into your code.
6. **Consistency (M6)**: Refactor metadata writes into `txn_commit`. Use `test_m6_crash.c` to verify that interrupted `mkdir` calls don't leak inodes.

File Count Summary

- Total files: 35
- Directories: 9
- Estimated lines of code: ~4,500 lines of C (excluding tests)