Chapter 1

File System II

1.1 Block Allocation Strategies

1.1.1 Limitations of Traditional Block Allocation

Files in modern operating systems typically occupy multiple blocks scattered across a disk. This creates several challenges for efficient file access and management:

- **Linked List Approach:** When blocks are linked together, accessing a file requires traversing all preceding blocks.
  - If each block access takes 100 μs, reading 5 blocks requires 500 μs.
- **File Allocation Table (FAT):** To improve performance, systems often cache the FAT in memory.
  - This approach consumes significant memory resources.
  - For each data block, metadata must be stored in the FAT.
  - Let's analyze the memory and performance implications for a large file:

Memory and Performance Analysis for FAT

Given:

- File size: 1 TB (2^{40} bytes)
- Block size: 4 KB (2^{12} bytes)
- FAT entry size: 4 bytes per block (typical)
- Block access time: 100 μs

Number of blocks needed to store the file:

$$\text{Blocks} = \frac{\text{File size}}{\text{Block size}} = \frac{1 \text{ TB}}{4 \text{ KB}} = \frac{2^{40} \text{ bytes}}{2^{12} \text{ bytes}} \tag{1.1}$$

$$= 2^{40-12} = 2^{28} \text{ blocks} \tag{1.2}$$

Memory required for FAT entries (metadata):

$$\text{FAT size} = \text{Number of blocks} \times \text{Entry size} \tag{1.3}$$

$$= 2^{28} \text{ blocks} \times 4 \text{ bytes/block} \tag{1.4}$$

$$= 2^{28+2} \text{ bytes} = 2^{30} \text{ bytes} = 1 \text{ GB} \tag{1.5}$$

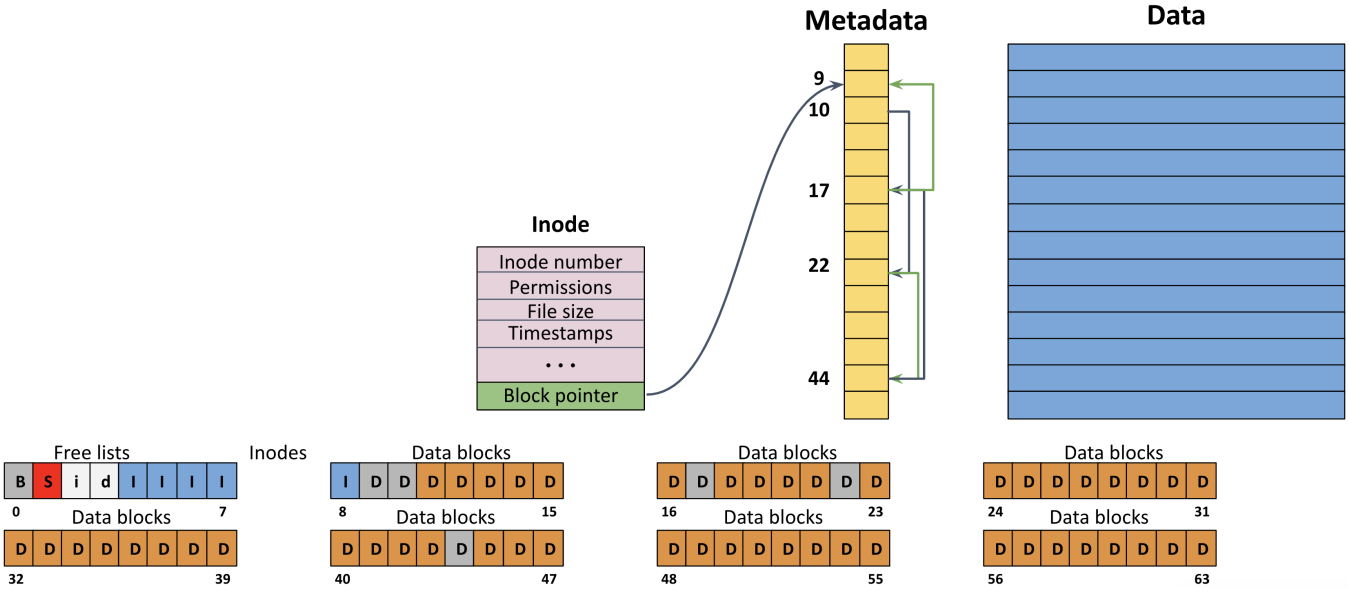
Time to access all metadata (worst case):

$$\text{Access time} = \text{Number of metadata blocks} \times \text{Block access time} \tag{1.6}$$

$$= \frac{1 \text{ GB}}{4 \text{ KB}} \times 100 \ \mu\text{s} = \frac{2^{30}}{2^{12}} \times 100 \ \mu\text{s} \tag{1.7}$$

$$= 2^{18} \times 100 \ \mu\text{s} \approx 26.2 \text{ seconds} \tag{1.8}$$

Implications: For a 1 TB file, the FAT approach requires 1 GB of memory just to store metadata. Reading all this metadata would take approximately 26 seconds, making file operations extremely slow.



1.1.2 Design Goals for Efficient Block Allocation

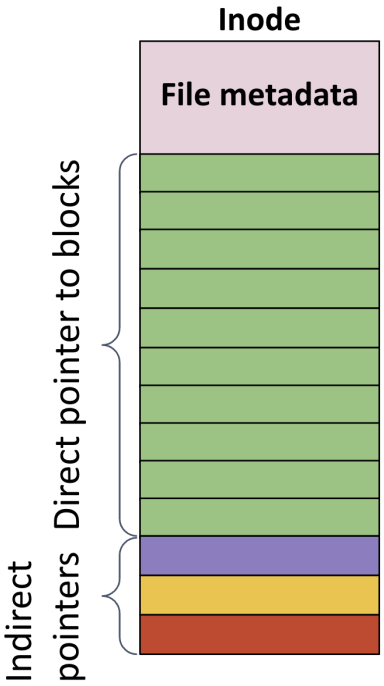
A well-designed block allocation strategy should balance several competing requirements:

- Minimize memory overhead for metadata
- Provide fast access to all parts of a file
- Support both small and large files efficiently
- Scale gracefully as file size increases

1.1.3 The Inode Approach

Key Observation: File systems must efficiently handle two common types of files:

1. **Small files** (< 50 KB)
   - Can be accessed directly with a small set of pointers
   - Direct inode pointers point to data blocks
2. **Large files**
   - Metadata blocks are allocated as the file grows
   - Similar to multi-level page tables
   - Minimizes memory waste through indirection



Inode Pointer Structure

An inode contains a fixed set of pointers that provide access to data blocks using a hierarchical addressing scheme:

Pointer Type	Description	File Size Range
Direct	First 12 pointers point directly to data blocks, providing immediate, single-step access with no indirection overhead.	Small files ($\leq$ 48 KB)
Single-Indirect	Pointer #13 points to a block of pointers where each entry points to a data block (one level of indirection).	Medium files (up to several MB)
Double-Indirect	Pointer #14 points to a block of pointers; each entry in that block points to another block, which in turn contains pointers to data blocks (two levels of indirection).	Large files (up to several GB)
Triple-Indirect	Pointer #15 points to a block of pointers; each entry points to another block of pointers, then to yet another block before finally reaching data blocks (three levels of indirection).	Very large files (up to TB range)

1.1.4 Benefits of the Inode Structure

- **Space Efficiency:** Metadata grows only as needed for larger files
- **Access Speed:** Small files can be accessed with minimal indirection
- **Scalability:** Can address extremely large files with limited overhead
- **Balanced Approach:** Optimizes for both small and large file access patterns

1.2 File Allocation Approach: Multi-level Indexing

The multi-level indexing scheme employs a tree-like structure to organize file data blocks, enhancing the efficiency of block retrieval. This approach uses a combination of direct, single indirect, double indirect, and triple indirect pointers to reference data blocks, thereby adapting the indexing depth to the file size.

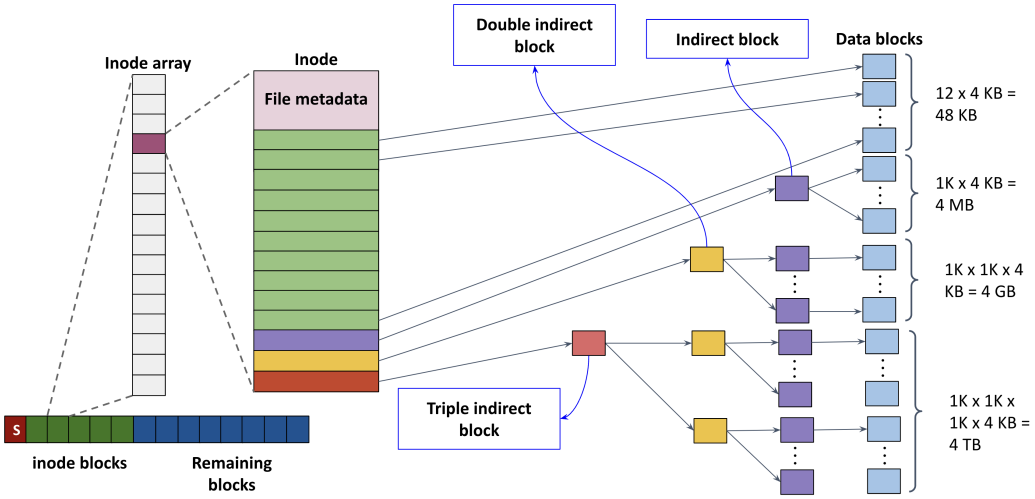
Key Features and Advantages

- **Efficient Block Location:** The tree structure allows rapid location of data blocks. Once an indirect block is read, it can reference hundreds of data blocks, making sequential read operations highly efficient.
- **Asymmetric Overhead:** The design is asymmetric, meaning that small files benefit from minimal overhead by primarily using direct pointers, while larger files leverage additional levels of indirection without incurring a prohibitive metadata cost.
- **Fixed Structure and Simplicity:** The fixed, hierarchical layout simplifies implementation. Metadata is stored separately from data, ensuring there is no conflation between file data and file system metadata.
- **No External Fragmentation:** Since data blocks are allocated without external fragmentation, the overall space utilization is improved.
- **Performance:** The structure provides reasonable read performance with low seek times, balancing the extra reads required for indirect accesses with the overall efficiency of accessing multiple blocks once an indirect block is in memory.

Dynamic Allocation and Practical Considerations

The allocation dynamics are designed to be adaptive:

- **Small Files:** For a file that contains only a few kilobytes of data, direct pointers are used. For example, reading a 4 KB block from a file accessed via a direct pointer incurs minimal overhead.
- **Large Files:** As the file size grows, additional levels of indexing are activated. With a three-level (triple indirect) indexing, even a file requiring 16 KB of data can be managed efficiently. The extra levels allow the file system to scale, enabling support for very large files without a linear increase in metadata.
- **Mixed Access Patterns:** The tree-like indexing provides a good balance between random access (via direct pointers) and sequential reads (via high-degree indirect blocks), which is beneficial for different file access patterns.



The multi-level indexing file allocation method enhances both performance and scalability by adapting the index structure to the file size, ensuring low overhead for small files while supporting efficient access for large files.

1.3 File Operations in a Filesystem

Reading and writing files in a filesystem involve complex sequences of operations that extend beyond simply accessing data. These operations require traversing directory structures, accessing metadata, and managing disk blocks. This section explores the mechanics of these fundamental operations.

1.3.1 Reading from a File

When an application reads data from a file, the operating system performs multiple disk operations to locate and retrieve the requested data. The process begins with opening the file and continues with reading data blocks as needed.

Opening a File for Reading

Before data can be read, the file must be opened:

Example 1.3.1.1 (Opening a file)**.** *`open("/cs202/w07", O_RDONLY)`*

This system call initiates a sequence of operations:

- The filesystem traverses the directory tree to locate the inode for "w07"
- It reads the inode to verify access permissions
- Upon successful verification, it returns a file descriptor that serves as a reference for subsequent operations

Reading Data

Each `read()` operation requires multiple steps:

- The filesystem reads the file's inode to locate the appropriate data blocks
- It reads the data block(s) corresponding to the current file offset
- It updates the last access time in the inode
- It updates the file offset in the in-memory open file table for the file descriptor

Example 1.3.1.2 (Reading the First Two Data Blocks from "/cs202/w07"). Let's look at the complete sequence of operations required to open a file and read its first two data blocks.

Step 1: Opening the File

1. **Root inode access:** The system reads the inode of the root directory (/) to locate its data blocks.
2. **Root directory data:** The filesystem reads the root directory's data blocks to find the entry for "cs202".
3. **cs202 inode access:** Using information from the root directory, it reads the inode for the "cs202" subdirectory.
4. **cs202 directory data:** It reads the data blocks of the "cs202" directory to locate the entry for "w07".
5. **w07 inode access:** Finally, it reads the inode associated with "w07", which contains the metadata and pointers to the file's data blocks.

At this point, the file is open and the system has established the necessary references to access its data.

Step 2: First read() Call

1. **w07 inode read:** The system reads the inode again to retrieve the pointer to the first data block and verify metadata.
2. **Data block access:** It reads the actual first data block of file "w07".
3. **Inode update:** It writes to the inode to update the last access timestamp.

Step 3: Second read() Call

1. **w07 inode read:** The system reads the inode again to retrieve the pointer to the second data block.
2. **Data block access:** It reads the second data block of file "w07".
3. **Inode update:** It writes to the inode to update the last access timestamp again.

The sequence of operations for file reads can be visualized as follows:

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs202 data	w07 data[0]	w07 data[1]
open("cs202/w07")			read()						
						read()			
				read()					
							read()		
					read()				
read()					read()				
								read()	
					write()				
read()					read()				
									read()
					write()				

1.3.2 Writing to a File

Writing to a file involves more complex operations than reading, particularly when new data blocks need to be allocated.

Opening a File for Writing

Similar to reading, writing begins with opening the file:

Example 1.3.2.1 (Opening a file for writing). `open("/cs202/w07", O_WRONLY)`

This assumes the file already exists. If it doesn't, additional operations would be required to create it.

Writing Data

Each logical write operation can generate multiple physical I/O operations:

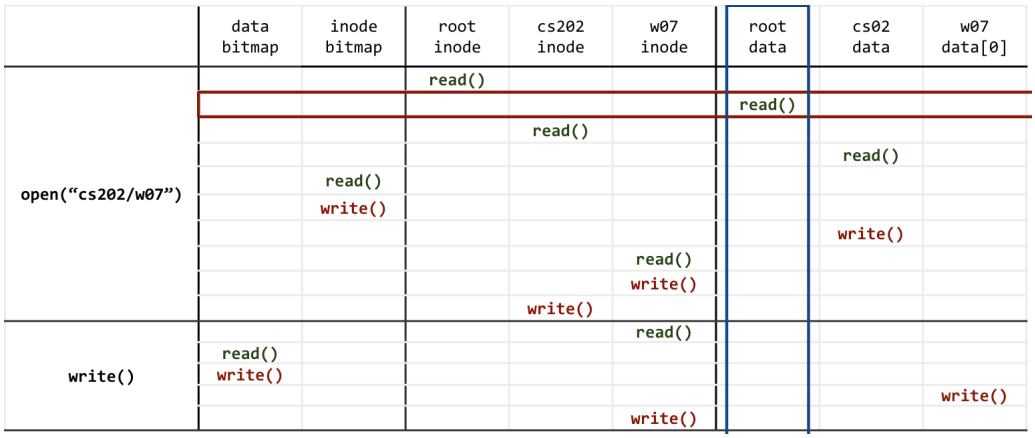
1. Read the free data block bitmap to locate available space
2. Write to the data block bitmap to mark the block as allocated
3. Read the file's inode to access its metadata
4. Write to the file's inode to update its block pointers
5. Write the actual data to the newly allocated block

File Creation and Additional Complexity

Creating a new file involves even more operations:

- Reading and writing the free inode bitmap to allocate an inode
- Writing the new inode with initial metadata
- Reading and updating the parent directory's data blocks
- If the parent directory is full, allocating new blocks for it

Example 1.3.2.2 (Creating and Writing to a New File "/cs202/w07"). Now, let's look at the complete sequence of operations required to create a new file and write its first data block.



Step 1: Creating the File

1. **Root inode access:** Reads the root directory's inode to locate its data blocks.
2. **Root directory data:** Reads the root directory's data to find the entry for "cs202".
3. **cs202 inode access:** Reads the inode for the "cs202" directory.
4. **cs202 directory data:** Reads "cs202" directory data to verify "w07" doesn't already exist.
5. **Inode bitmap operations:** Reads the inode bitmap to find a free inode, then writes to mark it as allocated.
6. **Directory update:** Updates the "cs202" directory data to include an entry for "w07" linked to the new inode.
7. **New inode initialization:** Writes initial metadata to the new inode (permissions, owner, timestamps).
8. **Parent directory update:** Updates the metadata for "cs202" (modification time, entry count).

Step 2: Writing Data to the New File

1. **w07 inode access:** Reads the new file's inode to access its metadata.
2. **Data bitmap operations:** Reads the data bitmap to find a free data block, then writes to mark it as allocated.
3. **Data write:** Writes the actual file content to the newly allocated data block.
4. **Inode update:** Updates the "w07" inode with the new file size, data block pointers, and timestamps.

1.4 File System Performance

File system performance is a critical aspect of operating system design that directly impacts user experience and application efficiency. This section explores how performance is defined, measured, and optimized in file systems.

1.4.1 Performance Metrics and Evaluation

Performance in file systems can be evaluated from multiple perspectives, each focusing on different aspects of system behavior:

Definition (File System Performance). *The measure of how efficiently a file system can execute operations such as reading, writing, and metadata manipulation, typically expressed in terms of latency, throughput, and resource utilization.*

When evaluating file system performance, several factors must be considered:

- **Operation count:** The number of I/O operations required to complete a task
- **Operation speed:** The time required to complete individual I/O operations
- **Program-level impact:** Effect on the performance of a single program
- **System-level impact:** Effect on overall system performance across all programs

These factors can be quantified using the following key metrics:

- **Latency:** The time delay between initiating and completing an operation
- **Throughput:** The amount of data processed per unit time (e.g., MB/s)
- **IOPS (I/O Operations Per Second):** The number of read/write operations a storage system can perform in one second

1.4.2 Performance Optimization Strategies

File systems employ various strategies to optimize performance, each addressing different performance bottlenecks

Definition (Block Cache). *A memory area that temporarily stores recently accessed disk blocks to reduce the need for physical disk operations when the same data is requested again.*
Caching significantly improves performance by reducing the need for slow disk operations:

- Frequently accessed blocks remain in memory, allowing `read()` operations to complete without disk I/O
- Modern systems often dedicate all unused memory to the file system buffer cache
- The cache maps file identifiers (inode, block offset) to physical memory locations (page frame numbers)

Operation Batching

Grouping multiple operations together can significantly improve overall system throughput:

Example 1.4.2.1 (Write Batching)**.** *Instead of writing data to disk immediately after each user interaction, an application can queue multiple write operations for 5 seconds and then perform them as a batch. This reduces the total number of disk accesses, improving throughput at the cost of slightly increased latency for individual operations.*

The benefits of operation batching include:

- Reduced disk seek time by grouping operations on physically proximate disk sectors
- Amortized per-operation overhead across multiple operations
- Opportunity for operation optimization and reordering

Delayed Idempotent Operations

Definition (Idempotent Operation). *An operation that can be performed multiple times without changing the final outcome beyond the initial application.*
Delaying or batching idempotent operations provides performance benefits without compromising correctness:

Example 1.4.2.2 (Timestamp Updates)**.** *Updating a file's "last accessed" timestamp can be delayed or batched because only the most recent timestamp is relevant. Multiple updates within a short time window can be coalesced into a single disk write.*

Strategic Indirection

Adding levels of indirection enables optimization opportunities:

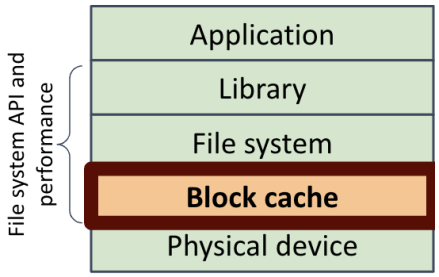
- Maintaining abstractions that decouple logical operations from physical ones
- Allowing the system to reorder or coalesce operations
- Providing flexibility in how and when operations are physically executed

1.4.3 The Block Cache Architecture

The block cache serves as a critical performance optimization layer in file systems

Example 1.4.3.1 (Block Cache Operation)**.** When an application repeatedly reads the same inode block

1. **First read:** The block is loaded from disk into the block cache
2. **Subsequent reads:** The system checks if the block is in the cache using the mapping: $\{inode, block\_offset\} \rightarrow page\_frame\_number$
3. If found, the data is returned directly from memory without disk I/O
4. The block remains in cache until memory pressure forces eviction



Key Block Cache Characteristics

- Dynamically adjusts size based on system memory availability
- Implements replacement policies to maximize cache hit rates
- Manages consistency between cached blocks and their disk versions
- May implement read-ahead or prefetching to anticipate future access patterns

These performance optimization strategies collectively ensure that file systems can deliver high throughput and low latency despite the inherent performance limitations of physical storage devices.

1.4.4 Optimizing I/O Operations Through Batching

Definition (I/O Batching). *The process of combining multiple I/O operations into larger, more efficient transfers to minimize overall system overhead and maximize throughput.*
Modern file systems employ batching strategies to address two key performance limitations:

- High latency cost per individual I/O operation
- Limited I/O operations per second (IOPS) capacity

Storage-Specific Optimizations

Different storage technologies benefit from distinct batching strategies:

- **Hard Disk Drives (HDD):**
  - Optimizes for sequential access by grouping operations on consecutive disk blocks
  - Minimizes seek time by processing physically proximate blocks together
  - Performance heavily influenced by disk fragmentation - the degree to which an inode's blocks are non-contiguous
- **Solid State Drives (SSD):**
  - Leverages internal parallelism for concurrent operations
  - Benefits from larger transfer sizes due to internal architecture
  - Less sensitive to physical block placement

1.4.5 Asynchronous Operations and Write Delays

While read operations typically require immediate process blocking, write operations present opportunities for optimization through delayed execution:

Example 1.4.5.1 (Asynchronous Write Operations)**.** *When an application writes data:*

1. *Data is initially stored in memory buffers*
2. *Write operations are queued for asynchronous processing*
3. *System performs actual disk writes within a defined interval (typically 30 seconds)*
4. *Operations may be reordered to optimize throughput*

Definition (Write Delay). *A performance optimization technique where write operations are temporarily held in memory and executed asynchronously to improve system throughput.*

Important Note: While write delays improve performance, they introduce a risk of data loss in case of system crashes before cached data is written to disk.

1.4.6 Cache Impact on Data Persistence

File systems cache multiple critical data structures to enhance performance:

- Free block and inode bitmaps
- Directory entries
- Inode metadata
- Data blocks

While caching significantly improves read performance, it introduces complexity for write operations due to the need to maintain data consistency between memory and disk.

1.4.7 Write Caching Policies

File systems implement different caching strategies to balance performance and data consistency:

Definition (Write-Back Cache). *A caching policy where modifications are initially made to the cache and later written to disk, prioritizing performance over immediate consistency.*

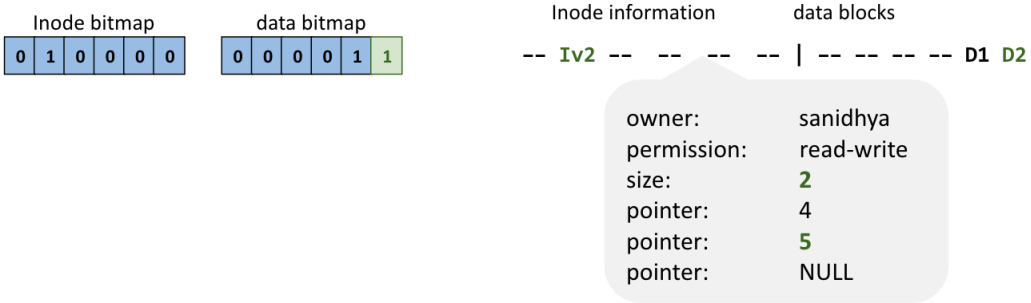
Definition (Write-Through Cache). *A caching policy where modifications are immediately written to both cache and disk, ensuring consistency at the cost of performance.*

Cache Policy	Advantages	Disadvantages
Write-Back	Higher performance Better I/O optimization	Risk of data loss during system crashes
Write-Through	Guaranteed consistency Immediate persistence	Lower performance Higher I/O overhead

Applications can force immediate disk writes using the `fsync` system call when data consistency is critical.

1.5 Crash Consistency

Suppose we are appending a data block to a file. This operation involves several steps: adding a new data block $D2$, updating the inode, and updating the data bitmap.

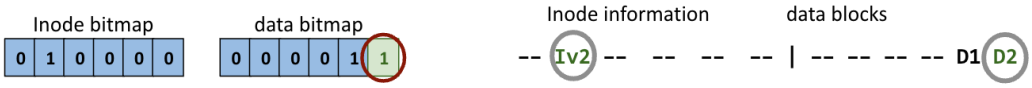


What happens if a crash or power outage occurs during these writes? The key issue is that file system operations often involve multiple write operations, and a failure between these operations can lead to an inconsistent state.

1.5.1 Single Write Scenario

Consider the case where only one write operation is successfully written to disk before a crash. Let's examine a few possibilities:

- **Data Block $D2$ is written**: The data is written, but there is no valid inode pointing to it. $D2$ appears as a free block in the metadata. The write is essentially lost, but the file system metadata structures remain consistent.
- **Inode ($Iv2$) is written**: If only the updated inode $Iv2$ is written, following the block pointer will lead to reading garbage data. This results in an inconsistent file system because the data bitmap indicates that the block is free, while the inode claims it is in use.
- **Updated Data Bitmap is written**: If only the updated data bitmap is written, the file system becomes inconsistent because the data bitmap indicates that a data block is in use, but no inode points to it.



1.5.2 Multiple Writes Scenario

Now, let's consider scenarios where two write operations succeed before a crash:

- **Inode and Data Bitmap updates succeed**: The file system remains consistent from a metadata perspective. However, reading the new block will return garbage data because the actual data block $D2$ was not successfully written.
- **Inode and Data Block updates succeed**: This leads to an inconsistent file system because the inode points to the new data block, but the data bitmap might not reflect that the block is in use.
- **Data Bitmap and Data Block updates succeed**: This also results in an inconsistent file system because the data bitmap marks the data block as used, but no inode points to it.

Caching exacerbates these issues because data can be written asynchronously, making it harder to predict the order of writes.
If the file system is interrupted between these writes, it can lead to an inconsistent state due to:

- Power loss and hard reboot
- Kernel panic
- File system bugs

Therefore, a mechanism is needed to recover from or fix these inconsistent states.

1.5.3 The Consistent Update Problem

The fundamental problem is that several file system operations update multiple data structures. Caching can worsen the issue because data can be written asynchronously. If a file system operation is interrupted between writes, it may leave the data in an inconsistent state. This can occur due to power loss, hard reboots, kernel panics, or file system bugs.
Therefore, the goal is to have a mechanism to recover from (or fix) an inconsistent state.

1.5.4 Consistency Solution #1: File System Checker (FSCK)

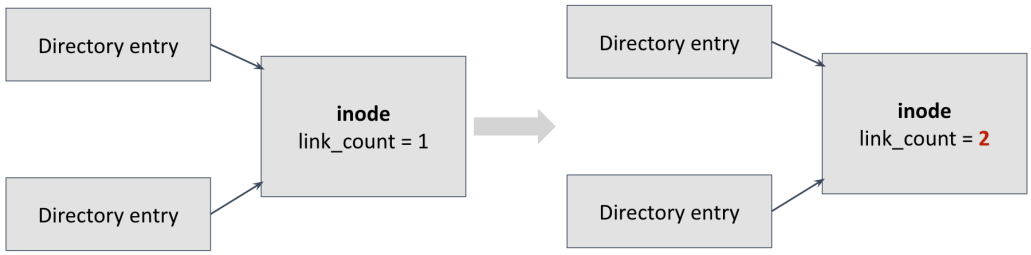
One approach to address file system inconsistencies is using a file system checker (FSCK). FSCK is a utility that checks the consistency of the file system after a certain number of mount operations or after a crash. It performs hundreds of consistency checks across different fields, such as:

- Do superblocks match?
- Is the file system size reasonable?
- Are link counts equal to the number of directory entries?

1.5.5 The File System Checker

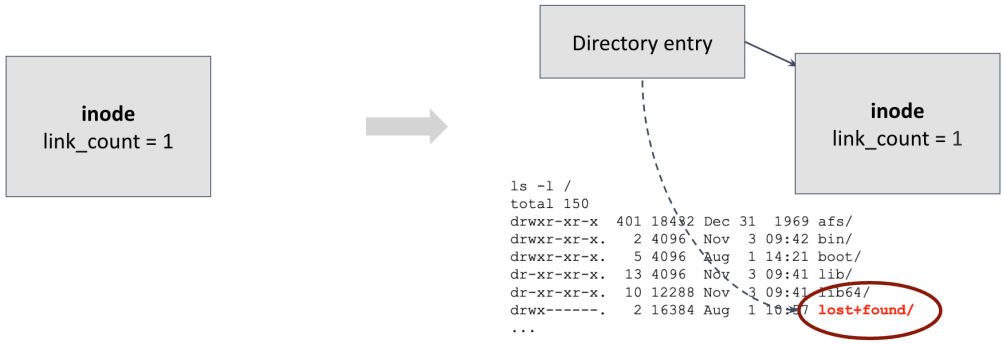
The file system checker (`fsck`) is a crucial utility for maintaining file system integrity. It is automatically invoked after a specific number of mount operations or following a system crash to ensure the file system's consistency. The `fsck` performs numerous checks across various file system components, addressing potential issues like incorrect link counts, data bitmap errors, duplicate pointers, and invalid pointers.
The following are examples of the consistency checks `fsck` performs:

- **Link Count Inconsistencies:** The `fsck` verifies that the number of directory entries pointing to an inode matches the inode's link count. For instance, if two directory entries point to the same inode but the inode's link count is set to 1, `fsck` detects this inconsistency.



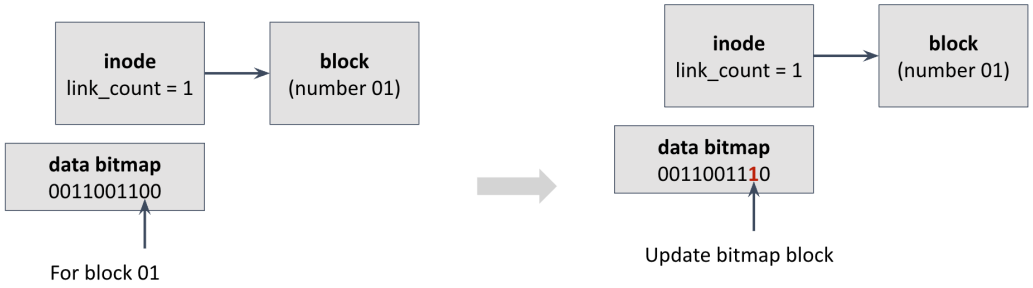
Fix the link count by increasing to 2

- **Lost Inodes:** If an inode has a link count greater than zero but no directory entries point to it, the `fsck` moves the corresponding file to the `lost+found` directory, allowing for potential recovery by the system administrator.



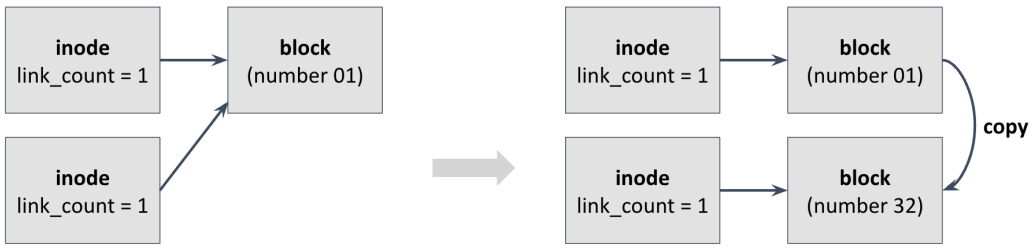
Link the file in lost+found directory

- **Data Bitmap Errors:** The `fsck` ensures the data bitmap accurately reflects the allocation status of blocks. If an inode points to a block, but the corresponding bit in the data bitmap is 0 (indicating the block is free), `fsck` corrects the bitmap to reflect the block's usage.



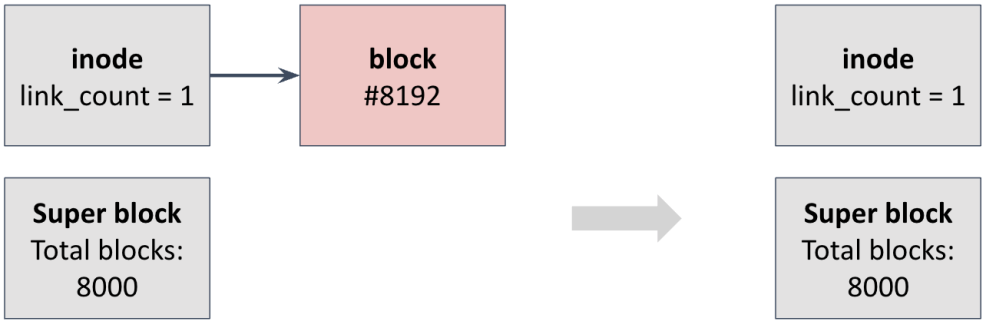
Update a reference block in the bitmap

- **Duplicate Pointers:** The `fsck` identifies and resolves situations where multiple inodes point to the same data block, which can lead to data corruption. In such cases, `fsck` may create a duplicate of the block, updating one of the inodes to point to the new duplicate, thus preserving data integrity.



Make a copy of the data block

- **Invalid Pointers:** The `fsck` checks for inodes pointing to blocks with numbers exceeding the total number of blocks in the file system. Such pointers are invalid and can cause crashes or data corruption. The `fsck` removes these invalid pointers to prevent further issues.



Remove the reference of the data block

1.5.6 Problems with FSCK

While `fsck` is essential, it has limitations:

- **Functionality:** `fsck` aims to bring the file system to a consistent state, which is not always the "correct" state. Determining the appropriate corrections can be challenging, and in severe cases, reformatting the disk may seem like the easiest solution, albeit with significant data loss.

- **Performance:** `fsck` can be slow, sometimes taking hours to complete, especially on large file systems. This prolonged downtime can be disruptive.

1.6 Consistency Solution #2: Journaling

To address the limitations of `fsck`, journaling offers an alternative approach to maintaining file system consistency with the following goals:

- Minimize the amount of work required for recovery after a crash.

- Achieve the *correct* state of the file system, not just a consistent one.

The core idea behind journaling is to record changes in a journal (or log) before applying them to the actual file system. This journal serves as a historical record of operations, allowing the system to recover to a known good state in the event of a crash. No need to scan the entire disk

Definition (Journaling). *Journaling is a technique used in file systems where all intended modifications are first recorded in a sequential log (the "journal") before being committed to the main file system. This write-ahead logging ensures atomicity and durability of operations, facilitating recovery after a crash.*

Before modifying any data (read, write, delete, etc.), the changes are first recorded in the journal. The journal is a special area on the disk that stores data in a write-ahead fashion. Journaling leverages the atomicity of transactions to provide crash consistency.

1.6.1 A Principled Approach: Transactions

Journaling relies on the concept of transactions to ensure data integrity.
Definition (Transaction). *A transaction is a group of operations treated as a single logical unit of work. It must adhere to the ACID properties: Atomicity, Consistency, Isolation, and Durability.*

- **Atomic:** A transaction is indivisible; either all operations within it are executed, or none are.
- **Consistent:** A transaction must maintain the integrity of the data. It moves the system from one valid state to another.
- **Isolated:** Concurrent transactions should not interfere with each other. The effects of one transaction should not be visible to others until it is complete.
- **Durable:** Once a transaction is committed, its effects are permanent and survive system failures.

Transactions can have two outcomes:

- **Commit:** The transaction is successfully completed, and its changes are made permanent.
- **Abort:** The transaction is terminated, and any changes made during the transaction are rolled back, restoring the system to its previous state.

1.6.2 How Journaling Works

Journaling groups file system operations into atomic and consistent units using transactions. `TxBeg` and `TxEnd` markers denote the start and end of a transaction. The process involves writing to the journal first and then writing the actual file system blocks (checkpoint) in a specific order. Journaling can be applied to both data and metadata blocks.

1.6.3 Data Journaling: An Example

Consider adding a new block `D2` to a file. This can be viewed as similar to operations in `git`. The steps involved in data journaling are:

1. Write the following blocks to the journal: `TxBeg | Iv2 | Bv2 | D2 | TxEnd`. Here, `Iv2` represents the inode update, `Bv2` represents the bitmap information, and `D2` is the new data block. Writing each record to a separate block ensures atomicity.
2. Write the blocks `Iv2`, `Bv2`, and `D2` to their respective locations in the file system (checkpoint).
3. Mark the transaction as free in the journal (i.e., remove it).

In case of a crash:

- If the crash occurs before the log is updated, the changes are ignored as if the transaction never happened.
- If the crash occurs after the log is updated but before the checkpoint, the changes are replayed from the log back to the disk during recovery.

1.6.4 Simplified Journaling Example

Consider the goal of atomically writing the value 10 to block 0 and the value 5 to block 1.

Time	Block 0	Block 1	Extra	Extra	Extra
0	12	3	0	0	0
1	10	3	0	0	0
2	10	5	0	0	0

A naive approach of directly writing to the blocks is problematic because a crash between the two writes could leave the file system in an inconsistent state.
Journaling solves this by first writing the changes to the journal within a transaction.

Time	Block 0	Block 1	Block 0'	Block 1'	Valid
0	12	3	0	0	0
1	12	3	10	0	0
2	12	3	10	5	0
3	12	3	10	5	1
4	10	3	10	5	1
5	10	5	10	5	1
6	12	3	10	5	0

The steps are as follows:

1. Write the transaction begin marker (`TxBeg`) to the journal.
2. Write the data for block 0 (value 10) to the journal.
3. Write the data for block 1 (value 5) to the journal.
4. Write a valid block indicator to the journal, signifying that the transaction completed successfully in the journal.
5. Write the data for block 0 (value 10) to block 0 in the main file system.
6. Write the data for block 1 (value 5) to block 1 in the main file system.

Time	Block 0	Block 1	Block 0'	Block 1'	Valid
0	12	3	0	0	0
1	12	3	10	0	0
2	12	3	10	5	0
3	12	3	10	5	1
4	10	3	10	5	1
5	10	5	10	5	1
6	12	3	10	5	0

Crash Scenarios:

- **Crash before time unit 3:** The file system retains the old data, as the transaction was not fully written to the journal.
- **Crash after time unit 3 but before time unit 6:** Upon recovery, the system detects the incomplete transaction in the journal and replays the changes to blocks 0 and 1, ensuring the new data is present.
- **Crash after time unit 6:** The new data is already present in the file system, and no recovery is needed.