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):

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

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

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

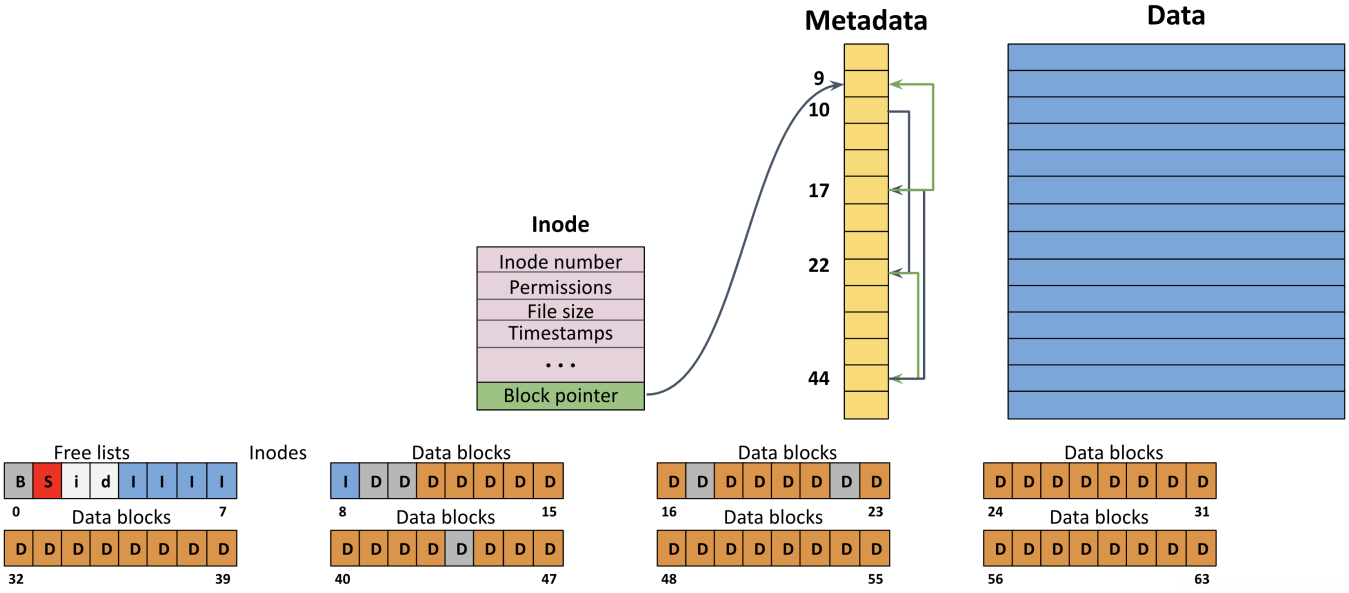
Time to access all metadata (worst case):

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

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

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

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

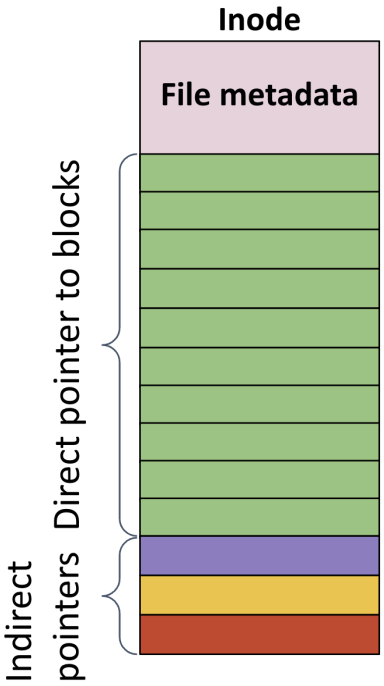
A well-designed block allocation strategy should balance several competing requirements:

- 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.

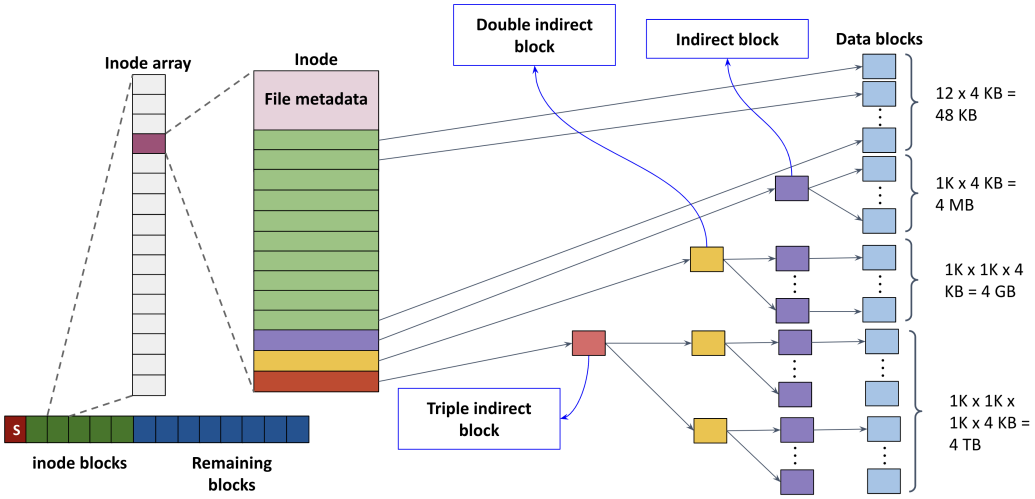
Key Features and Advantages

- **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:

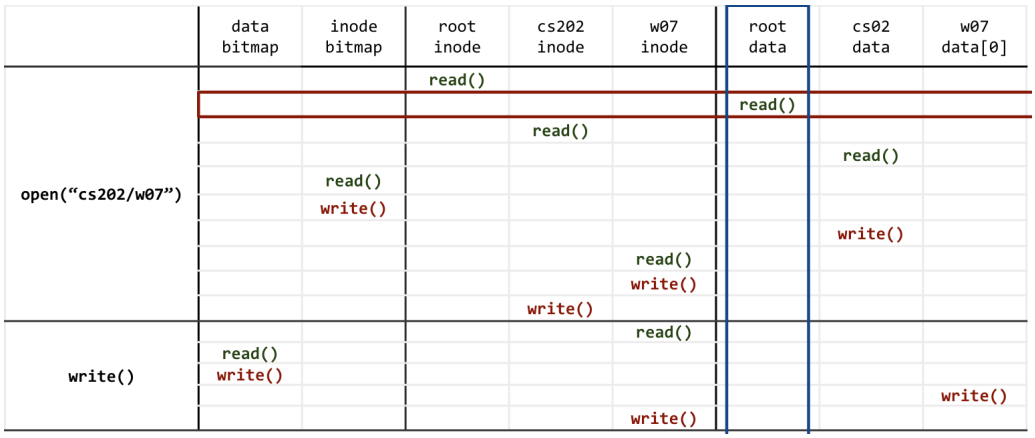
1. Read the free data block bitmap to locate available space
2. Write to the data block bitmap to mark the block as allocated
3. Read the file's inode to access its metadata
4. Write to the file's inode to update its block pointers
5. Write the actual data to the newly allocated block

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:

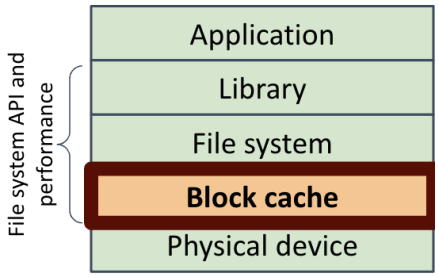
- Maintaining abstractions that decouple logical operations from physical ones
- Allowing the system to reorder or coalesce operations
- Providing flexibility in how and when operations are physically executed

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 Batching Operations

Each I/O operation is costly (latency), with limited concurrency (IOPS)
Idea: Perform fewer operations with larger transfers
HDD: when consecutive blocks on disk belong to the same inode Notion of "disk fragmentation" : a metric of what fraction of inode content are not on consecutive locations on disk
SDD: Can further exploit underlying hardware parallelism

1.4.5 Delaying Operations

A process must block on a read operation. But what about a write?
Idea: Delay all write operations Perform them asynchronously (typical: wait at most 30 seconds) Reorder operations to maximize throughput Consequence: content will be lost if the OS crashes

1.4.6 Block cache does affect data persistence

File systems maintain many data structures Bitmap of free blocks and inodes Directories Inodes Data blocks Data structure are cached for performance: Works great for read operations ... ... But what about writes?

1.4.7 Writes are mostly affected due to caching

Different caching policies possible Write-back caches Delay writes: Higher performance at the cost of potential inconsistency Write-through caches Write synchronously but poor performance (fsync)