# Section 11: File Systems, Performance, and Journaling

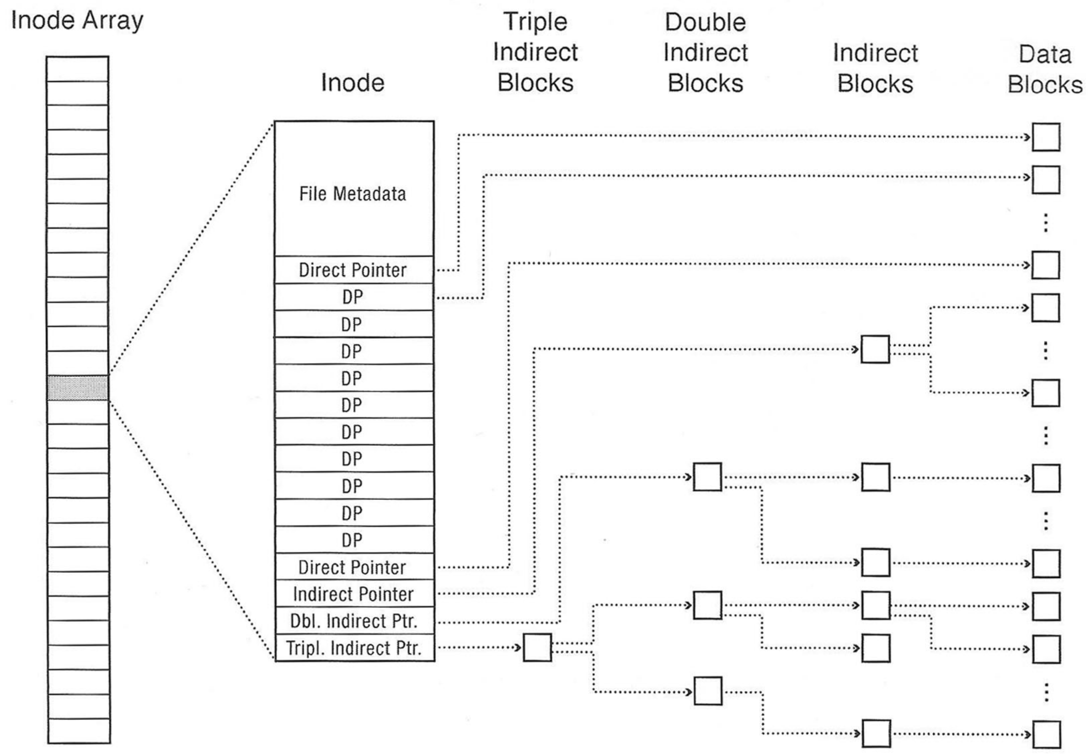
November 13-15, 2019

## Contents

## 1 Vocabulary

- **Unix File System (Fast File System)** - The Unix File System is a file system used by many Unix and Unix-like operating systems. Many modern operating systems use file systems that are based off of the Unix File System.
- **inode** - An inode is the data structure that describes the metadata of a file or directory. Each inode contains several metadata fields, including the owner, file size, modification time, file mode, and reference count. Each inode also contains several data block pointers, which help the file system locate the file's data blocks.

  Each inode typically has 12 direct block pointers, 1 singly indirect block pointer, 1 doubly indirect block pointer, and 1 triply indirect block pointer. Every direct block pointer directly points to a data block. The singly indirect block pointer points to a block of pointers, each of which points to a data block. The doubly indirect block pointer contains another level of indirection, and the triply indirect block pointer contains yet another level of indirection.



- **Transaction** - A transaction is a unit of work within a database management system. Each transaction is treated as an indivisible unit which executes independently from other transactions. The ACID properties are usually used to describe reliable transactions.
- **ACID** - An acronym standing for the four key properties of a reliable transaction.
  - *Atomicity* - The transaction must either occur in its entirety, or not at all.

  - *Consistency* - Transactions must take data from one consistent state to another, and cannot compromise data integrity or leave data in an intermediate state.
  - *Isolation* - Concurent transactions should not interfere with each other; it should appear as if all transactions are serialized.
  - *Durability* - The effect of a committed transaction should persist despite crashes.

- **Idempotent** - An idempotent operation can be repeated withiout an effect after the first iteration.
- **Logging File System** - A logging file system (or journaling file system) is a file system in which all updates are performed via a transaction log ("journal") to ensure consistency in case the system crashes or loses power. Each file system transaction is written to an append-only redo log. Then, the transaction can be committed to disk. In the event of a crash, a file system recovery program can scan the journal and re-apply any transactions that may not have completed successfully. Each transaction must be idempotent, so the recovery program can safely re-apply them.

## 2 Inode-Based File System

1. What are the advantages of an inode-based file system design compared to FAT?

   Fast random access to files. Support for hard links.

2. What is the difference between a hard link and a soft link?

   Hard links point to the same inode, while soft links simply list a directory entry. Hard links use reference counting. Soft links do not and may have problems with dangling references if the referenced file is moved or deleted. Soft links can span file systems, while hard links are limited to the same file system. Fast random access to files. Support for hard links.

3. Why do we have direct blocks? Why not just have indirect blocks?

   Faster for small files.

4. Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

   (a) How large of a disk can this file system support?

   $2^{32}$ blocks x $2^{11}$ bytes/block = $2^{43}$ = 8 Terabytes.

   (b) What is the maximum file size?

   There are 512 pointers per block (i.e. 512 4-byte pointers in 2048 byte block), so: blockSize × (numDirect + numIndirect + numDoublyIndirect + numTriplyIndirect)

$$\begin{aligned} 2048 \times (12 + 512 + 512^2 + 512^3) &= 2^{11} \times (2^2 \times 3 + 2^9 + 2^{9\times 2} + 2^{9\times 3}) \\ &= 2^{13} \times 3 + 2^{20} + 2^{29} + 2^{38} \\ &= 24K + 513M + 256G \end{aligned}$$

5. Rather than writing updated files to disk immediately, many UNIX systems use a delayed *write-behind policy* in which dirty disk blocks are flushed to disk once every $x$ seconds. List two advantages and one disadvantage of such a scheme.

   Advantage 1: The disk scheduling algorithm (i.e. SCAN) has more dirty blocks to work with at any one time and can thus do a better job of scheduling the disk arm.
   Advantage 2: Temporary files may be written and deleted before data is written to disk.
   Disadvantage: File data may be lost if the computer crashes before data is written to disk.

6. List the set of disk blocks that must be read into memory in order to read the file `/home/cs162/test.txt` in its entirety from a UNIX BSD 4.2 file system (10 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer). Assume the file is 15,234 bytes long and that disk blocks are 1024 bytes long. Assume that the directories in question all fit into a single disk block each. Note that this is not always true in reality.

   1. Read in file header for root (always at fixed spot on disk).
   2. Read in first data block for root ( / ).
   3. Read in file header for home.
   4. Read in data block for home.
   5. Read in file header for cs162.
   6. Read in data block for cs162.
   7. Read in file header for test.txt.
   8. Read in data block for test.txt.
   9. - 17. Read in second through 10th data blocks for test.txt.
   18. Read in indirect block pointed to by 11th entry in test.txt's file header.
   19. - 23. Read in 11th – 15th test.txt data blocks. The 15th data block is partially full.

## 3 I/O Performance

This question will explore the performance consequences of using traditional disks for storage. Assume we have a hard drive with the following specifications:

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queueing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

   The time to read the sector can be broken down into three parts: seek delay, rotational delay, and data transfer delay. We are already given the expected seek delay: 8 ms.

   We can assume that, on average, the hard disk must complete 1/2 revolution before the sector we are interested in reading moves under the read/write head.

   Given that the disk makes 7200 revolutions per minute, the time to complete a revolution is 60 sec/7200 Revolution $\approx$ 8.33 ms per revolution.

   The time to complete 1/2 revolution, the expected rotational delay, is $\sim 4.17$ ms.

   If the controller can transfer 50 MiB per second, it will take:

   $$4 \times 2^{10} \text{ bytes} \times \frac{1 \text{ sec.}}{50 \times 2^{20} \text{ bytes}} \approx 0.00781 \text{ ms}$$

   to transfer 4 KiB of data.

   In total, it takes 8 ms + 4.17 ms + 0.00781 ms $\approx$ 12.18 ms to read the 4 KiB sector, yielding a throughput of 4 KiB/12.18 ms $\approx$ 328.5 KiB/s

2. What is the expected throughput of the hard drive when reading 4 KiB sectors from the same track on disk (i.e., the read/write head is already positioned over the correct track when the operation starts)?

   Now, we can ignore seek delay and only need to account for rotational delay and data transfer delay.

   We already know that the expected rotational delay is 4.17 ms and we know that the expected data transfer delay is 0.00781 ms.

   Therefore, it takes a total of 4.17 ms+0.00781 ms $\approx$ 4.18 ms to read the 4 KiB sector, yielding a throughput of 4 KiB/4.18 ms $\approx$ 957 KiB/s.

3. What is the expected throughput of the hard drive when reading the very next 4 KiB sector (i.e. the read/write head is immediately over the proper track and sector at the start of the operation)?

   Now, we can ignore both rotational and seek delays. The throughput of the hard disk in this case is limited only by the controller, meaning we can take full advantage of its 50 MiB/s transfer rate.

   Note that this is roughly a 156$\times$ improvement over the random read scenario!

4. What are some ways the Unix Fast File System (FFS) was designed to deal with the discrepancy in performance we just saw?

- Attempt to keep contents of a file contiguous on disk (first-fit block allocation)
- Break disk into a set of *block groups* — sets of adjacent tracks, each with its own free space bitmap, inodes, and data blocks
- Keep a file's header information (inode) in same block group as its data blocks
- Keep files in the same directory in the same block group

## 4 Logs and Journaling

You create two new files, $F_1$ and $F_2$, right before your laptop's battery dies. You plug in and reboot your computer, and the operating system finds the following sequence of log entries in the file system's journal.

1. Find free blocks $x_1$, $x_2$, $\ldots$, $x_n$ to store the contents of $F_1$, and update the free map to mark these blocks as used.
2. Allocate a new inode for the file $F_1$, pointing to its data blocks.
3. Add a directory entry to $F_1$'s parent directory referring to this inode.
4. *Commit*
5. Find free blocks $y_1$, $y_2$, $\ldots$, $y_n$ to store the contents of $F_2$, and update the free map to mark these blocks as used.
6. Allocate a new inode for the file $F_2$, pointing to its data blocks.

What are the possible states of files $F_1$ and $F_2$ *on disk* at boot time?

- File $F_1$ may be fully intact on disk, with data blocks, an inode referring to them, and an entry in its parent directory referring to this inode.
- There may also be no trace of $F_1$ on disk (outside of the journal), if its creation was recorded in the journal but not yet applied.
- $F_1$ may also be in an intermediate state, e.g., its data blocks may have been allocated in the free map, but there may be no inode for $F_1$, making the data blocks unreachable.
- $F_2$ is a simpler case. There is no *Commit* message in the log, so we know these operations have not yet been applied to the file system.

Say the following entries are also found at the end of the log:

7. Add a directory entry to $F_2$'s parent directory referring to $F_2$'s inode.
8. *Commit*

How does this change the possible states of file $F_2$ on disk at boot time?

The situation for $F_2$ is now the same as $F_1$: the file and its metadata could be fully intact, there could be no trace of $F_2$ on disk, or any intermediate between these two states.

Say the log contained only entries (5) through (8) shown above. What are the possible states of file $F_1$ on disk at the time of the reboot?

We can now assume that $F_1$ is fully intact on disk. The log entries for its creation are only removed from the journal when the operation has been fully applied on disk.

What is the purpose of the *Commit* entries in the log?

- The *Commit* entry makes the creation of each file *atomic*. These changes to the file system's on-disk structures are either completely applied or not applied at all.
- The creation of a file involves multiple steps (allocating data blocks, setting up the inode, etc.) that are not inherently atomic, nor is the action of recording these actions in the journal, but we want to treat these steps as a single logical transaction.
- Appending the final *Commit* entry to the log (a single write to disk) *is* assumed to be an atomic operation and serves as the "tipping point" that guarantees the transaction is eventually applied.

When recovering from a system crash and applying the updates recorded in the journal, does the OS need to check if these updates were partially applied before the failure?

No. The operation for each log entry (e.g., updating an inode or a directory entry) is assumed to be *idempotent*. This greatly simplifies the recovery process, as it is safe to simply replay each committed transaction in the log, whether or not it was previously applied.