Section 9: File Systems, I/O Performance

April 3, 2020

Contents

1 Vocabulary

- **FAT** - In FAT, the disk space is still viewed as an array. The very first field of the disk is the boot sector, which contains essential information to boot the computer. A super block, which is fixed sized and contains the metadata of the file system, sits just after the boot sector. It is immediately followed by a file allocation table (FAT). The last section of the disk space is the data section, consisting of small blocks with size of 4 KiB.

  In FAT, a file is viewed as a linked list of data blocks. Instead of having a "next block pointer" in each data block to make up the linked list, FAT stores these pointers in the entries of the file allocation table, so that the data blocks can contain 100% data. There is a 1-to-1 correspondence between FAT entries and data blocks. Each FAT entry stores a data block index. Their meaning is interpreted as:

  If $N > 0$, N is the index of next block

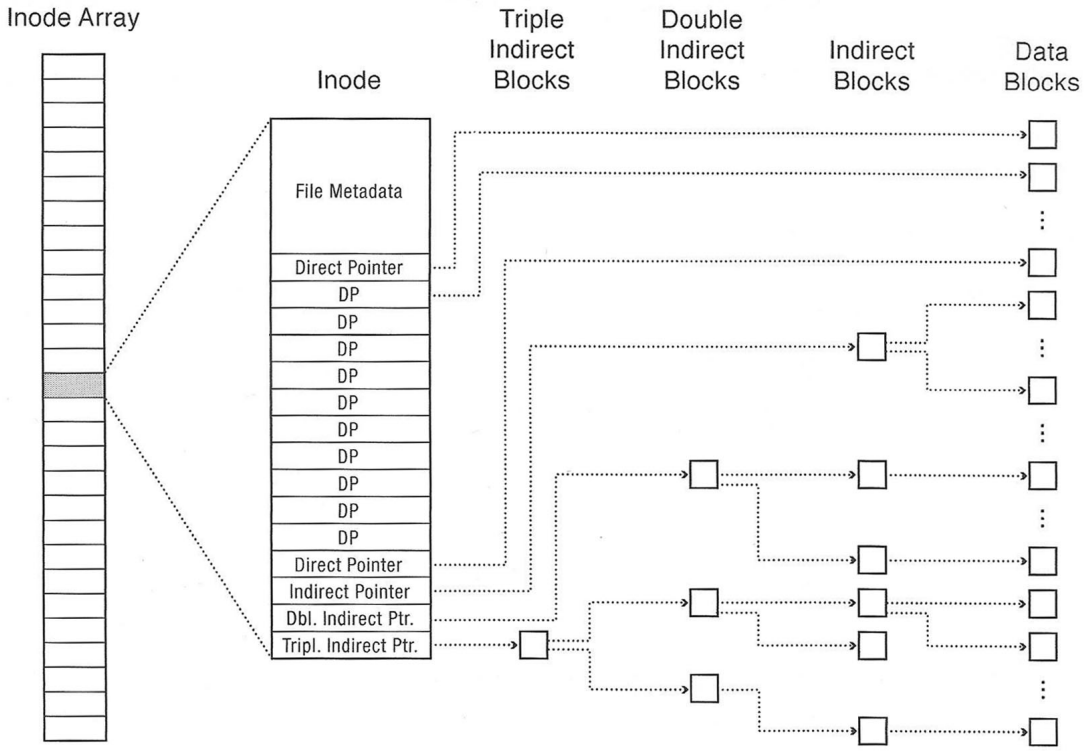
  If $N = 0$, it means that this is the end of a file

  If $N = -1$, it means this block is free

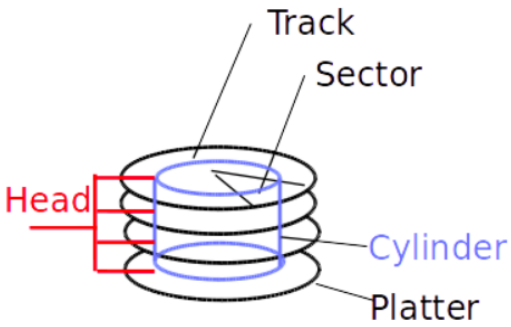
- **Unix File System (Fast File System)** - The Unix File System is a file system used by many Unix and Unix-like operating systems. Many modern operating systems use file systems that are based off of the Unix File System.

- **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.



- **I/O** - In the context of operating systems, input/output (I/O) consists of the processes by which the operating system receives and transmits data to connected devices.
- **Response Time** - Response time measures the time between a requested I/O operating and its completion, and is an important metric for determining the performance of an I/O device.
- **Throughput** - Another important metric is throughput, which measures the rate at which operations are performed over time.
- **Hard Disk Drive (HDD)** - A storage device that stores data on magnetic disks. Each disk consists of multiple **platters** of data. Each platter includes multiple concentric **tracks** that are further divided into **sectors**. Data is accessed (for reading or writing) one sector at a time. The **head** of the disk can transfer data from a sector when positioned over it.



- **Seek Time** - The time it takes for an HDD to reposition its disk head over the desired track.
- **Rotational Latency** - The time it takes for the desired sector to rotate under the disk head.
- **Transfer Rate** - The rate at which data is transferred under the disk head.

2 FAT File System

Calculate the maximum file size for a file in FAT. Now calculate the maximum file size for a file in the Unix file system. (Assume a block size of 4KiB. In FAT, assume file sizes are encoded as 4 bytes. In FFS block pointers are 4 bytes long.)

FAT : 4GB
FFS : Since a disk block is 4KB (2^{12} bytes) and a block number is 4 (2^2) bytes, there are $2^{10} = 1024$ entries per indirect block. Therefore, the maxium number of blocks of a file that could be referenced by the i-node is $12 + 2^{10} + (2^{10})^2 + (2^{10})^3 = 12 + 2^{10} + 2^{20} + 2^{30}$. Thus, the maximum size of the file would be $(12 + 2^{10} + 2^{20} + 2^{30}) \times 2^{12} = (12 \times 2^{12}) + 2^{22} + 2^{32} + 2^{42} = 48\text{KB} + 4\text{MB} + 4\text{GB} + 4\text{TB}$.

Suppose that there is a 1TB disk, with 4KB disk blocks. How big is the file allocation table in this case? Would it be feasible to cache the entire file allocation table to improve performance ?

There has to be a FAT entry for each disk block. Since the disk is 2^{40} bytes and a disk block is 2^{12} bytes, the number of disk blocks (and thus the number of FAT entries) is $2^{40}/2^{12} = 2^{28}$. Since there are 2^{28} entries, a block number (disk address) requires a minimum of $\log(2^{28})$ bits = 4 bytes. In this case, the minimum amount of space occupied by the FAT is the number of entries (2^{28}) times the 4 bytes per entry, namely $2^{30} = 1$GB. Imagine if this has to be in memory for improved performance!

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

   Faster for small files.

3. 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)

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

4. 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.

5. 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.

4 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× 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