Section 9: File Systems

William Walker

July 23, 2019

Contents

1 Warmup **2**

2 Vocabulary **3**

3 Problems **6**

3.1 Comparison of File Systems 6

3.2 Hard links 7

3.3 Disk Locality 7

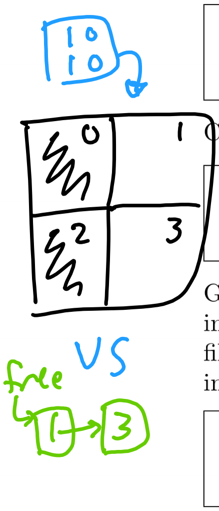
1 Warmup

What file access pattern is particularly suited to chained file allocation on disk?

Sequential reads/writes

What file allocation strategy is most appropriate for random access files?

indexed allocation ⇒ same (ish) lookup time for every access (no need to chase pointers)



Compare bitmap-based allocation of blocks on disk with a free block list.

bitmap always takes up space (0/1 state needed for all blocks) free list shrinks ⇒ can use up that space for more data.

Given that the maximum file size supported by the combination of direct, single indirection, double indirection, and triple indirection in an inode-based filesystem is approximately the same as the maximum file size supported by a filesystem that only uses triple indirection, why not simply use only triple indirection to locate all file blocks?

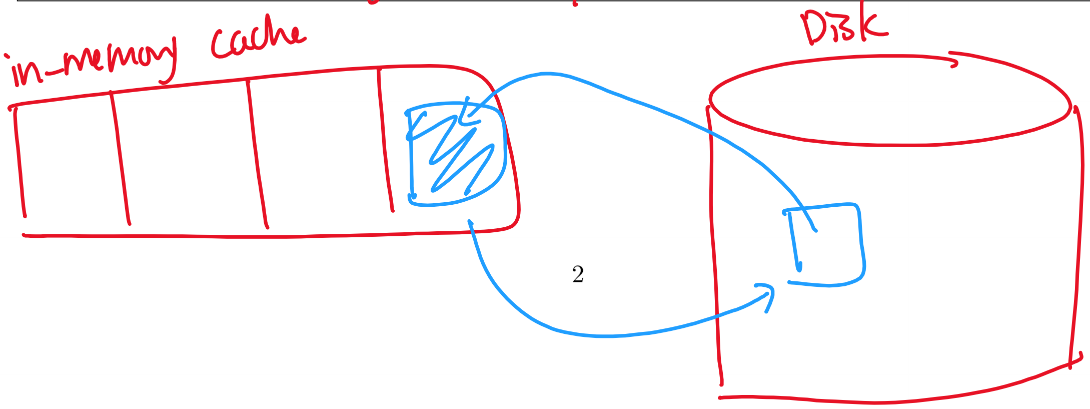
MOVED TO MON

What is the reference count field in the inode? You should consider its relationship to directory entries in you answer.

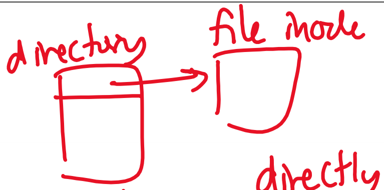
MOVED TO MON

The filesystem buffer cache does both buffering and caching. Describe why buffering is needed. Describe how buffering can improve performance (potentially to the detriment of file system robustness). Describe how the caching component of the buffer cache improves performance.

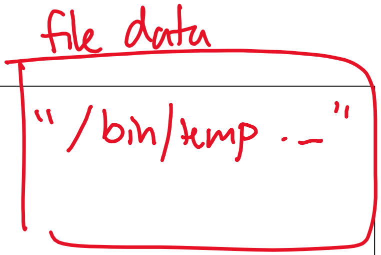
Buffer: less ops to disk, but more data loss on improper shutdown
– also can't just write 1B to disk, need to write full block
⇒ read into mem, do ops, write full block back
Cache: use block again (temporal locality) – e.g. use metadata block twice
use nearby data to your location (spatial locality)



What is the difference between a hard link and a symbolic link?



hard: directly to file inode

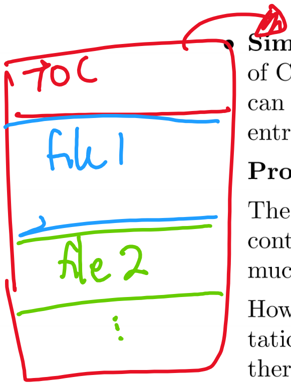


Soft link: file references another file

What data structures that must be updated when a file /foo/bar.txt is created in FFS? Assume that /foo/ already exists and that the new file is one block long (4KB).

MONDAY

2 Vocabulary



- **Simple File Syetem** - The disk is treated as a big array. At the beginning of the disk is the Table of Content (TOC) field, followed by data field. Files are stored in data field contigously, but there can be unused space between files. In the TOC field, there are limited chunks of file discription entries, with each entry discribing the name, start location and size of a file.

  Pros and Cons

  The main advantage of this implementation is simplicity. Whenever there is a new file created, a continuous space on disk is allocated for that file, which makes I/O (read and write) operations much faster.

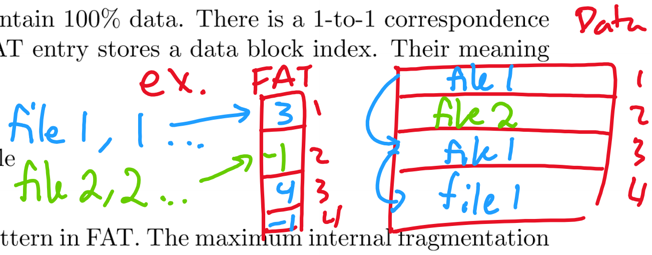
  However, this implementation also has many disadvantages. First of all, it has external fragmentation problem. Because only continuous space can be utilized, it may come to the situation that there is enough free space in sum, but none of the continuous space is large enough to hold the whole file. Second, once a file is created, it cannot be easily extended because the space after this file may already be occupied by another file. Third, there is no hierarchy of directories and no notion of file type.

- **External Fragmentation** - External fragmentation is the phenomenon in which free storage becomes divided into many small pieces over time. It occurs when an application allocates and deallocates regions of storage of varying sizes, and the allocation algorithm responds by leaving the allocated and deallocated regions interspersed. The result is that although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application.

- **Internal Fragmentation** - Internal fragmentation is the space wasted inside of allocated memory blocks because of the restriction on the minimum allowed size of allocated blocks.

- **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 $N = -1$, it means that this is the end of a file

If $N = 0$, it means this block is free

Thus, a file can be stored in a non-continuous pattern in FAT. The maximum internal fragmentation equals to 4095 bytes (4K bytes - 1 byte).

Directory in the FAT is a file that contains directory entries. The format of directory entries look as follows:

Name — Attributes — Index of 1st block — Size

Pros and Cons

Now we have a review of the pros and cons about FAT. Readers will find most of the following features have been already talked about above. So we only give a very simple list of these features.

Pros: no external fragmentation, can grow file size, has hierarchy of directories

Cons: no pre-allocation, disk space allocation is not contiguous (accordingly read and write operations will slow), assume File Allocation Table fits in RAM. Otherwise lseek and extending a file would take intolerably long time due to frequent memory operation.

- **Unix File System** (or FFS-Fast File System) - A file system used by many Unix and Unix-like operating systems, although there may be some implementation differences between different operating systems.

  Basically, disk space is divided into five parts

  Boot Sector

  Super Block

  MONDAY

  Free Block Bitmap: This part indicates which block is reserved and which is not. Each bit represents one block. If a bit in the Free Block Bitmap is set to 1, it indicates the corresponding block is free; otherwise the corresponding block has been allocated. Free Block Bitmap is much more efficient than the File Allocation Table since one bit is used to tell whether a block is allocated or not. This feature allows Unix-like file system to scale up to large disk space without wasting too much capacity. Assume each block has size of 8 KiB, a 1 TiB disk would contain 2^{27} blocks. So the Free Block Bitmap only takes 16 MiB space, which is 1/65536 of total disk space. Meanwhile, with FAT, it would take 1 GiB to store File Allocation Table (1/1024 of disk space).

  Inode Table: The inode table stores all the inodes. We will explain the inode below.

  Data Blocks

  MONDAY

- **Inodes** - An inode is the data structure that describes the meta data of files (regular files and directories). One inode represents one file or one directory. An inode is composed of several fields such as ownership, size, modification time, mode (permissions), reference count, and data block pointers (which point to the data blocks that store the content of the file). Note that the inode does not include a file/directory name. A file's inode number can be found using the `ls -i` command.

  Each inode has 12 direct block pointers (different file system may have different number of direct block pointers). Every direct block pointer directly points to a data block.

  For larger files with more than 12 data blocks, inodes also support indirect block pointers. Indirect block pointers point to data blocks which store direct block pointers, rather than to data blocks themselves. There is 1 singly-indirect, 1 doubly-indirect, and 1 triply-indirect block pointer.

- **NTFS** -

  NTFS (New Technology File System) is a proprietary file system developed by Microsoft.

  Each file on an NTFS volume is represented by a record in a special file called the master file table (MFT). NTFS reserves the first 16 records of the table for special information.

  The first record of this table describes the master file table itself, followed by a MFT mirror record.

  If the first MFT record is corrupted, NTFS reads the second record to find the MFT mirror file, whose first record is identical to the first record of the MFT. The locations of the data segments for both the MFT and MFT mirror file are recorded in the boot sector.

  http://ntfs.com/ntfs-mft.htm for more info

- **Unlink** - Unlink is the syscall used to delete a file. It decrements the reference count in the inode by 1 and removes the file entry from its parent directory. The inode and data itself are garbage-collected when reference count equals to 0.

3 Problems

3.1 Comparison of File Systems

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.)

mon

FAT32 : $2^{32}-1$: 4GiB - 32 bit file size, no larger

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 ?

$2^{40}/2^{12} = 2^{28}$ blocks ⇒ each entry needs $\log(2^{28})$ bits ≈ 4B

⇒ $2^{28} \cdot 4B = 1GB$ large FAT Table

In lecture three file descriptor structures were discussed: (a) Indexed files. (b) Linked files. (c) Contiguous (extent-based) allocation.

Each of the structures has its advantages and disadvantages depending on the goals for the file system and the expected file access pattern. For each of the following situations, rank the three structures in order of preference. Be sure to include the justification for your rankings.

(a) You have a file system where the most important criteria is the performance of sequential access to very large files.

1) extents - contiguously allocated
2) linked - simple lookup to next node
3) indexed - more complex lookup for larger files

(b) You have a file system where the most important criteria is the performance of random access to very large files.

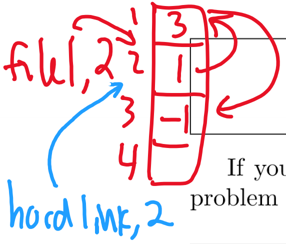
1) extents - contiguous allocation → easy to get to data
2) indexed - no need to follow list
3) linked - traverse entire chan of blocks = bad

(c) You have a file system where the most important criteria is the utilization of the disk capacity (i.e. getting the most file bytes on the disk).

1) linked - less wasted space for pointers
2) indexed - uses full blocks for ind, dbl, tpl ⇒ more wasted space
3) extents - external fragmentation

3.2 Hard links

A hard link is an entry in a directory that refers to the same file as another directory entry (in the same directory or not). If you were to try to implement hard links in an OS that uses a FAT, what would be contained in the directory entry for a hard link?



# of first block

If you didn't change the structure of a FAT (as found, for example, in DOS), what would be the problem with your hard link implementation?

no metadata about # references ⇒ delete ⇒ other links might not know this file is gone.

3.3 Disk Locality

Suppose you have a directory `/foo/` that contains 100 files, each consisting of 1 block size worth of text. Your disk is divided into tracks of 25 blocks each, and it contains a track buffer. You want to run `wc /foo/*`.

Consider two file systems - (a) FAT ; and (b) FFS that uses block groups that can each hold 20 data blocks each (each block group takes up 1 sector and the remaining 5 blocks contain the inodes and other metadata). Assume that the entire FAT table is cached in memory, and that the directory listing of `/foo/` are cached in memory (but none of the data or inodes of the files inside of `foo` are in memory).

(Average rotational latency : 10ms, Seek latency : 10ms, Time to read or write one track : 10ms)

Step 1: What is the average time needed for `wc /foo/*` to process all 100 files, assuming an optimal layout of the data on disk?

Monday

Step 2 : Now suppose you save the output of wc into a file `/foo/output.txt`. The contents of the output takes 1 block size worth of data. Assuming that all of the data you read in Step 1 is still cached in memory and that there is plenty of free disk space, what is the worst case time needed to create this new file and update the relevant metadata for it?

Monday