Section 10: File Systems and Queuing Theory

October 23, 2015

Contents

1 Warmup

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

Sequential file access is suited to chained file allocation. The chain is effectively a linked list that is traversed as the file is read. Random access is poor, because each prior link of the chain must be considered to get to any point of the file.

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

Indexed allocation is appropriate for random access, since it takes constant time (slowing down very slightly with increased indirection levels) to access any part of the file.

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

Bitmap based block allocation is a fixed size proportional to the size of the disk. This means wasted space when the disk is full. A free block list shrinks as space is used up, so when the disk is full, the size of the free block list is tiny. However, contiguous allocation is easier to perform with a bitmap.

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?

Triple indirection is slower, as it may result in multiple seeks to get to the desired block. Seeks take a long time. Tiny files that only use a few KB will perform much better if they do not have to do seeks to find each level of indirection.

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

The reference count is the number of hard links presently pointing at a particular inode. This includes parent pointers "..", self pointers ".", and children pointers (directory listing). When this reaches zero, it is an indication that that inode can be deleted, since nothing points at it anymore.

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.

Buffering is required when the unit of transfer/update is different between two entities (e.g. updating 1 byte in a disk block requires buffering the disk block in memory). Buffering can improve performance as writes to disk can be accumulated and written to disk in batches. However, this reduces the file system's robustness against power failures, because buffered writes may not make it to disk until much later.

Caching is keeping the data of recently used disk blocks in memory, so that subsequent reads to those blocks can be serviced by the cache and do not have to access the disk. The acceleration is based on the principle of locality, which says that if data is used, data nearby will be used soon, and the fact that memory is several orders of magnitude faster than disk.

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

A hardlink isn't a pointer to a file, it's a directory entry (a file) pointing to the same inode. Even if you change the name of the other file, a hardlink still points to the file. If you replace the other file with a new version (by copying a file over it), a hardlink will not point to the new file. You can only have hardlinks within the same filesystem. With hardlinks you don't have concept of the original files and links: all are equal (think of it as a reference to an object). It's a very low level concept.

On the other hand, a symlink points to a path on the file system (a directory path plus a file name); it resolves the name of the file each time you access it through the symlink. If you move the file, the symlink will not follow. If you replace the file with another one, keeping the name, the symlink will point to the new file. Symlinks can span across different filesystems. With symlinks you have very clear distinction between the actual file and symlink, which stores no information about the file it points to, except its path.

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).

The file is new, so it needs a new inode. Thus you need to update the inode bitmap and write the contents of the new inode to disk. The file has data, so you have to update the data bitmap and write the data block to disk. This is a total of four writes to the block group. The directory entry `/foo/` needs to be updated. Thus data block for "foo" needs to updated, and if required a new indirect block needs to be added (in which case the associated bitmap also needs to be updated). The metadata in the inode of foo needs to be updated to reflect the last-modified-time metadata field and the reference count.

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

  If $N = -1$, 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

  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

- **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

- **Log-structured file system** - A log-structured filesystem is a file system in which data and metadata are written to a circular buffer, called a log. The circular log is written sequentially, and starts over at the beginning when the end of the log is reached.

  This has several important side effects:

  - Write throughput on optical and magnetic disks is improved because they can be batched into large sequential runs and costly seeks are kept to a minimum.
  - Writes create multiple, chronologically-advancing versions of both file data and meta-data.
  - Recovery from crashes is simpler. Upon its next mount, the file system does not need to walk all its data structures to fix any inconsistencies, but can reconstruct its state from the last consistent point in the log.

- **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.

- **Queuing Theory** Here are some useful symbols: (both the symbols used in lecture and in the book are listed)

  - μ is the average service rate (jobs per second)
  - T_{ser} or S is the average service time, so $T_{ser} = \frac{1}{\mu}$
  - λ is the average arrival rate (jobs per second)
  - U or u or ρ is the utilization (fraction from 0 to 1), so $U = \frac{\lambda}{\mu} = \lambda S$
  - T_q or W is the average queuing time (aka waiting time) which is how much time a task needs to wait before getting serviced (it does not include the time needed to actually perform the task)
  - T_{sys} or R is the response time, and it's equal to $T_q + T_{ser}$ or $W + S$
  - L_q or Q is the average length of the queue, and it's equal to λT_q (this is Little's law)

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.)

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} = 1GB. Imagine if this has to be in memory for improved performance!

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. c (extent-based)
2. b (linked)
3. a (indexed)

It is easy to see that (c) is the best structure for sequential access to very large files, since in (c) files are contiguously allocated and the next block to read is physically the next on the disk. No seek time to find the next block, and each block will be read sequentially as the disk head moves.

Both (b) and (a) require some look up operation in order to know where the next block is. However, (b) may be slightly more expensive, since for "very large files", multiple disk accesses are required to read the indirect blocks.

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

1. c (extent-based)
2. a (indexed)
3. b (linked)

(c) is still the best structure here: just need to use an offset.

(a) will probably need to look at some levels of indirect blocks in order to find the right block to access (we are dealing with very large files).

(b) is absolutely the worst structure. In fact, in order to find a random block, we will need to traverse a linked list of blocks, which will take a time linear in the offset size.

(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. b (linked)
2. a (indexed)
3. c (extent-based)

(c) can suffer heavily of external fragmentation, especially for large files. So it is not the best structure for getting the most bytes on the disk, since lots of space will be wasted. However, for small files and large block size, (c) might prove to be better than (a) and (b).

(a) and (b) stuctures are generally more suitable for this question. The metadata overhead for (b) is likely to be smaller than the one for (a), since it only needs pointers to the next allocated block rather than an entire block (or blocks) which may or may not be totally used.

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?

Using a FAT, each entry in a directory contains the first block number for that file. This is because the blocks of a file is determined by its first block, i.e. the entry point into the FAT. Thus, a hard link might be implemented in a FAT system by having a directory entry contain the same first block number as another directory entry.

If you didnt change the structure of a FAT (as found, for example, in DOS), what would be the problem with your hard link implementation?

The problem is that a FAT entry, unlike an i-node, doesnt keep track of how many directory entries refer to that entry. Thus, if a user deletes a file even if another hard link to that file exists the block of that file will be put back on the list of free blocks (and eventually get overwritten).

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?

FAT : There are 100 data blocks to read, so that's 4 tracks. To read 4 tracks, we need $4(10\text{ms} + 10\text{ms} + 10\text{ms}) = 120\text{ms}$ on average.
FFS : There are 100 data blocks and 100 inodes to read. Since inodes and data blocks tend to be stored on the same block group, we only need to read $100/20 = 5$ sectors, which takes

5(10ms + 10ms + 10ms) = 150ms on average.

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?

FAT : We need to create a new FAT entry for the new file, update the directory listing of `/foo/`, and write one data block for the new file. In the worst case, they are all on 3 different sectors, so this is 3(10ms + 10ms + 10ms) = 90ms.
FFS : The new inode and new data block should be in the same block group, but in the worst case, the `/foo/` directory listing is in a block group that is full. So, we need to write two new block groups, which takes 2(10ms + 10ms + 10ms) = 60ms

3.4 Queuing Theory

Explain intuitively why response time is nonlinear with utilization. Draw a plot of utilization (x axis) vs response time (y axis) and label the endpoints on the x axis.

Even with high utilization (99%), some of the time (1%), the server is idle, which is a waste. All this wasted time adds up, and in the steady state, the queue becomes very long.
Graph should be linear-ish close to $u = 0$ and grow asymptotically toward ∞ at $u = 1$.

If 50 jobs arrive at a system every second and the average response time for any particular job is 100ms, how many jobs are in the system (either queued or being serviced) on average at a particular moment? Which law describes this relationship?

$50 \times 0.1 = 5$ (5 jobs at any time). This is Little's law.

Is it better to have N servers that can process 1 job per millisecond or 1 server that can process N jobs per millisecond? (Assume servers never crash or fail.) Give reasons to justify your answer.

One server that can process N jobs per millisecond is faster. Better response time ($\frac{1}{N}$ms vs 1ms) and better utilization, which gives you lower queuing delays on average.

What is the average queueing time for a work queue with 1 server, average arrival rate of λ, average service time S, and squared coefficient of variation of service time C?

$T_q = T_{ser}(\frac{u}{1-u})(\frac{C+1}{2})$ where $u = \lambda S$

What does it mean if $C = 0$? What does it mean if $C = 1$?

If $C = 0$, then your arrival rate is regular and deterministic, which means that tasks arrive at a constant rate.
If $C = 1$, then your arrival rate can be modeled as a Poisson distribution, and the interval between arrivals can be modeled as a exponential distribution.

3.5 Tying it all together

Assume that you have a disk with the following parameters:

- 1TB in size

- 6000RPM
- Data transfer rate of 4MB/s (4×10^6 bytes/sec)
- Average seek time of 3ms
- I/O controller with 1ms of controller delay
- Block size of 4000 bytes

What is the average rotational delay?

$\frac{1}{2} \times \frac{60\text{sec/minute}}{6000\text{RPM}} = 5\text{ms}$

What is the average time it takes to read 1 random block? Assume no queuing delay.

$\frac{4{,}000\text{bytes}}{4{,}000{,}000\text{bytes/sec}} = 1\text{ms}$, and $1 + 3 + 5 + 1 = 10\text{ms}$

Will the actual measured average time to read a block from disk (excluding queuing delay) tend to be lower, equal, or higher than this? Why?

It will be lower, because the operating system will use a disk scheduling algorithm to improve locality. This model assumes seeking from one track to another random track.

Assume that the average I/O operations per second demanded is 50 IOPS. Assume a squared coefficient of variation of $C = 1.5$. What is the average queuing time and the average queue length?

$$T_q = T_{ser}(\frac{u}{1-u})(\frac{C+1}{2})$$
$$u = \lambda T_{ser}$$
$$u = 50\text{IOPS} \times 0.01\text{sec}$$
$$u = 0.5$$
$$T_q = 10\text{ms}(\frac{0.5}{1-0.5})(\frac{1.5+1}{2})$$
$$T_q = 12.5\text{ms}$$
$$L_q = T_q\lambda$$
$$L_q = 0.0125 \times 50$$
$$L_q = 0.625 \text{ operations}$$