41

Locality and The Fast File System

When the UNIX operating system was first introduced, the UNIX wizard himself Ken Thompson wrote the first file system. We will call that the "old UNIX file system", and it was really simple. Basically, its data structures looked like this on the disk:

The super block (S) contained information about the entire file system: how big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth. The inode region of the disk contained all the inodes for the file system. Finally, most of the disk was taken up by data blocks.

The good thing about the old file system was that it was simple, and supported the basic abstractions the file system was trying to deliver: files and the directory hierarchy. This easy-to-use system was a real step forward from the clumsy, record-based storage systems of the past, and the directory hierarchy a true advance over simpler, one-level hierarchies provided by earlier systems.

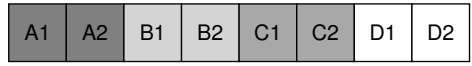
41.1 The Problem: Poor Performance

The problem: performance was terrible. As measured by Kirk McKusick and his colleagues at Berkeley [MJLF84], performance started off bad and got worse over time, to the point where the file system was delivering only 2% of overall disk bandwidth!

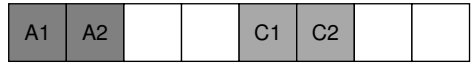
The main issue was that the old UNIX file system treated the disk like it was a random-access memory; data was spread all over the place without regard to the fact that the medium holding the data was a disk, and thus had real and expensive positioning costs. For example, the data blocks of a file were often very far away from its inode, thus inducing an expensive seek whenever one first read the inode and then the data blocks of a file (a pretty common operation).

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed. The free list would end up pointing to a bunch of blocks spread across the disk, and as files got allocated, they would simply take the next free block. The result was that a logically contiguous file would be accessed by going back and forth across the disk, thus reducing performance dramatically.

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:



As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say we now wish to allocate a file E, of size four blocks:

You can see what happens: E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4. This fragmentation problem happened all the time in the old UNIX file system, and it hurt performance. (A side note: this problem is exactly what disk defragmentation tools help with; they will reorganize on-disk data to place files contiguously and make free space one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes)

One other problem: the original block size was too small (512 bytes). Thus, transferring data from the disk was inherently inefficient. Smaller blocks were good because they minimized **internal fragmentation** (waste within the block), but bad for transfer as each block might require a positioning overhead to reach it. We can summarize the problem as follows:

THE CRUX:
HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE

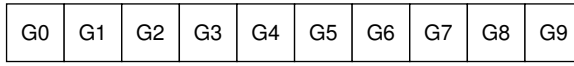
How can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? How do we make the file system "disk aware"?

41.2 FFS: Disk Awareness Is The Solution

A group at Berkeley decided to build a better, faster file system, which they cleverly called the **Fast File System (FFS)**. The idea was to design the file system structures and allocation policies to be "disk aware" and thus improve performance, which is exactly what they did. FFS thus ushered in a new era of file system research; by keeping the same *interface* to the file system (the same APIs, including `open()`, `read()`, `write()`, `close()`, and other file system calls) but changing the internal *implementation*, the authors paved the path for new file system construction, work that continues today. Virtually all modern file systems adhere to the existing interface (and thus preserve compatibility with applications) while changing their internals for performance, reliability, or other reasons.

41.3 Organizing Structure: The Cylinder Group

The first step was to change the on-disk structures. FFS divides the disk into a bunch of groups known as **cylinder groups** (some modern file systems like Linux ext2 and ext3 just call them **block groups**). We can thus imagine a disk with ten cylinder groups:



These groups are the central mechanism that FFS uses to improve performance; by placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.

Thus, FFS needs to have the ability to allocate files and directories within each of these groups. Each group looks like this:

We now describe the components of a cylinder group. A copy of the **super block** (S) is found in each group for reliability reasons (e.g., if one gets corrupted or scratched, you can still mount and access the file system by using one of the others).

Within each group, we need to track whether the inodes and data blocks of the group are allocated. A per-group **inode bitmap** (ib) and **data bitmap** (db) serve this role for inodes and data blocks in each group. Bitmaps are an excellent way to manage free space in a file system because it is easy to find a large chunk of free space and allocate it to a file, perhaps avoiding some of the fragmentation problems of the free list in the old file system.

Finally, the inode and data block regions are just like in the previous very simple file system. Most of each cylinder group, as usual, is comprised of data blocks.

ASIDE: **FFS FILE CREATION**

As an example, think about what data structures must be updated when a file is created; assume, for this example, that the user creates a new file `/foo/bar.txt` and that the file is one block long (4KB). The file is new, and thus needs a new inode; thus, both the inode bitmap and the newly-allocated inode will be written to disk. The file also has data in it and thus it too must be allocated; the data bitmap and a data block will thus (eventually) be written to disk. Hence, at least four writes to the current cylinder group will take place (recall that these writes may be buffered in memory for a while before the write takes place). But this is not all! In particular, when creating a new file, we must also place the file in the file-system hierarchy; thus, the directory must be updated. Specifically, the parent directory `foo` must be updated to add the entry for `bar.txt`; this update may fit in an existing data block of `foo` or require a new block to be allocated (with associated data bitmap). The inode of `foo` must also be updated, both to reflect the new length of the directory as well as to update time fields (such as last-modified-time). Overall, it is a lot of work just to create a new file! Perhaps next time you do so, you should be more thankful, or at least surprised that it all works so well.

41.4 Policies: How To Allocate Files and Directories

With this group structure in place, FFS now has to decide how to place files and directories and associated metadata on disk to improve performance. The basic mantra is simple: *keep related stuff together* (and its corollary, keep unrelated stuff far apart).

Thus, to obey the mantra, FFS has to decide what is "related" and place it within the same block group; conversely, unrelated items should be placed into different block groups. To achieve this end, FFS makes use of a few simple placement heuristics.

The first is the placement of directories. FFS employs a simple approach: find the cylinder group with a low number of allocated directories (because we want to balance directories across groups) and a high number of free inodes (because we want to subsequently be able to allocate a bunch of files), and put the directory data and inode in that group. Of course, other heuristics could be used here (e.g., taking into account the number of free data blocks).

For files, FFS does two things. First, it makes sure (in the general case) to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data (as in the old file system). Second, it places all files that are in the same directory in the cylinder group of the directory they are in. Thus, if a user creates four files, `/dir1/1.txt`, `/dir1/2.txt`, `/dir1/3.txt`, and `/dir99/4.txt`, FFS would try to place the first three near one another (same group) and the fourth far away (in some other group).

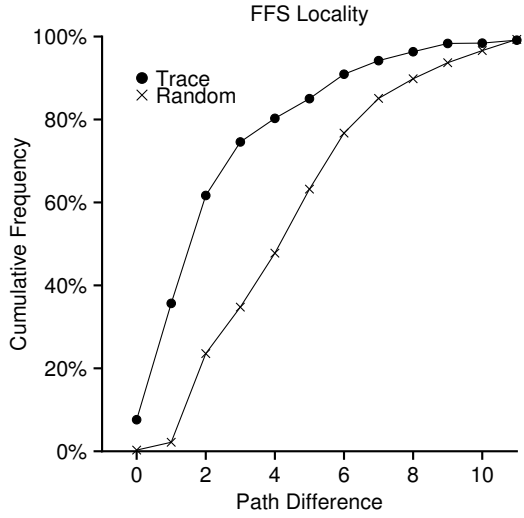


Figure 41.1: **FFS Locality For SEER Traces**

It should be noted that these heuristics are not based on extensive studies of file-system traffic or anything particularly nuanced; rather, they are based on good old-fashioned common sense (isn't that what CS stands for after all?). Files in a directory *are* often accessed together (imagine compiling a bunch of files and then linking them into a single executable). Because they are, FFS will often improve performance, making sure that seeks between related files are short.

41.5 Measuring File Locality

To understand better whether these heuristics make sense, we decided to analyze some traces of file system access and see if indeed there is namespace locality; for some reason, there doesn't seem to be a good study of this topic in the literature.

Specifically, we took the SEER traces [K94] and analyzed how "far away" file accesses were from one another in the directory tree. For example, if file `f` is opened, and then re-opened next in the trace (before any other files are opened), the distance between these two opens in the directory tree is zero (as they are the same file). If a file `f` in directory `dir` (i.e., `dir/f`) is opened, and followed by an open of file `g` in the same directory (i.e., `dir/g`), the distance between the two file accesses is one, as they share the same directory but are not the same file. Our distance metric, in other words, measures how far up the directory tree you have to travel to find the *common ancestor* of two files; the closer they are in the tree, the lower the metric.

Figure 41.1 shows the locality observed in the SEER traces over all workstations in the SEER cluster over the entirety of all traces. The graph plots the difference metric along the x-axis, and shows the cumulative percentage of file opens that were of that difference along the y-axis. Specifically, for the SEER traces (marked "Trace" in the graph), you can see that about 7% of file accesses were to the file that was opened previously, and that nearly 40% of file accesses were to either the same file or to one in the same directory (i.e., a difference of zero or one). Thus, the FFS locality assumption seems to make sense (at least for these traces).

Interestingly, another 25% or so of file accesses were to files that had a distance of two. This type of locality occurs when the user has structured a set of related directories in a multi-level fashion and consistently jumps between them. For example, if a user has a `src` directory and builds object files (`.o` files) into a `obj` directory, and both of these directories are sub-directories of a main `proj` directory, a common access pattern will be `proj/src/foo.c` followed by `proj/obj/foo.o`. The distance between these two accesses is two, as `proj` is the common ancestor. FFS does *not* capture this type of locality in its policies, and thus more seeking will occur between such accesses.

We also show what locality would be for a "Random" trace for the sake of comparison. We generated the random trace by selecting files from within an existing SEER trace in random order, and calculating the distance metric between these randomly-ordered accesses. As you can see, there is less namespace locality in the random traces, as expected. However, because eventually every file shares a common ancestor (e.g., the root), there is some locality eventually, and thus random trace is useful as a comparison point.

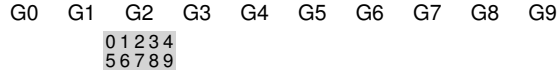
41.6 The Large-File Exception

In FFS, there is one important exception to the general policy of file placement, and it arises for large files. Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others). Filling a block group in this manner is undesirable, as it prevents subsequent "related" files from being placed within this block group, and thus may hurt file-access locality.

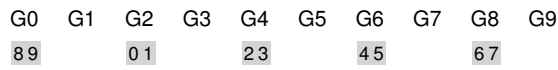
Thus, for large files, FFS does the following. After some number of blocks are allocated into the first block group (e.g., 12 blocks, or the number of direct pointers available within an inode), FFS places the next "large" chunk of the file (e.g., those pointed to by the first indirect block) in another block group (perhaps chosen for its low utilization). Then, the next chunk of the file is placed in yet another different block group, and so on.

Let's look at some pictures to understand this policy better. Without the large-file exception, a single large file would place all of its blocks into one part of the disk. We use a small example of a file with 10 blocks to illustrate the behavior visually.

Here is the depiction of FFS without the large-file exception:



With the large-file exception, we might see something more like this, with the file spread across the disk in chunks:



The astute reader will note that spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access (e.g., when a user or application reads chunks 0 through 9 in order). And you are right! It will. We can help this a little, by choosing our chunk size carefully.

Specifically, if the chunk size is large enough, we will still spend most of our time transferring data from disk and just a relatively little time seeking between chunks of the block. This process of reducing an overhead by doing more work per overhead paid is called **amortization** and is a common technique in computer systems.

Let's do an example: assume that the average positioning time (i.e., seek and rotation) for a disk is 10 ms. Assume further that the disk transfers data at 40 MB/s. If our goal was to spend half our time seeking between chunks and half our time transferring data (and thus achieve 50% of peak disk performance), we would thus need to spend 10 ms transferring data for every 10 ms positioning. So the question becomes: how big does a chunk have to be in order to spend 10 ms in transfer? Easy, just use our old friend, math, in particular the dimensional analysis we spoke of in the chapter on disks:

$$\frac{40\ \cancel{MB}}{\cancel{sec}} \cdot \frac{1024\ KB}{1\ \cancel{MB}} \cdot \frac{1\ \cancel{sec}}{1000\ \cancel{ms}} \cdot 10\ \cancel{ms} = 409.6\ KB \tag{41.1}$$

Basically, what this equation says is this: if you transfer data at 40 MB/s, you need to transfer only 409.6KB every time you seek in order to spend half your time seeking and half your time transferring. Similarly, you can compute the size of the chunk you would need to achieve 90% of peak bandwidth (turns out it is about 3.69MB), or even 99% of peak bandwidth (40.6MB!). As you can see, the closer you want to get to peak, the bigger these chunks get (see Figure 41.2 for a plot of these values).

FFS did not use this type of calculation in order to spread large files across groups, however. Instead, it took a simple approach, based on the structure of the inode itself. The first twelve direct blocks were placed in the same group as the inode; each subsequent indirect block, and all the blocks it pointed to, was placed in a different group. With a block size of 4KB, and 32-bit disk addresses, this strategy implies that every 1024 blocks of the file (4MB) were placed in separate groups, the lone exception being the first 48KB of the file as pointed to by direct pointers.

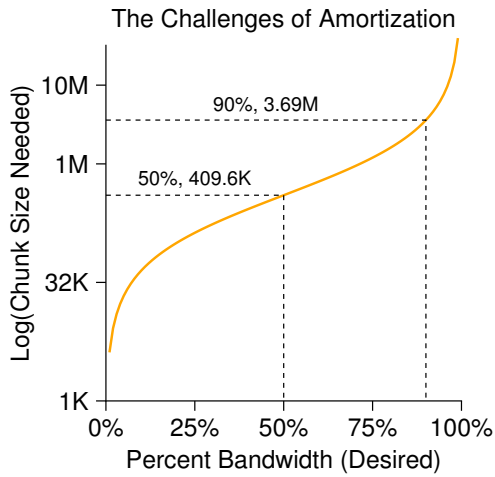


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

We should note that the trend in disk drives is that transfer rate improves fairly rapidly, as disk manufacturers are good at cramming more bits into the same surface, but the mechanical aspects of drives related to seeks (disk arm speed and the rate of rotation) improve rather slowly [P98]. The implication is that over time, mechanical costs become relatively more expensive, and thus, to amortize said costs, you have to transfer more data between seeks.

41.7 A Few Other Things About FFS

FFS introduced a few other innovations too. In particular, the designers were extremely worried about accommodating small files; as it turned out, many files were 2KB or so in size back then, and using 4KB blocks, while good for transferring data, was not so good for space efficiency. This **internal fragmentation** could thus lead to roughly half the disk being wasted for a typical file system.

The solution the FFS designers hit upon was simple and solved the problem. They decided to introduce **sub-blocks**, which were 512-byte little blocks that the file system could allocate to files. Thus, if you created a small file (say 1KB in size), it would occupy two sub-blocks and thus not waste an entire 4KB block. As the file grew, the file system will continue allocating 512-byte blocks to it until it acquires a full 4KB of data. At that point, FFS will find a 4KB block, *copy* the sub-blocks into it, and free the sub-blocks for future use.

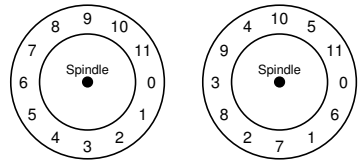


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

You might observe that this process is inefficient, requiring a lot of extra work for the file system (in particular, a lot of extra I/O to perform the copy). And you'd be right again! Thus, FFS generally avoided this pessimal behavior by modifying the `libc` library; the library would buffer writes and then issue them in 4KB chunks to the file system, thus avoiding the sub-block specialization entirely in most cases.

A second neat thing that FFS introduced was a disk layout that was optimized for performance. In those times (before SCSI and other more modern device interfaces), disks were much less sophisticated and required the host CPU to control their operation in a more hands-on way. A problem arose in FFS when a file was placed on consecutive sectors of the disk, as on the left in Figure 41.3.

In particular, the problem arose during sequential reads. FFS would first issue a read to block 0; by the time the read was complete, and FFS issued a read to block 1, it was too late: block 1 had rotated under the head and now the read to block 1 would incur a full rotation.

FFS solved this problem with a different layout, as you can see on the right in Figure 41.3. By skipping over every other block (in the example), FFS has enough time to request the next block before it went past the disk head. In fact, FFS was smart enough to figure out for a particular disk *how many* blocks it should skip in doing layout in order to avoid the extra rotations; this technique was called **parameterization**, as FFS would figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme.

You might be thinking: this scheme isn't so great after all. In fact, you will only get 50% of peak bandwidth with this type of layout, because you have to go around each track twice just to read each block once. Fortunately, modern disks are much smarter: they internally read the entire track in and buffer it in an internal disk cache (often called a **track buffer** for this very reason). Then, on subsequent reads to the track, the disk will just return the desired data from its cache. File systems thus no longer have to worry about these incredibly low-level details. Abstraction and higher-level interfaces can be a good thing, when designed properly.

Some other usability improvements were added as well. FFS was one of the first file systems to allow for **long file names**, thus enabling more expressive names in the file system instead of a the traditional fixed-size approach (e.g., 8 characters). Further, a new concept was introduced

TIP: MAKE THE SYSTEM USABLE
Probably the most basic lesson from FFS is that not only did it introduce the conceptually good idea of disk-aware layout, but it also added a number of features that simply made the system more usable. Long file names, symbolic links, and a rename operation that worked atomically all improved the utility of a system; while hard to write a research paper about (imagine trying to read a 14-pager about "The Symbolic Link: Hard Link's Long Lost Cousin"), such small features made FFS more useful and thus likely increased its chances for adoption. Making a system usable is often as or more important than its deep technical innovations.

called a **symbolic link**. As discussed in a previous chapter, hard links are limited in that they both could not point to directories (for fear of introducing loops in the file system hierarchy) and that they can only point to files within the same volume (i.e., the inode number must still be meaningful). Symbolic links allow the user to create an "alias" to any other file or directory on a system and thus are much more flexible. FFS also introduced an atomic `rename()` operation for renaming files. Usability improvements, beyond the basic technology, also likely gained FFS a stronger user base.

41.8 Summary

The introduction of FFS was a watershed moment in file system history, as it made clear that the problem of file management was one of the most interesting issues within an operating system, and showed how one might begin to deal with that most important of devices, the hard disk. Since that time, hundreds of new file systems have developed, but still today many file systems take cues from FFS (e.g., Linux ext2 and ext3 are obvious intellectual descendants). Certainly all modern systems account for the main lesson of FFS: treat the disk like it's a disk.

References

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems.
August, 1984. Volume 2, Number 3.
pages 181-197.
McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.

[P98] "Hardware Technology Trends and Database Opportunities"
David A. Patterson
Keynote Lecture at the ACM SIGMOD Conference (SIGMOD '98)
June, 1998
A great and simple overview of disk technology trends and how they change over time.

[K94] "The Design of the SEER Predictive Caching System"
G. H. Kuenning
MOBICOMM '94, Santa Cruz, California, December 1994
According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.