Introduction

The first file system created by the UNIX guru, Ken Thompson. The "old UNIX file system" was basic. On the disk, its data structures looked like this:

INSERT IMAGE

The super block (S) held information about the entire file system, like the volume size, inode count, and a link to the free list of blocks. The inode area of the disk included all file system inodes. Finally, data blocks occupied most of the disk.

The previous file system was basic and supported the fundamental abstractions of files and directory hierarchies. This easy-to-use system was a huge improvement over older record-based storage systems, and the directory hierarchy was a huge improvement over simpler one-level hierarchies.

The Poor Performance Problem

The old system's performance degraded with time, to the point where the file system delivered only 2% of total disk bandwidth!

Problem was, the original UNIX file system viewed the disk like random-access memory; data was distributed all around, ignoring the fact that it was a disk, causing actual and expensive placement costs. For example, reading a file's inode and subsequently its data blocks required a costly seek (a pretty common operation).

Worse, the file system would become fragmented due to poor free space management. Allocating files just took the next free block from the free list. As a result, a logically contiguous file would be accessed by traversing the disk, lowering performance.
Consider the following data block region with four files (A, B, C, and D) of size 2 blocks each:

INSERT IMAGE

Delete B and D for the following layout:

INSERT IMAGE

As you can see, the vacant area is split into two two-block sections instead of one big four-block chunk. Assume you want to allocate a four-block file E:

You can see: Because E is distributed over the disk, you don't receive peak (sequential) performance while accessing it. Instead, read E1 and E2, then seek, then E3 and E4. The old UNIX file system had this fragmentation issue, which hampered performance. Disk defragmentation software help solve this problem by rearranging data on the disk to make space for one or a few contiguous regions, moving data around and rewriting inodes and such to reflect the changes.

Another issue: the block size was too small (512 bytes). So data transfer from disk was intrinsically inefficient. Smaller blocks were useful for reducing internal fragmentation (waste within the block), but unsuitable for transfer because each unit required positioning. So, the issue:

MANAGING ON-DISK DATA FOR BETTER PERFORMANCE

How can we organize data structures in a file system to improve performance?
In addition to the data structures, what kinds of allocation policies do we

need?
How do we make the file system “disk aware”?

FFS: Disk Awareness

The Fast File System’s (FFS)   goal was to make the file system structures and allocation policies “disk aware” to boost performance.
FFS preserved the same file system interface (the same APIs, including open(), read(), write(), close(), and other file system calls) while changing the internal implementation.

Modern file systems almost always keep the same interface (and program compatibility) while upgrading their internals for performance, reliability, or other reasons.

Organizing Structure: The Cylinder Group

The initial step was to modify on-disk files. FFS partitions the disk into cylinder groups. A single cylinder is a group of tracks on a hard drive that are all the same distance from the drive's center. FFS groups N consecutive cylin- ders, so the disk can be seen as a collection of cylinder groups. Here's a simple example with four outside tracks and a cylinder group of three cylinders:

INSERT IMAGE

The file system needs more information than current drives provide; disks export a logical address space of blocks and hide specifics of their geometry from clients. The address space of a disk is divided into block groups by current file systems (such Linux ext2, ext3, and ext4). The image below shows an example of a block group consisting of 8 blocks (notice that genuine groups would have many more blocks):

INSERT IMAGE

Cylinder or block groups, they are the core FFS performance mechanisms. Importantly, by grouping files together, FFS can avoid long disk seeks when accessing them one after the other.
FFS must be able to put files and directories into groups and track the necessary information about them in order to use these groups. Among the structures included in FFS are inode space, data block space, and structures that track whether each of these is allocated or free. Within a single cylinder group, FFS keeps:

INSERT IMAGE

Let us now dissect the single cylinder group's components. For reliability, FFS copies the super block (S) in each group. The super block is required to mount the file system; by retaining many copies, you may still mount and access the file system if one copy becomes corrupt.
FFS must track if the group's inodes and data blocks are allocated. Inodes and data blocks in each group have their own per-group inode and data bitmaps. It is straightforward to detect big chunks of free space and allocate them to a file using bitmaps, eliminating some of the fragmentation issues of the old file system's free list.

Inode and data block sections are identical to the previous very simple file system (VSFS). As usual, data blocks make up most of each cylinder group.

Policies: File and Directory Allocation

Now that the group structure is in place, FFS must select where to put files, directories, and metadata to improve performance. Keep related items together is the basic rule (and its corol- lary, keep unrelated stuff far apart). It must decide what is “related” and place it in the same block group; unrelated items must be placed in distinct block groups. FFS uses a few simple placement algorithms to do this.
The first is directory placement. This is done by placing the directory data and inode in the cylinder group with the least amount of allocated directories (to balance directories among groups) and the most free inodes (to later allocate files). Other heuristics could be applied (e.g., taking into account the number of free data blocks).
FFS handles files in two ways. First, it allocates data blocks in the same group as the inode, preventing long seeks between inode and data (as in the old file system). Second, it puts all files in the same directory in the same cylinder group. FFS will try to position the first three close together (same group) and the fourth far away (different group) (in some other group).
Allow me to illustrate this allocation. The three directories (the root directory /, /a, and /b) and four files are placed within them following FFS regulations. Assume that each group has only 10 inodes and 10 data blocks. Assume the ordinary files are two blocks apiece, while the directories are one block each. A for /a, f for /b/f, and so on) are used to represent files and directories in this diagram.

```
group inodes data
  0 /——— /———
  1 acde
  —— accddee—
  2 bf
  ——— bff——-
  3 ———- ———-
  4 ———- ———-
  5 ———- ———-
  6 ———- ———-
  7 ———- ———-
```

Notably, the FFS policy places data blocks near inodes, and files in the same directory are near one other (Group 1, and Group 2, respectively).
Let us now examine an inode allocation policy that simply spreads inodes

among groups, attempting to prevent overflowing inode tables. Thus, the final allocation might be as follows:

group inodes data
0 /——— /———
1 a——— a———
2 b——— b
———
3 c——— cc ———
4 d
——— dd
——
5 e——— ee———
6 f——— ff

As seen in the figure, while this policy keeps file (and directory) data near its inode, files within a directory are randomly scattered around the disk, destroying name-based proximity. Files /a/c, /a/d, and /a/e now have three groups instead of one.

INSERT IMAGE

The FFS policy heuristics are based on common sense (isn’t that what CS stands for?) rather than comprehensive studies of file-system traffic. Imagine compiling a bundle of files and linking them into a single executable. Because namespace-based locality exists, FFS can typically increase performance by reducing search times between related files.

Measuring File Locality

Let’s examine some file system access traces to check if there is namespace locality. There doesn’t seem to be much research on this topic.

So we’ll use the SEER traces to see how widely apart file accesses were in the directory tree. This is because the distance between two opens in the directory tree is zero when file f is opened and then re-opened (before any other files are opened) (as they are the same file). Using the same directory but not the same file, opening file f in dir (i.e., dir/f) and then opening file g in the same directory (i.e., dir/g) creates a one-file distance. In other words, the closer two files are in the directory tree, the lower the metric.

Figure 41.1 depicts the localization found in SEER traces across all workstations in the SEER cluster. On the x-axis is the difference measure, and on the y-axis is the cumulative percentage of file opens that were that difference. Specifically, the SEER traces (marked “Trace” in the graph) show that almost 7% of file visits were to previously accessed files, and approximately 40% were to the same file or one in the same directory (i.e., a difference of zero or one). That makes sense (at least for these traces).

Interestingly, almost 25% of file accesses were to files separated by two. If a user has organized a group of linked directories into levels and jumps between them, this is the result. Assuming the user has a src directory and an obj directory, both of which are sub-directories of a main proj directory, a common access pattern is proj/src/foo.c, followed by a proj/obj/foo.o. Proj is the common ancestor of these two approaches. Because FFS does not record this form of locality, extra searching occurs between such accesses.

The graph also indicates a “Random” trace’s location. The random trace was created by randomly picking files from an existing SEER trace and measuring the distance metric between them. As expected, the random traces have less namespace locality. Because every file has a common ancestor (e.g., the root), random is handy as a comparison point.

Large-File Exception

Large files are a significant exception to the usual file placement approach in FFS. A huge file would normally fill the block group it is placed in (and maybe others). This filling of a block group is undesirable because it prohibits subsequent “related” files from being stored within it, affecting file-access locality.
So, with huge files, FFS accomplishes this. A few blocks later, FFS allocates the next “large” chunk of file (e.g., those indicated to by the first indirect block) to another block group (perhaps chosen for its low utilization). So on and so forth.
Examine some diagrams to help grasp this policy. A huge file would normally arrange all of its blocks in one area of the disk. We study a 30 block file (/a) in an FFS with 10 inodes and 40 data blocks per group. Here’s FFS without the large-file exception:

```
group inodes data
   0 /a——— /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a———
   1 ———- ———- ———- ———- ———-
   2 ———- ———- ———- ———- ———-
```

As shown in the image, /a fills most of the data blocks in Group 0, but not in other groups. If new files are generated in the root directory (/), the group’s data will be squeezed.
Instead of spreading the file among groups, FFS spreads the file between groups, resulting in low utilization within each group:

```
group inodes data
   0 /a——— /aaaaa—- ———- ———- ———-
   1 ———- aaaaa—- ———- ———- ———-
   2 ———- aaaaa—- ———- ———- ———-
   3 ———- aaaaa—- ———- ———- ———-
   4 ———- aaaaa—- ———- ———- ———-
   5 ———- aaaaa—- ———- ———- ———-
   6 ———- ———- ———- ———- ———-
```

The astute reader (you) will notice that distributing file blocks throughout the disk slows performance, especially in the case of sequential file access (e.g., when a user or application reads chunks 0 through 29 in order). And you are correct, dear reader! But you can solve this by carefully picking chunk size.
The file system will spend most of its time moving data from disk and only a little amount of time looking between pieces of the block. Amortization is a typical approach used in computer systems to reduce overhead by increasing effort per overhead paid.

Assume that the average disk positioning time (seek and rotation) is 10 ms. Assume the disk moves data at 40 MB/s. To attain 50% peak disk performance, you would need to spend 10 ms sending data for every 10 ms seeking between chunks. So, how big must a chunk be to spend 10 ms in transfer? Use math, especially dimensional analysis from the chapter on disks:

FORMULA

So, if you transfer data at 40 MB/s, you need to transfer only 409.6KB every time you seek to spend half your time looking and transferring. Similarly, you may calculate the chunk size required to attain 90% peak bandwidth (3.69MB) or even 100% peak bandwidth (40.6MB!). It's easy to notice how larger these chunks get as you near peak (see Figure 41.2 for a plot of these values).

IMAGE

However, FFS did not employ this method to distribute huge files across groups. Instead, it used a basic technique based on the inode's structure. The inode's first twelve direct blocks were grouped together; the inode's following indirect blocks were grouped separately. To use this approach, every 1024 blocks of the file (4MB) were separated, save the first 48KB which were pointed to by direct pointers.
The trend in disk drives is for the transfer rate to improve rapidly, whereas the mechanical components of seeks (disk arm speed and rate of rotation) improve slowly. The consequence is that when mechanical expenses increase, you must transfer more data between searches to amortize them.

Summary

The advent of FFS marked a turning point in file system history, showing the importance of file management within an operating system and illustrating how to cope with the most vital of devices, the hard disk. Hundreds of different file systems have emerged since then, yet many still borrow from FFS (e.g., Linux ext2 and ext3 are obvious intellectual descendants). The key lesson of FFS is to treat the disk as a disk.