**Chapter 41: Locality and the fast file system**

**41.1 The Problem: Poor Performance**

The performance was terrible. The main issue is that the old UNIX file system treated the disk like it was a random-access memory that it spread all over the place.

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed.

For example, if some blocks are deleted, then a big blocks is allocated, it will cause fragmentation as it is spread across the disk.

Graphical user interface, application

Description automatically generated

To solve this, we use **disk defragmentation tool** to reorganize ondisk data to place files contiguously and make free space for one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes.

There is another problem that the original block size was too small. Thus, transferring data from the disk was inherently inefficient. Smaller blocks are good as they prevent internal fragmentation.

**41.2 FFS: Disk Awareness Is The Solution**

The idea of **Fast File System (FFS)** is to design the file system structures and allocation policies to be “disk aware” and thus improve performance ; by keeping the same interface to the file system but changing the internal implementation

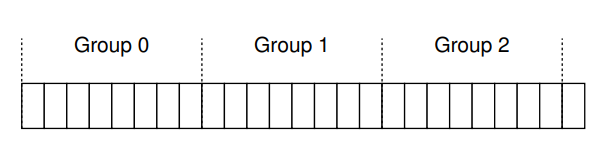
**41.3 Organizing Structure: The Cylinder Group**

The first step was to change the on-disk structures. FFS divides the disk into a number of **cylinder groups**. A single **cylinder** is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive. FFS aggregates N consecutive cylinders into a group, and thus the entire disk can thus be viewed as a collection of cylinder groups. For example,

A picture containing text, metalware, coil spring

Description automatically generated

Note that modern drives do not export enough information for the file system to truly understand whether a particular cylinder is in use. Therefore, the modern systems organize the drive into **block groups**, each of which is just a consecutive portion of the disk’s address space:



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.

To use these groups to store files and directories, FFS needs to have the ability to place files and directories into a group, and track all necessary information about them therein. This is accomplished by including all the structures in a file system such as space for inodes, data blocks, etc.

Shape, rectangle

Description automatically generated

FFS keeps a copy of the super block (S) in each group for reliability reasons. The super block is needed to mount the file system. By keeping multiple copies, if one copy becomes corrupt, you can still mount and access the file system by using a working replica.

Within each group, FFS needs 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.

The inode and data block regions are just like those in the previous very-simple file system (VSFS). Most of each cylinder group, as usual, is comprised of data blocks.

**41.4 Policies: How To Allocate Files and Directories**

The basic is that we have to keep related stuff together. Thus, the FFS has to decide what is related and place it within group and keep unrelated items away from each other. To achieve this end, FFS makes use of a few simple placement heuristics.

The first is the placement of directories. It finds the cylinder group with a low number of allocated directories (to balance directories across groups) and a high number of free inodes (to subsequently be able to allocate a bunch of files), and put the directory data and inode in that group.

For files, FFS does two things. First, it makes sure to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data. Second, it places all files that are in the same directory in the cylinder group of the directory they are in.

For example, assume that there are only 10 inodes and 10 data blocks in each group and that the three directories (the root directory /, /a, and /b) and four files (/a/c, /a/d, /a/e, /b/f) are placed within them per the FFS policies. Assume the regular files are each two blocks in size, and that the directories have just a single block of data. The result is shown as follows:

Letter

Description automatically generated with medium confidence

On the other hand, consider inode policy that simply spreads inodes across groups:

Diagram

Description automatically generated

FFS is designed based on common sense

**41.5 Measuring File Locality**

We will measure how far away file accesses were from one another in the directory tree. For example, the same file has distance of 0. If they are in the same folder, the distance is 1 (/a/b and /a/c).

Chart

Description automatically generated

The result is shown above. We can see that a large percentage of the file has shorter distance.

**41.6 The Large-File Exception**

For large block, after some number of blocks are allocated into the first block group, FFS places the next “large” chunk of the file in another block group. Then, the next chunk of the file is placed in yet another different block group, and so on.

Assuming we have a small example of a file (/a) with 30 blocks in an FFS configured with 10 inodes and 40 data blocks per group. Here is the depiction of FFS without the large-file exception:

A screenshot of a computer

Description automatically generated with medium confidence

If some other files are now created in the root directory, there is not much room for their data in the group. With the large-file exception (here set to five blocks in each chunk), FFS instead spreads the file spread across groups, and the resulting utilization within any one group is not too high:

A picture containing table

Description automatically generated

This might cost performance. Specifically, if the chunk size is large enough, the file system will spend most of its 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.

If we 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, or even 99% of peak bandwidth. As you can see, the closer you want to get to peak, the bigger these chunks get:

Diagram

Description automatically generated with low confidence

**41.7 A Few Other Things About FFS**

FFS does not work really well with small files. 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 is to use **sub-blocks**, which were 512-byte blocks that the file system could allocate to files.

If a small file (1KB) is created, it would occupy two sub-blocks and will not waste 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.

However, this is expensive and require a lot of extra work. 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 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.

Shape, circle

Description automatically generated

FFS solved this problem with a different layout. By skipping over every other block, 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.

Modern disks internally read the entire track in and buffer it in an internal disk cache (often called a **track buffer** for this very reason).

Finally, 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 the traditional fixed-size approach.