The OLD UNIX FS

Considering the UNIX (old implementation) as a standard we will now explore the pitsfalls and how to exploit the physical structure of a disk to make it faster – the so called, FFS. 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 one-level hierarchies provided by earlier systems.
The old UNIX FS was simple and it resembles the structure (see the image) right 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, the data blocks occupying most of the disk blocks

Poor Performance of the old UNIX FS

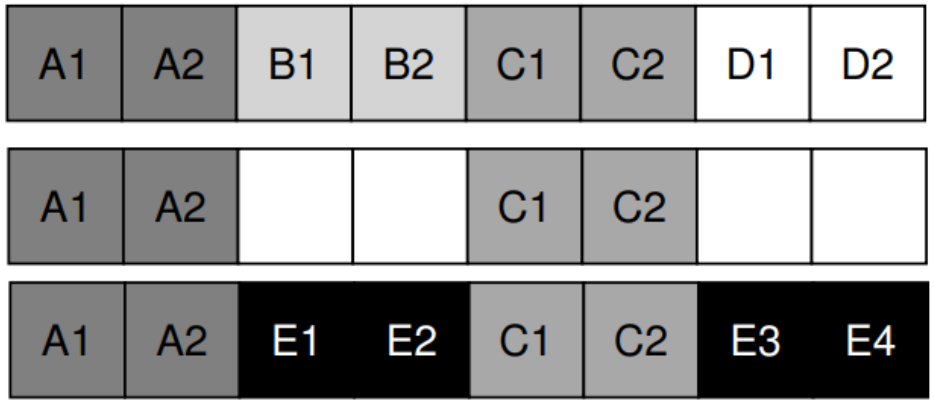
The performance of the old UNIX FS was really poor and as low as delivering only 2% of the obverall disk bandwidth.
The main issue was that the old UNIX file system treated the disk like it was a random-access memory; data was spread all over the place without regard to the fact that the medium holding the data was a disk,

- thus had real and expensive positioning costs as 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)
- the FS 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

Problems of old UNIX FS ... contd.

Consider the following data block region, which contains 4 files (A, B, C, and D), each of size 2 blocks (1st figure). If B and D are deleted we have a layout as shown in 2nd fugure. 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 (3rd figure). See what happens:

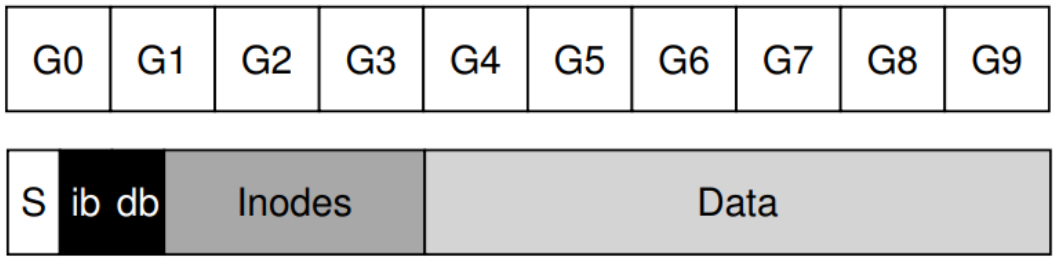
- E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4.
- This fragmentation problem happened all the time in the old UNIX file system reducing the performance



SOLUTION: DISK structure awareness

FFS divides the disk into a bunch of groups known as cylinder groups (a.k.a block groups in Linux ext2 and ext3). Let's take an example with ten cylinder groups (first figure)
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 the 2nd figure.
- A copy of the superblock S is kept in each group for reliability
- The inode bitmap (ib) and data bitmap (db) track whether each inode or data block is free, respectively
- Most of each cylinder group, as usual, is comprised of data blocks.



Allocating files and directories in FFS

IN FFS the basic action for improved performance is "keep related things together" and "un-related things far apart".
In order to comply the idea of what is realted the FFS must do somethings, For 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.

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).

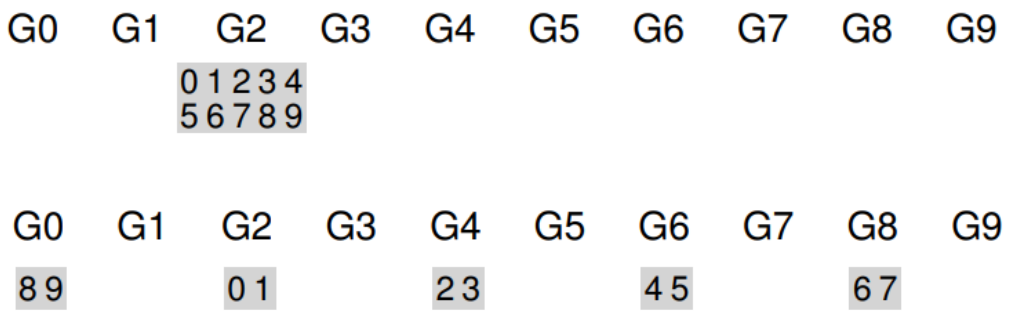
Allocating large files in FFS

FFS violates the policy of keeping related things (say a large file) together for improved performance; this exception is meaningful as we can see later.
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.

- Usually 12 (the number of direct pointers available within an inode) blocks are allocated into the first block groups.
- 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.

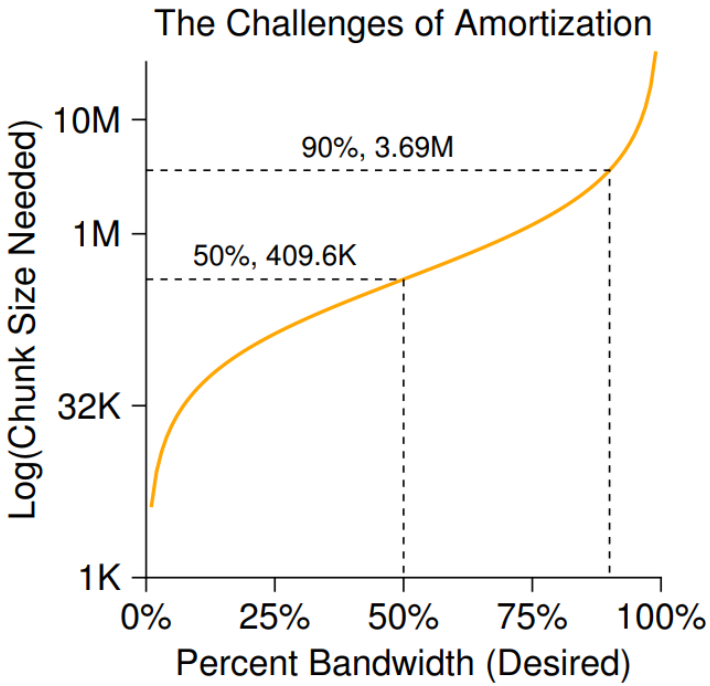
Allocating large files in FFS ...contd.

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 (see top figure). However, with the large-file exception, we might see something more like this, with the file spread across the disk in chunks (see bottom figure). In the relatively common case of sequential file access (e.g., when a user or application reads chunks 0 through 9 in order) the performance hurts. So, we need to chose our chunk size carefully.
(i) if the chunk size is large enough, we 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.



A good chunk size: Ammortization

Say, the goal is to spend 50% of the time to position the r/w head and 50% to transfer data. For a positioning time of 10 ms and a disk transfer rate of 40 MB/sec we could transfer 409.6 KB in 10 ms. In FFS for a 32-bit disk address & a block size of 4 KB – 1024 blocks of the file were placed in separate groups, the lone exception being the first 48-KB of the file as pointed to by direct pointers. The increase in storage density with time is far more than the reduction of positioning time – so the objective would be transferring more data between seeks.



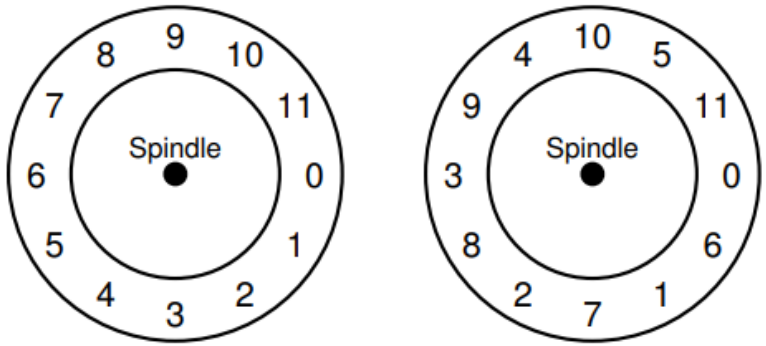
Accommodating small files in the FS

FFS has introduced the conception of sub-blocks to avoid internal fragmentation for the numerous small files (say, ¡= 2KB), in a typical FS, put in 4 KB blocks. FFS introduced 512-byte small blocks that the file system could allocate to files. Thus, for a small file (say 1 KB in size), it would occupy two sub-blocks and thus not waste an entire 4-KB block. For a growing file the file system will continue allocating 512-byte blocks to it until it acquires a full 4-KB of data. At that point, FFS will find a 4-KB block, copy the sub-blocks into it, and free the sub-blocks for future use.
While the proposal is good allocating sub-blocks incrementally leads to a lot of disk i/o – including copying sub-blocks to full block. In practice,

- Instead of allocating sub-blocks the write requests are buffered and disk block allocation is delayed till the requirement reaches to 4 KB. Thus
- avoiding intermittent disk i/o

Optimized Disk sector layout

FFS has also introduced parametric disk sectoring to improve performance. A problem arose in FFS when a file was placed on consecutive sectors of the disk (as on the left in the figure below). 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 shown in the right of the figure) 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.



Smart disks, long name and symbolic links

Even with paramteric sector numbering reading a full block might require more than one rotation and seems not to be a very good option. However, modern disks are intelligent and they do caching the sector data and returns after reading a full block. This relieves the higher level FS to take care of disk intricacies. The FFS also allows long name string (instead of traditional 8 characters). Moreover, FFS allows symbolic links – more flexible than the hard links 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