The UNIX Fast File System (FFS)

1 Introduction

Hello, and welcome to this lesson on the UNIX Fast File System, of FFS for short.

The Fast File System was proposed and developed in the mid-eighties by a group of researchers at the University of California at Berkeley, as part of their work on BSD UNIX.

We will begin by reminding ourselves briefly of the limitations of the original UNIX file system. Most of those limitations came because the file system was not aware that it was dealing with disks, which had long head movement latencies.

FFS improved on the original in three directions, of which two were performance driven, and a third direction was functional in nature:

- First, they significantly increased the block size, which improved the performance significantly.
- Second, they changed the file allocation so as to make it aware that disks take a long time to move their arms and to rotate the requested sector under the head.
- Finally, they added a number of sorely missing functionalities.

At the end of this lesson you will a good understanding of the UNIX Fast File System and of the design decisions taken to improve on the original file system.

Let's begin.

2 Recall: Problems with the Original UNIX File System

When we explored the original Unix File System, we noticed that its performance is severely limited by a number of design choices that had been made by its designers.

One big problem was its use of a very small block size. Initially it was 512 byte, and at some point increased to 1024 byte in some versions of UNIX.

Another big problem was that it kept the inodes stored separately from the data blocks. This caused long seek times when the disk wanted to read the data after having looked up the meta data for the file in the inode table.

Similarly, neither inodes nor data blocks of files in common directories were not kept together, meaning that it was expensive to search directories, for example.

Even the data blocks of a file were not stored together, which caused a lot of head movement to transfer a file.

Finally, the free list was quickly scrambled after some use which increased the overhead of finding free blocks.

In general, we notice that much of the performance limitations were cause by the original file system treating the storage device as a Random Access Device, similar to memory, rather than taking into consideration that disks have long seek latencies.

3 "Fast FS" (FFS, ca 1984)

How did the BSD people set out to improve on the original UNIX file system? . They pursued basically a three-pronged approach: . First they significantly increased the block size. This, as we will see, led to excessive fragmentation, which had to be addressed. . Second, they make the file system more disk-aware by trying to co-locate inode blocks and data blocks where possible. . Finally, they added a number of functional improvements.

3.1 FFS: Increase Block Size

Let?s talk first about the block size, which was increased from the original 512 byte or 1kB to at least 4kB, generally a power-of-two multiple of 4kB.

The reason for this increase what the overhead to read a single block: Each block access requires the head to be moved to the correct track, which expresses itself in form of seek latency. If possible, one wants to amortize the seek latency. With larger blocks one accesses more data before a new block must be accessed and another seek latency incurred. In addition, larger blocks make for larger index blocks, which fit more block numbers. Data in larger files can therefore be read after fewer accesses to index blocks.

In FFS the system administrator could define the size of the block during file system creation, and the block size would be stored as a parameter in the superblock of the file system.

Reasons for improvement:

1. Each disk operation transfers double the data. (amortizes seek times)
2. Less need to access indirect blocks (reduces number of additional seeks)

3.1.1 FFS: Block Size and Fragmentation

Increasing the block size significantly increased internal fragmentation, however. This was particularly the case for file systems with many small files, where many files can be smaller than large blocks.

As this table from the original publication on FFS shows, a file system with a bit less than 800MB of data had increasingly more waste due to internal fragmentation as one increased the block size. For 4kB blocks the authors measured a waste of 45% due to internal fragmentation.

Clearly, something had to be done about this.

The approach taken by the FFS designers was to do big blocks, but to also not forget small blocks altogether.

In fact, small blocks made it back in through the backdoor in form of so-called "fragments". Blocks are partitioned into smaller fragments, whose size can be defined at file system creation time. Fragment sizes are a power-of-two multiple of 512B.

One occasionally sees the block size over fragment size to designate these file system parameters, in this case the file system has 4kB blocks and 1kB fragments.

The combined management of blocks and fragments is a slightly tricky because we need to manage, fundamentally, two block sizes.

In this case we have 7 free blocks, which can be partitioned into 4 fragments each.

Let's assume that we need to store a file, which requires two blocks and two fragments. The whole blocks can be stored in two free blocks. For the remaining two fragments we acquire two available fragments, for which we need to start a new block. The remaining fragments can be used by other files.

When a new request comes in, we do exactly that. The first two blocks are stored in available blocks. The fragment for the rest is picked from the first block.

When we now append data to the first file, we try to use fragments in the partially filled block. As this is not sufficient, we copy the existing fragments to a new block and and store the new data in fragments in the new block.

Naturally, this management of fragments and blocks requires an additional data structure to maintain the status of allocated and freed fragments and blocks.

3.2 FFS Organization: Making the File System Disk-Aware

So far for increasing the block size. We also said that FFS would have a disk aware file allocation scheme.

This is achieved through what are called "cylinder groups" and more generally a system-aware file system parameterization.

3.2.1 Cylinder Groups

Let's focus first on cylinder groups.

Cylinder groups are groups of adjacent disk cylinders. From our earlier discussion of disks we learned that a disk cylinder in a multi-surface disk is the set of tracks that can be accessed without moving the disk arms. As a result, all the blocks stored within a cylinder can be accessed without seek latency. Blocks that are stored in the same cylinder group can be accessed with very little seek latency, given that they are either stored in the same cylinder or in neighboring cylinders.

Cylinder groups now allow us to allocate files and directories in a way that significantly reduces arm movement and therefore seek latencies.

The principle is to try to "keep related stuff together in the same cylinder group".

By this we mean that inodes and data blocks are stored in the same cylinder group, as are data blocks of the same files as are blocks of files that belong to the same directory, including the directory file itself.

What de-facto happens is that we have many mini-file-systems, one per cylinder group. Each group maintains a copy of something similar to a superblock, an inode bitmap, data bitmap, inode list, and the data block.

As we will see, the "keep related stuff together" principle is very effective in reducing seek latencies and therefore increasing disk performance.

3.3 Minimizing Rotational Latency

The second goal, after minimizing seek latency, is to minimize rotational latency as well.

For this, newly allocated blocks are positioned so that they minimize the rotational latency after the disk accesses the previous block.

We say that the new block is "rotationally well positioned".

This figure shows an example of two allocation sequences. The one on the left is the natural ordering where, the blocks will be allocated in order of the rotation of the disk. This sequence would not work for a system with very quickly rotating disks and a CPU with either a slow bus or slow interrupt service handling. By the time the CPU is ready to access the next block, it may have spun already under the head, and the system has to wait for nearly an entire rotation until in can access the next blocks.

The allocation sequence on the right is interleaved. The CPU has enough time to finish handling the first request until the next block comes under the head. This results in a much better system performance.

In order to decide on an optimal positioning of blocks, one need to know a number of system parameters. For example it is important to know the number of blocks per track, of course, but one also needs to know how fast the disk spins. These parameters can often be queried from from the disk.

One also needs to know, at least approximately, some CPU or bus parameters. It is important, for example, to know how long it takes for the system to service an interrupt to be ready for the next disk transfer.

Once all this data is available, FSS can decide on an optimal block placement.

4 FFS: Performance

The designers of FFS compared its performance to that of the original UNIX file system with 1kB blocks. These measurements were taken in the mid eighties, on a machine, the VAX 750, which has about 1/20th of the computing power of an 20MHz Arduino. While we can safely ignore the absolute values, the relative performance numbers are interesting.

As expected, both read performance, as well as write performance increased dramatically with the new file system, in particular with 8kB block sizes. Reads were about 8 times faster and writes were a bit over four times faster.

When we include the results with an improved disk controller, the results look even more impressive.

If we look at the CPU utilization for the "write" experiments, we see that it is extremely high. This is an indication that the file operations are not limited by disk operations anymore. The bottleneck has moved from the disk to the CPU.

5 FFS: Functional Enhancements

In addition to performance improvements, FFS introduced a number of functional improvements.

One was long file names. We remember that in the original UNIX file system the directory entries were of fixed size, and file names were limited to 14 characters.

Also, there was no provision for locking files. If one wanted to serialize the access to a file across processes, a separate "lock" file had created, and processes would try to create this file. If they succeeded, they "owned" the lock, so-to-say. If they failed because the lock file existed already, they kept trying.

This did not work well, for three reasons:

- First, if a process crashed while owning a lock file, the lock file would hang around and had to be deleted by an administrator.
- Second, processes would keep polling for the lock, using up system resources.
- Finally, system administrators would never fail to create files, so this mechanism would not work for them, and a separate mechanism would have to be created for them anyways. FFS introduced a mechanism that allowed to lock the file directly.

Symbolic links (similar to shortcuts in Windows) were borrowed from MULTICS. Hard links are limited, as they for example don't work across physical file systems.

Previously, renaming a file required 3 system calls, which, when a process was interrupted somewhere in-between them, would lead to all all kinds of inconsistencies. So a **"Rename"** system call was added.

Finally, as UNIX developed into a multi-user operating system, quotas became an issue. **Quotas** in FFS allow the administrator to limit number of inodes and number of blocks that a user can allocate.

6 The UNIX Fast File System: Summary

With this we have come to the conclusion of our exploration of the UNIX Fast File System.

We set off by reminding ourselves of the performance limitations of the original UNIX file system. Many of these came from treating the disk as a random access device.

FFS improved on the original file system by increasing the block size, making the file system disk aware and generally system aware to reduce seek latency and rotational latency, and by finally adding sorely needed functional improvements, such as file locking an others.

We truly hope that you enjoyed our exploration of the UNIX Fast File System.

As we will learn in the following lessons, the file system landscape has significantly changed since the mid eighties. One major change is the prevalence of large amounts of cheap and very fast main memory, which allows the system to cache large parts of the file system. Given this environment, the UNIX fast file system has become largely irrelevant as well, as it has been replaced by various forms of log-based file systems, which effectively leverage memory caches.

FFS remains important, however, because it illustrates how simple but well thought-through improvements to an established system can yield some pretty spectacular performance improvements.