Unix File System Implementation

Hello, and welcome to this lesson on the Unix File System Implementation.

In this lesson, we will focus on the implementation of the original Unix file system, as designed in the mid-seventies.

We will come to see that this was quite an engineering feat for that time.

We will first look at how the file system is laid out on the storage device.

We will see that there will be four parts: (1) the boot block, (2) the superblock, (3) the blocks that contain the inode list, which is all about file meta data, and (4) the actual data blocks.

The role of the boot block is clear, so we will first focus on the superblock and how it maintains file-system information and the free space.

We will then look at the role of inodes to maintain file metadata and explore how inodes are managed.

We will then briefly discuss how naming is realized through directories.

Finally, we will do a brief analysis of pros and cons of this file system. We will see that for larger systems the cons prevail, which explains why a need emerged after some time to improve on the basic UNIX file system.

At the end of this lesson you will have a detailed understanding of the original Unix file system, including file allocation, free space management, management of file meta data, and a scalable naming infrastructure. You will also understand why efforts were underway quite soon after its first deployment to improve its performance.

Let's begin

File-System Layout of UNIX FS

If we look at a storage device as a sequence of blocks, starting from the top,

then the Unix files system layout (and similarly layouts of many other files systems) consists of 4 parts,

namely boot block, superblock, inode list or some other structure for file metadata, and data blocks where the actual data is stored.

[slide 3]

If we look at a storage device as a sequence of blocks, starting from the top, then the Unix files system layout (and similarly layouts of many other files systems) consists of 4 parts, namely (1) boot block, (2)

Superblock, (3) inode list or some other structure for file metadata, and (4) data blocks where the actual data is stored.

[slide 4]

The superblock contains the information about the file system. We also call this information the “file system meta data”.

This includes file system size, free space information, information about location of file metadata records, called inodes in the case of Unix, and so on.

The content of the superblock can be represented as a C struct as follows:

At the beginning, we have the size of the inode list, in blocks that are reserved to store the list of inodes.

Then follows the size of the file system in blocks.

Then comes an array of free blocks and a counter of free blocks. We notice that the array is limited to 100 blocks. We see later how the system uses linked index blocks to manage more than 100 free blocks.

Similarly, we have an array and a counter for the inodes, which store, as we will see soon, the metadata for the files. (5)

Then there are a few locks that are used while the file system is in use, and an indicator if the file system has been modified. If the value of “fmod” indicates that file system has been modified, the system knows that it has to write the superblock to disk.

Finally, the time indicates when the file system has been modified last.

Free Space Management

Let’s look how the system manages free blocks using the counter “nfree” and the array “free”. The array has space for 99 free block numbers, stored in free[1] to free[99]. The entry in free[0] is reserved to store the block number of a block that contains additional free blocks. Here both nfree and free[0] are zero, indicating that there are no free blocks.

Let’s assume that a block is freed, and so nfree is incremented to 1, and the number of the newly freed block is stored in free [nfree], in free[1] in other words.

When another block is freed, nfree is incremented to 2, and free[2] points to the newly freed block.

The same happens when a third block is freed, and so on (5) until we have 99 free blocks.

When now the 100^{th} block is freed, we have no space in the array but we have the newly freed block that we now can use to store parts of the free list.

We copy the contents of the array in the superblock over to the newly freed block, in this case block number 9156.

Update the entry free [0] to point to the new block, (10) clear all the other entries in free, and set the counter back to zero.

If now another block is freed, we start again.

Whenever we free a block, this process just goes backwards. When we need to allocate a block and the counter is zero, we copy the content of the block pointed to by free [0] into the array free, set the counter to 99, and continue. When both the counter and the value of free [0] are zero, we are out of blocks, and an error is signaled.

[slide 7]

The inode management is a bit different than that of free blocks.

We do have a counter and an array of inode numbers. By an inode number we mean the position of the inode in the inode list.

This array is used more as a primarily as a cache, as we will see.

Say the counter is zero, indicating that we don't know of free inodes.

Each inode has a flag, however, to indicate if the inode is free.

We can therefore query these flags to see if any free inodes exist.

If there are any, we can update the values in the inodes array to point to the free inodes. In this case we have at least 100 free inodes, which we reflect in the updated counter.

If we now need an inode, we can decrement the counter ninode, use inodes [ninode] as the new inode, and delete the entry in the array.

When we free inodes we simply set the flag and optionally update the array. This is not needed because we simply refill it again when we the counter reaches zero again.

Unix inodes

What kind of meta data is stored in the inode?

First, there is a number of flag bits, which indicate whether the inode is used, what permissions users have for this file, whether the file is a so-called large file, what type the file is, or whether it is a special file, and so one.

Next come the links to the file, which are basically the number of names that the file has. We will discuss this in more detail in a minute.

Then come the user and the group identified of the owner of the file, and the size of the file, spread over three bytes.

Then comes the index array for the allocation table, which we will explore in a minute. (5)

And finally, the time of last access and of last modification for this file.

The multilevel indexed allocation table is rooted in the array "address", but a bit differently than how we described it in the lesson on file allocation.

The file can be treated as a so-called "small file", in which case the array points to up to 8 data blocks. A zero valued pointer means that there is no data block.

This allows to represent file of size up to 8 blocks, which makes for 4kB.

If the file is marked as “large file” in the flags (10), then the block number is interpreted differently. The block number is divided by 256 and the result is now used to index into the address array.

The entries in the address array now don’t point to data blocks but to index blocks, whose entries point to data blocks.

The last entry in the address array is a doubly indirect pointer, which points to an index block, whose entries point to index blocks, whose entries point to data blocks.

[slide 9]

We said that zero-valued pointers don’t point to any blocks. This can be used to very efficiently store sparse files, that is, files that have mostly zeros.

In this case we only require blocks that are pointed to. If we now assume that all zero pointers point to blocks with zero values, then we need only 6 data blocks and 4 index blocks to store a huge file, whose content is mostly zeros.

One thing is puzzling about inodes.

[slide 10]

We have all kind of meta-information, but the file name is missing. Where is the file name stored?

File names are stored totally separately from the file in so-called directory files.

Each directory file contains a sequence of entries that are basically pairs of an inode number and the file name associated with the file associated with that inode number.

In early implementations of Unix such a directory entry was 16 bytes in length, with 2 bytes for the inode number and 14 characters for the name.

A file is created by creating the inode and associating it with a newly created entry in a directory file.

Files can have multiple Names (Hard Links)

Files can have multiple names.

In this example, we have a directory entry in directory /dirA that associates inode number 12345 with name “name1” in directory “/dirA”.

The link counter is one because this file is pointed to by one link, which means that it has one name.

We can now create a new name for the file by linking the new name to the existing file.

In UNIX there is a system call to do this, called “link”.

If we link the name “/dirB/name2” to the file “/dirA/name1”, we create an entry in directory “/dirB”, which associates the inode number 12345 with name “name2”.

As a result, the file now has two names, which is reflected in the link counter in the inode.

Deleting Files

Names are deleted by unlinking.

For example, we unlink the name “/dirA/name1” from the file, which removes the entry in directory “/dirA” and decrements the link counter in the inode.

Similarly, when we unlink the name “/dirB/name2” from the file, we remove the entry for name2 from directory “/dirB” and decrement the link counter in the inode.

The counter now is zero, which means that it cannot be accessed by anybody anymore, and we can delete the blocks of the file and the inode as well.

File-System Layout of Unix FS

We can of course also create directory files, which assigns inodes to directory files as well. These directory files are given names by adding entries to their parent directories. This all results, in the Directed Acyclic Graph of directories that we know from systems like Unix. The question now is: How do we know where the root of the directory graph is? Where is the graph grounded?

This is handled by defining the first inode in the inode list to represent the root directory. All other files and directories now dangle of the root directory, which means that all the inodes can be reached from this first inode.

Pros/Cons of Original Unix File System

Let’s briefly evaluate the original Unix file system.

Some of the benefits of this system we have discussed in our discussion of file allocation.

Data in small files can be easily accessed from the inode.

At worst we need to disk operations, one to read the inode, and the other to read the data block.

Larger files can be accessed efficiently as well, as the number of blocks to read is at most 4, including the inode.

Also, as we discussed earlier, there is no external fragmentation.

As time passed, and disks and file systems became larger, a number of problems came up. (5)

First, the 512-byte block size soon became a bottleneck. The overhead to read just 512 bytes was significant. Also, the number of entries that could be held in an index block was small, which limited the file size and increased the number of blocks that needed to be read to get access to a particular location in the file.

Another problem was that inodes were kept separately from the data blocks. The head had to move far to move from the inode to the data block, which caused long seek times.

Also, inodes of files in the same directory were not kept together, which caused a lot of seek times when traversing directories.

Data blocks were not stored together either. This increased the cost of sequential access to files because the head had to move all over the disk as it read subsequent blocks.

Finally, the free list quickly got scrambled, thus increasing the overhead of finding free blocks.

[slide 15]

Added together, these effects were so bad that some measurements indicated that the file system under heavy I/O was using as little as 2% of the available bandwidth.

It is important to point out that the I/O bus at the time was a ST-506, which maxed out at 5Mbit/s. The achieved IO bandwidth in these experiments was as low as 13kB/sec.

No wonder this a lot of activity get under way to develop a next-generation file system for Unix systems. We will talk about some of these efforts in the next lessons.

Unix File System Implementation Conclusion

We have come to the conclusion of this lesson on the original UNIX File system, where have explored quite in detail its implementation.

We have started with the file system layout on the storage device, and then looked at how the system managed free blocks.

We learned about inodes and which role they play in the file system, and how inodes are stored and managed on disk.

Finally, we have evaluated the pros and cons. It became clear that the ability to scatter blocks across the disk to avoid external fragmentation now became part of an overall performance problem, where excessive disk head movement slowed down the file system performance significantly.

In the next lessons, we will look at improvement to the basic Unix file system that addressed these performance issues.

We very much hope that you enjoyed this lesson on the original Unix File system.

While todays file systems are implemented differently, the original Unix File system is a mile stone and has been guiding many more recent developments.

In the next lessons we will learn how this basic system has been successively improved to address performance and reliability limitations.

Thank you for watching.