Hard Disks and Filesystems

I-Nodes and Superblocks

Users with typical, run-of-the-mill computers will tend to use a hard disk. But very few people know how exactly files are stored on a disk or what options are available. We will be attempting to shed some light on the subject in this article. BY MARTIN SCHULZE

Besides the actual data, a filesystem stores a variety of information about itself. Status information, access times, and structures are all vital parameters that operating systems and programs will read from the filesystem for use at runtime.

The Superblock

The superblock stores a variety of meta-information. As this block is vital to filesystem use, it is automatically copied to various parts of the filesystem.

If the superblock is damaged or overwritten, it is normally possible to restore the filesystem from a copy. The superblock is normally located at position 0, with a copy at 8193 (or 32768, depending on creation parameters).

You can use *e2fsck -b 8193* to check a filesystem for errors, even if the first copy of the superblock has been damaged. If the system is mounted with write permissions, the program will restore the first superblock after performing the check.

The superblock also specifies the filesystem type and records the number of times the filesystem has been mounted without being checked. The following section provides an overviw of the most important superblock components (and Figure 1 illustrates them).

I-Node Count and *Blocks Count* contain the total number of **I-Nodes** and **blocks** on this filesystem. Similarly, *Free Blocks* and *Free I-Nodes* show the number of free blocks.

When a filesystem is created, typically a certain percentage of the total block count is reserved for use by the administrator, *root*. This figure is stored in the *Reserved-Blocks Count* field.

The next few fields in the illustration apply to the boot process. *Mount Count* indicates how often this filesystem has been mounted since last being checked. *Max Mount Count* stipulates a threshold. When the threshold value is reached, the operating system will perform a filesystem check, even though the filesystem is "clean" (see *State* field), having been unmounted cleanly.

The *lastcheck* and *checkinterval* work in a similar way. The first field stores the date of the last filesystem check, and the second shows when the next check is scheduled. Both of these values are needed for safety reasons, to ensure that the filesystem is given a thorough service after a long period of use, thus avoiding possible errors (as caused by **bit flips**) – this is something like having your automobile serviced.

The final important field is labeled *First I-Node* and contains the I-Node with the root directory (*/*) for this filesystem – in other words the I-Node representing the access point to this filesystem from the Linux user's perspective.

This I-Node points at a datablock that stores links between names and the remaining I-Nodes in the directory. Entries often point to I-Nodes that represent directories themselves, and this way all the files on the filesystem can be located.

Journaling Filesystems

The Ext2 (Second Extended) Filesystem works really well, and was specially created for Linux, however, hard disks have grown considerably over the years, and today many users have large partitions on Linux. One problem has become particularly evident – if a computer is powered off without unmounting the filesystem cleanly, the filesystem check will take quite a while when the system is rebooted.

That is fine for a home computer, used for storing song tracks or movies, but it is definitely not so in the case of a production server. If the filesystem creates a so-called journal, only the journal needs to be parsed in case of failure to restore the filesystem to its current state.

Many people believe that – as a journal is created – regular filesystem check is unnecessary for journaling filesystems, and of course, recovering from a crash is a lot easier if you use Ext3, JFS, XFS or ReiserFS. However, this does not provide any protection from flipped bits or inconsistencies arising due to other malfunctions.

This is one of the reasons why the Third Extended Filesystem (Ext3), an enhancement on Ext2, continues to perform filesystem checks. You specify how often a check will be performed when creating the filesystem and this data is stored in the superblock, as described previously. However, Linux will not check the filesystem on booting, but only when *Max Mount Count* has been exceeded, or *lastcheck* and *checkinterval* indicate that another check is due.

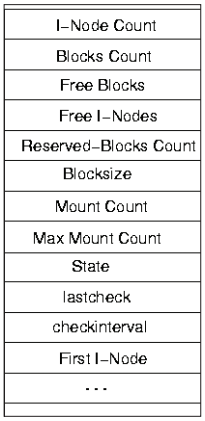

Figure 1: Generic structure of a superblock

Hardlinks and Symlinks

Figure 2 shows you the general filesystem structure. When Linux saves a file, the kernel filesystem driver reserves an I-Node and multiple data blocks. The I-Node points to the data blocks that will store the actual data. Additionally, the filename is stored in the directory along with the I-Node.

UNIX is well-known for a special filesystem feature, links. You can use a link to add a file to multiple directories allowing you to use multiple names to access that file. This makes sense in more ways than one.

One useful application for links is the case where a program assumes multiple roles and these roles are similar apart from a few operations. Instead of writing two or more programs, the program can read its own name when called, and react accordingly. This is what programs such as *gzip* and *gunzip*, or the mtools (*mdir* and *mcopy*) do.

Debian's alternative mechanism follows a similar principle. The program name (such as */usr/bin/vi*) is a link to a file in */etc/alternatives*, where a link to the program itself is stored. If you have difficulty remembering long pathnames, you can create a link in a directory that will point at the required file and save you a lot of typing.

Using links has a lot of advantages in comparison to creating a copy of a file. For one thing, you do not need to copy the file, and if it happens to be a CD image that can save you a lot of disk capacity. For another thing, you do not need to synchronize the copy with the original, as any modifications will be immediately visible to the link. In fact, you will be using the original file.

But there are two distinct link types, and each type is stored in a different way on the filesystem. The distinct types are known as hardlinks and symlinks. In the case of a hardlink multiple directory entries point at the same I-Node, whereas a symlink stores the name of the original file.

Hardlinks

A hardlink the simplest and most efficient way of referencing a file by multiple names. This method involves creating a new directory entry, just like for a new file. However, the directory entry does not call for a new I-Node or data blocks, instead using the I-Node for the original file, and allowing two directory entries to point at the original I-Node.

Of course, the system needs to record this fact somehow, as the I-Node and any data blocks belonging to it cannot be deleted until the last directory entry that points at the I-Node has also been deleted. The I-Node uses the *links_count* field for this purpose – the field is incrementing whenever a hardlink is created. Creating this link increases the value to 2 and not to 1, as two directory entries now point at the I-Node.

From the filesystem's perspective both directory entries are equivalent. When you delete an entry, the filesystem will remove the entry from the directory and reduce the I-Node counter by one. The I-Node is deleted and the memory it uses released when the entry counter drops to zero.

Figure 3 shows how the filesystem responds to the following link command.

```
ln foo bar
```

foo is the original file, and *bar* is the additional, new name for the file. As you can see, a hardlink requires only an additional directory entry, that is, the memory overhead is fairly low.

I-Nodes are only unique within the same filesystem; thus hardlinks cannot

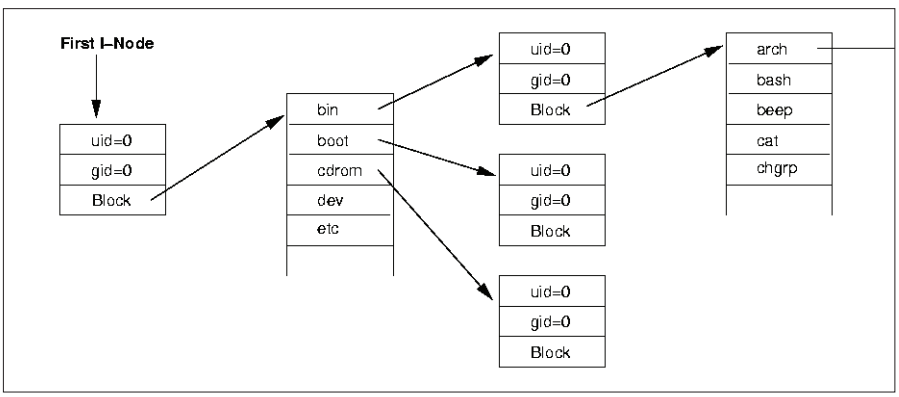

Figure 2: Internal structure of the UNIX filesystem

GLOSSARY

I-Node: *On Unix oriented filesystems every file (and each directory, being a special type of file) is mapped to an inode. Inodes map directory entries to data blocks on a partition. Inodes themselves are special data blocks that store metadata for the files they point to. You can envisage them as a kind of data structure that stores additional information for a file, such as its length, access permissions, ownership, access times, and pointers to the blocks in use. Inodes are normally used by the internal filesystem only.*

Block: *A block is the smallest unit of space that can be assigned on a filesystem. By having small block sizes such as 1 KByte, we are limited to the size of the hard disk that is accessable, but less space is wasted with small files.*

Bit flip: *This refers to the inversion of a magnetic state on a hard disk, and can be caused by magnetic fields in the vicinity of the computer.*

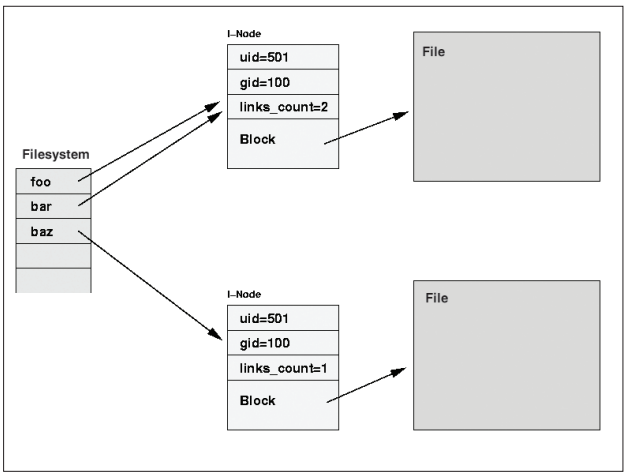

Figure 3: Hardlinks and the filesystem

be used across multiple filesystems, in contrast to symbolic links (symlinks), which can link across filesystems.

The value of *links_count* is typically output by *ls -l* and displayed in the second field (the first field shows the permissions).

Symbolic Links

Symbolic links, or symlinks, are the second way of using multiple names to access the same file, without having to copy the original file. The symlink requests an I-Node, just like a normal file, and creates an appropriate directory entry.

However, the I-Node used by the symlink points to the name of the original file. If the name is short enough, it can even be stored in the I-Node itself – the I-Node can store up to 29 characters. This is shown in Figure 4 where you can see what the following command does.

```
ln -s foo bar
```

In the case of longer pathnames, and particularly if the name contains the absolute path and not only the filename of the original file, the I-Node will be unable to store the name. Instead, a datablock will be assigned to the symlink and stored in the path to the original file. Figure 5 shows what the following command does:

```
ln -s /org/infodrom.org/home
/joey/foo bar
```

Incidentally, you can use relative pathnames, such as *../joey/foo*, however, if you do so, be aware of the following: If the link is moved to another position in the directory, the path will no longer apply, leaving the link pointing at a black hole.

We have not mentioned the *links_count* field in the context of symlinks so far, as the symlink is handled like a normal file, leaving a value of 1 in the field. If the original file is deleted, the directory entry for the symlink also disappears, but the symbolic link itself remains, as deleting the file, technically, does not affect the link. However, the link will be pointing at a an empty space now; this is what is referred to as a "dangling symlink".

Filesystem Check

When a Linux system is booted, the root filesystem is the first mounted read only, to allow programs to execute and configuration files to be parsed. One of the next steps the boot process goes through is to check all the local UNIX-type filesystems (Minix, Ext2, Ext3, XiaFS, ReiserFS, XFS, JFS).

If one of these filesystems is not "clean", that is, if it was not gracefully unmounted from the system, a check will be performed, as inconsistencies must be assumed. Recent modifications may not have been incorporated. With a journaling filesystem, Linux will first read the journal and check whether all of these actions have been written out to disk. The check should restore the filesystem to a consistent state.

The filesystem check for Ext2 comprises five steps. This allows Linux to ensure that the information stored in the data structures will be consistent. The general consistency of the filesystem is checked first; this means checking the entire system in a process that can be extremely time-consuming on larger filesystems.

During the process, the *fsck* program ensures that the value in the *links_count* of the I-Nodes corresponds to the number of links in the directories. *fsck* updates the field if it discovers a mismatch. This also applies to file sizes.

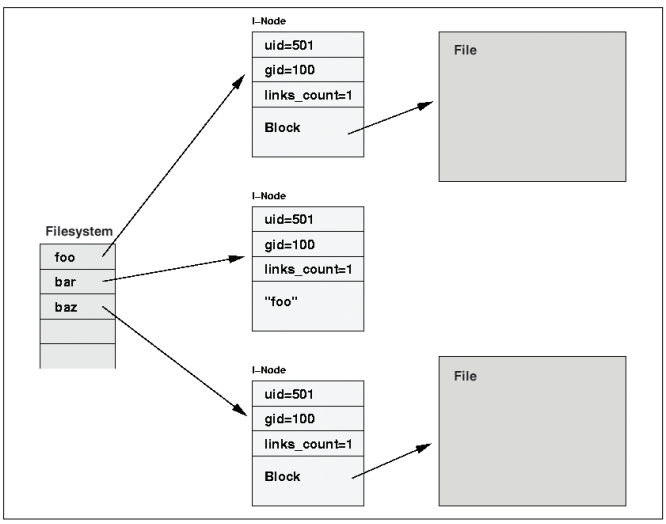

Figure 4: A "quick" symlink on the filesystem

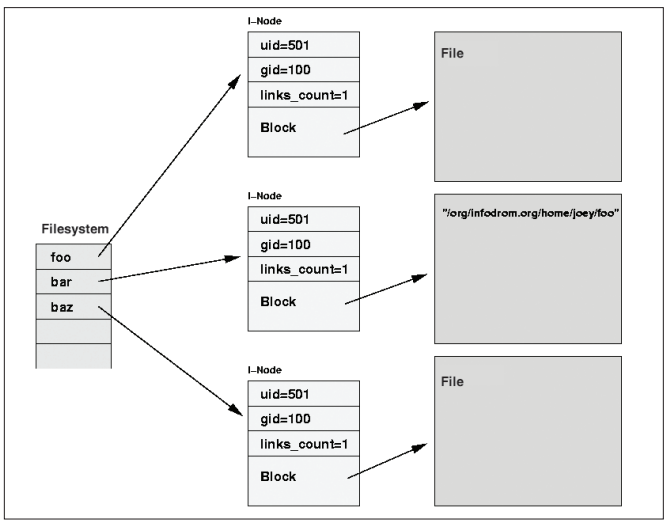

Figure 5: A "slow" symlink on the filesystem

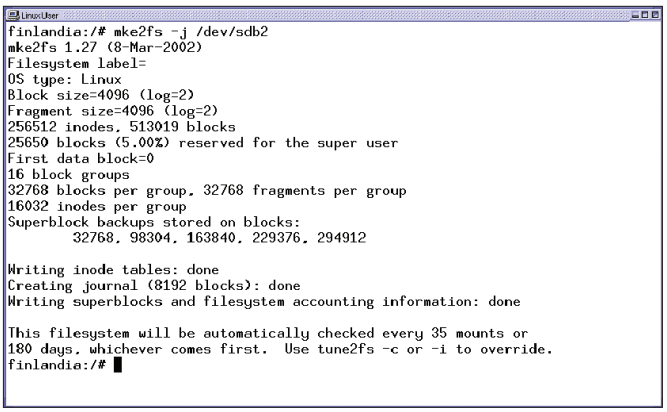

Figure 6: Creating an Ext3 journaling filesystem

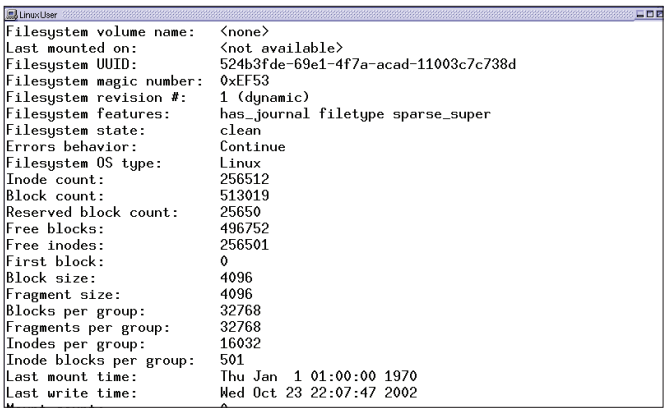

Figure 7: dumpe2fs output for an Ext3 partition (abbreviated)

If the check discovers I-Nodes without matching directory entries, the nodes are placed in a special directory called *lost+found*, allowing the administrator to decide whether to delete the file or move it to an appropriate location.

Creating a Filesystem

Linux uses the *mkfs* programs to create filesystems. To give a uniform naming scheme, the name of the program is typically the *mkfs.* prefix and the filesystem type to be created, for example *mkfs.ext3* would create an Ext3 filesystem.

Linux machines tend to use the "Third Extended Filesystem" (Ext3) or the Reiser Filesystem (ReiserFS). Various parameters are available for specifying how the program will create the filesytem, or more specifically the number of I-Nodes, and the maximum number of files and directories the filesystem can store.

The *-b* parameter defines the block size for the filesystem; this defaults to 1024 bytes (although newer systems may use a different default). Thus a logical data block will always use two physical hard disk blocks. The *-b* parameter allows you to change this setting, although values are restricted to multiples of 1024, that is, you can choose from 1024, 2048, and 4096.

In case of filesystems with a large number of large files, it might be advisable to opt for a larger block size, as this can improve access speeds. This is expensive, however, as files can only use whole blocks. Thus, in our example, a 5000 byte file would waste 3192 bytes in the second logical block.

Another useful option specifies the area reserved for *root*. Normally, five percent of the filesystem is reserved for the superuser. This ensures that the system will continue to run, although your hard disk capacity is almost exhausted, as programs executed by *root* can still write to the filesystem.

You can change this setting using the *-m* parameter. You might want to do without this safety margin for filesystems that will store user data, rather than system data, such as */home* or */var/spool/news*.

The *-i* parameter specifies the number of I-Nodes to create relative to the data payload. If the filesystem will be mainly used for storing large files, it does not make much sense to assign an I-Node for 4 KB of data – 10 KB or even 100 KB per I-Node would make more sense.

On the other hand, it does not make much sense to assign an I-Node for 4 KB of data if your filesyste will be storing a whole bunch of tiny file (like newsgroup postings). You might like to assign an I-Node for every 2048 bytes of data in this case. The number of I-Nodes cannot be changed on the fly.

The *-j* parameter is used to create a journaling filesystem, that is Ext3 instead of Ext2. As the Ext2 and Ext3 filesystems are compatible, you can convert an Ext2 filesystem to a journaling Ext3 filesytem later. The *tune2fs* command is used for this purpose.

Investigating the Filesystem

Normally, users are unable to take a closer look at the way the filesystem works, but Linux allows you to investigate the current filesystem configuration, even on a production system. This functionality is provided by the *dumpe2fs* program, which mainly displays the information stored in the superblock. Assuming that the superblock is undamaged, the location of the superblock copy is also indicated.

As you can see from this sample *dumpe2fs* output for our current filesystem, the copy of the superblock is not located at block 8193, but at block 32768. This is caused by a filesystem preference.

Recovery

Knowing where to find a copy of the superblock, which *dumpe2fs* refers to as the *Backup Superblock*, can be extremely important. If the original superblock is damaged, you will be unable to check the filesystem, and thus unable to mount it. However, your data is not lost, as you still have a copy of the superblock.

You can restore the filesystem by explicitly pointing to the location of the backup superblock. The syntax for this task is as follows:

```
e2fsck -b 8193 /dev/hda3
```

or (depending on the location of your superblock copy)

```
e2fsck -b 32768 /dev/hda3
```

If this works out, you should be able to continue using the filesystem. ■

THE AUTHOR

Martin Schulze enjoys developing and enhancing free software; failing that, he will settle for promoting free software by helping to organize the LinuxTag, for example, presenting keynotes and holding workshops. You can contact Martin at joey@infodrom.org.