Linux inodes | Baeldung on Linux

12–16 minutes

---

1. Introduction

We often think of files as objects with content. However, what we mostly deal with daily are only links. **Underneath, the objects for files are actually inodes**.

In this tutorial, we deal with Linux inodes – what they are, as well as why and when we use them. We start with the concept of storage. After that, we discuss how filesystems apply to storage devices and check out their rough inner workings. Next, inodes take the spotlight for a comprehensive discussion. Finally, we explore some filesystem objects based on inodes and note some special considerations.

We tested the code in this tutorial on Debian 11 (Bullseye) with GNU Bash 5.1.4. It is POSIX-compliant and should work in any such environment.

2. Storage

Most machines work with at least main and secondary memory. We most often call main memory random access memory (RAM), while secondary memory or *storage* consists of hard disks, solid-state drives, and similar devices.

Regardless of their kind, storage devices usually have a controller. It is responsible for exposing an interface of the respective memory module to the machine. Controllers present the device as same-sized blocks.

We should note that a *sector* or minimal block of secondary memory is most commonly 512 bytes big. This will be important later on.

To make better use of storage devices, an operating system (OS)

has drivers to talk to the controller. However, a driver still presents a very raw structure of memory. That structure, in turn, is further refined for use via another abstraction.

3. Filesystem

While drivers are the way operating systems see and control storage, filesystems are the way they order it. To do that, **a filesystem commonly uses an old mechanism we can find in books – an index**.

3.1. Metadata Organization

Consider how sections of a book are listed with important attributes such as page numbers and titles. All of this information is within the index. On the other hand, the index itself is neatly packed at one end and takes just a fraction of the book, yet it allows for fast and easy browsing.

Even without all of the pages in hand, using the index, we can get a sense of:

- how long the book is
- which sections are short, and which – long
- how sections are ordered
- which sections are subsections of which other sections
- basic characteristics of the sections (such as the title)

Now imagine some of the book's sections are empty. The resulting structure is usually the most basic idea of a filesystem.

In particular, the index represents the *metadata* associated with a filesystem. **The sections, free or allocated to files, are also called *blocks* and represent physical parts of memory**. The filesystem index organizes those for fast and easy browsing, just like pages in a book.

3.2. Examples

Indeed, most filesystems have some kind of supporting organization, which points to the blocks:

- *FAT* (File Allocation Table) has a File Allocation Table (hence the

name *FAT*)

- *exFAT* (Extended File Allocation Table) basically extends File Allocation Tables
- *NTFS* (New Technology File System) uses a much more advanced Master File Table

  Linux supports many filesystems, but only some of them are native. One notable example is the *ext* (Extended Filesystem) family systems: ext2, ext3, and ext4. Another good example is the *XFS* (X filesystem). From here on out, we'll only be discussing how native Linux filesystems work because of their widespread use and a concept they share – the key to their metadata.

  Let's explore this concept.

  The term *inode* comes from *index node*. Since we already talked about filesystem organizations being like indices in books, we can see how inodes fit nicely into this idea.

  In particular, we can equate an inode to the row number of an index entry in a book. **This way, inodes allow us to index the main index**. Operating systems assign each inode a unique integer, so they can be used as keys to the inner filesystem structures we already discussed.

  To get most of the information an inode points to for a given file, we can use the *stat* command, which internally executes the stat *system call*:

```
$ stat file.ext
File: file.ext
Size: 166 Blocks: 8 IO Block: 4096 regular file
Device: 810h/2064d Inode: 666 Links: 1
Access: (0777/-rwxrwxrwx) Uid: ( 1000/ x) Gid: (
1000/ x)
Access: 2021-11-11 10:00:00.101020201 +0200
Modify: 2021-11-11 10:00:00.101020201 +0200
Change: 2021-11-11 10:00:00.101020201 +0200
Birth: 2021-11-11 10:00:00.101020201 +0200
```

  Let's break this information down into its constituents top to bottom, left to right. The output starts with the file's name and size. The latter is in bytes (166) and blocks (8) of a given size

(8*512 = 4096 bytes). Like we already mentioned, the physical block is most often 512 bytes, but we can specify IO Block sizes when formatting a device or partition with a given filesystem. The link isn't direct, but it does exist. At the end of the second row, we see that we used *stat* on a *regular file*.

Next, we have the device identification in hex (810) and decimal (2064). Of course, we see the inode number (666). Last on the fourth row is the number of links to this inode. We'll discuss links in the next section.

There is a whole row dedicated to the ownership and access rights. Finally, we see four rows dedicated to the access, modification, change, and creation (birth) date. Note that to change a file means to modify its metadata.

Indeed, many filesystem objects have their own unique inode. In fact, regardless of their specific nature, all of them are files.

5. Filesystem Objects

Up until now, we talked about filesystem metadata. However, all filesystems exist to organize the actual data we store with them. To us, the metadata is mostly hidden. What is visible are the filesystem objects.

The main such object in most filesystems is the file. **In native Linux filesystems, the term *file* means any object that has an associated inode**. They can be of many different types:

- Regular
- Directory
- Block special
- Character special
- Named pipe (FIFO)
- Symbolic link
- Socket

Let's explore the common ones.

5.1. Regular File

A regular file is the collection of data we usually associate with the word *file*. Apart from their inode's metadata, **regular files have non-system contents, which we read and write via editors and other tools**.

Importantly, we can see how a file's contents are spread via *debugfs* (Filesystem Debugger):

```
$ debugfs /dev/sda
debugfs 1.46.2 (20-Nov-2021)
debugfs: inode_dump -b /file.ext
0000 0af3 0100 0400 0000 0000 0000 0000 0000
................
0020 0100 0000 0034 1100 0000 0000 0000 0000
.....4..........
0040 0000 0000 0000 0000 0000 0000 0000 0000
................
*
```

First, we start *debugfs* with a filesystem block device as its argument. After that, we use the *inode_dump* command with the *-b* flag to show which blocks, by their identifiers, a file consists of.

We usually store most files inside directories.

5.2. Directory

Directories are containers in a filesystem – they can store most objects, including other directories. Linux directories are a list of filenames to inode number mappings. Users see this structure in the form of a tree.

In fact, we can use the *tree* command to confirm this:

```
$ tree -L 2 -d /etc/systemd/
/etc/systemd/
|-- network
|-- system
| |-- default.target.wants
| |-- getty.target.wants
| |-- multi-user.target.wants
| |-- network-online.target.wants
| |-- remote-fs.target.wants
```

```
| |-- sockets.target.wants
| |-- sysinit.target.wants
| `-- timers.target.wants
`-- user
```

The *-d* flag is for listing only directories, while the number after *-L* indicates the level of depth below the specified path (last argument).

In native Linux filesystems, each directory has the following unique associated information:

- . (self inode reference)
- .. (upper-level directory inode reference)
- list of contained objects

Note that the last point is what a directory's blocks contain: a table with filenames and their associated inode numbers. In fact, this is the only place filesystems store filenames. Regular file inodes do not store their names, only their other metadata:

```
$ debugfs /dev/sda
debugfs 1.46.2 (20-Nov-2021)
debugfs: stat /file.ext
Inode: 666 Type: regular Mode: 0644 Flags:
0x80000
Generation: 890607921 Version:
0x00000000:00000001
User: 1000 Group: 0 Project: 0 Size: 166
File ACL: 0
Links: 1 Blockcount: 8
Fragment: Address: 0 Number: 0 Size: 0
ctime: 0x61a50568:29f85c40 -- Mon Nov 29 18:52:56
2021
atime: 0x61a5055f:8491b840 -- Mon Nov 29 18:52:47
2021
mtime: 0x61a5055f:8491b840 -- Mon Nov 29 18:52:47
2021
crtime: 0x61a5055f:8491b840 -- Mon Nov 29
18:52:47 2021
Size of extra inode fields: 32
```

```
Inode checksum: 0x6af32853
EXTENTS:
(0):1064843
```

As we can confirm above, the filename is not part of the inode data, although the *stat* command does add it for clarity.

We can make more complex tree structures via links.

5.3. Link

Sometimes we might want a filesystem object to exist in more than one place at the same time without actually duplicating data. In these cases, we use links.

In fact, we can think of links like shortcuts to another object. There are two types of links: hard and soft (symbolic).

While soft links point directly to the original object by its current name, hard links are connected with its inode and are just another name for the same file. This is an exception to our earlier statement about where filenames exist – symlinks also keep them. In practice, objects disappear once all of their hard links are removed since symbolic links always point to a hard link. Interestingly, symlinks have their own special inodes.

Furthermore, we can not only create file links but also link to a directory. Doing this can make complex circular structures out of directory trees.

In addition, the inode mechanism can result in some specific behavior, which we'll discuss next.

6. *inode* Filesystem Specifics

When using a filesystem that is based on inodes, we sometimes have to consider its inner workings. Let's look at some special cases.

6.1. Maximum *inodes*

It is possible for a filesystem to run out of inodes, despite free space is available. This usually happens when lots of small files don't take up too much storage, but they deduct from the total inode count.

In practice, we can see total inode count information via the *df* (Disk Free) command:

```
$ df -ih
Filesystem Inodes IUsed IFree IUse% Mounted on
/dev/sda 16M 106K 16M 1% /
```

The above output states that we've used 106 thousand out of the 16 million possible inodes. Of course, we can free inodes to have more available.

6.2. Reallocation

When we delete an inode, the filesystem can allocate it for another file in the future. However, **deleting a file by name isn't the same as deleting an inode**. The former only removes a pointer (hard or soft link), while the latter destroys metadata about the file in the inodes table. Think of it like this:

- files are just names linked to inodes
- inodes consist of file descriptions and their content block pointers
- blocks hold file contents

Consequently, we can scan through the inodes table to restore a deleted file but can't do the same for an inode's metadata under normal circumstances. However, even losing block pointers might be reversible via scans and some detection algorithm before the data is permanently rewritten.

6.3. Inline Files

Another useful function sometimes available in filesystems is inline files. **They are available if the size of mandatory inode metadata is smaller than the inode size**:

```
$ debugfs /dev/sda
debugfs 1.46.2 (20-Nov-2021)
debugfs: stat /file.ext
Inode: 666 Type: regular Mode: 0644 Flags:
0x80000
Generation: 890607921 Version:
0x00000000:00000001
User: 1000 Group: 0 Project: 0 Size: 166
```

```
File ACL: 0
Links: 1 Blockcount: 8
Fragment: Address: 0 Number: 0 Size: 0
ctime: 0x61a50568:29f85c40 -- Mon Nov 29 18:52:56
2021
atime: 0x61a5055f:8491b840 -- Mon Nov 29 18:52:47
2021
mtime: 0x61a5055f:8491b840 -- Mon Nov 29 18:52:47
2021
crtime: 0x61a5055f:8491b840 -- Mon Nov 29
18:52:47 2021
Size of extra inode fields: 32
Inode checksum: 0x6af32853
EXTENTS:
(0):1064843
```

The extra space, 32 bytes in the above example, can contain data and, for very small files, this means we don't need additional disk space to store data and metadata. The feature is recent (May 2016) and may have to be enabled explicitly via *e2fsprogs*.

6.4. *stat* and *debugfs stat*

While most of the information from *stat* in *debugfs* is the same as from the *stat* command, there are some differences.

One of them is the generation number. It can distinguish inode numbers as seen by two different operating systems, e.g., client and server. We use this in specific scenarios where these differences make sense.

Another group is the fragment fields. They show when blocks use the now-obsolete block fragmentation feature.

The last field we see is EXTENTS. It's there when we enable a feature, which allows the filesystem to store contiguous blocks with only the identifiers of the first and last block in the sequence.

7. Summary

In this article, we discussed inodes and their major role in filesystems. To that end, we looked at multiple levels where the information in an inode is essential.

In conclusion, **inodes are the glue between files the user sees and what the system actually stores about those files**.