CHAPTER 5
Filesystems

In this chapter, we focus on files and filesystems. The UNIX concept of "everything is a file" lives on in Linux, and while that's not true 100% of the time, most resources in Linux are indeed files. Files can be everything from the content of the letter you write to your school to the funny GIF you download (from an obviously safe and trusted site).

There are other things that are also exposed as files in Linux—for example, devices and pseudo-devices such as in `echo "Hello modern Linux users" > /dev/pts/0`, which prints "Hello modern Linux users" to the screen. While you may not associate these resources with files, you can access them with the same methods and tools you know from regular files. For example, the kernel exposes certain runtime information (as discussed in "Process Management" on page 17) about a process, such as its PID or the binary used to run the process.

What all these things have in common is a standardized, uniform interface: opening a file, gathering information about a file, writing to a file, and so forth. In Linux, filesystems provide this uniform interface. This interface, together with the fact that Linux treats files as a stream of bytes, without any expectations about the structure, enables us to build tools that work with a range of different file types.

In addition, the uniform interface that filesystems provide reduces your cognitive load, making it faster for you to learn how to use Linux.

In this chapter, we first define some relevant terms. Then, we look at how Linux implements the "everything is a file" abstraction. Next, we review special-purpose filesystems the kernel uses to expose information about processes or devices. We then move on to regular files and filesystems, something you would typically associate with documents, data, and programs. We compare filesystem options and discuss common operations.

Basics

Before we get into the filesystem terminology, let's first make some implicit assumptions and expectations about filesystems more explicit:

- While there are exceptions, most of the widely used filesystems today are hierarchical. That is, they provide the user with a single filesystem tree, starting at the root (`/`).
- In the filesystem tree, you find two different types of objects: directories and files. Think of directories as an organizational unit, allowing you to group files. If you'd like to apply the tree analogy, directories are the nodes in the tree, whereas the leaves are either files or directories.
- You can navigate a filesystem by listing the content of a directory (`ls`), changing into that directory (`cd`), and printing the current working directory (`pwd`).
- Permissions are built-in: as discussed in "Permissions" on page 80, one of the attributes a filesystem captures is ownership. Consequently, ownership enforces access to files and directories via the assigned permissions.
- Generally, filesystems are implemented in the kernel.

While filesystems are usually, for performance reasons, implemented in the kernel space, there's also an option to implement them in user land. See the Filesystem in Userspace (FUSE) documentation and the libfuse project site.

With this informal high-level explanation out of the way, we now focus on some more crisp definitions of terms that you'll need to understand:

Drive
: A (physical) block device such as a hard disk drive (HDD) or a solid-state drive (SSD). In the context of virtual machines, a drive also can be emulated—for example, */dev/sda* (SCSI device) or */dev/sdb* (SATA device) or */dev/hda* (IDE device).

Partition
: You can logically split up drives into partitions, a set of storage sectors. For example, you may decide to create two partitions on your HDD, which then would show up as */dev/sdb1* and */dev/sdb2*.

Volume
: A volume is somewhat similar to a partition, but it is more flexible, and it is also formatted for a specific filesystem. We'll discuss volumes in detail in "Logical Volume Manager" on page 99.

Super block
: When formatted, filesystems have a special section in the beginning that captures the metadata of the filesystem. This includes things like filesystem type, blocks, state, and how many inodes per block.

Inodes
: In a filesystem, inodes store metadata about files, such as size, owner, location, date, and permissions. However, inodes do not store the filename and the actual data. This is kept in directories, which really are just a special kind of regular file, mapping inodes to filenames.

That was a lot of theory, so let's see these concepts in action. First, here's how to see what drives, partitions, and volumes are present in your system:

```
$ lsblk --exclude 7 ❶
NAME                      MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda                         8:0    0 223.6G  0 disk             ❷
├─sda1                      8:1    0   512M  0 part /boot/efi   ❸
└─sda2                      8:2    0 223.1G  0 part             ❹
  ├─elementary--vg-root   253:0    0 222.1G  0 lvm  /
  └─elementary--vg-swap_1 253:1    0   976M  0 lvm  [SWAP]
```

❶ List all block devices but exclude pseudo (loop) devices.

❷ We have a disk drive called *sda* with some 223 GB overall.

❸ There are two partitions here, with *sda1* being the boot partition.

❹ The second partition, called *sda2*, contains two volumes (see "Logical Volume Manager" on page 99 for details).

Now that we have an overall idea of the physical and logical setup, let's have a closer look at the filesystems in use:

```
$ findmnt -D -t nosquashfs ❶
SOURCE                          FSTYPE     SIZE  USED  AVAIL USE% TARGET
udev                            devtmpfs   3.8G     0   3.8G   0% /dev
tmpfs                           tmpfs    778.9M  1.6M 777.3M   0% /run
/dev/mapper/elementary--vg-root ext4     217.6G 13.8G 192.7G   6% /
tmpfs                           tmpfs      3.8G 19.2M   3.8G   0% /dev/shm
tmpfs                           tmpfs        5M    4K     5M   0% /run/lock
tmpfs                           tmpfs      3.8G     0   3.8G   0% /sys/fs/cgroup
/dev/sda1                       vfat       511M    6M 504.9M   1% /boot/efi
tmpfs                           tmpfs    778.9M   76K 778.8M   0% /run/user/1000
```

❶ List filesystems but exclude squashfs types (specialized read-only compressed filesystem originally developed for CDs, now also for snapshots).

We can go a step further and look at individual filesystem objects such as directories or files:

```
$ stat myfile
  File: myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file ❶
Device: fc01h/64513d    Inode: 555036      Links: 1 ❷
Access: (0664/-rw-rw-r--)  Uid: ( 1000/     mh9)   Gid: ( 1001/     mh9)
Access: 2021-08-29 09:26:36.638447261 +0000
Modify: 2021-08-29 09:26:36.638447261 +0000
Change: 2021-08-29 09:26:36.638447261 +0000
 Birth: 2021-08-29 09:26:36.638447261 +0000
```

❶ File type information

❷ Information about device and inode

In the previous command, if we used `stat .` (note the dot), we would have gotten the respective directory file information, including its inode, number of blocks used, and so forth.

Table 5-1 lists some basic filesystem commands that allow you to explore the concepts we introduced earlier.

Table 5-1. Selection of low-level filesystem and block device commands

Command	Use case
`lsblk`	List all block devices
`fdisk`, `parted`	Manage disk partitions
`blkid`	Show block device attributes such as UUID
`hwinfo`	Show hardware information
`file -s`	Show filesystem and partition information
`stat`, `df -i`, `ls -i`	Show and list inode-related information

Another term you'll come across in the context of filesystems is that of *links*. Sometimes you want to refer to files with different names or provide shortcuts. There are two types of links in Linux:

Hard links
: Reference inodes and can't refer to directories. They also do not work across filesystems.

Symbolic links, or symlinks
: Special files with their content being a string representing the path of another file.

Now let's see links in action (some outputs shortened):

```
$ ln myfile somealias ❶
$ ln -s myfile somesoftalias ❷

$ ls -al *alias ❸
-rw-rw-r-- 2 mh9 mh9 0 Sep  5 12:15 somealias
lrwxrwxrwx 1 mh9 mh9 6 Sep  5 12:45 somesoftalias -> myfile

$ stat somealias ❹
  File: somealias
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: fd00h/64768d    Inode: 6302071     Links: 2
...
$ stat somesoftalias ❺
  File: somesoftalias -> myfile
  Size: 6               Blocks: 0          IO Block: 4096   symbolic link
Device: fd00h/64768d    Inode: 6303540     Links: 1
...
```

❶ Create a hard link to *myfile*.

❷ Create a soft link to the same file (notice the `-s` option).

❸ List the files. Notice the different file types and the rendering of the name. We could also have used `ls -ali *alias`, which would show that the inodes were the same on the two names associated with the hard link.

❹ Show the file details of the hard link.

❺ Show the file details of the soft link.

Now that you're familiar with filesystem terminology let's explore how Linux makes it possible to treat any kind of resource as a file.

The Virtual File System

Linux manages to provide a file-like access to many sorts of resources (in-memory, locally attached, or networked storage) through an abstraction called the virtual file system (VFS). The basic idea is to introduce a layer of indirection between the clients (syscalls) and the individual filesystems implementing operations for a concrete

device or other kind of resource. This means that VFS separates the generic operation (open, read, seek) from the actual implementation details.

VFS is an abstraction layer in the kernel that provides clients a common way to access resources, based on the file paradigm. A file, in Linux, doesn't have any prescribed structure; it's just a stream of bytes. It's up to the client to decide what the bytes mean. As shown in Figure 5-1, VFS abstracts access to different kinds of filesystems:

Local filesystems, such as `ext3`*, XFS, FAT, and NTFS*
: These filesystems use drivers to access local block devices such as HDDs or SSDs.

In-memory filesystems, such as `tmpfs`*, that are not backed by long-term storage devices but live in main memory (RAM)*
: We'll cover these and the previous category in "Regular Files" on page 108.

Pseudo filesystems like `procfs`*, as discussed in "Pseudo Filesystems" on page 104*
: These filesystems are also in-memory in nature. They're used for kernel interfacing and device abstractions.

Networked filesystems, such as NFS, Samba, Netware (nee Novell), and others
: These filesystems also use a driver; however, the storage devices where the actual data resides is not locally attached but remote. This means that the driver involves network operations. For this reason, we'll cover them in Chapter 7.

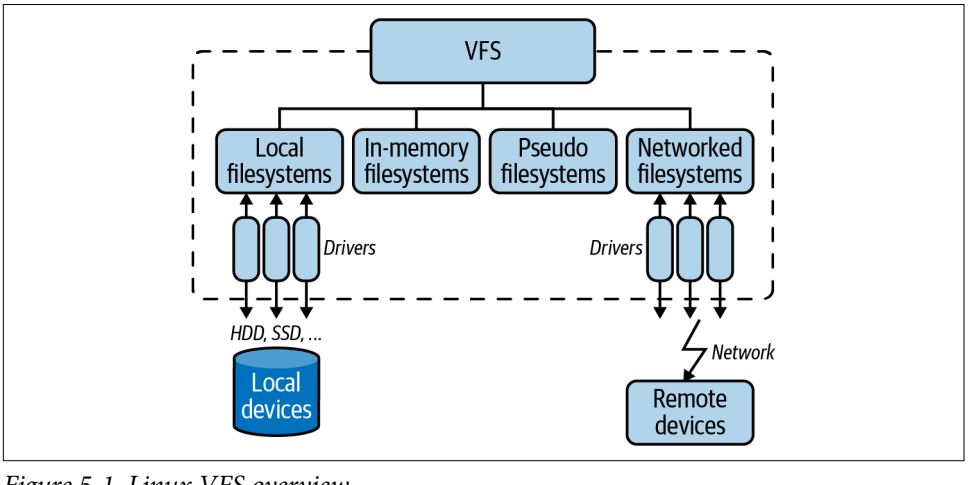

Figure 5-1. Linux VFS overview

Describing the makeup of the VFS isn't easy. There are over 100 syscalls related to files; however, in its core, the operations can be grouped into a handful of categories, as listed in Table 5-2.

Table 5-2. Select syscalls making up the VFS interface

Category	Example syscalls
Inodes	`chmod`, `chown`, `stat`
Files	`open`, `close`, `seek`, `truncate`, `read`, `write`
Directories	`chdir`, `getcwd`, `link`, `unlink`, `rename`, `symlink`
Filesystems	`mount`, `flush`, `chroot`
Others	`mmap`, `poll`, `sync`, `flock`

Many VFS syscalls dispatch to the filesystem-specific implementation. For other syscalls, there are VFS default implementations. Further, the Linux kernel defines relevant VFS data structures—see *include/linux/fs.h*—such as the following:

`inode`
: The core filesystem object, capturing type, ownership, permissions, links, pointers to blocks containing the file data, creation and access statistics, and more

`file`
: Representing an open file (including path, current position, and inode)

`dentry` *(directory entry)*
: Stores its parent and children

`super_block`
: Representing a filesystem including mount information

Others
: Including `vfsmount` and `file_system_type`

With the VFS overview done, let's have a closer look at the details, including volume management, filesystem operations, and common file system layouts.

Logical Volume Manager

We previously talked about carving up drives using partitions. While doing this is possible, partitions are hard to use, especially when resizing (changing the amount of storage space) is necessary.

Logical volume manager (LVM) uses a layer of indirection between physical entities (such as drives or partitions) and the file system. This yields a setup that allows for risk-free, zero-downtime expanding and automatic storage extension through the pooling of resources. The way LVM works is depicted in Figure 5-2, with key concepts explained in the passage that follows.

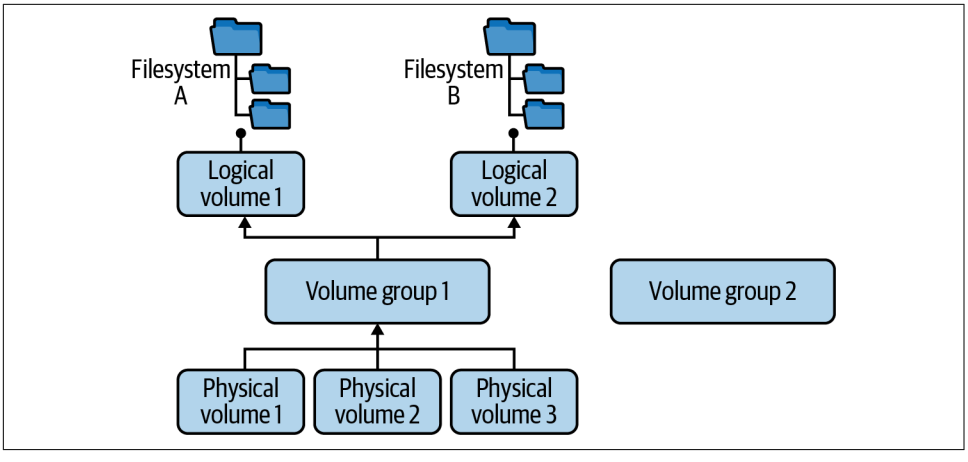

Figure 5-2. Linux LVM overview

Physical volumes (PV)
: Can be a disk partition, an entire disk drive, and other devices.

Logical volumes (LV)
: Are block devices created from VGs. These are conceptually comparable to partitions. You have to create a filesystem on an LV before you can use it. You can easily resize LVs while in use.

Volume groups (VG)
: Are a go-between between a set of PVs and LVs. Think of a VG as pools of PVs collectively providing resources.

To manage volumes with LVM, a number of tools are required; however, they are consistently named and relatively easy to use:

PV management tools

- `lvmdiskscan`
- `pvdisplay`
- `pvcreate`
- `pvscan`

VG management tools

- `vgs`
- `vgdisplay`
- `vgcreate`
- `vgextend`

LV management tools

- `lvs`
- `lvscan`
- `lvcreate`

Let's see some LVM commands in action, using a concrete setup:

```
$ sudo lvscan ❶
  ACTIVE            '/dev/elementary-vg/root' [<222.10 GiB] inherit
  ACTIVE            '/dev/elementary-vg/swap_1' [976.00 MiB] inherit

$ sudo vgs ❷
  VG            #PV #LV #SN Attr   VSize    VFree
  elementary-vg   1   2   0 wz--n- <223.07g 16.00m

$ sudo pvdisplay ❸
  --- Physical volume ---
  PV Name               /dev/sda2
  VG Name               elementary-vg
  PV Size               <223.07 GiB / not usable 3.00 MiB
  Allocatable           yes
  PE Size               4.00 MiB
  Total PE              57105
  Free PE               4
  Allocated PE          57101
  PV UUID               2OrEfB-77zU-jun3-a0XC-QiJH-erDP-1ujfAM
```

❶ List logical volumes; we have two here (*root* and *swap_1*) using volume group *elementary-vg*.

❷ Display volume groups; we have one here called *elementary-vg*.

❸ Display physical volumes; we have one here (*/dev/sda2*) that's assigned to the volume group *elementary-vg*.

Whether you use a partition or an LV, two more steps, which we'll cover next, are necessary to use a filesystem.

Filesystem Operations

In the following section, we'll discuss how to create a filesystem, given a partition or a logical volume (created using LVM). There are two steps involved: creating the filesystem—in other non-Linux operating systems, this step is sometimes called *formatting*—and then mounting it, or inserting it into the filesystem tree.

Creating filesystems

In order to use a filesystem, the first step is to create one. This means that you're setting up the management pieces that make up a filesystem, taking a partition or a volume as the input. Consult Table 5-1 if you're unsure how to gather the necessary information about the input, and once you have everything together, use `mkfs` to create a filesystem.

`mkfs` takes two primary inputs: the type of filesystem you want to create (see one of the options we discuss in "Common Filesystems" on page 109) and the device you want to create the filesystem on (for example, a logical volume):

```
mkfs -t ext4 \ ❶
    /dev/some_vg/some_lv ❷
```

❶ Create a filesystem of type `ext4`.

❷ Create the filesystem on the logical volume */dev/some_vg/some_lv*.

As you can see from the previous command, there's not much to it to create a filesystem, so the main work for you is to figure out what filesystem type to use.

Once you have created the filesystem with `mkfs`, you can then make it available in the filesystem tree.

Mounting filesystems

Mounting a filesystem means attaching it to the filesystem tree (which starts at /). Use the `mount` command to attach a filesystem. `mount` takes two main inputs: the device you want to attach and the place in the filesystem tree. In addition, you can provide other inputs, including mount options (via `-o`) such as read-only, and bind mounts—via `--bind`—for mounting directories into the filesystem tree. We'll revisit this latter option in the context of containers.

You can use `mount` on its own as well. Here's how to list existing mounts:

```
$ mount -t ext4,tmpfs ❶
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=797596k,mode=755)
/dev/mapper/elementary--vg-root on / type ext4 (rw,relatime,errors=remount-ro) ❷
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,relatime,size=5120k)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
```

❶ List mounts but only show certain filesystem types (`ext4` and `tmpfs` here).

❷ An example mount: the LVM VG */dev/mapper/elementary--vg-root* of type `ext4` is mounted at the root.

You must make sure that you mount a filesystem using the type it has been created with. For example, if you're trying to mount an SD card using `mount -t vfat /dev/sdX2 /media`, you have to know the SD card is formatted using `vfat`. You can let `mount` try all filesystems until one works using the `-a` option.

Further, the mounts are valid only for as long as the system is running, so in order to make it permanent, you need to use the fstab file (*/etc/fstab*). For example, here is mine (output slightly edited to fit):

```
$ cat /etc/fstab
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
/dev/mapper/elementary--vg-root / ext4 errors=remount-ro 0 1
# /boot/efi was on /dev/sda1 during installation
UUID=2A11-27C0  /boot/efi vfat umask=0077 0 1
/dev/mapper/elementary--vg-swap_1 none swap sw 0 0
```

Now you know how to manage partitions, volumes, and filesystems. Next up, we review common ways to organize filesystems.

Common Filesystem Layouts

Once you have a filesystem in place, an obvious challenge is to come up with a way to organize its content. You may want to organize things like where programs are stored, configuration data, system data, and user data. We will refer to this organization of directories and their content as the *filesystem layout*. Formally, the layout is called the Filesystem Hierarchy Standard (FHS). It defines directories, including their structure and recommended content. The Linux Foundation maintains the FHS, and it's a good starting point for Linux distributions to follow.

The idea behind FHS is laudable. However, in practice you will find that the filesystem layout very much depends on the Linux distribution you're using. Thus, I strongly recommend you use the `man hier` command to learn about your concrete setup.

To provide you with a high-level idea of what you can expect when you see certain top-level directories, I compiled a list of common ones in Table 5-3.

Table 5-3. Common top-level directories

Directory	Semantics
bin, sbin	System programs and commands (usually links to */usr/bin* and */usr/sbin*)
boot	Kernel images and related components

Directory	Semantics
dev	Devices (terminals, drives, etc.)
etc	System configuration files
home	User home directories
lib	Shared system libraries
mnt, *media*	Mount points for removable media (e.g., USB sticks)
opt	Distro specific; can host package manager files
proc, *sys*	Kernel interfaces; see also "Pseudo Filesystems" on page 104
tmp	For temporary files
usr	User programs (usually read-only)
var	User programs (logs, backups, network caches, etc.)

With that, let's move on to some special kinds of filesystems.

Pseudo Filesystems

Filesystems are a great way to structure and access information. By now you have likely already internalized the Linux motto that "everything is a file." We looked at how Linux provides a uniform interface via VFS in "The Virtual File System" on page 97. Now, let's take a closer look at how an interface is provided in cases where the VFS implementor is not a block device (such as an SD card or an SSD drive).

Meet pseudo filesystems: they only pretend to be filesystems so that we can interact with them in the usual manner (`ls`, `cd`, `cat`), but really they are wrapping some kernel interface. The interface can be a range of things, including the following:

- Information about a process
- An interaction with devices such as keyboards
- Utilities such as special devices you can use as data sources or sinks

Let's have a closer look at the three major pseudo filesystems Linux has, starting with the oldest.

procfs

Linux inherited the */proc* filesystem (`procfs`) from UNIX. The original intention was to publish process-related information from the kernel, to make it consumable for system commands such as `ps` or `free`. It has very few rules around structure, allows read-write access, and over time many things found their way into it. In general, you find two types of information there:

- Per-process information in */proc/PID/*. This is process-relevant information that the kernel exposes via directories with the PID as the directory name. Details concerning the information available there are listed in Table 5-4.
- Other information such as mounts, networking-related information, TTY drivers, memory information, system version, and uptime.

You can glean per-process information as listed in Table 5-4 simply by using commands like `cat`. Note that most are read-only; the write semantics depend on the underlying resource.

Table 5-4. Per-process information in procfs (most notable)

Entry	Type	Information
`attr`	Directory	Security attributes
`cgroup`	File	Control groups
`cmdline`	File	Command line
`cwd`	Link	Current working directory
`environ`	File	Environment variables
`exe`	Link	Executable of the process
`fd`	Directory	File descriptors
`io`	File	Storage I/O (bytes/char read and written)
`limits`	File	Resource limits
`mem`	File	Memory used
`mounts`	File	Mounts used
`net`	Directory	Network stats
`stat`	File	Process status
`syscall`	File	Syscall usage
`task`	Directory	Per-task (thread) information
`timers`	File	Timers information

To see this in action, let's inspect the process status. We're using `status` here rather than `stat`, which doesn't come with human-readable labels:

```
$ cat /proc/self/status | head -10 ❶
Name:   cat
Umask:  0002
State:  R (running) ❷
Tgid:   12011
Ngid:   0
Pid:    12011 ❸
PPid:   3421 ❹
TracerPid:      0
```

```
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
```

❶ Get the process status about the currently running command, showing only the first 10 lines.

❷ The current state (running, on-CPU).

❸ The PID of the current process.

❹ The process ID of the parent process of the command; in this case, it's the shell where I ran the `cat` command in.

Here is one more example of using `procfs` to glean information, this time from the networking space:

```
$ cat /proc/self/net/arp
IP address       HW type     Flags       HW address            Mask     Device
192.168.178.1    0x1         0x2         3c:a6:2f:8e:66:b3     *        wlp1s0
192.168.178.37   0x1         0x2         dc:54:d7:ef:90:9e     *        wlp1s0
```

As shown in the previous command, we can glean ARP information about the current process from this special */proc/self/net/arp*.

`procfs` is very useful if you're low-level debugging or developing system tooling. It is relatively messy, so you'll need the kernel docs or, even better, the kernel source code at hand to understand what each file represents and how to interpret the information in it.

Let's move on to a more recent, more orderly way the kernel exposes information.

sysfs

Where `procfs` is pretty Wild West, the */sys* filesystem (`sysfs`) is a Linux-specific, structured way for the kernel to expose select information (such as about devices) using a standardized layout.

Here are the directories in `sysfs`:

block/
: This directory symbolic links to discovered block devices.

bus/
: In this directory, you find one subdirectory for each physical bus type supported in the kernel.

class/
: This directory contains device classes.

dev/
: This directory contains two subdirectories: *block/* for block devices and *char/* for character devices on the system, structured with `major-ID:minor-ID`.

devices/
: In this directory, the kernel provides a representation of the device tree.

firmware/
: Via these directories, you can manage firmware-specific attributes.

fs/
: This directory contains subdirectories for some filesystems.

module/
: In these directories you find subdirectories for each module loaded in the kernel.

There are more subdirectories in `sysfs`, but some are newish and/or would benefit from better documentation. You'll find certain information duplicated in `sysfs` that is also available in `procfs`, but other information (such as memory information) is only available in `procfs`.

Let's see `sysfs` in action (output edited to fit):

```
$ ls -al /sys/block/sda/ | head -7 ❶
total 0
drwxr-xr-x 11 root root    0 Sep  7 11:49 .
drwxr-xr-x  3 root root    0 Sep  7 11:49 ..
-r--r--r--  1 root root 4096 Sep  8 16:22 alignment_offset
lrwxrwxrwx  1 root root    0 Sep  7 11:51 bdi ->  ../../../virtual/bdi/8:0 ❷
-r--r--r--  1 root root 4096 Sep  8 16:22 capability ❸
-r--r--r--  1 root root 4096 Sep  7 11:49 dev ❹
```

❶ List information about block device `sda`, showing only the first seven lines.

❷ The `backing_dev_info` link using `MAJOR:MINOR` format.

❸ Captures device capabilities, such as if it is removable.

❹ Contains the device major and minor number (`8:0`); see also the block device drivers reference for what the numbers mean.

Next up in our little pseudo filesystem review are devices.

devfs

The */dev* filesystem (`devfs`) hosts device special files, representing devices ranging from physical devices to things like a random number generator or a write-only data sink.

The devices available and managed via `devfs` are:

Block devices
: Handle data in blocks—for example, storage devices (drives)

Character devices
: Handle things character by character, such as a terminal, a keyboard, or a mouse

Special devices
: Generate data or allow you to manipulate it, including the famous */dev/null* or */dev/random*

Let's now see `devfs` in action. For example, assume you want to get a random string. You could do something like the following:

```
tr -dc A-Za-z0-9 < /dev/urandom | head -c 42
```

The previous command generates a 42-character random sequence containing uppercase and lowercase as well as numerical characters. And while */dev/urandom* looks like a file and can be used like one, it indeed is a special file that, using a number of sources, generates (more or less) random output.

What do you think about the following command:

```
echo "something" > /dev/tty
```

That's right! The string "something" appeared on your display, and that is by design. */dev/tty* stands for the terminal, and with that command we sent something (quite literally) to it.

With a good understanding of filesystems and their features, let's now turn our attention to filesystems that you want to use to manage regular files such as documents and data files.

Regular Files

In this section, we focus on regular files and filesystems for such file types. Most of the day-to-day files we're dealing with when working fall into this category: office documents, YAML and JSON configuration files, images (PNG, JPEG, etc.), source code, plain text files, and so on.

Linux comes with a wealth of options. We'll focus on local filesystems, both those native for Linux as well as those in other operating systems (such as Windows/DOS) that Linux allows you to use. First, let's have a look at some common filesystems.

Common Filesystems

The term *common filesystem* doesn't have a formal definition. It's simply an umbrella term for filesystems that are either the defaults used in Linux distributions or widely used in storage devices such as removable devices (USB sticks and SD cards) or read-only devices, like CDs and DVDs.

In Table 5-5 I provide a quick overview and comparison of some common filesystems that enjoy in-kernel support. Later in this section, we'll review some popular filesystems in greater detail.

Table 5-5. Common filesystems for regular files

Filesystem	Linux support since	File size	Volume size	Number of files	Filename length
`ext2`	1993	2 TB	32 TB	10^{18}	255 characters
`ext3`	2001	2 TB	32 TB	variable	255 characters
`ext4`	2008	16 TB	1 EB	4 billion	255 characters
`btrfs`	2009	16 EB	16 EB	2^{18}	255 characters
XFS	2001	8 EB	8 EB	2^{64}	255 characters
ZFS	2006	16 EB	2^{128} Bytes	10^{14} files per directory	255 characters
NTFS	1997	16 TB	256 TB	2^{32}	255 characters
`vfat`	1995	2 GB	N/A	2^{16} per directory	255 characters

The information provided in Table 5-5 is meant to give you a rough idea about the filesystems. Sometimes it's hard to pinpoint the exact time a filesystem would be officially considered part of Linux; sometimes the numbers make sense only with the relevant context applied. For example, there are differences between theoretical limits and implementation.

Now let's take a closer look at some widely used filesystems for regular files:

`ext4`
: A widely used filesystem, used by default in many distributions nowadays. It's a backward-compatible evolution of `ext3`. Like `ext3`, it offers journaling—that is, changes are recorded in a log so that in the worst-case scenario (think: power outage), the recovery is fast. It's a great general-purpose choice. See the `ext4` manual for usage.

XFS
: A journaling filesystem that was originally designed by Silicon Graphics (SGI) for their workstations in the early 1990s. Offering support for large files and high-speed I/O, it's now used, for example, in the Red Hat distributions family.

ZFS
: Originally developed by Sun Microsystems in 2001, ZFS combines filesystem and volume manager functionality. While now there is the OpenZFS project, offering a path forward in an open source context, there are some concerns about ZFS's integration with Linux.

FAT
: This is really a family of FAT filesystems for Linux, with `vfat` being used most often. The main use case is interoperability with Windows systems, as well as removable media that uses FAT. Many of the native considerations around volumes do not apply.

Drives are not the only place one can store data, so let's have a look at in-memory options.

In-Memory Filesystems

There are a number of in-memory filesystems available; some are general purpose and others have very specific use cases. In the following, we list some widely used in-memory filesystems (in alphabetical order):

`debugfs`
: A special-purpose filesystem used for debugging; usually mounted with `mount -t debugfs none /sys/kernel/debug`.

`loopfs`
: Allows mapping a filesystem to blocks rather than devices. See also a mail thread on the background.

`pipefs`
: A special (pseudo) filesystem mounted on `pipe:` that enables pipes.

`sockfs`
: Another special (pseudo) filesystem that makes network sockets look like files, sitting between the syscalls and the sockets.

`swapfs`
: Used to realize swapping (not mountable).

`tmpfs`
: A general-purpose filesystem that keeps file data in kernel caches. It's fast but nonpersistent (power off means data is lost).

Let's move on to a special category of filesystems, specifically relevant in the context of "Containers" on page 131.

Copy-on-Write Filesystems

Copy-on-write (CoW) is a nifty concept to increase I/O speed and at the same time use less space. The way it works is depicted in Figure 5-3, with further explanation in the passage that follows.

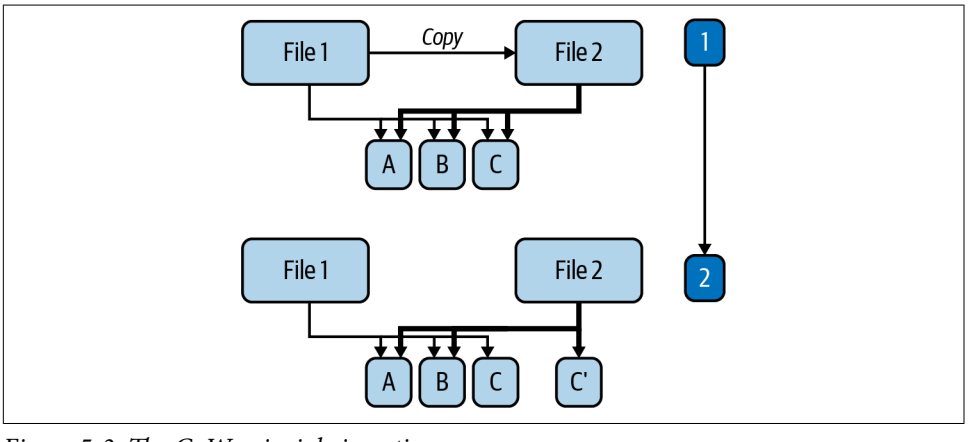

Figure 5-3. The CoW principle in action

1. The original file, File 1, consisting of blocks A, B, and C, is copied to a file called File 2. Rather than copying the actual blocks, only the metadata (pointers to the blocks) is copied. This is fast and doesn't use up much space since only metadata is created.
2. When File 2 is modified (let's say something in block C is changed), only then is block C copied: a new block called C′ is created, and while File 2 still points to (uses) the unmodified blocks A and B, it now uses a new block (C′) to capture new data.

Before we get to implementations, we need to understand a second concept relevant in this context: union mounts. This is the idea that you can combine (mount) multiple directories into one location so that, to the user of the resulting directory, it appears that said directory contains the combined content (or: union) of all the participating directories. With union mounts, you often come across the terms *upper filesystem* and *lower filesystem*, hinting at the layering order of the mounts. You'll find more details in the article "Unifying Filesystems with Union Mounts".

With union mounts, the devil is in the details. You have to come up with rules around what happens when a file exists in multiple places or what writing to or removing files means.

Let's have a quick look at implementations of CoW in the context of Linux filesystems. We'll have a closer look at some of these in the context of Chapter 6, when we discuss their use as a building block for container images.

Unionfs
: Originally developed at Stony Brook University, Unionfs implements a union mount for CoW filesystems. It allows you to transparently overlay files and directories from different filesystems using priorities at mount time. It was widely popular and used in the context of CD-ROMs and DVDs.

OverlayFS
: A union mount filesystem implementation for Linux introduced in 2009 and added to the kernel in 2014. With OverlayFS, once a file is opened, all operations are directly handled by the underlying (lower or upper) filesystems.

AUFS
: Another attempt to implement an in-kernel union mount, AUFS (short for advanced multilayered unification filesystem; originally AnotherUnionFS) has not been merged into the kernel yet. It is used to default in Docker (see "Docker" on page 138; nowadays Docker defaults to OverlayFS with storage driver `overlay2`).

`btrfs`
: Short for b-tree filesystem (and pronounced *butterFS* or *betterFS*), `btrfs` is a CoW initially designed by Oracle Corporation. Today, a number of companies contribute to the `btrfs` development, including Facebook, Intel, SUSE, and Red Hat.

  It comes with a number of features such as snapshots (for software-based RAID) and automatic detection of silent data corruptions. This makes `btrfs` very suitable for professional environments—for example, on a server.

Conclusion

In this chapter, we discussed files and filesystems in Linux. Filesystems are a great and flexible way to organize access to information in a hierarchical manner. Linux has many technologies and projects around filesystems. Some are open source based, but there is also a range of commercial offerings.

We discussed the basic building blocks, from drives to partitions and volumes. Linux realizes the "everything is a file" abstraction using VFS, supporting virtually any kind of filesystem, local or remote.

The kernel uses pseudo filesystems such as */proc* and */sys* to expose information about processes or devices. You can interact with these (in-memory) filesystems that represent kernel APIs just like with filesystems such as `ext4` (that you use to store files).

We then moved on to regular files and filesystems, where we compared common local filesystem options, as well as in-memory and CoW filesystem basics. Linux's filesystem support is comprehensive, allowing you to use (at least read) a range of filesystems, including those originating from other operating systems such as Windows.

You can dive deeper into the topics covered in this chapter with the following resources:

Basics

- "UNIX File Systems: How UNIX Organizes and Accesses Files on Disk"
- "KHB: A Filesystems Reading List"

VFS

- "Overview of the Linux Virtual File System"
- "Introduction to the Linux Virtual Filesystem (VFS)"
- "LVM" on ArchWiki
- "LVM2 Resource Page"
- "How to Use GUI LVM Tools"
- "Linux Filesystem Hierarchy"
- "Persistent BPF Objects"

Regular files

- "Filesystem Efficiency—Comparison of EXT4, XFS, BTRFS, and ZFS" thread on reddit
- "Linux Filesystem Performance Tests"
- "Comparison of File Systems for an SSD" thread on Linux.org
- "Kernel Korner—Unionfs: Bringing Filesystems Together"
- "Getting Started with btrfs for Linux"

Equipped with knowledge around filesystems, we're now ready to bring things together and focus on how to manage and launch applications.