That's it for `sudo` for now. If you need to use its more advanced features, see the sudoers(5) and sudo(8) manual pages. (The actual mechanics of user switching are covered in Chapter 7.)

2.21 Looking Forward

You should now know how to do the following at the command line: run programs, redirect output, interact with files and directories, view process listings, view manual pages, and generally make your way around the user space of a Linux system. You should also be able to run commands as the superuser. You may not yet know much about the internal details of user-space components or what goes on in the kernel, but with the basics of files and processes under your belt, you're on your way. In the next few chapters, you'll be working with both kernel and user-space system components using the command-line tools that you just learned.

Chapter 3. Devices

This chapter is a basic tour of the kernel-provided device infrastructure in a functioning Linux system. Throughout the history of Linux, there have been many changes to how the kernel presents devices to the user. We'll begin by looking at the traditional system of device files to see how the kernel provides device configuration information through sysfs. Our goal is to be able to extract information about the devices on a system in order to understand a few rudimentary operations. Later chapters will cover interacting with specific kinds of devices in greater detail.

It's important to understand how the kernel interacts with user space when presented with new devices. The udev system enables user-space programs to automatically configure and use new devices. You'll see the basic workings of how the kernel sends a message to a user-space process through udev, as well as what the process does with it.

3.1 Device Files

It is easy to manipulate most devices on a Unix system because the kernel presents many of the device I/O interfaces to user processes as files. These device files are sometimes called *device nodes*. Not only can a programmer use regular file operations to work with a device, but some devices are also accessible to standard programs like `cat`, so you don't have to be a programmer to use a device. However, there is a limit to what you can do with a file interface, so not all devices or device capabilities are accessible with standard file I/O.

Linux uses the same design for device files as do other Unix flavors. Device files are in the */dev* directory, and running `ls /dev` reveals more than a few files in */dev*. So how do you work with devices?

To get started, consider this command:

```
$ echo blah blah > /dev/null
```

As does any command with redirected output, this sends some stuff from the standard output to a file. However, the file is */dev/null*, a device, and the kernel decides what to do with any data written to this device. In the case of */dev/null*, the kernel simply ignores the input and throws away the data.

To identify a device and view its permissions, use `ls -l`:

Example 3-1. Device files

```
$ ls -l
brw-rw----    1 root disk 8, 1 Sep  6 08:37 sda1
crw-rw-rw-    1 root root 1, 3 Sep  6 08:37 null
prw-r--r--    1 root root    0 Mar  3 19:17 fdata
srw-rw-rw-    1 root root    0 Dec 18 07:43 log
```

Note the first character of each line (the first character of the file's mode) in Example 3-1. If this character is `b`, `c`, `p`, or `s`, the file is a device. These letters stand for *block*, *character*, *pipe*, and *socket*, respectively, as described in more detail below.

Block device

- Programs access data from a *block device* in fixed chunks. The *sda1* in the preceding example is a *disk device*, a type of block device. Disks can be easily split up into blocks of data. Because a block device's

total size is fixed and easy to index, processes have random access to any block in the device with the help of the kernel.

Character device

- Character devices work with data streams. You can only read characters from or write characters to character devices, as previously demonstrated with */dev/null*. Character devices don't have a size; when you read from or write to one, the kernel usually performs a read or write operation on the device. Printers directly attached to your computer are represented by character devices. It's important to note that during character device interaction, the kernel cannot back up and reexamine the data stream after it has passed data to a device or process.

Pipe device

- *Named pipes* are like character devices, with another process at the other end of the I/O stream instead of a kernel driver.

Socket device

- *Sockets* are special-purpose interfaces that are frequently used for interprocess communication. They're often found outside of the */dev* directory. Socket files represent Unix domain sockets; you'll learn more about those in Chapter 10.

The numbers before the dates in the first two lines of Example 3-1 are the *major* and *minor* device numbers that help the kernel identify the device. Similar devices usually have the same major number, such as *sda3* and *sdb1* (both of which are hard disk partitions).

NOTE

Not all devices have device files because the block and character device I/O interfaces are not appropriate in all cases. For example, network interfaces don't have device files. It is theoretically possible to interact with a network interface using a single character device, but because it would be exceptionally difficult, the kernel uses other I/O interfaces.

3.2 The sysfs Device Path

The traditional Unix */dev* directory is a convenient way for user processes to reference and interface with devices supported by the kernel, but it's also a very simplistic scheme. The name of the device in */dev* tells you a little about the device, but not a lot. Another problem is that the kernel assigns devices in the order in which they are found, so a device may have a different name between reboots.

To provide a uniform view for attached devices based on their actual hardware attributes, the Linux kernel offers the sysfs interface through a system of files and directories. The base path for devices is */sys/devices*. For example, the SATA hard disk at */dev/sda* might have the following path in sysfs:

```
/sys/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/s
da
```

As you can see, this path is quite long compared with the */dev/sda* filename, which is also a directory. But you can't really compare the two paths because they have different purposes. The */dev* file is there so that user processes can use the device, whereas the */sys/devices* path is used to view information and manage the device. If you list the contents of a device path such as the preceding one, you'll see something like the following:

`alignment_offset`	`discard_alignment`	`holders`	`removable`	`size`	`uevent`

bdi	events	inflight	ro	slaves	
capability	events_async	power	sda1	stat	
dev	events_poll_msecs	queue	sda2	subsystem	
device	ext_range	range	sda5	trace	

The files and subdirectories here are meant to be read primarily by programs rather than humans, but you can get an idea of what they contain and represent by looking at an example such as the */dev* file. Running `cat dev` in this directory displays the numbers `8:0`, which happen to be the major and minor device numbers of */dev/sda*.

There are a few shortcuts in the */sys* directory. For example, */sys/block* should contain all of the block devices available on a system. However, those are just symbolic links; run `ls -l /sys/block` to reveal the true sysfs paths.

It can be difficult to find the sysfs location of a device in */dev*. Use the `udevadm` command to show the path and other attributes:

```
$ udevadm info --query=all --name=/dev/sda
```

NOTE

*The `udevadm` program is in /*sbin*; you can put this directory at the end of your path if it's not already there.*

You'll find more details about `udevadm` and the entire udev system in 3.5 udev.

3.3 dd and Devices

The program `dd` is extremely useful when working with block and character devices. This program's sole function is to read from an input file or stream and write to an output file or stream, possibly doing some encoding conversion on the way.

`dd` copies data in blocks of a fixed size. Here's how to use `dd` with a character device and some common options:

```
$ dd if=/dev/zero of=new_file bs=1024 count=1
```

As you can see, the `dd` option format differs from the option formats of most other Unix commands; it's based on an old IBM Job Control Language (JCL) style. Rather than use the dash (-) character to signal an option, you name an option and set its value to something with the equals (=) sign. The preceding example copies a single 1024-byte block from */dev/zero* (a continuous stream of zero bytes) to *new_file*.

These are the important `dd` options:

- **`if=`*`file`*** The input file. The default is the standard input.
- **`of=`*`file`*** The output file. The default is the standard output.
- **`bs=`*`size`*** The block size. `dd` reads and writes this many bytes of data at a time. To abbreviate large chunks of data, you can use `b` and `k` to signify 512 and 1024 bytes, respectively. Therefore, the example above could read `bs=1k` instead of `bs=1024`.

- **ibs=*size*, obs=*size*** The input and output block sizes. If you can use the same block size for both input and output, use the `bs` option; if not, use `ibs` and `obs` for input and output, respectively.
- **count=*num*** The total number of blocks to copy. When working with a huge file—or with a device that supplies an endless stream of data, such as */dev/zero*—you want `dd` to stop at a fixed point or you could waste a lot of disk space, CPU time, or both. Use `count` with the `skip` parameter to copy a small piece from a large file or device.
- **skip=**
- ***num*** Skip past the first *num* blocks in the input file or stream and do not copy them to the output.

WARNING

`dd` is very powerful, so make sure you know what you're doing when you run it. It's very easy to corrupt files and data on devices by making a careless mistake. It often helps to write the output to a new file if you're not sure what it will do.

3.4 Device Name Summary

It can sometimes be difficult to find the name of a device (for example, when partitioning a disk). Here are a few ways to find out what it is:

- Query `udevd` using `udevadm` (see 3.5 udev).
- Look for the device in the */sys* directory.
- Guess the name from the output of the `dmesg` command (which prints the last few kernel messages) or the kernel system log file (see 7.2 System Logging). This output might contain a description of the devices on your system.
- For a disk device that is already visible to the system, you can check the output of the `mount` command.
- Run `cat /proc/devices` to see the block and character devices for which your system currently has drivers. Each line consists of a number and name. The number is the major number of the device as described in 3.1 Device Files. If you can guess the device from the name, look in */dev* for the character or block devices with the corresponding major number, and you've found the device files.

Among these methods, only the first is reliable, but it does require udev. If you get into a situation where udev is not available, try the other methods but keep in mind that the kernel might not have a device file for your hardware.

The following sections list the most common Linux devices and their naming conventions.

3.4.1 Hard Disks: /dev/sd*

Most hard disks attached to current Linux systems correspond to device names with an *sd* prefix, such as */dev/sda*, */dev/sdb*, and so on. These devices represent entire disks; the kernel makes separate device files, such as */dev/sda1* and */dev/sda2*, for the partitions on a disk.

The naming convention requires a little explanation. The *sd* portion of the name stands for *SCSI disk*. *Small Computer System Interface (SCSI)* was originally developed as a hardware and protocol standard for communication between devices such as disks and other peripherals. Although traditional SCSI hardware isn't used in most modern machines, the SCSI protocol is everywhere due to its adaptability. For example, USB storage devices use it to communicate. The story on SATA disks is a little more complicated, but the Linux kernel still uses SCSI commands at a certain point when talking to them.

To list the SCSI devices on your system, use a utility that walks the device paths provided by sysfs. One of the most succinct tools is `lsscsi`. Here is what you can expect when you run it:

```
$ lsscsi

[0:0:0:0]❶    disk❷    ATA      WDC WD3200AAJS-2 01.0  /dev/sda❸

[1:0:0:0]     cd/dvd   Slimtype DVD A  DS8A5SH   XA15  /dev/sr0

[2:0:0:0]     disk     FLASH    Drive UT_USB20   0.00  /dev/sdb
```

The first column ❶ identifies the address of the device on the system, the second ❷ describes what kind of device it is, and the last ❸ indicates where to find the device file. Everything else is vendor information.

Linux assigns devices to device files in the order in which its drivers encounter devices. So in the previous example, the kernel found the disk first, the optical drive second, and the flash drive last.

Unfortunately, this device assignment scheme has traditionally caused problems when reconfiguring hardware. Say, for example, that you have a system with three disks: */dev/sda*, */dev/sdb*, and */dev/sdc*. If */dev/sdb* explodes and you must remove the disk so that the machine can work again, the former */dev/sdc* moves to */dev/sdb*, and there is no longer a */dev/sdc*. If you were referring to the device names directly in the *fstab* file (see 4.2.8 The /etc/fstab Filesystem Table), you'd have to make some changes to that file in order to get things (mostly) back to normal. To solve this problem, most modern Linux systems use the Universally Unique Identifier (UUID, see 4.2.4 Filesystem UUID) for persistent disk device access.

This discussion has barely scratched the surface of how to use disks and other storage devices on Linux systems. See Chapter 4 for more information about using disks. Later in this chapter, we'll examine how SCSI support works in the Linux kernel.

3.4.2 CD and DVD Drives: /dev/sr*

Linux recognizes most optical storage drives as the SCSI devices */dev/sr0*, */dev/sr1*, and so on. However, if the drive uses an older interface, it might show up as a PATA device, as discussed below. The */dev/sr** devices are read only, and they are used only for reading from discs. For the write and rewrite capabilities of optical devices, you'll use the "generic" SCSI devices such as */dev/sg0*.

3.4.3 PATA Hard Disks: /dev/hd*

The Linux block devices */dev/hda*, */dev/hdb*, */dev/hdc*, and */dev/hdd* are common on older versions of the Linux kernel and with older hardware. These are fixed assignments based on the master and slave devices on interfaces 0 and 1. At times, you might find a SATA drive recognized as one of these disks. This means that the SATA drive is running in a compatibility mode, which hinders performance. Check your BIOS settings to see if you can switch the SATA controller to its native mode.

3.4.4 Terminals: /dev/tty*, /dev/pts/*, and /dev/tty

Terminals are devices for moving characters between a user process and an I/O device, usually for text output to a terminal screen. The terminal device interface goes back a long way, to the days when terminals were typewriter-based devices.

Pseudoterminal devices are emulated terminals that understand the I/O features of real terminals. But rather than talk to a real piece of hardware, the kernel presents the I/O interface to a piece of software, such as the shell terminal window that you probably type most of your commands into.

Two common terminal devices are */dev/tty1* (the first virtual console) and */dev/pts/0* (the first pseudoterminal device). The */dev/pts* directory itself is a dedicated filesystem.

The */dev/tty* device is the controlling terminal of the current process. If a program is currently reading from and writing to a terminal, this device is a synonym for that terminal. A process does not need to be attached to

a terminal.

Display Modes and Virtual Consoles

Linux has two primary display modes: *text mode* and an *X Window System server* (graphics mode, usually via a display manager). Although Linux systems traditionally booted in text mode, most distributions now use kernel parameters and interim graphical display mechanisms (bootsplashes such as plymouth) to completely hide text mode as the system is booting. In such cases, the system switches over to full graphics mode near the end of the boot process.

Linux supports *virtual consoles* to multiplex the display. Each virtual console may run in graphics or text mode. When in text mode, you can switch between consoles with an ALT-Function key combination—for example, ALT-F1 takes you to */dev/tty1*, ALT-F2 goes to */dev/tty2*, and so on. Many of these may be occupied by a `getty` process running a login prompt, as described in 7.4 getty and login.

A virtual console used by the X server in graphics mode is slightly different. Rather than getting a virtual console assignment from the init configuration, an X server takes over a free virtual console unless directed to use a specific virtual console. For example, if you have `getty` processes running on *tty1* and *tty2*, a new X server takes over *tty3*. In addition, after the X server puts a virtual console into graphics mode, you must normally press a CTRL-ALT-Function key combination to switch to another virtual console instead of the simpler ALT-Function key combination.

The upshot of all of this is that if you want to see your text console after your system boots, press CTRL-ALT-F1. To return to the X11 session, press ALT-F2, ALT-F3, and so on, until you get to the X session.

If you run into trouble switching consoles due to a malfunctioning input mechanism or some other circumstance, you can try to force the system to change consoles with the `chvt` command. For example, to switch to *tty1*, run the following as root:

```
# chvt 1
```

3.4.5 Serial Ports: /dev/ttyS*

Older RS-232 type and similar serial ports are special terminal devices. You can't do much on the command line with serial port devices because there are too many settings to worry about, such as baud rate and flow control.

The port known as COM1 on Windows is */dev/ttyS0*; COM2 is */dev/ttyS1*; and so on. Plug-in USB serial adapters show up with *USB* and *ACM* with the names */dev/ttyUSB0*, */dev/ttyACM0*, */dev/ttyUSB1*, */dev/ttyACM1*, and so on.

3.4.6 Parallel Ports: /dev/lp0 and /dev/lp1

Representing an interface type that has largely been replaced by USB, the unidirectional parallel port devices */dev/lp0* and */dev/lp1* correspond to LPT1: and LPT2: in Windows. You can send files (such as a file to be printed) directly to a parallel port with the `cat` command, but you might need to give the printer an extra form feed or reset afterward. A print server such as CUPS is much better at handling interaction with a printer.

The bidirectional parallel ports are */dev/parport0* and */dev/parport1*.

3.4.7 Audio Devices: /dev/snd/*, /dev/dsp, /dev/audio, and More

Linux has two sets of audio devices. There are separate devices for the *Advanced Linux Sound Architecture (ALSA)* system interface and the older *Open Sound System (OSS)*. The ALSA devices are in the */dev/snd* directory, but it's difficult to work with them directly. Linux systems that use ALSA support OSS backward-compatible devices if the OSS kernel support is currently loaded.

Some rudimentary operations are possible with the OSS *dsp* and *audio* devices. For example, the computer

plays any WAV file that you send to */dev/dsp*. However, the hardware may not do what you expect due to frequency mismatches. Furthermore, on most systems, the device is often busy as soon as you log in.

NOTE

Linux sound is a messy subject due to the many layers involved. We've just talked about the kernel-level devices, but typically there are user-space servers such as pulse-audio that manage audio from different sources and act as intermediaries between the sound devices and other user-space processes.

3.4.8 Creating Device Files

In modern Linux systems, you do not create your own device files; this is done with devtmpfs and udev (see 3.5 udev). However, it is instructive to see how it was once done, and on a rare occasion, you might need to create a named pipe.

The `mknod` command creates one device. You must know the device name as well as its major and minor numbers. For example, creating */dev/sda1* is a matter of using the following command:

```
# mknod /dev/sda1 b 8 2
```

The `b 8 2` specifies a block device with a major number 8 and a minor number 2. For character or named pipe devices, use `c` or `p` instead of `b` (omit the major and minor numbers for named pipes).

As mentioned earlier, the `mknod` command is useful only for creating the occasional named pipe. At one time, it was also sometimes useful for creating missing devices in single-user mode during system recovery.

In older versions of Unix and Linux, maintaining the */dev* directory was a challenge. With every significant kernel upgrade or driver addition, the kernel could support more kinds of devices, meaning that there would be a new set of major and minor numbers to be assigned to device filenames. Maintaining this was difficult, so each system had a `MAKEDEV` program in */dev* to create groups of devices. When you upgraded your system, you would try to find an update to `MAKEDEV` and then run it in order to create new devices.

This static system became ungainly, so a replacement was in order. The first attempt to fix it was devfs, a kernel-space implementation of */dev* that contained all of the devices that the current kernel supported. However, there were a number of limitations, which led to the development of udev and devtmpfs.

3.5 udev

We've already talked about how unnecessary complexity in the kernel is dangerous because you can too easily introduce system instability. Device file management is an example: You can create device files in user space, so why would you do this in the kernel? The Linux kernel can send notifications to a user-space process (called `udevd`) upon detecting a new device on the system (for example, when someone attaches a USB flash drive). The user-space process on the other end examines the new device's characteristics, creates a device file, and then performs any device initialization.

That was the theory. Unfortunately, in practice, there is a problem with this approach—device files are necessary early in the boot procedure, so `udevd` must start early. To create device files, `udevd` could not depend on any devices that it was supposed to create, and it would need to perform its initial startup very quickly so that the rest of the system wouldn't get held up waiting for `udevd` to start.

3.5.1 devtmpfs

The devtmpfs filesystem was developed in response to the problem of device availability during boot (see 4.2 Filesystems for more details on filesystems). This filesystem is similar to the older devfs support, but it's simplified. The kernel creates device files as necessary, but it also notifies `udevd` that a new device is

available. Upon receiving this signal, `udevd` does not create the device files, but it does perform device initialization and process notification. Additionally, it creates a number of symbolic links in */dev* to further identify devices. You can find examples in the directory */dev/disk/by-id*, where each attached disk has one or more entries.

For example, consider this typical disk:

```
lrwxrwxrwx 1 root root 9 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-
WMAV2FU80671 -> ../../sda

lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-
WMAV2FU80671-part1 ->

  ../../sda1

lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-
WMAV2FU80671-part2 ->

  ../../sda2

lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-
WMAV2FU80671-part5 ->

  ../../sda5
```

`udevd` names the links by interface type, and then by manufacturer and model information, serial number, and partition (if applicable).

But how does `udevd` know which symbolic links to create, and how does it create them? The next section describes how `udevd` does its work. However, you don't need to know that to continue on with the book. In fact, if this is your first time looking at Linux devices, you're encouraged to move to the next chapter to start learning about how to use disks.

3.5.2 udevd Operation and Configuration

The `udevd` daemon operates as follows:

1. The kernel sends `udevd` a notification event, called a *uevent*, through an internal network link.
2. `udevd` loads all of the attributes in the uevent.
3. `udevd` parses its rules, and it takes actions or sets more attributes based on those rules.

An incoming uevent that `udevd` receives from the kernel might look like this:

```
ACTION=change

DEVNAME=sde

DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.2/1-
1.2:1.0/host4/

  target4:0:0/4:0:0:3/block/sde

DEVTYPE=disk

DISK_MEDIA_CHANGE=1

MAJOR=8

MINOR=64
```

```
SEQNUM=2752

SUBSYSTEM=block

UDEV_LOG=3
```

You can see here that there is a change to a device. After receiving the uevent, `udevd` knows the sysfs device path and a number of other attributes associated with the properties, and it is now ready to start processing rules.

The rules files are in the */lib/udev/rules.d* and */etc/udev/rules.d* directories. The rules in */lib* are the defaults, and the rules in */etc* are overrides. A full explanation of the rules would be tedious, and you can learn much more from the udev(7) manual page, but let's look at the symbolic links from the */dev/sda* example in 3.5.1 devtmpfs. Those links were defined by rules in */lib/udev/rules.d/60-persistent-storage.rules*. Inside, you'll see the following lines:

```
# ATA devices using the "scsi" subsystem

KERNEL=="sd*[!0-9]|sr*",   ENV{ID_SERIAL}!="?*",   SUBSYSTEMS=="scsi",
ATTRS{vendor}=="ATA",

  IMPORT{program}="ata_id --export $tempnode"

# ATA/ATAPI devices (SPC-3 or later) using the "scsi" subsystem

KERNEL=="sd*[!0-9]|sr*", ENV{ID_SERIAL}!="?*", SUBSYSTEMS=="scsi",

  ATTRS{type}=="5",                          ATTRS{scsi_level}=="[6-9]*",
IMPORT{program}="ata_id --export $tempnode"
```

These rules match ATA disks presented through the kernel's SCSI subsystem (see 3.6 In-Depth: SCSI and the Linux Kernel). You can see that there are a few rules to catch different ways that the devices may be represented, but the idea is that `udevd` will try to match a device starting with `sd` or `sr` but without a number (with the `KERNEL=="sd*[!0-9]|sr*"` expression), as well as a subsystem (`SUBSYSTEMS=="scsi"`), and finally, some other attributes. If all of those conditional expressions are true, `udevd` moves to the next expression:

```
IMPORT{program}="ata_id --export $tempnode"
```

This is not a conditional, but rather, a directive to import variables from the */lib/udev/ata_id* command. If you have such a disk, try it yourself on the command line:

```
$ sudo /lib/udev/ata_id --export /dev/sda

ID_ATA=1

ID_TYPE=disk

ID_BUS=ata

ID_MODEL=WDC_WD3200AAJS-22L7A0

ID_MODEL_ENC=WDC\x20WD3200AAJS22L7A0\x20\x20\x20\x20\x20\x20\x20\x20\x
20\x20

  \x20\x20\x20\x20\x20\x20\x20\x20

ID_REVISION=01.03E10

ID_SERIAL=WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
```

```
--snip--
```

The import now sets the environment so that all of the variable names in this output are set to the values shown. For example, any rule that follows will now recognize `ENV{ID_TYPE}` as disk.

Of particular note is `ID_SERIAL`. In each of the rules, this conditional appears second:

```
ENV{ID_SERIAL}!="?*"
```

This means that *ID_SERIAL* is true only if is not set. Therefore, if it *is* set, the conditional is false, the entire current rule is false, and `udevd` moves to the next rule.

So what's the point? The object of these two rules (and many around them in the file) is to find the serial number of the disk device. With `ENV{ID_SERIAL}` set, `udevd` can now evaluate this rule:

```
KERNEL=="sd*|sr*|cciss*", ENV{DEVTYPE}=="disk", ENV{ID_SERIAL}=="?*",

  SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}"
```

You can see that this rule requires `ENV{ID_SERIAL}` to be set, and it has one directive:

```
SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}"
```

Upon encountering this directive, `udevd` adds a symbolic link for the incoming device. So now you know where the device symbolic links came from!

You may be wondering how to tell a conditional expression from a directive. Conditionals are denoted by two equal signs (==) or a bang equal (!=), and directives by a single equal sign (=), a plus equal (+=), or a colon equal (:=).

3.5.3 udevadm

The `udevadm` program is an administration tool for `udevd`. You can reload `udevd` rules and trigger events, but perhaps the most powerful features of `udevadm` are the ability to search for and explore system devices and the ability to monitor uevents as `udevd` receives them from the kernel. The only trick is that the command syntax can get a bit involved.

Let's start by examining a system device. Returning to the example in 3.5.2 udevd Operation and Configuration, in order to look at all of the udev attributes used and generated in conjunction with the rules for a device such as */dev/sda*, run the following command:

```
$ udevadm info --query=all --name=/dev/sda
```

The output looks like this:

```
P:
/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda

N: sda

S: disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671

S: disk/by-id/scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671

S: disk/by-id/wwn-0x50014ee057faef84 S: disk/by-path/pci-0000:00:1f.2-
scsi-0:0:0:0

E:   DEVLINKS=/dev/disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
/dev/disk/by-id/scsi

  -SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671              /dev/disk/by-id/wwn-
```

```
0x50014ee057faef84 /dev/disk/by
  -path/pci-0000:00:1f.2-scsi-0:0:0:0
E: DEVNAME=/dev/sda
E:
DEVPATH=/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/blo
ck/sda
E: DEVTYPE=disk
E: ID_ATA=1
E: ID_ATA_DOWNLOAD_MICROCODE=1
E: ID_ATA_FEATURE_SET_AAM=1
--snip--
```

The prefix in each line indicates an attribute or other characteristic of the device. In this case, the `P:` at the top is the sysfs device path, the `N:` is the device node (that is, the name given to the */dev* file), `S:` indicates a symbolic link to the device node that `udevd` placed in */dev* according to its rules, and `E:` is additional device information extracted in the `udevd` rules. (There was far more output in this example than was necessary to show here; try the command for yourself to get a feel for what it does.)

3.5.4 Monitoring Devices

To monitor uevents with `udevadm`, use the `monitor` command:

```
$ udevadm monitor
```

Output (for example, when you insert a flash media device) looks like this abbreviated sample:

```
KERNEL[658299.569485]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2 (usb)
KERNEL[658299.569667]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2/2-1.2:1.0 (usb)
KERNEL[658299.570614]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2/2-1.2:1.0/host15
  (scsi)
KERNEL[658299.570645]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2/2-1.2:1.0/
  host15/scsi_host/host15 (scsi_host)
UDEV  [658299.622579]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2 (usb)
UDEV  [658299.623014]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2/2-1.2:1.0 (usb)
UDEV  [658299.623673]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2/2-1.2:1.0/host15
  (scsi)
```

```
UDEV  [658299.623690]  add    /devices/pci0000:00/0000:00:1d.0/usb2/2-
1/2-1.2/2-1.2:1.0/

  host15/scsi_host/host15 (scsi_host)

--snip--
```

There are two copies of each message in this output because the default behavior is to print both the incoming message from the kernel (marked with KERNEL) and the message that `udevd` sends out to other programs when it's finished processing and filtering the event. To see only kernel events, add the `--kernel` option, and to see only outgoing events, use `--udev`. To see the whole incoming uevent, including the attributes as shown in 3.5.2 udevd Operation and Configuration, use the `--property` option.

You can also filter events by subsystem. For example, to see only kernel messages pertaining to changes in the SCSI subsystem, use this command:

```
$ udevadm monitor --kernel --subsystem-match=scsi
```

For more on `udevadm`, see the udevadm(8) manual page.

There's much more to udev. For example, the D-Bus system for interprocess communication has a daemon called `udisks-daemon` that listens to the outgoing `udevd` events in order to automatically attach disks and to further notify other desktop software that a new disk is now available.

3.6 In-Depth: SCSI and the Linux Kernel

In this section, we'll take a look at the SCSI support in the Linux kernel as a way to explore part of the Linux kernel architecture. You don't need to know any of this information in order to use disks, so if you're in a hurry to use one, move on to Chapter 4. In addition, the material here is more advanced and theoretical in nature that what you've seen so far, so if you want to stay hands-on, you should definitely skip to the next chapter.

Let's begin with a little background. The traditional SCSI hardware setup is a host adapter linked with a chain of devices over an SCSI bus, as shown in Figure 3-1. The host adapter is attached to a computer. The host adapter and devices each have an SCSI ID, and there can be 8 or 16 IDs per bus, depending on the SCSI version. You might hear the term *SCSI target* used to refer to a device and its SCSI ID.

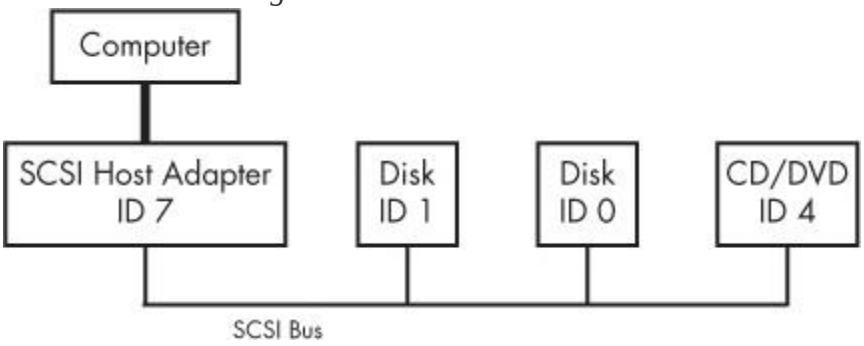

Figure 3-1. SCSI Bus with host adapter and devices

The host adapter communicates with the devices through the SCSI command set in a peer-to-peer relationship; the devices send responses back to the host adapter. The computer is not directly attached to the device chain, so it must go through the host adapter in order to communicate with disks and other devices. Typically, the computer sends SCSI commands to the host adapter to relay to the devices, and the devices relay responses back through the host adapter.

Newer versions of SCSI, such as *Serial Attached SCSI (SAS)*, offer exceptional performance, but you probably won't find true SCSI devices in most machines. You'll more often encounter USB storage devices that use

SCSI commands. In addition, devices supporting ATAPI (such as CD/DVD-ROM drives) use a version of the SCSI command set.

SATA disks also appear on your system as SCSI devices by means of a translation layer in libata (see 3.6.2 SCSI and ATA). Some SATA controllers (especially high-performance RAID controllers) perform this translation in hardware.

How does this all fit together? Consider the devices shown on the following system:

```
$ lsscsi
[0:0:0:0]   disk    ATA      WDC WD3200AAJS-2 01.0  /dev/sda
[1:0:0:0]   cd/dvd  Slimtype DVD A  DS8A5SH   XA15  /dev/sr0
[2:0:0:0]   disk    USB2.0   CardReader CF    0100  /dev/sdb
[2:0:0:1]   disk    USB2.0   CardReader SM XD 0100  /dev/sdc
[2:0:0:2]   disk    USB2.0   CardReader MS    0100  /dev/sdd
[2:0:0:3]   disk    USB2.0   CardReader SD    0100  /dev/sde
[3:0:0:0]   disk    FLASH    Drive UT_USB20   0.00  /dev/sdf
```

The numbers in brackets are, from left to right, the SCSI host adapter number, the SCSI bus number, the device SCSI ID, and the LUN (logical unit number, a further subdivision of a device). In this example, there are four attached adapters (scsi0, scsi1, scsi2, and scsi3), each of which has a single bus (all with bus number 0), and just one device on each bus (all with target 0). The USB card reader at 2:0:0 has four logical units, though—one for each kind of flash card that can be inserted. The kernel has assigned a different device file to each logical unit.

Figure 3-2 illustrates the driver and interface hierarchy inside the kernel for this particular system configuration, from the individual device drivers up to the block drivers. It does not include the SCSI generic (sg) drivers.

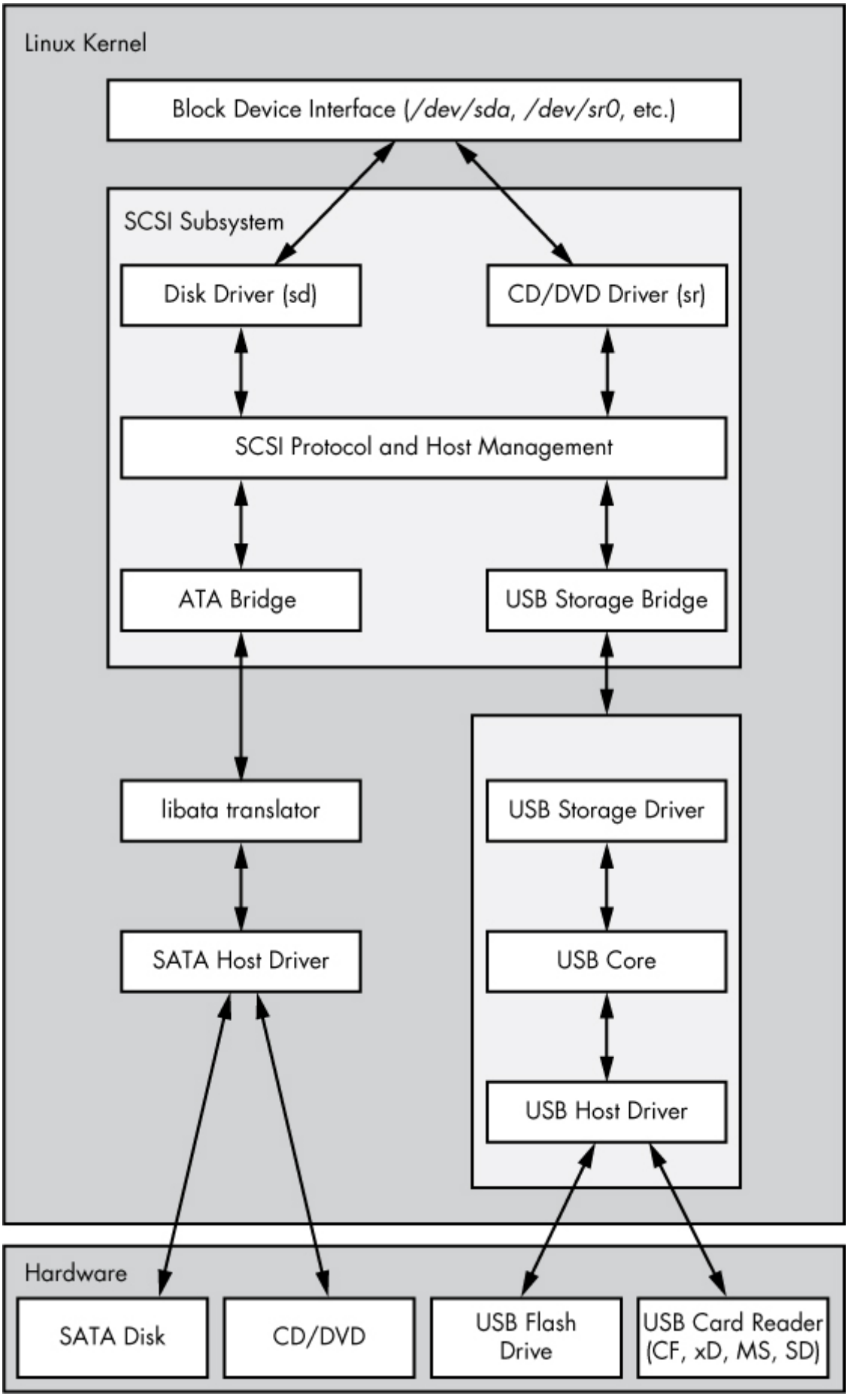

Figure 3-2. Linux SCSI subsystem schematic

Although this is a large structure and may look overwhelming at first, the data flow in the figure is very linear. Let's begin dissecting it by looking at the SCSI subsystem and its three layers of drivers:

- The top layer handles operations for a class of device. For example, the sd (SCSI disk) driver is at this layer; it knows how to translate requests from the kernel block device interface into disk-specific commands in the SCSI protocol, and vice versa.
- The middle layer moderates and routes the SCSI messages between the top and bottom layers, and keeps track of all of the SCSI buses and devices attached to the system.
- The bottom layer handles hardware-specific actions. The drivers here send outgoing SCSI protocol messages to specific host adapters or hardware, and they extract incoming messages from the hardware. The reason for this separation from the top layer is that although SCSI messages are uniform for a device class (such as the disk class), different kinds of host adapters have varying procedures for sending the same messages.

The top and bottom layers contain many different drivers, but it's important to remember that, for any given device file on your system, the kernel uses one top-layer driver and one lower-layer driver. For the disk at */dev/sda* in our example, the kernel uses the sd top-layer driver and the ATA bridge lower-layer driver.

There are times when you might use more than one upper-layer driver for one hardware device (see 3.6.3 Generic SCSI Devices). For true hardware SCSI devices, such as a disk attached to an SCSI host adapter or a hardware RAID controller, the lower-layer drivers talk directly to the hardware below. However, for most hardware that you find attached to the SCSI subsystem, it's a different story.

3.6.1 USB Storage and SCSI

In order for the SCSI subsystem to talk to common USB storage hardware, as shown in Figure 3-2, the kernel needs more than just a lower-layer SCSI driver. The USB flash drive represented by */dev/sdf* understands SCSI commands, but to actually communicate with the drive, the kernel needs to know how to talk through the USB system.

In the abstract, USB is quite similar to SCSI—it has device classes, buses, and host controllers. Therefore, it should be no surprise that the Linux kernel includes a three-layer USB subsystem that closely resembles the SCSI subsystem, with device-class drivers at the top, a bus management core in the middle, and host controller drivers at the bottom. Much as the SCSI subsystem passes SCSI commands between its components, the USB subsystem passes USB messages between its components. There's even an `lsusb` command that is similar to `lsscsi`.

The part we're really interested in here is the USB storage driver at the top. This driver acts as a translator. On one side, the driver speaks SCSI, and on the other, it speaks USB. Because the storage hardware includes SCSI commands inside its USB messages, the driver has a relatively easy job: It mostly repackages data.

With both the SCSI and USB subsystems in place, you have almost everything you need to talk to the flash drive. The final missing link is the lower-layer driver in the SCSI subsystem because the USB storage driver is a part of the USB subsystem, not the SCSI subsystem. (For organizational reasons, the two subsystems should not share a driver.) To get the subsystems to talk to one another, a simple, lower-layer SCSI bridge driver connects to the USB subsystem's storage driver.

3.6.2 SCSI and ATA

The SATA hard disk and optical drive shown in Figure 3-2 both use the same SATA interface. To connect the SATA-specific drivers of the kernel to the SCSI subsystem, the kernel employs a bridge driver, as with the USB drives, but with a different mechanism and additional complications. The optical drive speaks ATAPI, a version of SCSI commands encoded in the ATA protocol. However, the hard disk does not use ATAPI and does not encode any SCSI commands!

The Linux kernel uses part of a library called libata to reconcile SATA (and ATA) drives with the SCSI subsystem. For the ATAPI-speaking optical drives, this is a relatively simple task of packaging and extracting

SCSI commands into and from the ATA protocol. But for the hard disk, the task is much more complicated because the library must do a full command translation.

The job of the optical drive is similar to typing an English book into a computer. You don't need to understand what the book is about in order to do this job, nor do you even need to understand English. But the task for the hard disk is more like reading a German book and typing it into the computer as an English translation. In this case, you need to understand both languages as well as the book's content.

Despite this difficulty, libata performs this task and makes it possible to attach the SCSI subsystem to ATA/SATA interfaces and devices. (There are typically more drivers involved than just the one SATA host driver shown in Figure 3-2, but they're not shown for the sake of simplicity.)

3.6.3 Generic SCSI Devices

When a user-space process communicates with the SCSI subsystem, it normally does so through the block device layer and/or another other kernel service that sits on top of an SCSI device class driver (like *sd* or *sr*). In other words, most user processes never need to know anything about SCSI devices or their commands.

However, user processes can bypass device class drivers and give SCSI protocol commands directly to devices through their *generic devices*. For example, consider the system described in 3.6 In-Depth: SCSI and the Linux Kernel, but this time, take a look at what happens when you add the `-g` option to `lsscsi` in order to show the generic devices:

```
$ lsscsi -g

[0:0:0:0]    disk    ATA      WDC WD3200AAJS-2 01.0  /dev/sda ❶/dev/sg0

[1:0:0:0]    cd/dvd  Slimtype DVD A  DS8A5SH   XA15  /dev/sr0 /dev/sg1
[2:0:0:0]    disk    USB2.0   CardReader CF    0100  /dev/sdb /dev/sg2
[2:0:0:1]    disk    USB2.0   CardReader SM XD 0100  /dev/sdc /dev/sg3
[2:0:0:2]    disk    USB2.0   CardReader MS    0100  /dev/sdd /dev/sg4
[2:0:0:3]    disk    USB2.0   CardReader SD    0100  /dev/sde /dev/sg5
[3:0:0:0]    disk    FLASH    Drive UT_USB20   0.00  /dev/sdf /dev/sg6
```

In addition to the usual block device file, each entry lists an SCSI generic device file in the last column at ❶. For example, the generic device for the optical drive at */dev/sr0* is */dev/sg1*.

Why would you want to use an SCSI generic device? The answer has to do with the complexity of code in the kernel. As tasks get more complicated, it's better to leave them out of the kernel. Consider CD/DVD writing and reading. Not only is writing significantly more difficult than reading, but no critical system services depend on the action of writing. A user-space program might do the writing a little more inefficiently than a kernel service, but that program will be far easier to build and maintain than a kernel service, and bugs will not threaten kernel space. Therefore, to write to an optical disc in Linux, you run a program that talks to a generic SCSI device, such as */dev/sg1*. Due to the relative simplicity of reading compared to writing, however, you still read from the device using the specialized *sr* optical device driver in the kernel.

3.6.4 Multiple Access Methods for a Single Device

The two points of access (*sr* and *sg*) for an optical drive from user space are illustrated for the Linux SCSI subsystem in Figure 3-3 (any drivers below the SCSI lower layer have been omitted). Process A reads from the drive using the *sr* driver, and process B writes to the drive with the *sg* driver. However, processes such as these two would not normally run simultaneously to access the same device.