6 – Redirection

In this lesson we are going to unleash what may be the coolest feature of the command line. It's called *I/O redirection*. The "I/O" stands for *input/output* and with this facility we can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- `cat` – Concatenate files
- `sort` – Sort lines of text
- `uniq` – Report or omit repeated lines
- `grep` – Print lines matching a pattern
- `wc` – Print newline, word, and byte counts for each file
- `head` – Output the first part of a file
- `tail` – Output the last part of a file
- `tee` – Read from standard input and write to standard output and files

Standard Input, Output, and Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types:

- The program's results, that is, the data the program is designed to produce
- Status and error messages that tell us how the program is getting along

If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of "everything is a file," programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called s*tandard error* (*stderr*). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input* (*stdin*), which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file instead of the screen, we use the `>` redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file `ls-output.txt` instead of the screen:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here, we created a long listing of the `/usr/bin` directory and sent the results to the file `ls-output.txt`. Let's examine the redirected output of the command, shown here:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   167878 2018-02-01 15:07 ls-output.txt
```

Good — a nice, large, text file. If we look at the file with `less`, we will see that the file `ls-output.txt` does indeed contain the results from our `ls` command.

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test, but this time with a twist. We'll change the name of the directory to one that does not exist:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. This makes sense since we specified the nonexistent directory `/bin/usr`, but why was the error message displayed on the screen rather than being redirected to the file `ls-output.txt`? The answer is that the `ls` program does not send its error messages to standard output. Instead, like most well-written Unix programs,

it sends its error messages to standard error. Since we only redirected standard output and not standard error, the error message was still sent to the screen. We'll see how to redirect standard error in just a minute, but first let's look at what happened to our output file:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   0 2018-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because when we redirect output with the “>” redirection operator, the destination file is always rewritten from the beginning. Since our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file), we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the `>>` redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Using the `>>` operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the `>` operator had been used. Let's put it to the test:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   503634 2018-02-01 15:45 ls-output.txt
```

We repeated the command three times resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of a dedicated redirection operator. To redirect standard error we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output and error, the shell references them internally as file descriptors 0, 1, and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor "2" is placed immediately before the redirection operator to perform the redirection of standard error to the file `ls-error.txt`.

Redirecting Standard Output and Standard Error to One File

There are cases in which we may want to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. Shown here is the traditional way, which works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Using this method, we perform two redirections. First we redirect standard output to the file `ls-output.txt` and then we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation `2>&1`.

Notice that the order of the redirections is significant. The redirection of standard error must always occur *after* redirecting standard output or it doesn't work. The following example redirects standard error to the file `ls-output.txt`:

```
>ls-output.txt 2>&1
```

However, if the order is changed to the following, standard error is directed to the screen.

```
2>&1 >ls-output.txt
```

Recent versions of `bash` provide a second, more streamlined method for performing this combined redirection shown here:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

In this example, we use the single notation `&>` to redirect both standard output and standard error to the file `ls-output.txt`. We can also append the standard output and standard error streams to a single file like so:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

Disposing of Unwanted Output

Sometimes “silence is golden,” and we don't want output from a command, we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called “/dev/null”. This file is a system device often referred to as a *bit bucket*, which accepts input and does nothing with it. To suppress error messages from a command, we do this:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

`/dev/null` In Unix Culture

The bit bucket is an ancient Unix concept and because of its universality, it has appeared in many parts of Unix culture. When someone says he/she is sending your comments to `/dev/null`, now you know what it means. For more examples, see the Wikipedia article on `/dev/null`.

Redirecting Standard Input

Up to now, we haven't encountered any commands that make use of standard input (actually we have, but we’ll reveal that surprise a little bit later), so we need to introduce one.

`cat` – Concatenate Files

The `cat` command reads one or more files and copies them to standard output like so:

```
cat [file...]
```

In most cases, we can think of `cat` as being analogous to the `TYPE` command in DOS. We can use it to display files without paging. For example, the following will display the contents of the file `ls-output.txt`:

```
[me@linuxbox ~]$ cat ls-output.txt
```

`cat` is often used to display short text files. Since `cat` can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on Usenet), and we want to join them back together. If the files were named:

`movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099`

we could join them back together with this command as follows:

```
cat movie.mpeg.0* > movie.mpeg
```

Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we enter `cat` with no arguments?

```
[me@linuxbox ~]$ cat
```

Nothing happens, it just sits there like it's hung. It might seem that way, but it's really doing exactly what it's supposed to do.

If `cat` is not given any arguments, it reads from standard input and since standard input is, by default, attached to the keyboard, it's waiting for us to type something! Try adding the following text and pressing Enter:

```
[me@linuxbox ~]$ cat
```

```
The quick brown fox jumped over the lazy dog.
```

Next, type a `Ctrl-d` (i.e., hold down the Ctrl key and press "d") to tell `cat` that it has reached *end of file* (EOF) on standard input:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

In the absence of filename arguments, `cat` copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Let's say we wanted to create a file called `lazy_dog.txt` containing the text in our example. We would do this:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Type the command followed by the text we want to place in the file. Remember to type `Ctrl-d` at the end. Using the command line, we have implemented the world's dumbest word processor! To see our results, we can use `cat` to copy the file to stdout again.

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Now that we know how `cat` accepts standard input, in addition to filename arguments, let's try redirecting standard input.

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Using the `<` redirection operator, we change the source of standard input from the keyboard to the file `lazy_dog.txt`. We see that the result is the same as passing a single filename argument. This is not particularly useful compared to passing a filename argument, but it serves to demonstrate using a file as a source of standard input. Other commands make better use of standard input, as we will soon see.

Before we move on, check out the man page for `cat`, because it has several interesting

options.

Pipelines

The capability of commands to read data from standard input and send to standard output is utilized by a shell feature called *pipelines*. Using the pipe operator | (vertical bar), the standard output of one command can be *piped* into the standard input of another.

```
command1 | command2
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's `less`. We can use `less` to display, page by page, the output of any command that sends its results to standard output:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

The Difference Between > and |

At first glance, it may be hard to understand the redirection performed by the pipeline operator | versus the redirection operator >. Simply put, the redirection operator connects a command with a file, while the pipeline operator connects the output of one command with the input of a second command.

```
command1 > file1
```

```
command1 | command2
```

A lot of people will try the following when they are learning about pipelines, “just to see what happens”:

```
command1 > command2
```

Answer: sometimes something really bad.

Here is an actual example submitted by a reader who was administering a Linux-based server appliance. As the superuser, he did this:

```
# cd /usr/bin
# ls > less
```

The first command put him in the directory where most programs are stored and the second command told the shell to overwrite the file `less` with the output of the `ls` command. Since the `/usr/bin` directory already contained a file named `less` (the `less` program), the second command overwrote the `less` program file with the text from `ls`, thus destroying the `less` program on his system.

The lesson here is that the redirection operator silently creates or overwrites files, so you need to treat it with a lot of respect.

Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters*. Filters take input, change it somehow, and then output it. The first one we will try is `sort`. Imagine we wanted to make a combined list of all the executable programs in `/bin` and `/usr/bin`, put them in sorted order and view the resulting list:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Since we specified two directories (`/bin` and `/usr/bin`), the output of `ls` would have consisted of two sorted lists, one for each directory. By including `sort` in our pipeline, we changed the data to produce a single, sorted list.

`uniq` - Report or Omit Repeated Lines

The `uniq` command is often used in conjunction with `sort`. `uniq` accepts a sorted list of data from either standard input or a single filename argument (see the `uniq` man page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the `/bin` and `/usr/bin` directories), we will add `uniq` to our pipeline.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

In this example, we use `uniq` to remove any duplicates from the output of the `sort` command. If we want to see the list of duplicates instead, we add the "-d" option to `uniq` like so:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc – Print Line, Word, and Byte Counts

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. Here's an example:

```
[me@linuxbox ~]$ wc ls-output.txt
 7902  64566 503634 ls-output.txt
```

In this case, it prints out three numbers: lines, words, and bytes contained in `ls-output.txt`. Like our previous commands, if executed without command line arguments, `wc` accepts standard input. The "-l" option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of items we have in our sorted list, we can do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep – Print Lines Matching a Pattern

`grep` is a powerful program used to find text patterns within files. It's used like this:

```
grep pattern [file...]
```

When `grep` encounters a "pattern" in the file, it prints out the lines containing it. The patterns that `grep` can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called *regular expressions* in Chapter 19.

Let's say we wanted to find all the files in our list of programs that had the word `zip` embedded in the name. Such a search might give us an idea of some of the programs on our system that had something to do with file compression. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for `grep`:

- `-i`, which causes `grep` to ignore case when performing the search (normally searches are case sensitive)
- `-v`, which tells `grep` to print only those lines that do not match the pattern.

`head` / `tail` – Print First / Last Part of Files

Sometimes we don't want all the output from a command. We may only want the first few lines or the last few lines. The `head` command prints the first ten lines of a file, and the `tail` command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the `-n` option.

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root       31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2007-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2007-11-26 14:27 a2p
-rwxr-xr-x 1 root root       25368 2006-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root        5234 2007-06-27 10:56 znew
-rwxr-xr-x 1 root root         691 2005-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root         930 2007-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root         930 2007-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root           6 2016-01-31 05:22 zsoelim -> soelim
```

These can be used in pipelines as well:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

`tail` has an option which allows us to view files in real time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the `messages` file in `/var/log` (or the `/var/log/syslog` file if `messages` is missing). Superuser privileges are required to do this on some Linux distributions, because the `/var/log/messages` file may contain security information:

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in
1652 seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to
both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1
port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in
1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART
Prefailure Attribute: 8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to
both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user
me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user
root by me(uid=500)
```

Using the "-f" option, `tail` continues to monitor the file, and when new lines are appended, they immediately appear on the display. This continues until we press `Ctrl-c`.

`tee` – Read from Stdin and Output to Stdout and Files

In keeping with our plumbing metaphor, Linux provides a command called `tee` which

creates a “tee” fitting on our pipe. The `tee` program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including `tee` to capture the entire directory listing to the file `ls.txt` before `grep` filters the pipeline's contents:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Summing Up

As always, check out the documentation of each of the commands we have covered in this chapter. We have seen only their most basic usage. They all have a number of interesting options. As we gain Linux experience, we will see that the redirection feature of the command line is extremely useful for solving specialized problems. There are many commands that make use of standard input and output, and almost all command line programs use standard error to display their informative messages.

Linux Is About Imagination

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on, and play with it. Pretty graphics, cute sounds. After a while, though, you get tired of the game that came with it, so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, “I want a game that does this!” only to be told that no such game exists because there is no “market

demand” for it. Then you say, “But I only need to change this one thing!” The person behind the counter says you can't change it. The games are all sealed up in their cartridges. You discover that your toy is limited to the games others have decided that you need.

Linux, on the other hand, is like the world's largest Erector Set. You open it, and it's just a huge collection of parts. There's a lot of steel struts, screws, nuts, gears, pulleys, motors, and a few suggestions on what to build. So, you start to play with it. You build one of the suggestions and then another. After a while you discover that you have your own ideas of what to make. You don't ever have to go back to the store, as you already have everything you need. The Erector Set takes on the shape of your imagination. It does what you want.

Your choice of toys is, of course, a personal thing, so which toy would you find more satisfying?

11 – The Environment

As we discussed earlier, the shell maintains a body of information during our shell session called the *environment*. Programs use data stored in the environment to determine facts about the system's configuration. While most programs use *configuration files* to store program settings, some programs also look for values stored in the environment to adjust their behavior. Knowing this, we can use the environment to customize our shell experience.

In this chapter, we will work with the following commands:

- `printenv` – Print part or all of the environment
- `set` – Set shell options
- `export` – Export environment to subsequently executed programs
- `alias` – Create an alias for a command

What is Stored in the Environment?

The shell stores two basic types of data in the environment; though, with `bash`, the types are largely indistinguishable. They are *environment variables* and *shell variables*. Shell variables are bits of data placed there by `bash`, and environment variables are everything else. In addition to variables, the shell stores some programmatic data, namely *aliases* and *shell functions*. We covered aliases in Chapter 5, "Working with Commands." and we will cover shell functions (which are related to shell scripting) in Part 4.

Examining The Environment

To see what is stored in the environment, we can use either the `set` builtin in `bash` or the `printenv` program. The `set` command will show both the shell and environment variables, while `printenv` will only display the latter. Since the list of environment contents will be fairly long, it is best to pipe the output of either command into `less`.

```
[me@linuxbox ~]$ printenv | less
```

Doing so, we should get that looks like this:

```
USER=me
PAGER=less
LSCOLORS=Gxfxcxdxbxegedabagacad
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/usr/local/games
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
JOB=dbus
PWD=/home/me
XMODIFIERS=@im=ibus
GNOME_KEYRING_PID=1850
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
 MASTER_HOST=linuxbox
IM_CONFIG_PHASE=1
COMPIZ_CONFIG_PROFILE=ubuntu
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
XDG_SEAT=seat0
HOME=/home/me
SHLVL=2
LANGUAGE=en_US
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LESS=-R
LOGNAME=me
COMPIZ_BIN_PATH=/usr/bin/
LC_CTYPE=en_US.UTF-8
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/
usr/share/
QT4_IM_MODULE=xim
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-IwaesmWaT0
LESSOPEN=| /usr/bin/lesspipe %s
INSTANCE=
```

What we see is a list of environment variables and their values. For example, we see a variable called `USER`, which contains the value `me`. The `printenv` command can also list the value of a specific variable.

```
[me@linuxbox ~]$ printenv USER
me
```

The `set` command, when used without options or arguments, will display both the shell and environment variables, as well as any defined shell functions. Unlike `printenv`, its output is courteously sorted in alphabetical order.

```
[me@linuxbox ~]$ set | less
```

It is also possible to view the contents of a variable using the `echo` command, like this:

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

One element of the environment that neither `set` nor `printenv` displays is aliases. To see them, enter the `alias` command without arguments.

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-
dot --show-tilde'
```

Some Interesting Variables

The environment contains quite a few variables, and though the environment will differ from the one presented here, we will likely see the variables listed in Table 11-1 in our environment.

Table 11-1: Environment Variables

Variable	Contents
`DISPLAY`	The name of the display if we are running a graphical environment. Usually this is ":0", meaning the first display generated by the X server.

`EDITOR`	The name of the program to be used for text editing.
`SHELL`	The name of the user's default shell program.
`HOME`	The pathname of your home directory.
`LANG`	Defines the character set and collation order of your language.
`OLDPWD`	The previous working directory.
`PAGER`	The name of the program to be used for paging output. This is often set to `/usr/bin/less`.
`PATH`	A colon-separated list of directories that are searched when we enter the name of a executable program.
`PS1`	This stands for "prompt string 1." This defines the contents of the shell prompt. As we will later see, this can be extensively customized.
`PWD`	The current working directory.
`TERM`	The name of your terminal type. Unix-like systems support many terminal protocols; this variable sets the protocol to be used with your terminal emulator.
`TZ`	Specifies your time zone. Most Unix-like systems maintain the computer's internal clock in *Coordinated Universal Time* (UTC) and then display the local time by applying an offset specified by this variable.
`USER`	Your username.

Don't worry if some of these values are missing. They vary by distribution.

How Is The Environment Established?

When we log on to the system, the `bash` program starts, and reads a series of configuration scripts called *startup files*, which define the default environment shared by all users. This is followed by more startup files in our home directory that define our personal environment. The exact sequence depends on the type of shell session being started. There are two kinds.

- **A login shell session** A login shell session is one in which we are prompted for our username and password. This happens when we start a virtual console session, for example.
- **A non-login shell session** A non-login shell session typically occurs when we

launch a terminal session in the GUI.

Login shells read one or more startup files as shown in Table 11-2.

Table 11-2: Startup Files for Login Shell Sessions

File	Contents
`/etc/profile`	A global configuration script that applies to all users.
`~/.bash_profile`	A user's personal startup file. This can be used to extend or override settings in the global configuration script.
`~/.bash_login`	If `~/.bash_profile` is not found, `bash` attempts to read this script.
`~/.profile`	If neither `~/.bash_profile` nor `~/.bash_login` is found, `bash` attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

Non-login shell sessions read the startup files listed in Table 11-3.

Table 11-3: Startup Files for Non-Login Shell Sessions

File	Contents
`/etc/bash.bashrc`	A global configuration script that applies to all users.
`~/.bashrc`	A user's personal startup file. It can be used to extend or override settings in the global configuration script.

In addition to reading the startup files in Table 11-3, non-login shells inherit the environment from their parent process, usually a login shell.

Take a look and see which of these startup files are installed. Remember—since most of the filenames listed above start with a period (meaning that they are hidden), we will need to use the "-a" option when using `ls`.

The `~/.bashrc` file is probably the most important startup file from the ordinary user's point of view, since it is almost always read. Non-login shells read it by default and most startup files for login shells are written in such a way as to read the `~/.bashrc` file as well.

What's in a Startup File?

If we take a look inside a typical `.bash_profile` (taken from a CentOS 6 system), it

looks something like this:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
```

Lines that begin with a "#" are *comments* and are not read by the shell. These are there for human readability. The first interesting thing occurs on the fourth line, with the following code:

```
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

This is called an *if compound command*, which we will cover fully when we get to shell scripting in Part 4, but for now, here is a translation:

```
If the file "~/.bashrc" exists, then
    read the "~/.bashrc" file.
```

We can see that this bit of code is how a login shell gets the contents of `.bashrc`. The next thing in our startup file has to do with the `PATH` variable.

Ever wonder how the shell knows where to find commands when we enter them on the command line? For example, when we enter `ls`, the shell does not search the entire computer to find `/bin/ls` (the full pathname of the `ls` command); rather, it searches a list of directories that are contained in the `PATH` variable.

The `PATH` variable is often (but not always, depending on the distribution) set by the `/etc/profile` startup file with this code:

```
PATH=$PATH:$HOME/bin
```

PATH is modified to add the directory $HOME/bin to the end of the list. This is an example of parameter expansion, which we touched on in Chapter 7. “Seeing the World As tthe Shell Sees It.” To demonstrate how this works, try the following:

```
[me@linuxbox ~]$ foo="This is some "
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo"text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

Using this technique, we can append text to the end of a variable's contents.

By adding the string $HOME/bin to the end of the PATH variable's contents, the directory $HOME/bin is added to the list of directories searched when a command is entered. This means that when we want to create a directory within our home directory for storing our own private programs, the shell is ready to accommodate us. All we have to do is call it bin, and we’re ready to go.

Note: Many distributions provide this PATH setting by default. Debian based distributions, such as Ubuntu, test for the existence of the ~/bin directory at login and dynamically add it to the PATH variable if the directory is found.

Lastly, we have:

```
export PATH
```

The export command tells the shell to make the contents of PATH available to child processes of this shell.

Modifying the Environment

Since we know where the startup files are and what they contain, we can modify them to customize our environment.

Which Files Should We Modify?

As a general rule, to add directories to your `PATH` or define additional environment variables, place those changes in `.bash_profile` (or the equivalent, according to your distribution; for example, Ubuntu uses `.profile`). For everything else, place the changes in `.bashrc`.

Note: Unless you are the system administrator and need to change the defaults for all users of the system, restrict your modifications to the files in your home directory. It is certainly possible to change the files in `/etc` such as `profile`, and in many cases it would be sensible to do so, but for now, let's play it safe.

Text Editors

To edit (i.e., modify) the shell's startup files, as well as most of the other configuration files on the system, we use a program called a *text editor*. A text editor is a program that is, in some ways, like a word processor in that it allows us to edit the words on the screen with a moving cursor. It differs from a word processor by only supporting pure text and often contains features designed for writing programs. Text editors are the central tool used by software developers to write code and by system administrators to manage the configuration files that control the system.

A lot of different text editors are available for Linux; most systems have several installed. Why so many different ones? Because programmers like writing them and since programmers use them extensively, they write editors to express their own desires as to how they should work.

Text editors fall into two basic categories: graphical and text based. GNOME and KDE both include some popular graphical editors. GNOME ships with an editor called `gedit`, which is usually called “Text Editor” in the GNOME menu. KDE usually ships with three, which are (in order of increasing complexity) `kedit`, `kwrite`, and `kate`.

There are many text-based editors. The popular ones we'll encounter are `nano`, `vi`, and `emacs`. The `nano` editor is a simple, easy-to-use editor designed as a replacement for the `pico` editor supplied with the PINE email suite. The `vi` editor (which on most Linux systems replaced by a program named `vim`, which is short for “vi improved”) is the traditional editor for Unix-like systems. It will be the subject of our next chapter. The `emacs` editor was originally written by Richard Stallman. It is a gigantic, all-purpose, does-everything programming environment. While readily available, it is seldom installed on most Linux systems by default.

Using a Text Editor

Text editors can be invoked from the command line by typing the name of the editor followed by the name of the file we want to edit. If the file does not already exist, the editor will assume that we want to create a new file. Here is an example using `gedit`:

```
[me@linuxbox ~]$ gedit some_file
```

This command will start the `gedit` text editor and load the file named "some_file", if it exists.

Graphical text editors are pretty self-explanatory, so we won't cover them here. Instead, we will concentrate on our first text-based text editor, `nano`. Let's fire up `nano` and edit the `.bashrc` file. But before we do that, let's practice some "safe computing." Whenever we edit an important configuration file, it is always a good idea to create a backup copy of the file first. This protects us in case we mess up the file while editing. To create a backup of the `.bashrc` file, do this:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

It doesn't matter what we call the backup file; just pick an understandable name. The extensions ".bak", ".sav", ".old", and ".orig" are all popular ways of indicating a backup file. Oh, and remember that `cp` will *overwrite existing files* silently.

Now that we have a backup file, we'll start the editor.

```
[me@linuxbox ~]$ nano .bashrc
```

Once `nano` starts, we'll get a screen like this:

```
  GNU nano 2.0.3           File: .bashrc

# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
```

```
# User specific aliases and functions









                          [ Read 8 lines ]
^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

> **Note:** If your system does not have `nano` installed, you may use a graphical editor instead.

The screen consists of a header at the top, the text of the file being edited in the middle, and a menu of commands at the bottom. Since `nano` was designed to replace the text editor supplied with an email client, it is rather short on editing features.

The first command we should learn in any text editor is how to exit the program. In the case of `nano`, we press `Ctrl-x` to exit. This is indicated in the menu at the bottom of the screen. The notation `^X` means `Ctrl-x`. This is a common notation for control characters used by many programs.

The second command we need to know is how to save our work. With `nano` it's `Ctrl-o`. With this knowledge, we're ready to do some editing. Using the down arrow key and/or the `PageDown` key, move the cursor to the end of the file, and then add the following lines to the `.bashrc` file:

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

> **Note:** Your distribution may already include some of these, but duplicates won't

hurt anything.

Table 11-4 details the meaning of our additions:

Table 11-4: Additions to Our `.bashrc`

Line	Meaning
`umask 0002`	Sets the `umask` to solve the problem with the shared directories we discussed in Chapter 9, “Permissions.”
`export HISTCONTROL=ignoredups`	Causes the shell's history recording feature to ignore a command if the same command was just recorded.
`export HISTSIZE=1000`	Increases the size of the command history from the usual default of 500 lines to 1,000 lines.
`alias l.='ls -d .* --color=auto'`	Creates a new command called `l.`, which displays all directory entries that begin with a dot.
`alias ll='ls -l --color=auto'`	Creates a new command called `ll`, which displays a long-format directory listing.

As we can see, many of our additions are not intuitively obvious, so it would be a good idea to add some comments to our `.bashrc` file to help explain things to the humans. Using the editor, change our additions to look like this:

```
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase
# history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Ah, much better! With our changes complete, press `Ctrl-o` to save our modified `.bashrc` file, and press `Ctrl-x` to exit `nano`.

Why Comments Are Important

Whenever you modify configuration files it's a good idea to add some comments to document your changes. Sure, you'll probably remember what you changed tomorrow, but what about six months from now? Do yourself a favor and add some comments. While you're at it, it’s not a bad idea to keep a log of what changes you make.

Shell scripts and `bash` startup files use a “#” symbol to begin a comment. Other configuration files may use other symbols. Most configuration files will have comments. Use them as a guide.

You will often see lines in configuration files that are *commented out* to prevent them from being used by the affected program. This is done to give the reader suggestions for possible configuration choices or examples of correct configuration syntax. For example, the `.bashrc` file of Ubuntu 18.04 contains these lines:

```
# some more ls aliases
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

The last three lines are valid alias definitions that have been commented out. If you remove the leading “#” symbols from these three lines, a technique called *uncommenting*, you will activate the aliases. Conversely, if you add a “#” symbol to

> the beginning of a line, you can deactivate a configuration line while preserving the information it contains.

Activating Our Changes

The changes we have made to our `.bashrc` will not take effect until we close our terminal session and start a new one because the `.bashrc` file is only read at the beginning of a session. However, we can force `bash` to reread the modified `.bashrc` file with the following command:

```
[me@linuxbox ~]$ source ~/.bashrc
```

After doing this, we should be able to see the effect of our changes. Try one of the new aliases.

```
[me@linuxbox ~]$ ll
```

Summing Up

In this chapter, we learned an essential skill—editing configuration files with a text editor. Moving forward, as we read man pages for commands, take note of the environment variables that commands support. There may be a gem or two. In later chapters, we will learn about shell functions, a powerful feature that you can also include in the `bash` startup files to add to your arsenal of custom commands.

Further Reading

- The INVOCATION section of the `bash` man page covers the `bash` startup files in gory detail.

13 – Customizing the Prompt

In this chapter, we will look at a seemingly trivial detail—our shell prompt. This examination will reveal some of the inner workings of the shell and the terminal emulator program.

Like so many things in Linux, the shell prompt is highly configurable, and while we have pretty much taken it for granted, the prompt is a really useful device once we learn how to control it.

Anatomy of a Prompt

Our default prompt looks something like this:

```
[me@linuxbox ~]$
```

Notice that it contains our username, our hostname, and our current working directory, but how did it get that way? Very simply, it turns out. The prompt is defined by an environment variable named `PS1` (short for "prompt string 1"). We can view the contents of `PS1` with the `echo` command.

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \W]\$
```

> **Note:** Don't worry if your results are not the same as the example above. Every Linux distribution defines the prompt string a little differently, some quite exotically.

From the results, we can see that `PS1` contains a few of the characters we see in our prompt such as the brackets, the at-sign, and the dollar sign, but the rest are a mystery. The astute among us will recognize these as *backslash-escaped special characters* like those we saw in Chapter 7, "Seeing the World as the Shell Sees It." Table 13-1 provides a

partial list of the characters that the `bash` treats specially in the prompt string.

Table 13-1: Escape Codes Used in Shell Prompts

Sequence	Value Displayed
`\a`	ASCII bell. This makes the computer beep when it is encountered.
`\d`	Current date in day, month, date format. For example, “Mon May 26.”
`\h`	Hostname of the local machine minus the trailing domain name.
`\H`	Full hostname.
`\j`	Number of jobs running in the current shell session.
`\l`	Name of the current terminal device.
`\n`	A newline character.
`\r`	A carriage return.
`\s`	Name of the shell program.
`\t`	Current time in 24-hour hours:minutes:seconds format.
`\T`	Current time in 12-hour format.
`\@`	Current time in 12-hour AM/PM format.
`\A`	Current time in 24-hour hours:minutes format.
`\u`	Username of the current user.
`\v`	Version number of the shell.
`\V`	Version and release numbers of the shell.
`\w`	Name of the current working directory.
`\W`	Last part of the current working directory name.
`\!`	History number of the current command.
`\#`	Number of commands entered during this shell session.
`\$`	This displays a “$” character unless we have superuser privileges. In that case, it displays a “#” instead.
`\[`	Signals the start of a series of one or more non-printing characters. This is used to embed non-printing control characters that manipulate the terminal emulator in some way, such as moving the cursor or changing text colors.

\]	Signals the end of a non-printing character sequence.

Trying Some Alternative Prompt Designs

With this list of special characters, we can change the prompt to see the effect. First, we'll back up the existing prompt string so we can restore it later. To do this, we will copy the existing string into another shell variable that we create ourselves.

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

We create a new variable called `ps1_old` and assign the value of `PS1` to it. We can verify that the string has been copied by using the `echo` command.

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \W]\$
```

We can restore the original prompt at any time during our terminal session by simply reversing the process.

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

Now that we are ready to proceed, let's see what happens if we have an empty prompt string.

```
[me@linuxbox ~]$ PS1=
```

If we assign nothing to the prompt string, we get nothing. No prompt string at all! The prompt is still there, but displays nothing, just as we asked it to do. Since this is kind of disconcerting to look at, we'll replace it with a minimal prompt.

```
PS1="\$ "
```

That's better. At least now we can see what we are doing. Notice the trailing space within the double quotes. This provides the space between the dollar sign and the cursor when the prompt is displayed.

Let's add a bell to our prompt.

```
$ PS1="\[\a\]\$ "
```

Now we should hear a beep each time the prompt is displayed, though some systems disable this "feature." This could get annoying, but it might be useful if we needed notification when an especially long-running command has been executed. Note that we included the `\[` and `\]` sequences. Since the ASCII bell (`\a`) does not "print," that is, it does not move the cursor, we need to tell `bash` so it can correctly determine the length of the prompt.

Next, let's try to make an informative prompt with some hostname and time-of-day information.

```
$ PS1="\A \h \$ "
17:33 linuxbox $
```

Adding time-of-day to our prompt will be useful if we need to keep track of when we perform certain tasks. Finally, we'll make a new prompt that is similar to our original.

```
17:37 linuxbox $ PS1="<\u@\h \W>\$ "
<me@linuxbox ~>$
```

Try the other sequences listed in the table above and see whether you can come up with a brilliant new prompt.

Adding Color

Most terminal emulator programs respond to certain non-printing character sequences to control such things as character attributes (such as color, bold text, and the dreaded blinking text) and cursor position. We'll cover cursor position in a little bit, but first we'll look at color.

Terminal Confusion

Back in ancient times, when terminals were hooked to remote computers, there were many competing brands of terminals and they all worked differently. They had different keyboards, and they all had different ways of interpreting control information. Unix and Unix-like systems have two rather complex subsystems to

deal with the babel of terminal control (called `termcap` and `terminfo`). If you look in the deepest recesses of your terminal emulator settings, you may find a setting for the type of terminal emulation.

In an effort to make terminals speak some sort of common language, the American National Standards Institute (ANSI) developed a standard set of character sequences to control video terminals. Old-time DOS users will remember the `ANSI.SYS` file that was used to enable interpretation of these codes.

Character color is controlled by sending the terminal emulator an *ANSI escape code* embedded in the stream of characters to be displayed. The control code does not "print out" on the display; rather, it is interpreted by the terminal as an instruction. As we saw in the table above, the `\[` and `\]` sequences are used to encapsulate non-printing characters. An ANSI escape code begins with an octal 033 (the code generated by the `Esc` key), followed by an optional character attribute, followed by an instruction. For example, the code to set the text color to normal (attribute = 0), black text is as follows:

```
\033[0;30m
```

Table 13-2 lists the available text colors. Notice that the colors are divided into two groups, differentiated by the application of the bold character attribute (1), which creates the appearance of "light" colors.

Table 13- 2: Escape Sequences Used to Set Text Colors

Sequence	Text Color	Sequence	Text Color
`\033[0;30m`	Black	`\033[1;30m`	Dark gray
`\033[0;31m`	Red	`\033[1;31m`	Light red
`\033[0;32m`	Green	`\033[1;32m`	Light green
`\033[0;33m`	Brown	`\033[1;33m`	Yellow
`\033[0;34m`	Blue	`\033[1;34m`	Light blue
`\033[0;35m`	Purple	`\033[1;35m`	Light purple
`\033[0;36m`	Cyan	`\033[1;36m`	Light cyan
`\033[0;37m`	Light gray	`\033[1;37m`	White

Let's try to make a red prompt. We'll insert the escape code at the beginning.

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$ "
<me@linuxbox ~>$
```

That works, but notice that all the text that we type after the prompt will also display in red. To fix this, we will add another escape code to the end of the prompt that tells the terminal emulator to return to the previous color.

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

That's better!

It's also possible to set the text background color using the codes listed Table 13-3. The background colors do not support the bold attribute.

Table 13-3: Escape Sequences Used to Set Background Color

Sequence	Background Color	Sequence	Background Color
\033[0;40m	Black	\033[0;44m	Blue
\033[0;41m	Red	\033[0;45m	Purple
\033[0;42m	Green	\033[0;46m	Cyan
\033[0;43m	Brown	\033[0;47m	Light gray

We can create a prompt with a red background by applying a simple change to the first escape code.

```
<me@linuxbox ~>$ PS1="\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

Try the color codes and see what you can create!

> **Note:** Besides the normal (0) and bold (1) character attributes, text may be given underscore (4), blinking (5), and inverse (7) attributes. In the interests of good taste, many terminal emulators refuse to honor the blinking attribute, however.

Moving the Cursor

Escape codes can be used to position the cursor. This is commonly used to provide a clock or some other kind of information at a different location on the screen, such as in an upper corner each time the prompt is drawn. Table 13-4 lists the escape codes that position the cursor.

Table 13-4: Cursor Movement Escape Sequences

Escape Code	Action
`\033[`*l*`;`*c*`H`	Move the cursor to line *l* and column *c*
`\033[`*n*`A`	Move the cursor up *n* lines
`\033[`*n*`B`	Move the cursor down *n* lines
`\033[`*n*`C`	Move the cursor forward *n* characters
`\033[`*n*`D`	Move the cursor backward *n* characters
`\033[2J`	Clear the screen and move the cursor to the upper-left corner (line 0, column 0)
`\033[K`	Clear from the cursor position to the end of the current line
`\033[s`	Store the current cursor position
`\033[u`	Recall the stored cursor position

Using the codes in Table 13-4, we'll construct a prompt that draws a red bar at the top of the screen containing a clock (rendered in yellow text) each time the prompt is displayed. The code for the prompt is this formidable-looking string:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ "
```

Table 13-5 outlines what each part of the string does.

Table 13-5: Breakdown of Complex Prompt String

Sequence	Action
`\[`	Begin a non-printing character sequence. The purpose of this is to allow `bash` to properly calculate the size of the visible prompt. Without an accurate calculation, command line editing features cannot position the cursor correctly.

`\033[s`	Store the cursor position. This is needed to return to the prompt location after the bar and clock have been drawn at the top of the screen. *Be aware that some terminal emulators do not recognize this code.*
`\033[0;0H`	Move the cursor to the upper-left corner, which is line 0, column 0.
`\033[0;41m`	Set the background color to red.
`\033[K`	Clear from the current cursor location (the top-left corner) to the end of the line. Since the background color is now red, the line is cleared to that color, creating our bar. Note that clearing to the end of the line does not change the cursor position, which remains in the upper-left corner.
`\033[1;33m`	Set the text color to yellow.
`\t`	Display the current time. While this is a "printing" element, we still include it in the non-printing portion of the prompt since we don't want `bash` to include the clock when calculating the true size of the displayed prompt.
`\033[0m`	Turn off color. This affects both the text and the background.
`\033[u`	Restore the cursor position saved earlier.
`\]`	End the non-printing characters sequence.
`<\u@\h \W>\$`	Prompt string.

Saving the Prompt

Obviously, we don't want to be typing that monster all the time, so we'll want to store our prompt someplace. We can make the prompt permanent by adding it to our `.bashrc` file. To do so, add these two lines to the file:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ "

export PS1
```

Summing Up

Believe it or not, there is much more that can be done with prompts involving shell func-

tions and scripts that we haven't covered here, but this is a good start. Not everyone will care enough to change the prompt, since the default prompt is usually satisfactory. But for those of us who like to tinker, the shell provides the opportunity for many hours of casual fun.

Further Reading

- The *Bash Prompt HOWTO* from the Linux Documentation Project provides a pretty complete discussion of what the shell prompt can be made to do. It is available at:
  http://tldp.org/HOWTO/Bash-Prompt-HOWTO/
- Wikipedia has a good article on the ANSI Escape Codes:
  http://en.wikipedia.org/wiki/ANSI_escape_code

16 – Networking

When it comes to networking, there is probably nothing that cannot be done with Linux. Linux is used to build all sorts of networking systems and appliances, including firewalls, routers, name servers, network-attached storage (NAS) boxes and on and on.

Just as the subject of networking is vast, so are the number of commands that can be used to configure and control it. We will focus our attention on just a few of the most frequently used ones. The commands chosen for examination include those used to monitor networks and those used to transfer files. In addition, we are going to explore the `ssh` program that is used to perform remote logins. This chapter will cover the following commands:

- `ping` – Send an ICMP ECHO_REQUEST to network hosts
- `traceroute` – Print the route packets trace to a network host
- `ip` – Show / manipulate routing, devices, policy routing and tunnels
- `netstat` – Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships
- `ftp` – Internet file transfer program
- `wget` – Non-interactive network downloader
- `ssh` – OpenSSH SSH client (remote login program)

We're going to assume a little background in networking. In this, the Internet age, everyone using a computer needs a basic understanding of networking concepts. To make full use of this chapter we should be familiar with the following terms:

- Internet protocol (IP) address
- Host and domain name
- Uniform resource identifier (URI)

Please see the "Further Reading" section below for some useful articles regarding these terms.

> **Note:** Some of the commands we will cover may (depending on your distribution) require the installation of additional packages from your distribution's repositories, and some may require superuser privileges to execute.

Examining and Monitoring a Network

Even if you're not the system administrator, it's often helpful to examine the performance and operation of a network.

ping

The most basic network command is `ping`. The `ping` command sends a special network packet called an ICMP ECHO_REQUEST to a specified host. Most network devices receiving this packet will reply to it, allowing the network connection to be verified.

> **Note:** It is possible to configure most network devices (including Linux hosts) to ignore these packets. This is usually done for security reasons, to partially obscure a host from a potential attacker. It is also common for firewalls to be configured to block ICMP traffic.

For example, to see whether we can reach `linuxcommand.org` (one of our favorite sites ;-), we can use `ping` like this:

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Once started, `ping` continues to send packets at a specified interval (default is one second) until it is interrupted.

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1
ttl=43 time=107 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2
ttl=43 time=108 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4
ttl=43 time=106 ms
```

```
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5
ttl=43 time=105 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6
ttl=43 time=107 ms

--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

After it is interrupted (in this case after the sixth packet) by pressing `Ctrl-c`, `ping` prints performance statistics. A properly performing network will exhibit 0 percent packet loss. A successful "ping" will indicate that the elements of the network (its interface cards, cabling, routing, and gateways) are in generally good working order.

traceroute

The `traceroute` program (some systems use the similar `tracepath` program instead) lists all the "hops" network traffic takes to get from the local system to a specified host. For example, to see the route taken to reach `slashdot.org`, we would do this:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

The output looks like this:

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte
packets
 1  ipcop.localdomain (192.168.1.1)  1.066 ms  1.366 ms  1.720 ms
 2  * * *
 3  ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9)  14.622
ms  14.885 ms  15.169 ms
 4  po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154)  17.634
ms  17.626 ms  17.899 ms
 5  po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158)  15.992
ms  15.983 ms  16.256 ms
 6  po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5)  22.835
ms  14.233 ms  14.405 ms
 7  po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34)  16.154
ms  13.600 ms  18.867 ms
 8  te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)
21.951 ms  21.073 ms  21.557 ms
```

```
 9  pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)
22.917 ms  21.884 ms  22.126 ms
10  204.70.144.1 (204.70.144.1)  43.110 ms  21.248 ms  21.264 ms
11  cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93)  21.857 ms
cr2-pos-0-0-3-1.newyork.savvis.net (204.70.204.238)  19.556 ms cr1-
pos-0-7-3-1.newyork.savvis.net (204.70.195.93)  19.634 ms
12  cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109)  41.586 ms
42.843 ms cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)
43.115 ms
13  hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net
(204.70.195.122)  44.215 ms  41.833 ms  45.658 ms
14  csr1-ve241.elkgrovech3.savvis.net (216.64.194.42)  46.840 ms
43.372 ms  47.041 ms
15  64.27.160.194 (64.27.160.194)  56.137 ms  55.887 ms  52.810 ms
16  slashdot.org (216.34.181.45)  42.727 ms  42.016 ms  41.437 ms
```

In the output, we can see that connecting from our test system to `slashdot.org` requires traversing 16 routers. For routers that provided identifying information, we see their hostnames, IP addresses, and performance data, which includes three samples of round-trip time from the local system to the router. For routers that do not provide identifying information (because of router configuration, network congestion, firewalls, etc.), we see asterisks as in the line for hop number 2. In cases where routing information is blocked, we can sometimes overcome this by adding either the `-T` or `-I` option to the `traceroute` command.

ip

The `ip` program is a multi-purpose network configuration tool that makes use of the full range networking of features available in modern Linux kernels. It replaces the earlier and now deprecated `ifconfig` program. With `ip`, we can examine a system's network interfaces and routing table.

```
[me@linuxbox ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
```

```
state UP group default qlen 1000
    link/ether ac:22:0b:52:cf:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.14/24 brd 192.168.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::ae22:bff:fe52:cf84/64 scope link
       valid_lft forever preferred_lft forever
```

In the example above, we see that our test system has two network interfaces. The first, called `lo`, is the *loopback interface*, a virtual interface that the system uses to "talk to itself" and the second, called `eth0`, is the Ethernet interface.

When performing casual network diagnostics, the important things to look for are the presence of the word `UP` in the first line for each interface, indicating that the network interface is enabled, and the presence of a valid IP address in the `inet` field on the third line. For systems using Dynamic Host Configuration Protocol (DHCP), a valid IP address in this field will verify that the DHCP is working.

`netstat`

The `netstat` program is used to examine various network settings and statistics. Through the use of its many options, we can look at a variety of features in our network setup. Using the `-ie` option, we can examine the network interfaces in our system.

```
[me@linuxbox ~]$ netstat -ie
eth0    Link encap:Ethernet  HWaddr 00:1d:09:9b:99:67
        inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
        inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:153098921 (146.0 MB)  TX bytes:261035246 (248.9 MB)
        Memory:fdfc0000-fdfe0000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:111490 (108.8 KB)  TX bytes:111490 (108.8 KB)
```

Using the `-r` option will display the kernel's network routing table. This shows how the network is configured to send packets from network to network.

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination  Gateway     Genmask       Flags   MSS Window  irtt Iface

192.168.1.0  *           255.255.255.0 U         0 0          0 eth0
default      192.168.1.1 0.0.0.0       UG        0 0          0 eth0
```

In this simple example, we see a typical routing table for a client machine on a local area network (LAN) behind a firewall/router. The first line of the listing shows the destination `192.168.1.0`. IP addresses that end in zero refer to networks rather than individual hosts, so this destination means any host on the LAN. The next field, `Gateway`, is the name or IP address of the gateway (router) used to go from the current host to the destination network. An asterisk in this field indicates that no gateway is needed.

The last line contains the destination `default`. This means any traffic destined for a network that is not otherwise listed in the table. In our example, we see that the gateway is defined as a router with the address of `192.168.1.1`, which presumably knows what to do with the destination traffic.

Like `ip`, the `netstat` program has many options and we have looked only at a couple. Check out the `ip` and `netstat` man pages for a complete list.

Transporting Files Over a Network

What good is a network unless we can move files across it? There are many programs that move data over networks. We will cover two of them now and several more in later sections.

`ftp`

One of the true "classic" programs, `ftp` gets its name from the protocol it uses, the *File Transfer Protocol*. FTP was once the most widely used method of downloading files over the Internet. Most, if not all, web browsers support it, and you often see URIs starting with the protocol `ftp://`.

Before there were web browsers, there was the `ftp` program. `ftp` is used to communicate with *FTP servers*, machines that contain files that can be uploaded and downloaded over a network.

FTP (in its original form) is not secure because it sends account names and passwords in

cleartext. This means they are not encrypted and anyone *sniffing* the network can see them. Because of this, almost all FTP done over the Internet is done by *anonymous FTP servers*. An anonymous server allows anyone to log in using the login name "anonymous" and a meaningless password.

In the example below, we show a typical session with the `ftp` program downloading an Ubuntu iso image located in the `/pub/cd_images/Ubuntu-18.04` directory of the anonymous FTP server `fileserver`:

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-18.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-rw-r--    1 500      500      733079552 Apr 25 03:53 ubuntu-
18.04-desktop-amd64.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-18.04-desktop-amd64.iso
local: ubuntu-18.04-desktop-amd64.iso remote: ubuntu-18.04-desktop-
amd64.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-18.04-desktop-
amd64.iso (733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

Table 16-1 provides an explanation of the commands entered during this session.

Table 16-1: Examples of Interactive `ftp` Commands

Command	Meaning

`ftp fileserver`	Invoke the `ftp` program and have it connect to the FTP server `fileserver`.
`anonymous`	Login name. After the login prompt, a password prompt will appear. Some servers will accept a blank password; others will require a password in the form of an email address. In that case, try something like `user@example.com`.
`cd pub/cd_images/Ubuntu-18.04`	Change to the directory on the remote system containing the desired file. Note that on most anonymous FTP servers, the files for public downloading are found somewhere under the `pub` directory.
`ls`	List the directory on the remote system.
`lcd Desktop`	Change the directory on the local system to `~/Desktop`. In the example, the `ftp` program was invoked when the working directory was `~`. This command changes the working directory to `~/Desktop`.
`get ubuntu-18.04-desktop-amd64.iso`	Tell the remote system to transfer the file `ubuntu-18.04-desktop-amd64.iso` to the local system. Since the working directory on the local system was changed to `~/Desktop`, the file will be downloaded there.
`bye`	Log off the remote server and end the `ftp` program session. The commands `quit` and `exit` may also be used.

Typing `help` at the `ftp>` prompt will display a list of the supported commands. Using `ftp` on a server where sufficient permissions have been granted, it is possible to perform

many ordinary file management tasks. It's clumsy, but it does work.

lftp – A Better ftp

ftp is not the only command-line FTP client. In fact, there are many. One of the better (and more popular) ones is lftp by Alexander Lukyanov. It works much like the traditional ftp program but has many additional convenience features including multiple-protocol support (including HTTP), automatic retry on failed downloads, background processes, tab completion of path names, and many more.

wget

Another popular command-line program for file downloading is wget. It is useful for downloading content from both web and FTP sites. Single files, multiple files, and even entire sites can be downloaded. To download the first page of linuxcommand.org we could do this:

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51--  http://linuxcommand.org/index.php
           => `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

    [ <=>                                 ] 3,120         --.--K/s

11:02:51 (161.75 MB/s) - `index.php' saved [3120]
```

The program's many options allow wget to recursively download, download files in the background (allowing you to log off but continue downloading), and complete the download of a partially downloaded file. These features are well documented in its better-than-average man page.

Secure Communication with Remote Hosts

For many years, Unix-like operating systems have had the ability to be administered remotely via a network. In the early days, before the general adoption of the Internet, there were a couple of popular programs used to log in to remote hosts. These were the rlogin and telnet programs. These programs, however, suffer from the same fatal flaw that the ftp program does; they transmit all their communications (including login

names and passwords) in cleartext. This makes them wholly inappropriate for use in the Internet age.

ssh

To address this problem, a new protocol called Secure Shell (SSH) was developed. SSH solves the two basic problems of secure communication with a remote host.

1. It authenticates that the remote host is who it says it is (thus preventing so-called man-in-the-middle attacks).
2. It encrypts all of the communications between the local and remote hosts.

SSH consists of two parts. An SSH server runs on the remote host, listening for incoming connections, by default, on port 22, while an SSH client is used on the local system to communicate with the remote server.

Most Linux distributions ship an implementation of SSH called OpenSSH from the OpenBSD project. Some distributions include both the client and the server packages by default (for example, Red Hat), while others (such as Ubuntu) only supply the client. To enable a system to receive remote connections, it must have the `OpenSSH-server` package installed, configured and running, and (if the system either is running or is behind a firewall) it must allow incoming network connections on TCP port 22.

Tip: If you don't have a remote system to connect to but want to try these examples, make sure the `OpenSSH-server` package is installed on your system and use `localhost` as the name of the remote host. That way, your machine will create network connections with itself.

The SSH client program used to connect to remote SSH servers is called, appropriately enough, `ssh`. To connect to a remote host named `remote-sys`, we would use the `ssh` client program like so:

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be
established.
RSA key fingerprint is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

The first time the connection is attempted, a message is displayed indicating that the authenticity of the remote host cannot be established. This is because the client program has never seen this remote host before. To accept the credentials of the remote host, enter

"yes" when prompted. Once the connection is established, the user is prompted for a password:

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list
of known hosts.
me@remote-sys's password:
```

After the password is successfully entered, we receive the shell prompt from the remote system.

```
Last login: Sat Aug 30 13:00:48 2016
[me@remote-sys ~]$
```

The remote shell session continues until the user enters the `exit` command at the remote shell prompt, thereby closing the remote connection. At this point, the local shell session resumes, and the local shell prompt reappears.

It is also possible to connect to remote systems using a different username. For example, if the local user "me" had an account named "bob" on a remote system, user `me` could log in to the account `bob` on the remote system as follows:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Sat Aug 30 13:03:21 2016
[bob@remote-sys ~]$
```

As stated earlier, `ssh` verifies the authenticity of the remote host. If the remote host does not successfully authenticate, the following message appears:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    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
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
```

```
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of this
message.
Offending key in /home/me/.ssh/known_hosts:1
RSA host key for remote-sys has changed and you have requested strict
checking.
Host key verification failed.
```

This message is caused by one of two possible situations. First, an attacker may be attempting a man-in-the-middle attack. This is rare, since everybody knows that `ssh` alerts the user to this. The more likely culprit is that the remote system has been changed somehow; for example, its operating system or SSH server has been reinstalled. In the interests of security and safety, however, the first possibility should not be dismissed out of hand. Always check with the administrator of the remote system when this message occurs.

After it has been determined that the message is because of a benign cause, it is safe to correct the problem on the client side. This is done by using a text editor (`vim` perhaps) to remove the obsolete key from the `~/.ssh/known_hosts` file. In the example message above, we see this:

```
Offending key in /home/me/.ssh/known_hosts:1
```

This means that the first line of the `known_hosts` file contains the offending key. Delete this line from the file, and the `ssh` program will be able to accept new authentication credentials from the remote system.

Besides opening a shell session on a remote system, `ssh` allows us to execute a single command on a remote system. For example, to execute the `free` command on a remote host named `remote-sys` and have the results displayed on the local system, use this:

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
             total       used       free     shared    buffers     cached

Mem:        775536     507184     268352          0     110068     154596

-/+ buffers/cache:     242520     533016
Swap:      1572856          0    1572856
[me@linuxbox ~]$
```

It’s possible to use this technique in more interesting ways, such as the following exam-

ple in which we perform an ls on the remote system and redirect the output to a file on the local system:

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

Notice the use of the single quotes in the command above. This is done because we do not want the pathname expansion performed on the local machine; rather, we want it to be performed on the remote system. Likewise, if we had wanted the output redirected to a file on the remote machine, we could have placed the redirection operator and the filename within the single quotes.

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

Tunneling with SSH

Part of what happens when you establish a connection with a remote host via SSH is that an *encrypted tunnel* is created between the local and remote systems. Normally, this tunnel is used to allow commands typed at the local system to be transmitted safely to the remote system and for the results to be transmitted safely back. In addition to this basic function, the SSH protocol allows most types of network traffic to be sent through the encrypted tunnel, creating a sort of virtual private network (VPN) between the local and remote systems.

Perhaps the most common use of this feature is to allow X Window system traffic to be transmitted. On a system running an X server (that is, a machine displaying a GUI), it is possible to launch and run an X client program (a graphical application) on a remote system and have its display appear on the local system. It's easy to do; here's an example. Let's say we are sitting at a Linux system called `linuxbox` that is running an X server, and we want to run the `xload` program on a remote system named `remote-sys` to see the program's graphical output on our local system. We could do this:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 08 13:23:11 2016
[me@remote-sys ~]$ xload
```

After the `xload` command is executed on the remote system, its window appears on the local system. On some systems, you may need to use the "-Y" option rather than the "-X" option to do this.

`scp` and `sftp`

The OpenSSH package also includes two programs that can make use of an SSH-encrypted tunnel to copy files across the network. The first, `scp` (secure copy) is used much like the familiar `cp` program to copy files. The most notable difference is that the source or destination pathnames may be preceded with the name of a remote host, followed by a colon character. For example, if we wanted to copy a document named `document.txt` from our home directory on the remote system, `remote-sys`, to the current working directory on our local system, we could do this:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt                              100% 5581     5.5KB/s   00:00
[me@linuxbox ~]$
```

As with `ssh`, you may apply a username to the beginning of the remote host's name if the desired remote host account name does not match that of the local system.

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

The second SSH file-copying program is `sftp` which, as its name implies, is a secure replacement for the `ftp` program. `sftp` works much like the original `ftp` program that we used earlier; however, instead of transmitting everything in cleartext, it uses an SSH encrypted tunnel. `sftp` has an important advantage over conventional `ftp` in that it does not require an FTP server to be running on the remote host. It requires only the SSH server. This means that any remote machine that can connect with the SSH client can also be used as an FTP-like server. Here is a sample session:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
```

```
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-
desktop-i386.iso
/home/me/ubuntu-8.04-desktop-i386.iso 100%  699MB   7.4MB/s   01:35
sftp> bye
```

Tip: The SFTP protocol is supported by many of the graphical file managers found in Linux distributions. Using either GNOME or KDE, we can enter a URI beginning with `sftp://` into the location bar and operate on files stored on a remote system running an SSH server.

An SSH Client for Windows?

Let's say you are sitting at a Windows machine but you need to log in to your Linux server and get some real work done; what do you do? Get an SSH client program for your Windows box, of course! There are a number of these. The most popular one is probably PuTTY by Simon Tatham and his team. The PuTTY program displays a terminal window and allows a Windows user to open an SSH (or telnet) session on a remote host. The program also provides analogs for the `scp` and `sftp` programs.

PuTTY is available at http://www.chiark.greenend.org.uk/~sgtatham/putty/

Summing Up

In this chapter, we surveyed the field of networking tools found on most Linux systems. Since Linux is so widely used in servers and networking appliances, there are many more that can be added by installing additional software. But even with the basic set of tools, it is possible to perform many useful network-related tasks.

Further Reading

- For a broad (albeit dated) look at network administration, the Linux Documentation Project provides the *Linux Network Administrator's Guide*: http://tldp.org/LDP/nag2/index.html

- Wikipedia contains many good networking articles. Here are some of the basics:
  http://en.wikipedia.org/wiki/Internet_protocol_address
  http://en.wikipedia.org/wiki/Host_name
  http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

18 – Archiving and Backup

One of the primary tasks of a computer system's administrator is keeping the system's data secure. One way this is done is by performing timely backups of the system's files. Even if you're not a system administrator, it is often useful to make copies of things and move large collections of files from place to place and from device to device.

In this chapter, we will look at several common programs that are used to manage collections of files. These are the file compression programs:

- `gzip` – Compress or expand files
- `bzip2` – A block sorting file compressor

These are the archiving programs:

- `tar` – Tape archiving utility
- `zip` – Package and compress files

This is the file synchronization program:

- `rsync` – Remote file and directory synchronization

Compressing Files

Throughout the history of computing, there has been a struggle to get the most data into the smallest available space, whether that space be memory, storage devices, or network bandwidth. Many of the data services that we take for granted today, such as mobile phone service, high-definition television, or broadband Internet, owe their existence to effective *data compression* techniques.

Data compression is the process of removing *redundancy* from data. Let's consider an imaginary example. Say we had an entirely black picture file with the dimensions of 100 pixels by 100 pixels. In terms of data storage (assuming 24 bits, or 3 bytes per pixel), the image will occupy 30,000 bytes of storage.

100 * 100 * 3 = 30,000

An image that is all one color contains entirely redundant data. If we were clever, we could encode the data in such a way that we simply describe the fact that we have a block

of 10,000 black pixels. So, instead of storing a block of data containing 30,000 zeros (black is usually represented in image files as zero), we could compress the data into the number 10,000, followed by a zero to represent our data. Such a data compression scheme is called *run-length encoding* and is one of the most rudimentary compression techniques. Today's techniques are much more advanced and complex, but the basic goal remains the same—get rid of redundant data.

Compression algorithms (the mathematical techniques used to carry out the compression) fall into two general categories.

- *Lossless*: Lossless compression preserves all the data contained in the original. This means that when a file is restored from a compressed version, the restored file is exactly the same as the original, uncompressed version.
- *Lossy*: Lossy compression, on the other hand, removes data as the compression is performed to allow more compression to be applied. When a lossy file is restored, it does not match the original version; rather, it is a close approximation. Examples of lossy compression are JPEG (for images) and MP3 (for music).

In our discussion, we will look exclusively at lossless compression since most data on computers cannot tolerate any data loss.

gzip

The `gzip` program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding `gunzip` program is used to restore compressed files to their original, uncompressed form. Here is an example:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2018-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    3230 2018-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2018-10-14 07:15 foo.txt
```

In this example, we create a text file named `foo.txt` from a directory listing. Next, we run `gzip`, which replaces the original file with a compressed version named `foo.txt.gz`. In the directory listing of `foo.*`, we see that the original file has been replaced with the compressed version and that the compressed version is about one-fifth the size of the original. We can also see that the compressed file has the same permissions and time-

stamp as the original.

Next, we run the `gunzip` program to uncompress the file. Afterward, we can see that the compressed version of the file has been replaced with the original, again with the permissions and timestamp preserved.

`gzip` has many options, as described in Table 18-1.

Table 18-1: `gzip` Options

Option	Long Option	Description
`-c`	`--stdout` `--to-stdout`	Write output to standard output and keep the original files.
`-d`	`--decompress` `--uncompress`	Decompress. This causes `gzip` to act like `gunzip`.
`-f`	`--force`	Force compression even if a compressed version of the original file already exists.
`-h`	`--help`	Display usage information.
`-l`	`--list`	List compression statistics for each file compressed.
`-r`	`--recursive`	If one or more arguments on the command line is a directory, recursively compress files contained within them.
`-t`	`--test`	Test the integrity of a compressed file.
`-v`	`--verbose`	Display verbose messages while compressing.
`-`*`number`*		Set amount of compression. *number* is an integer in the range of 1 (fastest, least compression) to 9 (slowest, most compression). The values 1 and 9 may also be expressed as `--fast` and `--best`, respectively. The default value is 6.

Let’s return to our earlier example.

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz:   OK
```

```
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

Here, we replaced the file `foo.txt` with a compressed version named `foo.txt.gz`. Next, we tested the integrity of the compressed version, using the `-t` and `-v` options. Finally, we decompressed the file to its original form.

`gzip` can also be used in interesting ways via standard input and output.

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

This command creates a compressed version of a directory listing.

The `gunzip` program, which uncompresses gzip files, assumes that filenames end in the extension `.gz`, so it's not necessary to specify it, as long as the specified name is not in conflict with an existing uncompressed file.

```
[me@linuxbox ~]$ gunzip foo.txt
```

If our goal were only to view the contents of a compressed text file, we could do this:

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

Alternately, there is a program supplied with `gzip`, called `zcat`, that is equivalent to `gunzip` with the `-c` option. It can be used like the `cat` command on `gzip` compressed files.

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

Tip: There is a `zless` program, too. It performs the same function as the previous pipeline.

bzip2

The `bzip2` program, by Julian Seward, is similar to `gzip` but uses a different compression algorithm that achieves higher levels of compression at the cost of compression speed. In most regards, it works in the same fashion as `gzip`. A file compressed with

`bzip2` is denoted with the extension `.bz2`.

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me    me    15738 2018-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me    me     2792 2018-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

As we can see, `bzip2` can be used the same way as `gzip`. All the options (except for `-r`) that we discussed for `gzip` are also supported in `bzip2`. Note, however, that the compression-level option (`-`*`number`*) has a somewhat different meaning to `bzip2`. `bzip2` comes with `bunzip2` and `bzcat` for decompressing files.

`bzip2` also comes with the `bzip2recover` program, which will try to recover damaged `.bz2` files.

Don’t Be Compressive Compulsive

I occasionally see people attempting to compress a file that has already been compressed with an effective compression algorithm by doing something like this:

```
$ gzip picture.jpg
```

Don’t do it. You’re probably just wasting time and space! If you apply compression to a file that is already compressed, you will usually end up with a larger file. This is because all compression techniques involve some overhead that is added to the file to describe the compression. If you try to compress a file that already contains no redundant information, the compression will most often not result in any savings to offset the additional overhead.

Archiving Files

A common file-management task often used in conjunction with compression is *archiving*. Archiving is the process of gathering up many files and bundling them together into a single large file. Archiving is often done as part of system backups. It is also used when old data is moved from a system to some type of long-term storage.

tar

In the Unix-like world of software, the `tar` program is the classic tool for archiving files. Its name, short for *tape archive*, reveals its roots as a tool for making backup tapes. While it is still used for that traditional task, it is equally adept on other storage devices. We often see filenames that end with the extension `.tar` or `.tgz`, which indicate a "plain" tar archive and a gzipped archive, respectively. A tar archive can consist of a group of separate files, one or more directory hierarchies, or a mixture of both. The command syntax works like this:

`tar` *mode*[*options*] *pathname*...

Here *mode* is one of the following operating modes listed in Table 18-2 (only a partial list is shown here; see the `tar` man page for a complete list).

Table 18-2: `tar` Modes

Mode	Description
`c`	Create an archive from a list of files and/or directories.
`x`	Extract an archive.
`r`	Append specified pathnames to the end of an archive.
`t`	List the contents of an archive.

`tar` uses a slightly odd way of expressing options, so we'll need some examples to show how it works. First, let's re-create our playground from the previous chapter.

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

Next, let's create a tar archive of the entire playground.

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

This command creates a tar archive named `playground.tar` that contains the entire playground directory hierarchy. We can see that the mode and the `f` option, which is used to specify the name of the `tar` archive, may be joined together and do not require a leading dash. Note, however, that the mode must always be specified first, before any other option.

To list the contents of the archive, we can do this:

```
[me@linuxbox ~]$ tar tf playground.tar
```

For a more detailed listing, we can add the v (verbose) option.

```
[me@linuxbox ~]$ tar tvf playground.tar
```

Now, let's extract the playground in a new location. We will do this by creating a new directory named foo, changing the directory and extracting the tar archive.

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

If we examine the contents of ~/foo/playground, we see that the archive was successfully installed, creating a precise reproduction of the original files. There is one caveat, however. Unless we are operating as the superuser, files and directories extracted from archives take on the ownership of the user performing the restoration, rather than the original owner.

Another interesting behavior of tar is the way it handles pathnames in archives. The default for pathnames is relative, rather than absolute. tar does this by simply removing any leading slash from the pathname when creating the archive. To demonstrate, we will re-create our archive, this time specifying an absolute pathname.

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

Remember, ~/playground will expand into /home/me/playground when we press the Enter key, so we will get an absolute pathname for our demonstration. Next, we will extract the archive as before and watch what happens.

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home   playground
[me@linuxbox foo]$ ls home
```

```
me
[me@linuxbox foo]$ ls home/me
playground
```

Here we can see that when we extracted our second archive, it re-created the directory `home/me/playground` relative to our current working directory, `~/foo`, not relative to the root directory, as would have been the case with an absolute pathname. This may seem like an odd way for it to work, but it's actually more useful this way, because it allows us to extract archives to any location rather than being forced to extract them to their original locations. Repeating the exercise with the inclusion of the verbose option (`v`) will give a clearer picture of what's going on.

Let's consider a hypothetical, yet practical, example of `tar` in action. Imagine we want to copy the home directory and its contents from one system to another and we have a large USB hard drive that we can use for the transfer. On our modern Linux system, the drive is "automagically" mounted in the `/media` directory. Let's also imagine that the disk has a volume name of `BigDisk` when we attach it. To make the tar archive, we can do the following:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

After the tar file is written, we unmount the drive and attach it to the second computer. Again, it is mounted at `/media/BigDisk`. To extract the archive, we do this:

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

What's important to see here is that we must first change directory to `/` so that the extraction is relative to the root directory, since all pathnames within the archive are relative.

When extracting an archive, it's possible to limit what is extracted from the archive. For example, if we wanted to extract a single file from an archive, it could be done like this:

```
tar xf archive.tar pathname
```

By adding the trailing *pathname* to the command, `tar` will restore only the specified file. Multiple pathnames may be specified. Note that the pathname must be the full, exact relative pathname as stored in the archive. When specifying pathnames, wildcards are not normally supported; however, the GNU version of `tar` (which is the version most often

found in Linux distributions) supports them with the `--wildcards` option. Here is an example using our previous `playground.tar` file:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/pla
yground/dir-*/file-A'
```

This command will extract only files matching the specified pathname including the wildcard `dir-*`.

`tar` is often used in conjunction with `find` to produce archives. In this example, we will use `find` to produce a set of files to include in an archive.

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf
playground.tar '{}' '+'
```

Here we use `find` to match all the files in `playground` named `file-A` and then, using the `-exec` action, we invoke `tar` in the append mode (`r`) to add the matching files to the archive `playground.tar`.

Using `tar` with `find` is a good way of creating *incremental backups* of a directory tree or an entire system. By using `find` to match files newer than a timestamp file, we could create an archive that contains only those files newer than the last archive, assuming that the timestamp file is updated right after each archive is created.

`tar` can also make use of both standard input and output. Here is a comprehensive example:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-
from=- | gzip > playground.tgz
```

In this example, we used the `find` program to produce a list of matching files and piped them into `tar`. If the filename `-` is specified, it is taken to mean standard input or output, as needed. (By the way, this convention of using `-` to represent standard input/output is used by a number of other programs, too). The `--files-from` option (which may also be specified as `-T`) causes `tar` to read its list of pathnames from a file rather than the command line. Lastly, the archive produced by `tar` is piped into `gzip` to create the compressed archive `playground.tgz`. The `.tgz` extension is the conventional extension given to `gzip`-compressed tar files. The extension `.tar.gz` is also used some-

times.

While we used the `gzip` program externally to produce our compressed archive, modern versions of GNU `tar` support both `gzip` and `bzip2` compression directly with the use of the `z` and `j` options, respectively. Using our previous example as a base, we can simplify it this way:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf
playground.tgz -T -
```

If we had wanted to create a `bzip2`-compressed archive instead, we could have done this:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf
playground.tbz -T -
```

By simply changing the compression option from `z` to `j` (and changing the output file's extension to `.tbz` to indicate a `bzip2`-compressed file) we enabled `bzip2`-compression.

Another interesting use of standard input and output with the `tar` command involves transferring files between systems over a network. Imagine that we had two machines running a Unix-like system equipped with `tar` and `ssh`. In such a scenario, we could transfer a directory from a remote system (named `remote-sys` for this example) to our local system.

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar
xf -
me@remote-sys's password:
[me@linuxbox remote-stuff]$ ls
Documents
```

Here we were able to copy a directory named `Documents` from the remote system `remote-sys` to a directory within the directory named `remote-stuff` on the local system. How did we do this? First, we launched the `tar` program on the remote system using `ssh`. You will recall that `ssh` allows us to execute a program remotely on a networked computer and "see" the results on the local system—the standard output produced on the remote system is sent to the local system for viewing. We can take advan-

tage of this by having `tar` create an archive (the `c` mode) and send it to standard output, rather than a file (the `f` option with the dash argument), thereby transporting the archive over the encrypted tunnel provided by `ssh` to the local system. On the local system, we execute `tar` and have it expand an archive (the `x` mode) supplied from standard input (again, the `f` option with the dash argument).

zip

The `zip` program is both a compression tool and an archiver. The file format used by the program is familiar to Windows users, as it reads and writes `.zip` files. In Linux, however, `gzip` is the predominant compression program, with `bzip2` being a close second.

In its most basic usage, `zip` is invoked like this:

`zip` *`options zipfile file...`*

For example, to make a zip archive of our playground, we would do this:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

Unless we include the `-r` option for recursion, only the `playground` directory (but none of its contents) is stored. Although the addition of the extension `.zip` is automatic, we will include the file extension for clarity.

During the creation of the zip archive, `zip` will normally display a series of messages like this:

```
 adding: playground/dir-020/file-Z (stored 0%)
 adding: playground/dir-020/file-Y (stored 0%)
 adding: playground/dir-020/file-X (stored 0%)
 adding: playground/dir-087/ (stored 0%)
 adding: playground/dir-087/file-S (stored 0%)
```

These messages show the status of each file added to the archive. `zip` will add files to the archive using one of two storage methods: either it will “store” a file without compression, as shown here, or it will “deflate” the file that performs compression. The numeric value displayed after the storage method indicates the amount of compression achieved. Since our playground contains only empty files, no compression is performed on its contents.

Extracting the contents of a zip file is straightforward when using the `unzip` program.

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

One thing to note about `zip` (as opposed to `tar`) is that if an existing archive is specified, it is updated rather than replaced. This means the existing archive is preserved, but new files are added and matching files are replaced.

Files may be listed and extracted selectively from a zip archive by specifying them to `unzip`.

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Archive:  ../playground.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
        0  10-05-16 09:25   playground/dir-087/file-Z
 --------                   -------
        0                   1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-087/file-Z
Archive:  ../playground.zip
replace playground/dir-087/file-Z? [y]es, [n]o, [A]ll, [N]one,
[r]ename: y
 extracting: playground/dir-087/file-Z
```

Using the `-l` option causes `unzip` to merely list the contents of the archive without extracting the file. If no files are specified, `unzip` will list all files in the archive. The `-v` option can be added to increase the verbosity of the listing. Note that when the archive extraction conflicts with an existing file, the user is prompted before the file is replaced.

Like `tar`, `zip` can make use of standard input and output, though its implementation is somewhat less useful. It is possible to pipe a list of filenames to `zip` via the `-@` option.

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

Here we use `find` to generate a list of files matching the test `-name "file-A"` and then pipe the list into `zip`, which creates the archive `file-A.zip` containing the selected files.

`zip` also supports writing its output to standard output, but its use is limited because few programs can make use of the output. Unfortunately, the `unzip` program does not accept

standard input. This prevents `zip` and `unzip` from being used together to perform network file copying like `tar`.

`zip` can, however, accept standard input, so it can be used to compress the output of other programs.

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
 adding: - (deflated 80%)
```

In this example, we pipe the output of `ls` into `zip`. Like `tar`, `zip` interprets the trailing dash as "use standard input for the input file."

The `unzip` program allows its output to be sent to standard output when the `-p` (for pipe) option is specified.

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

We touched on some of the basic things that `zip`/`unzip` can do. They both have a lot of options that add to their flexibility, though some are platform specific to other systems. The man pages for both `zip` and `unzip` are pretty good and contain useful examples. However, the main use of these programs is for exchanging files with Windows systems, rather than performing compression and archiving on Linux, where `tar` and `gzip` are greatly preferred.

Synchronizing Files and Directories

A common strategy for maintaining a backup copy of a system involves keeping one or more directories synchronized with another directory (or directories) located on either the local system (usually a removable storage device of some kind) or a remote system. We might, for example, have a local copy of a website under development and synchronize it from time to time with the "live" copy on a remote web server.

In the Unix-like world, the preferred tool for this task is `rsync`. This program can synchronize both local and remote directories by using the *rsync remote-update protocol*, which allows `rsync` to quickly detect the differences between two directories and perform the minimum amount of copying required to bring them into sync. This makes `rsync` very fast and economical to use, compared to other kinds of copy programs.

`rsync` is invoked like this:

`rsync` *`options source destination`*

where *source* and *destination* are one of the following:

- A local file or directory
- A remote file or directory in the form of *[user@]host:path*
- A remote rsync server specified with a URI of *rsync://[user@]host[:port]/path*

Note that either the source or the destination must be a local file. Remote-to-remote copying is not supported.

Let's try `rsync` out on some local files. First, let's clean out our `foo` directory.

```
[me@linuxbox ~]$ rm -rf foo/*
```

Next, we'll synchronize the `playground` directory with a corresponding copy in `foo`.

```
[me@linuxbox ~]$ rsync -av playground foo
```

We've included both the `-a` option (for archiving—causes recursion and preservation of file attributes) and the `-v` option (verbose output) to make a *mirror* of the `playground` directory within `foo`. While the command runs, we will see a list of the files and directories being copied. At the end, we will see a summary message like this indicating the amount of copying performed:

```
sent 135759 bytes  received 57870 bytes  387258.00 bytes/sec
total size is 3230  speedup is 0.02
```

If we run the command again, we will see a different result.

```
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done

sent 22635 bytes  received 20 bytes  45310.00 bytes/sec
total size is 3230  speedup is 0.14
```

Notice that there was no listing of files. This is because `rsync` detected that there were no differences between `~/playground` and `~/foo/playground`, and therefore it didn't need to copy anything. If we modify a file in `playground` and run `rsync` again:

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
```

```
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
playground/dir-099/file-Z
sent 22685 bytes  received 42 bytes  45454.00 bytes/sec
total size is 3230  speedup is 0.14
```

we see that `rsync` detected the change and copied only the updated file.

There is a subtle but useful feature we can use when we specify an `rsync` source. Let's consider two directories.

```
[me@linuxbox ~]$ ls
source        destination
```

Directory `source` contains one file named `file1` and directory `destination` is empty. If we perform a copy of `source` to `destination` like so:

```
[me@linuxbox ~]$ rsync source destination
```

then `rsync` copies the directory `source` into `destination`.

```
[me@linuxbox ~]$ ls destination
source
```

However, if we append a trailing `/` to the source directory name, `rsync` will copy only the contents of the source directory and not the directory itself.

```
[me@linuxbox ~]$ rsync source/ destination
[me@linuxbox ~]$ ls destination
file1
```

This is handy if we want only the contents of a directory copied without creating another level of directories within the destination. We can think of it as being like `source/*` in its outcome, but this method will copy all of the source directory's content including the hidden files.

As a practical example, let's consider the imaginary external hard drive that we used earlier with `tar`. If we attach the drive to our system and it is mounted at `/media/`

BigDisk once again, we can perform a useful system backup by first creating a directory named /backup on the external drive and then using rsync to copy the most important stuff from our system to the external drive.

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup
```

In this example, we copied the /etc, /home, and /usr/local directories from our system to our imaginary storage device. We included the --delete option to remove files that may have existed on the backup device that no longer existed on the source device (this is irrelevant the first time we make a backup but will be useful on subsequent copies). Repeating the procedure of attaching the external drive and running this rsync command would be a useful (though not ideal) way of keeping a small system backed up. Of course, an alias would be helpful here, too. We could create an alias and add it to our .bashrc file to provide this feature.

```
alias backup='sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup'
```

Now all we have to do is attach our external drive and run the backup command to do the job.

Using rsync Over a Network

One of the real beauties of rsync is that it can be used to copy files over a network. After all, the r in rsync stands for “remote.” Remote copying can be done in one of two ways. The first way is with another system that has rsync installed, along with a remote shell program such as ssh. Let’s say we had another system on our local network with a lot of available hard drive space and we wanted to perform our backup operation using the remote system instead of an external drive. Assuming that it already had a directory named /backup where we could deliver our files, we could do this:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home
/usr/local remote-sys:/backup
```

We made two changes to our command to facilitate the network copy. First, we added the --rsh=ssh option, which instructs rsync to use the ssh program as its remote shell.

In this way, we were able to use an `ssh`-encrypted tunnel to securely transfer the data from the local system to the remote host. Second, we specified the remote host by prefixing its name (in this case the remote host is named `remote-sys`) to the destination pathname.

The second way that `rsync` can be used to synchronize files over a network is by using an *rsync server*. `rsync` can be configured to run as a daemon and listen to incoming requests for synchronization. This is often done to allow mirroring of a remote system. For example, Red Hat Software maintains a large repository of software packages under development for its Fedora distribution. It is useful for software testers to mirror this collection during the testing phase of the distribution release cycle. Since files in the repository change frequently (often more than once a day), it is desirable to maintain a local mirror by periodic synchronization, rather than by bulk copying of the repository. One of these repositories is kept at Duke University; we could mirror it using our local copy of `rsync` and their `rsync` server like this:

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av –delete rsync://archive.linux.duke.edu/
fedora/linux/development/rawhide/Everything/x86_64/os/ fedora-devel
```

In this example, we use the URI of the remote `rsync` server, which consists of a protocol (`rsync://`), followed by the remote host-name (`archive.linux.duke.edu`), followed by the pathname of the repository.

Summing Up

We've looked at the common compression and archiving programs used on Linux and other Unix-like operating systems. For archiving files, the `tar`/`gzip` combination is the preferred method on Unix-like systems while `zip`/`unzip` is used for interoperability with Windows systems. Finally, we looked at the `rsync` program (a personal favorite) which is very handy for efficient synchronization of files and directories across systems.

Further Reading

- The man pages for all of the commands discussed here are pretty clear and contain useful examples. In addition, the GNU Project has a good online manual for its version of `tar`. It can be found here:
  http://www.gnu.org/software/tar/manual/index.html

20 – Text Processing

All Unix-like operating systems rely heavily on text files for data storage. So it makes sense that there are many tools for manipulating text. In this chapter, we will look at programs that are used to "slice and dice" text. In the next chapter, we will look at more text processing, focusing on programs that are used to format text for printing and other kinds of human consumption.

This chapter will revisit some old friends and introduce us to some new ones:

- `cat` – Concatenate files and print on the standard output
- `sort` – Sort lines of text files
- `uniq` – Report or omit repeated lines
- `cut` – Remove sections from each line of files
- `paste` – Merge lines of files
- `join` – Join lines of two files on a common field
- `comm` – Compare two sorted files line by line
- `diff` – Compare files line by line
- `patch` – Apply a diff file to an original
- `tr` – Translate or delete characters
- `sed` – Stream editor for filtering and transforming text
- `aspell` – Interactive spell checker

Applications of Text

So far, we have learned a couple of text editors (`nano` and `vim`), looked at a bunch of configuration files, and have witnessed the output of dozens of commands, all in text. But what else is text used for? For many things, it turns out.

Documents

Many people write documents using plain text formats. While it is easy to see how a small text file could be useful for keeping simple notes, it is also possible to write large documents in text format. One popular approach is to write a large document in a text format and then embed a *markup language* to describe the formatting of the finished document. Many scientific papers are written using this method, as Unix-based text processing systems were among the first systems that supported the advanced typographical layout needed by writers in technical disciplines.

Web Pages

The world's most popular type of electronic document is probably the web page. Web pages are text documents that use either *Hypertext Markup Language* (*HTML*) or *Extensible Markup Language* (*XML*) as markup languages to describe the document's visual format.

Email

Email is an intrinsically text-based medium. Even non-text attachments are converted into a text representation for transmission. We can see this for ourselves by downloading an email message and then viewing it in `less`. We will see that the message begins with a *header* that describes the source of the message and the processing it received during its journey, followed by the *body* of the message with its content.

Printer Output

On Unix-like systems, output destined for a printer is sent as plain text or, if the page contains graphics, is converted into a text format *page description language* known as *PostScript*, which is then sent to a program that generates the graphic dots to be printed.

Program Source Code

Many of the command line programs found on Unix-like systems were created to support system administration and software development, and text processing programs are no exception. Many of them are designed to solve software development problems. The reason text processing is important to software developers is that all software starts out as text. *Source code*, the part of the program the programmer actually writes, is always in text format.

Revisiting Some Old Friends

Back in Chapter 6, "Redirection," we learned about some commands that are able to ac-

cept standard input in addition to command line arguments. We touched on them only briefly then, but now we will take a closer look at how they can be used to perform text processing.

cat

The `cat` program has a number of interesting options. Many of them are used to help better visualize text content. One example is the `-A` option, which is used to display non-printing characters in the text. There are times when we want to know whether control characters are embedded in our otherwise visible text. The most common of these are tab characters (as opposed to spaces) and carriage returns, often present as end-of-line characters in MS-DOS-style text files. Another common situation is a file containing lines of text with trailing spaces.

Let's create a test file using `cat` as a primitive word processor. To do this, we'll just enter the command `cat` (along with specifying a file for redirected output) and type our text, followed by `Enter` to properly end the line and then `Ctrl-d`, to indicate to `cat` that we have reached end-of-file. In this example, we enter a leading tab character and follow the line with some trailing spaces:

```
[me@linuxbox ~]$ cat > foo.txt
    The quick brown fox jumped over the lazy dog.
[me@linuxbox ~]$
```

Next, we will use `cat` with the `-A` option to display the text:

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog.   $
[me@linuxbox ~]$
```

As we can see in the results, the tab character in our text is represented by `^I`. This is a common notation that means `Ctrl-i` which, as it turns out, is the same as a tab character. We also see that a `$` appears at the true end of the line, indicating that our text contains trailing spaces.

MS-DOS Text vs. Unix Text

One of the reasons you may want to use `cat` to look for non-printing characters in text is to spot hidden carriage returns. Where do hidden carriage returns come from? DOS and Windows! Unix and DOS don't define the end of a line the same way in text files. Unix ends a line with a linefeed character (ASCII 10) while MS-DOS and its derivatives use the sequence carriage return (ASCII 13) and linefeed to terminate each line of text.

There are a several ways to convert files from DOS to Unix format. On many Linux systems, there are programs called `dos2unix` and `unix2dos`, which can convert text files to and from DOS format. However, if you don't have `dos2unix` on your system, don't worry. The process of converting text from DOS to Unix format is simple; it involves the removal of the offending carriage returns. That is easily accomplished by a couple of the programs discussed later in this chapter.

`cat` also has options that are used to modify text. The two most prominent are `-n`, which numbers lines, and `-s`, which suppresses the output of multiple blank lines. We can demonstrate thusly:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox


jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
     1  The quick brown fox
     2
     3  jumped over the lazy dog.
[me@linuxbox ~]$
```

In this example, we create a new version of our `foo.txt` test file, which contains two lines of text separated by two blank lines. After processing by `cat` with the `-ns` options, the extra blank line is removed and the remaining lines are numbered. While this is not much of a process to perform on text, it is a process.

sort

The `sort` program sorts the contents of standard input, or one or more files specified on the command line, and sends the results to standard output. Using the same technique that we used with `cat`, we can demonstrate processing of standard input directly from the keyboard as follows:

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

After entering the command, we type the letters `c`, `b`, and `a`, and then we press `Ctrl-d` to indicate end-of-file. We then view the resulting file and see that the lines now appear in sorted order.

Since `sort` can accept multiple files on the command line as arguments, it is possible to *merge* multiple files into a single sorted whole. For example, if we had three text files and wanted to combine them into a single sorted file, we could do something like this:

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

`sort` has several interesting options. Table 20-1 contains a partial list:

Table 20-1: Common `sort` Options

Option	Long Option	Description
`-b`	`--ignore-leading-blanks`	By default, sorting is performed on the entire line, starting with the first character in the line. This option causes `sort` to ignore leading spaces in lines and calculates sorting based on the first non-whitespace character on the line.
`-f`	`--ignore-case`	Make sorting case-insensitive.

`-n`	`--numeric-sort`	Perform sorting based on the numeric evaluation of a string. Using this option allows sorting to be performed on numeric values rather than alphabetic values.
`-r`	`--reverse`	Sort in reverse order. Results are in descending rather than ascending order.
`-k`	`--key=`*`field1`*`[,`*`field2`*`]`	Sort based on a key field located from *field1* to *field2* rather than the entire line. See the following discussion.
`-m`	`--merge`	Treat each argument as the name of a presorted file. Merge multiple files into a single sorted result without performing any additional sorting.
`-o`	`--output=`*`file`*	Send sorted output to *file* rather than standard output.
`-t`	`--field-separator=`*`char`*	Define the field-separator character. By default fields are separated by spaces or tabs.

Although most of these options are pretty self-explanatory, some are not. First, let's look at the `-n` option, used for numeric sorting. With this option, it is possible to sort values based on numeric values. We can demonstrate this by sorting the results of the `du` command to determine the largest users of disk space. Normally, the `du` command lists the results of a summary in pathname order.

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252     /usr/share/aclocal
96      /usr/share/acpi-support
8       /usr/share/adduser
196     /usr/share/alacarte
344     /usr/share/alsa
8       /usr/share/alsa-base
12488   /usr/share/anthy
8       /usr/share/apmd
```

```
21440   /usr/share/app-install
48      /usr/share/application-registry
```

In this example, we pipe the results into `head` to limit the results to the first 10 lines. We can produce a numerically sorted list to show the 10 largest consumers of space this way.

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940  /usr/share/locale-langpack
242660  /usr/share/doc
197560  /usr/share/fonts
179144  /usr/share/gnome
146764  /usr/share/myspell
144304  /usr/share/gimp
135880  /usr/share/dict
76508   /usr/share/icons
68072   /usr/share/apps
62844   /usr/share/foomatic
```

By using the `n` and `r` options, we produce a reverse numerical sort, with the largest values appearing first in the results. This sort works because the numerical values occur at the beginning of each line. But what if we want to sort a list based on some value found within the line? For example, here are the results of `ls -l`:

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root   root      34824 2016-04-04 02:42 [
-rwxr-xr-x 1 root   root     101556 2007-11-27 06:08 a2p
-rwxr-xr-x 1 root   root      13036 2016-02-27 08:22 aconnect
-rwxr-xr-x 1 root   root      10552 2007-08-15 10:34 acpi
-rwxr-xr-x 1 root   root       3800 2016-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root   root       7536 2016-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root   root       3576 2016-04-29 07:57 addpart
-rwxr-xr-x 1 root   root      20808 2016-01-03 18:02 addr2line
-rwxr-xr-x 1 root   root     489704 2016-10-09 17:02 adept_batch
```

Ignoring, for the moment, that `ls` can sort its results by size, we could use `sort` to sort this list by file size, as well.

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nrk 5 | head
```

```
-rwxr-xr-x 1 root   root    8234216 2016-04-07 17:42 inkscape
-rwxr-xr-x 1 root   root    8222692 2016-04-07 17:42 inkview
-rwxr-xr-x 1 root   root    3746508 2016-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root   root    3654020 2016-08-26 16:16 quanta
-rwxr-xr-x 1 root   root    2928760 2016-09-10 14:31 gdbtui
-rwxr-xr-x 1 root   root    2928756 2016-09-10 14:31 gdb
-rwxr-xr-x 1 root   root    2602236 2016-10-10 12:56 net
-rwxr-xr-x 1 root   root    2304684 2016-10-10 12:56 rpcclient
-rwxr-xr-x 1 root   root    2241832 2016-04-04 05:56 aptitude
-rwxr-xr-x 1 root   root    2202476 2016-10-10 12:56 smbcacls
```

Many uses of `sort` involve the processing of *tabular data*, such as the results of the previous `ls` command. If we apply database terminology to the previous table, we would say that each row is a *record* and that each record consists of multiple *fields*, such as the file attributes, link count, filename, file size, and so on. `sort` is able to process individual fields. In database terms, we are able to specify one or more *key fields* to use as *sort keys*. In the previous example, we specify the `n` and `r` options to perform a reverse numerical sort and specify `-k 5` to make `sort` use the fifth field as the key for sorting.

The `k` option is interesting and has many features, but first we need to talk about how `sort` defines fields. Let's consider the following simple text file consisting of a single line containing the author's name:

```
William Shotts
```

By default, `sort` sees this line as having two fields. The first field contains these characters:

`"William"`

The second field contains these characters:

`"Shotts"`

This means that whitespace characters (spaces and tabs) are used as delimiters between fields and that the delimiters are included in the field when sorting is performed.

Looking again at a line from our `ls` output, as follows, we can see that a line contains eight fields and that the fifth field is the file size:

```
-rwxr-xr-x 1 root   root    8234216 2016-04-07 17:42 inkscape
```

For our next series of experiments, let's consider the following file containing the history of three popular Linux distributions released from 2006 to 2008. Each line in the file has three fields: the distribution name, version number, and date of release in MM/DD/YYYY format.

```
SUSE    10.2     12/07/2006
Fedora  10       11/25/2008
SUSE    11.0     06/19/2008
Ubuntu  8.04     04/24/2008
Fedora  8        11/08/2007
SUSE    10.3     10/04/2007
Ubuntu  6.10     10/26/2006
Fedora  7        05/31/2007
Ubuntu  7.10     10/18/2007
Ubuntu  7.04     04/19/2007
SUSE    10.1     05/11/2006
Fedora  6        10/24/2006
Fedora  9        05/13/2008
Ubuntu  6.06     06/01/2006
Ubuntu  8.10     10/30/2008
Fedora  5        03/20/2006
```

Using a text editor (perhaps `vim`), we'll enter this data and name the resulting file `distros.txt`.

Next, we'll try sorting the file and observe these results:

```
[me@linuxbox ~]$ sort distros.txt
Fedora  10       11/25/2008
Fedora  5        03/20/2006
Fedora  6        10/24/2006
Fedora  7        05/31/2007
Fedora  8        11/08/2007
Fedora  9        05/13/2008
SUSE    10.1     05/11/2006
SUSE    10.2     12/07/2006
SUSE    10.3     10/04/2007
SUSE    11.0     06/19/2008
Ubuntu  6.06     06/01/2006
Ubuntu  6.10     10/26/2006
Ubuntu  7.04     04/19/2007
Ubuntu  7.10     10/18/2007
```

```
Ubuntu  8.04     04/24/2008
Ubuntu  8.10     10/30/2008
```

Well, it mostly worked. The problem occurs in the sorting of the Fedora version numbers. Since 1 comes before 5 in the character set, version 10 ends up at the top while version 9 falls to the bottom.

To fix this problem, we are going to have to sort on multiple keys. We want to perform an alphabetic sort on the first field and then a numeric sort on the second field. `sort` allows multiple instances of the `-k` option so that multiple sort keys can be specified. In fact, a key may include a range of fields. If no range is specified (as has been the case with our previous examples), `sort` uses a key that begins with the specified field and extends to the end of the line. Here is the syntax for our multi-key sort:

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora  5        03/20/2006
Fedora  6        10/24/2006
Fedora  7        05/31/2007
Fedora  8        11/08/2007
Fedora  9        05/13/2008
Fedora  10       11/25/2008
SUSE    10.1     05/11/2006
SUSE    10.2     12/07/2006
SUSE    10.3     10/04/2007
SUSE    11.0     06/19/2008
Ubuntu  6.06     06/01/2006
Ubuntu  6.10     10/26/2006
Ubuntu  7.04     04/19/2007
Ubuntu  7.10     10/18/2007
Ubuntu  8.04     04/24/2008
Ubuntu  8.10     10/30/2008
```

Though we used the long form of the option for clarity, `-k 1,1 -k 2n` would be exactly equivalent. In the first instance of the key option, we specified a range of fields to include in the first key. Since we wanted to limit the sort to just the first field, we specified `1,1` which means "start at field 1 and end at field 1." In the second instance, we specified `2n`, which means field 2 is the sort key and that the sort should be numeric. An option letter may be included at the end of a key specifier to indicate the type of sort to be performed. These option letters are the same as the global options for the `sort` program: `b` (ignore leading blanks), `n` (numeric sort), `r` (reverse sort), and so on.

The third field in our list contains a date in an inconvenient format for sorting. On com-

puters, dates are usually formatted in YYYY-MM-DD order to make chronological sorting easy, but ours are in the American format of MM/DD/YYYY. How can we sort this list in chronological order?

Fortunately, `sort` provides a way. The key option allows specification of *offsets* within fields, so we can define keys within fields.

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora  10      11/25/2008
Ubuntu  8.10    10/30/2008
SUSE    11.0    06/19/2008
Fedora  9       05/13/2008
Ubuntu  8.04    04/24/2008
Fedora  8       11/08/2007
Ubuntu  7.10    10/18/2007
SUSE    10.3    10/04/2007
Fedora  7       05/31/2007
Ubuntu  7.04    04/19/2007
SUSE    10.2    12/07/2006
Ubuntu  6.10    10/26/2006
Fedora  6       10/24/2006
Ubuntu  6.06    06/01/2006
SUSE    10.1    05/11/2006
Fedora  5       03/20/2006
```

By specifying `-k 3.7`, we instruct `sort` to use a sort key that begins at the seventh character within the third field, which corresponds to the start of the year. Likewise, we specify `-k 3.1` and `-k 3.4` to isolate the month and day portions of the date. We also add the `n` and `r` options to achieve a reverse numeric sort. The `b` option is included to suppress the leading spaces (whose numbers vary from line to line, thereby affecting the outcome of the sort) in the date field.

Some files don't use tabs and spaces as field delimiters; for example, here's the `/etc/passwd` file:

```
[me@linuxbox ~]$ head /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
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
```

```
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

The fields in this file are delimited with colons (`:`), so how would we sort this file using a key field? `sort` provides the `-t` option to define the field separator character. To sort the `passwd` file on the seventh field (the account's default shell), we could do this:

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

By specifying the colon character as the field separator, we can sort on the seventh field.

uniq

Compared to `sort`, the `uniq` program is lightweight. `uniq` performs a seemingly trivial task. When given a sorted file (or standard input), it removes any duplicate lines and sends the results to standard output. It is often used in conjunction with `sort` to clean the output of duplicates.

> **Tip:** While `uniq` is a traditional Unix tool often used with `sort`, the GNU version of `sort` supports a `-u` option, which removes duplicates from the sorted output.

Let's make a text file to try this as shown here:

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
```

```
a
b
c
```

Remember to type `Ctrl-d` to terminate standard input. Now, if we run `uniq` on our text file, we get this:

```
[me@linuxbox ~]$ uniq foo.txt
a
b
c
a
b
c
```

The results are no different from our original file; the duplicates were not removed. For `uniq` to do its job, the input must be sorted first.

```
[me@linuxbox ~]$ sort foo.txt | uniq
a
b
c
```

This is because `uniq` only removes duplicate lines that are adjacent to each other.

`uniq` has several options. Table 20-2 lists the common ones.

Table 20-2: Common `uniq` Options

Option	Long Option	Description
`-c`	`--count`	Output a list of duplicate lines preceded by the number of times the line occurs.
`-d`	`--repeated`	Output only repeated lines, rather than unique lines.
`-f` *n*	`--skip-fields=`*n*	Ignore *n* leading fields in each line. Fields are separated by whitespace as they are in `sort`; however, unlike `sort`, `uniq` has no option for setting an alternate field separator.

`-i`	`--ignore-case`	Ignore case during the line comparisons.
`-s` *n*	`--skip-chars=`*n*	Skip (ignore) the leading *n* characters of each line.
`-u`	`--unique`	Output only unique lines. Lines with duplicates are ignored.

Here we see `uniq` used to report the number of duplicates found in our text file, using the `-c` option:

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
      2 a
      2 b
      2 c
```

Slicing and Dicing

The next three programs we will discuss are used to peel columns of text out of files and recombine them in useful ways.

cut

The `cut` program is used to extract a section of text from a line and output the extracted section to standard output. It can accept multiple file arguments or input from standard input.

Specifying the section of the line to be extracted is somewhat awkward and is specified using the options listed in Table 20-3.

Table 20-3: `cut` Selection Options

Option	Long Option	Description
`-c` *list*	`--characters=`*list*	Extract the portion of the line defined by *list*. The list may consist of one or more comma-separated numerical ranges.
`-f` *list*	`--fields=`*list*	Extract one or more fields from the line as defined by *list*. The list may contain one

		or more fields or field ranges separated by commas.
`-d` *`delim`*	`--delimeter=`*`delim`*	When `-f` is specified, use *delim* as the field delimiting character. By default, fields must be separated by a single tab character.
	`--complement`	Extract the entire line of text, except for those portions specified by `-c` and/or `-f`.

As we can see, the way `cut` extracts text is rather inflexible. `cut` is best used to extract text from files that are produced by other programs, rather than text directly typed by humans. We'll take a look at our `distros.txt` file to see whether it is "clean" enough to be a good specimen for our `cut` examples. If we use `cat` with the `-A` option, we can see whether the file meets our requirements of tab-separated fields:

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

It looks good. There are no embedded spaces, just single tab characters between the fields. Since the file uses tabs rather than spaces, we'll use the `-f` option to extract a field.

```
[me@linuxbox ~]$ cut -f 3 distros.txt
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

Because our `distros` file is tab-delimited, it is best to use `cut` to extract fields rather than characters. This is because when a file is tab-delimited, it is unlikely that each line will contain the same number of characters, which makes calculating character positions within the line difficult or impossible. In our previous example, however, we now have extracted a field that luckily contains data of identical length, so we can show how character extraction works by extracting the year from each line.

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
2006
2008
2006
2008
2006
```

By running `cut` a second time on our list, we are able to extract character positions 7 through 10, which corresponds to the year in our date field. The `7-10` notation is an example of a range. The `cut` man page contains a complete description of how ranges can be specified.

Expanding Tabs

Our `distros.txt` file is ideally formatted for extracting fields using `cut`. But what if we wanted a file that could be fully manipulated with `cut` by characters, rather than fields? This would require us to replace the tab characters within the file with the corresponding number of spaces. Fortunately, the GNU Coreutils package includes a tool for that. Named `expand`, this program accepts either one or more file arguments or standard input and outputs the modified text to standard output.

If we process our `distros.txt` file with `expand`, we can use `cut -c` to extract any range of characters from the file. For example, we could use the following command to extract the year of release from our list by expanding the file and using `cut` to extract every character from the 23rd position to the end of the line:

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

Coreutils also provides the `unexpand` program to substitute tabs for spaces.

When working with fields, it is possible to specify a different field delimiter rather than the tab character. Here we will extract the first field from the `/etc/passwd` file:

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
daemon
bin
sys
sync
games
man
lp
mail
news
```

Using the `-d` option, we are able to specify the colon character as the field delimiter.

paste

The `paste` command does the opposite of `cut`. Rather than extracting a column of text from a file, it adds one or more columns of text to a file. It does this by reading multiple files and combining the fields found in each file into a single stream on standard output. Like `cut`, `paste` accepts multiple file arguments and/or standard input. To demonstrate how `paste` operates, we will perform some surgery on our `distros.txt` file to produce a chronological list of releases.

From our earlier work with `sort`, we will first produce a list of distros sorted by date and store the result in a file called `distros-by-date.txt`.

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > dis
tros-by-date.txt
```

Next, we will use `cut` to extract the first two fields from the file (the distro name and version) and store that result in a file named `distro-versions.txt`.

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.t
xt
[me@linuxbox ~]$ head distros-versions.txt
Fedora  10
Ubuntu  8.10
SUSE    11.0
Fedora  9
Ubuntu  8.04
Fedora  8
Ubuntu  7.10
SUSE    10.3
Fedora  7
Ubuntu  7.04
```

The final piece of preparation is to extract the release dates and store them in a file named `distro-dates.txt`.

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
```

```
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

We now have the parts we need. To complete the process, use `paste` to put the column of dates ahead of the distro names and versions, thus creating a chronological list. This is done simply by using `paste` and ordering its arguments in the desired arrangement.

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008   Fedora    10
10/30/2008   Ubuntu    8.10
06/19/2008   SUSE      11.0
05/13/2008   Fedora    9
04/24/2008   Ubuntu    8.04
11/08/2007   Fedora    8
10/18/2007   Ubuntu    7.10
10/04/2007   SUSE      10.3
05/31/2007   Fedora    7
04/19/2007   Ubuntu    7.04
12/07/2006   SUSE      10.2
10/26/2006   Ubuntu    6.10
10/24/2006   Fedora    6
06/01/2006   Ubuntu    6.06
05/11/2006   SUSE      10.1
03/20/2006   Fedora    5
```

join

In some ways, `join` is like `paste` in that it adds columns to a file, but it uses a unique way to do it. A *join* is an operation usually associated with *relational databases* where data from multiple *tables* with a shared key field is combined to form a desired result. The `join` program performs the same operation. It joins data from multiple files based on a shared key field.

To see how a join operation is used in a relational database, let's imagine a small database consisting of two tables, each containing a single record. The first table, called CUSTOMERS, has three fields: a customer number (CUSTNUM), the customer's first name

(FNAME), and the customer's last name (LNAME):

```
CUSTNUM   FNAME     LNAME
========  =====     ======
4681934   John      Smith
```

The second table is called ORDERS and contains four fields: an order number (ORDERNUM), the customer number (CUSTNUM), the quantity (QUAN), and the item ordered (ITEM).

```
ORDERNUM       CUSTNUM   QUAN ITEM
========       =======   ==== ====
3014953305     4681934   1    Blue Widget
```

Note that both tables share the field CUSTNUM. This is important, because it allows a relationship between the tables.

Performing a join operation would allow us to combine the fields in the two tables to achieve a useful result, such as preparing an invoice. Using the matching values in the CUSTNUM fields of both tables, a join operation could produce the following:

```
FNAME     LNAME     QUAN ITEM
=====     =====     ==== ====
John      Smith     1    Blue Widget
```

To demonstrate the `join` program, we'll need to make a couple of files with a shared key. To do this, we will use our `distros-by-date.txt` file. From this file, we will construct two additional files. One contains the release dates (which will be our shared key for this demonstration) and the release names, as shown here

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt > distros-
key-names.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008   Fedora
10/30/2008   Ubuntu
06/19/2008   SUSE
05/13/2008   Fedora
04/24/2008   Ubuntu
11/08/2007   Fedora
10/18/2007   Ubuntu
10/04/2007   SUSE
05/31/2007   Fedora
04/19/2007   Ubuntu
```

The second file contains the release dates and the version numbers, as shown here:

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt > distro
s-key-vernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008   10
10/30/2008   8.10
06/19/2008   11.0
05/13/2008   9
04/24/2008   8.04
11/08/2007   8
10/18/2007   7.10
10/04/2007   10.3
05/31/2007   7
04/19/2007   7.04
```

We now have two files with a shared key (the “release date” field). It is important to point out that the files must be sorted on the key field for `join` to work properly.

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-vernums.txt |
head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
```

Note also that, by default, `join` uses whitespace as the input field delimiter and a single space as the output field delimiter. This behavior can be modified by specifying options. See the `join` man page for details.

Comparing Text

It is often useful to compare versions of text files. For system administrators and software developers, this is particularly important. A system administrator may, for example, need to compare an existing configuration file to a previous version to diagnose a system problem. Likewise, a programmer frequently needs to see what changes have been made to

programs over time.

comm

The comm program compares two text files and displays the lines that are unique to each one and the lines they have in common. To demonstrate, we will create two nearly identical text files using cat.

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d
[me@linuxbox ~]$ cat > file2.txt
b
c
d
e
```

Next, we will compare the two files using comm:

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
        b
        c
        d
   e
```

As we can see, comm produces three columns of output. The first column contains lines unique to the first file argument, the second column contains the lines unique to the second file argument, and the third column contains the lines shared by both files. comm supports options in the form *-n*, where *n* is either 1, 2, or 3. When used, these options specify which columns to suppress. For example, if we wanted to only output the lines shared by both files, we would suppress the output of the first and second columns.

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c
d
```

diff

Like the `comm` program, `diff` is used to detect the differences between files. However, `diff` is a much more complex tool, supporting many output formats and the ability to process large collections of text files at once. `diff` is often used by software developers to examine changes between different versions of program source code and thus has the ability to recursively examine directories of source code, often referred to as *source trees*. One common use for `diff` is the creation of *diff files* or *patches* that are used by programs such as `patch` (which we'll discuss shortly) to convert one version of a file (or files) to another version.

If we use `diff` to look at our previous example files:

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

we see its default style of output: a terse description of the differences between the two files. In the default format, each group of changes is preceded by a *change command* in the form of *range operation range* to describe the positions and types of changes required to convert the first file to the second file, as outlined in Table 20-4.

Table 20-4: `diff` Change Commands

Change	Description
r1`a`*r2*	Append the lines at the position *r2* in the second file to the position *r1* in the first file.
r1`c`*r2*	Change (replace) the lines at position *r1* with the lines at the position *r2* in the second file.
r1`d`*r2*	Delete the lines in the first file at position *r1*, which would have appeared at range *r2* in the second file

In this format, a range is a comma-separated list of the starting line and the ending line. While this format is the default (mostly for POSIX compliance and backward compatibility with traditional Unix versions of `diff`), it is not as widely used as other, optional formats. Two of the more popular formats are the *context format* and the *unified format*.

When viewed using the context format (the `-c` option), we will see this:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt 2008-12-23 06:40:13.000000000 -0500
--- file2.txt 2008-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
- a
  b
  c
  d
--- 1,4 ----
  b
  c
  d
+ e
```

The output begins with the names of the two files and their timestamps. The first file is marked with asterisks and the second file is marked with dashes. Throughout the remainder of the listing, these markers will signify their respective files. Next, we see groups of changes, including the default number of surrounding context lines. In the first group, we see this:

```
*** 1,4 ***
```

which indicates lines 1 through 4 in the first file. Later we see this:

```
--- 1,4 ---
```

which indicates lines 1 through 4 in the second file. Within a change group, lines begin with one of four indicators shown in Table 20-5.

Table 20-5: `diff` Context Format Change Indicators

Indicator	Meaning
`blank`	A line shown for context. It does not indicate a difference between the two files.
`-`	A line deleted. This line will appear in the first file but not in the second file.
`+`	A line added. This line will appear in the second file but not in the first file.
`!`	A line changed. The two versions of the line will be displayed, each in its respective section of the change group.

The unified format is similar to the context format but is more concise. It is specified with the `-u` option.

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt 2008-12-23 06:40:13.000000000 -0500
+++ file2.txt 2008-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
 d
+e
```

The most notable difference between the context and unified formats is the elimination of the duplicated lines of context, making the results of the unified format shorter than those of the context format. In our previous example, we see file timestamps like those of the context format, followed by the string `@@ -1,4 +1,4 @@`. This indicates the lines in the first file and the lines in the second file described in the change group. Following this are the lines themselves, with the default three lines of context. Each line starts with one of three possible characters listed in Table 20-6.

Table 20-6: `diff` Unified Format Change Indicators

Character	Meaning
`blank`	This line is shared by both files.
`-`	This line was removed from the first file.
`+`	This line was added to the first file.

patch

The `patch` program is used to apply changes to text files. It accepts output from `diff` and is generally used to convert older version files into newer versions. Let’s consider a famous example. The Linux kernel is developed by a large, loosely organized team of contributors who submit a constant stream of small changes to the source code. The Linux kernel consists of several million lines of code, while the changes that are made by one contributor at one time are quite small. It makes no sense for a contributor to send each developer an entire kernel source tree each time a small change is made. Instead, a diff file is submitted. The diff file contains the change from the previous version of the kernel to the new version with the contributor's changes. The receiver then uses the

`patch` program to apply the change to his own source tree. Using `diff`/`patch` offers two significant advantages.

1. The diff file is small, compared to the full size of the source tree.
2. The diff file concisely shows the change being made, allowing reviewers of the patch to quickly evaluate it.

Of course, `diff`/`patch` will work on any text file, not just source code. It would be equally applicable to configuration files or any other text.

To prepare a diff file for use with `patch`, the GNU documentation (see Further Reading below) suggests using `diff` as follows:

`diff -Naur` *`old_file new_file > diff_file`*

where *old_file* and *new_file* are either single files or directories containing files. The `r` option supports recursion of a directory tree.

Once the diff file has been created, we can apply it to patch the old file into the new file.

`patch <` *`diff_file`*

We'll demonstrate with our test file.

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt
[me@linuxbox ~]$ patch < patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

In this example, we created a diff file named `patchfile.txt` and then used the `patch` program to apply the patch. Note that we did not have to specify a target file to `patch`, as the diff file (in unified format) already contains the filenames in the header. Once the patch is applied, we can see that `file1.txt` now matches `file2.txt`.

`patch` has a large number of options, and there are additional utility programs that can be used to analyze and edit patches.

Editing on the Fly

Our experience with text editors has been largely *interactive*, meaning that we manually move a cursor around and then type our changes. However, there are *non-interactive* ways to edit text as well. It's possible, for example, to apply a set of changes to multiple

files with a single command.

tr

The `tr` program is used to *transliterate* characters. We can think of this as a sort of character-based search-and-replace operation. Transliteration is the process of changing characters from one alphabet to another. For example, converting characters from lowercase to uppercase is transliteration. We can perform such a conversion with `tr` as follows:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

As we can see, `tr` operates on standard input, and outputs its results on standard output. `tr` accepts two arguments: a set of characters to convert from and a corresponding set of characters to convert to. Character sets may be expressed in one of three ways.

1. An enumerated list. For example, `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
2. A character range. For example, `A-Z`. Note that this method is sometimes subject to the same issues as other commands, because of the locale collation order, and thus should be used with caution.
3. POSIX character classes. For example, `[:upper:]`.

In most cases, both character sets should be of equal length; however, it is possible for the first set to be larger than the second, particularly if we want to convert multiple characters to a single character.

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

In addition to transliteration, `tr` allows characters to simply be deleted from the input stream. Earlier in this chapter, we discussed the problem of converting MS-DOS text files to Unix-style text. To perform this conversion, carriage return characters need to be removed from the end of each line. This can be performed with `tr` as follows:

`tr -d '\r' < dos_file > unix_file`

where *dos_file* is the file to be converted and *unix_file* is the result. This form of the command uses the escape sequence `\r` to represent the carriage return character. To see a complete list of the sequences and character classes `tr` supports, try the following:

```
[me@linuxbox ~]$ tr --help
```

ROT13: The Not-So-Secret Decoder Ring

One amusing use of `tr` is to perform *ROT13 encoding* of text. ROT13 is a trivial type of encryption based on a simple substitution cipher. Calling ROT13 "encryption" is being generous; "text obfuscation" is more accurate. It is used sometimes on text to obscure potentially offensive content. The method simply moves each character 13 places up the alphabet. Since this is half way up the possible 26 characters, performing the algorithm a second time on the text restores it to its original form. Use the following to perform this encoding with `tr`:

```
echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

Performing the same procedure a second time results in the following translation:

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

A number of email programs and Usenet news readers support ROT13 encoding. Wikipedia contains a good article on the subject:

http://en.wikipedia.org/wiki/ROT13

`tr` can perform another trick, too. Using the `-s` option, `tr` can "squeeze" (delete) repeated instances of a character.

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

Here we have a string containing repeated characters. By specifying the set "ab" to `tr`, we eliminate the repeated instances of the letters in the set, while leaving the character that is missing from the set ("c") unchanged. Note that the repeating characters must be adjoining. If they are not, the squeezing will have no effect.

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

sed

The name sed is short for *stream editor*. It performs text editing on a stream of text, either a set of specified files or standard input. sed is a powerful and somewhat complex program (there are entire books about it), so we will not cover it completely here.

In general, the way sed works is that it is given either a single editing command (on the command line) or the name of a script file containing multiple commands, and it then performs these commands upon each line in the stream of text. Here is a simple example of sed in action:

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

In this example, we produce a one-word stream of text using echo and pipe it into sed. sed, in turn, carries out the instruction s/front/back/ upon the text in the stream and produces the output "back" as a result. We can also recognize this command as resembling the "substitution" (search-and-replace) command in vi.

Commands in sed begin with a single letter. In the previous example, the substitution command is represented by the letter s and is followed by the search-and-replace strings, separated by the slash character as a delimiter. The choice of the delimiter character is arbitrary. By convention, the slash character is often used, but sed will accept any character that immediately follows the command as the delimiter. We could perform the same command this way:

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_'
back
```

By using the underscore character immediately after the command, it becomes the delimiter. The ability to set the delimiter can be used to make commands more readable, as we shall see.

Most commands in sed may be preceded by an *address*, which specifies which line(s) of the input stream will be edited. If the address is omitted, then the editing command is carried out on every line in the input stream. The simplest form of address is a line number. We can add one to our example.

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

Adding the address `1` to our command causes our substitution to be performed on the first line of our one-line input stream. If we specify another number and we see that the editing is not carried out, since our input stream does not have a line 2.

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

Addresses may be expressed in many ways. Table 20-7 lists the most common.

Table 20-7: `sed` Address Notation

Address	Description
n	A line number where *n* is a positive integer.
$	The last line.
/regexp/	Lines matching a POSIX basic regular expression. Note that the regular expression is delimited by slash characters. Optionally, the regular expression may be delimited by an alternate character, by specifying the expression with `\c`*regexp*`c`, where *c* is the alternate character.
addr1,*addr2*	A range of lines from *addr1* to *addr2*, inclusive. Addresses may be any of the single address forms listed earlier.
first~*step*	Match the line represented by the number *first*, then each subsequent line at *step* intervals. For example 1~2 refers to each odd numbered line, and 5~5 refers to the fifth line and every fifth line thereafter.
addr1,+n	Match *addr1* and the following *n* lines.
addr!	Match all lines except *addr*, which may be any of the forms listed earlier.

We'll demonstrate different kinds of addresses using the `distros.txt` file from earlier in this chapter. First, here's a range of line numbers:

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE    10.2     12/07/2006
Fedora  10       11/25/2008
SUSE    11.0     06/19/2008
```

```
Ubuntu  8.04     04/24/2008
Fedora  8    11/08/2007
```

In this example, we print a range of lines, starting with line 1 and continuing to line 5. To do this, we use the `p` command, which simply causes a matched line to be printed. For this to be effective, however, we must include the option `-n` (the "no auto-print" option) to cause `sed` not to print every line by default.

Next, we'll try a regular expression.

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE    10.2     12/07/2006
SUSE    11.0     06/19/2008
SUSE    10.3     10/04/2007
SUSE    10.1     05/11/2006
```

By including the slash-delimited regular expression `/SUSE/`, we are able to isolate the lines containing it in much the same manner as `grep`.

Finally, we'll try negation by adding an exclamation point (`!`) to the address.

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora  10       11/25/2008
Ubuntu  8.04     04/24/2008
Fedora  8        11/08/2007
Ubuntu  6.10     10/26/2006
Fedora  7        05/31/2007
Ubuntu  7.10     10/18/2007
Ubuntu  7.04     04/19/2007
Fedora  6        10/24/2006
Fedora  9        05/13/2008
Ubuntu  6.06     06/01/2006
Ubuntu  8.10     10/30/2008
Fedora  5        03/20/2006
```

Here we see the expected result: all the lines in the file except the ones matched by the regular expression.

So far, we've looked at two of the `sed` editing commands, `s` and `p`. Table 20-8 provides a more complete list of the basic editing commands.

Table 20-8: `sed` Basic Editing Commands

Command	Description
`=`	Output the current line number.
`a`	Append text after the current line.
`d`	Delete the current line.
`i`	Insert text in front of the current line.
`p`	Print the current line. By default, `sed` prints every line and only edits lines that match a specified address within the file. The default behavior can be overridden by specifying the `-n` option.
`q`	Exit `sed` without processing any more lines. If the `-n` option is not specified, output the current line.
`Q`	Exit `sed` without processing any more lines.
`s/`*`regexp/replacement`*`/`	Substitute the contents of *replacement* wherever *regexp* is found. *replacement* may include the special character `&`, which is equivalent to the text matched by *regexp*. In addition, *replacement* may include the sequences `\1` through `\9`, which are the contents of the corresponding subexpressions in *regexp*. For more about this, see the discussion of *back references* below. After the trailing slash following *replacement*, an optional flag may be specified to modify the `s` command's behavior.
`y/`*`set1/set2`*	Perform transliteration by converting characters from *set1* to the corresponding characters in *set2*. Note that unlike `tr`, `sed` requires that both sets be of the same length.

The `s` command is by far the most commonly used editing command. We will demonstrate just some of its power by performing an edit on our `distros.txt` file. We discussed earlier how the date field in `distros.txt` was not in a "computer-friendly" format. While the date is formatted MM/DD/YYYY, it would be better (for ease of sorting) if the format were YYYY-MM-DD. Performing this change on the file by hand would be both time consuming and error prone, but with `sed`, this change can be performed in one step.

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\
)$/\3-\1-\2/' distros.txt
SUSE    10.2     2006-12-07
Fedora  10       2008-11-25
SUSE    11.0     2008-06-19
Ubuntu  8.04     2008-04-24
Fedora  8        2007-11-08
SUSE    10.3     2007-10-04
Ubuntu  6.10     2006-10-26
Fedora  7        2007-05-31
Ubuntu  7.10     2007-10-18
Ubuntu  7.04     2007-04-19
SUSE    10.1     2006-05-11
Fedora  6        2006-10-24
Fedora  9        2008-05-13
Ubuntu  6.06     2006-06-01
Ubuntu  8.10     2008-10-30
Fedora  5        2006-03-20
```

Wow! Now that is an ugly looking command. But it works. In just one step, we have changed the date format in our file. It is also a perfect example of why regular expressions are sometimes jokingly referred to as a “write-only” medium. We can write them, but we sometimes cannot read them. Before we are tempted to run away in terror from this command, let’s look at how it was constructed. First, we know that the command will have this basic structure.

```
sed 's/regexp/replacement/' distros.txt
```

Our next step is to figure out a regular expression that will isolate the date. Because it is in MM/DD/YYYY format and appears at the end of the line, we can use an expression like this:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

This matches two digits, a slash, two digits, a slash, four digits, and the end of line. So that takes care of *regexp*, but what about *replacement*? To handle that, we must introduce a new regular expression feature that appears in some applications that use BRE. This feature is called *back references* and works like this: if the sequence *n* appears in *replacement* where *n* is a number from 1 to 9, the sequence will refer to the corresponding subexpression in the preceding regular expression. To create the subexpressions, we sim-

ply enclose them in parentheses like so:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

We now have three subexpressions. The first contains the month, the second contains the day of the month, and the third contains the year. Now we can construct *replacement* as follows:

```
\3-\1-\2
```

This gives us the year, a dash, the month, a dash, and the day.

Now, our command looks like this:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

We have two remaining problems. The first is that the extra slashes in our regular expression will confuse `sed` when it tries to interpret the `s` command. The second is that since `sed`, by default, accepts only basic regular expressions, several of the characters in our regular expression will be taken as literals, rather than as metacharacters. We can solve both these problems with a liberal application of backslashes to escape the offending characters.

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' dis
tros.txt
```

And there you have it!

Another feature of the `s` command is the use of optional flags that may follow the replacement string. The most important of these is the `g` flag, which instructs `sed` to apply the search-and-replace globally to a line, not just to the first instance, which is the default. Here is an example:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

We see that the replacement was performed, but only to the first instance of the letter `b`, while the remaining instances were left unchanged. By adding the `g` flag, we are able to

change all the instances.

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

So far, we have only given `sed` single commands via the command line. It is also possible to construct more complex commands in a script file using the `-f` option. To demonstrate, we will use `sed` with our `distros.txt` file to build a report. Our report will feature a title at the top, our modified dates, and all the distribution names converted to uppercase. To do this, we will need to write a script, so we'll fire up our text editor and enter the following:

```
# sed script to produce Linux distributions report

1 i\
\
Linux Distributions Report\

s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

We will save our `sed` script as `distros.sed` and run it like this:

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt

Linux Distributions Report

SUSE    10.2     2006-12-07
FEDORA  10       2008-11-25
SUSE    11.0     2008-06-19
UBUNTU  8.04     2008-04-24
FEDORA  8        2007-11-08
SUSE    10.3     2007-10-04
UBUNTU  6.10     2006-10-26
FEDORA  7        2007-05-31
UBUNTU  7.10     2007-10-18
UBUNTU  7.04     2007-04-19
SUSE    10.1     2006-05-11
FEDORA  6        2006-10-24
FEDORA  9        2008-05-13
```

```
UBUNTU  6.06     2006-06-01
UBUNTU  8.10     2008-10-30
FEDORA  5        2006-03-20
```

As we can see, our script produces the desired results, but how does it do it? Let's take another look at our script. We'll use `cat` to number the lines.

```
[me@linuxbox ~]$ cat -n distros.sed
    1   # sed script to produce Linux distributions report
    2
    3   1 i\
    4   \
    5   Linux Distributions Report\
    6
    7   s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
    8   y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Line one of our script is a *comment*. Like many configuration files and programming languages on Linux systems, comments begin with the # character and are followed by human-readable text. Comments can be placed anywhere in the script (though not within commands themselves) and are helpful to any humans who might need to identify and/or maintain the script.

Line 2 is a blank line. Like comments, blank lines may be added to improve readability.

Many `sed` commands support line addresses. These are used to specify which lines of the input are to be acted upon. Line addresses may be expressed as single line numbers, line number ranges, and the special line number `$`, which indicates the last line of input.

Lines 3, 4, 5, and 6 contain text to be inserted at the address 1, the first line of the input. The `i` command is followed by the sequence of a backslash and then a carriage return to produce an escaped carriage return, or what is called a *line-continuation character*. This sequence, which can be used in many circumstances including shell scripts, allows a carriage return to be embedded in a stream of text without signaling the interpreter (in this case `sed`) that the end of the line has been reached. The `i`, and the `a` (which appends text, rather than inserting it) and `c` (which replaces text) commands allow multiple lines of text as long as each line, except the last, ends with a line-continuation character. The sixth line of our script is actually the end of our inserted text and ends with a plain carriage return rather than a line-continuation character, signaling the end of the `i` command.

> **Note:** A line-continuation character is formed by a backslash followed *immediately* by a carriage return. No intermediary spaces are permitted.

Line 7 is our search-and-replace command. Since it is not preceded by an address, each line in the input stream is subject to its action.

Line 8 performs transliteration of the lowercase letters into uppercase letters. Note that unlike `tr`, the `y` command in `sed` does not support character ranges (for example, `[a-z]`), nor does it support POSIX character classes. Again, since the `y` command is not preceded by an address, it applies to every line in the input stream.

People Who Like sed Also Like...

`sed` is a capable program, able to perform fairly complex editing tasks to streams of text. It is most often used for simple, one-line tasks rather than long scripts. Many users prefer other tools for larger tasks. The most popular of these are `awk` and `perl`. These go beyond mere tools like the programs covered here and extend into the realm of complete programming languages. `perl`, in particular, is often used instead of shell scripts for many system management and administration tasks, as well as being a popular medium for web development. `awk` is a little more specialized. Its specific strength is its ability to manipulate tabular data. It resembles `sed` in that `awk` programs normally process text files line by line, using a scheme similar to the `sed` concept of an address followed by an action. While both `awk` and `perl` are outside the scope of this book, they are good skills for the Linux command line user to learn.

`aspell`

The last tool we will look at is `aspell`, an interactive spelling checker. The `aspell` program is the successor to an earlier program named `ispell` and can be used, for the most part, as a drop-in replacement. While the `aspell` program is mostly used by other programs that require spell-checking capability, it can also be used effectively as a standalone tool from the command line. It has the ability to intelligently check various types of text files, including HTML documents, C/C++ programs, email messages, and other kinds of specialized texts.

To spellcheck a text file containing simple prose, it could be used like this:

```
aspell check textfile
```

where *textfile* is the name of the file to check. As a practical example, let's create a simple text file named `foo.txt` containing some deliberate spelling errors.

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jimped over the laxy dog.
```

Next we'll check the file using `aspell`:

```
[me@linuxbox ~]$ aspell check foo.txt
```

As `aspell` is interactive in the check mode, we will see a screen like this:

```
The quick brown fox jimped over the laxy dog.

1) jumped                                     6) wimped
2) gimped                                     7) camped
3) comped                                     8) humped
4) limped                                     9) impede
5) pimped                                     0) umped
i) Ignore                                     I) Ignore all
r) Replace                                    R) Replace all
a) Add                                        l) Add Lower
b) Abort                                      x) Exit
?
```

At the top of the display, we see our text with a suspiciously spelled word highlighted. In the middle, we see ten spelling suggestions numbered zero through nine, followed by a list of other possible actions. Finally, at the bottom, we see a prompt ready to accept our choice.

If we press the `1` key, `aspell` replaces the offending word with the word "jumped" and moves on to the next misspelled word, which is laxy. If we select the replacement lazy, `aspell` replaces it and terminates. Once `aspell` has finished, we can examine our file and see that the misspellings have been corrected:

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

Unless told otherwise via the command line option `--dont-backup`, `aspell` creates a backup file containing the original text by appending the extension `.bak` to the filename.

Showing off our `sed` editing prowess, we'll put our spelling mistakes back in so we can reuse our file.

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

The `sed` option `-i` tells `sed` to edit the file "in-place," meaning that rather than sending the edited output to standard output, it will rewrite the file with the changes applied. We also see the ability to place more than one editing command on the line by separating them with a semicolon.

Next, we'll look at how `aspell` can handle different kinds of text files. Using a text editor such as `vim` (the adventurous may want to try `sed`), we will add some HTML markup to our file.

```
<html>
    <head>
        <title>Mispelled HTML file</title>
    </head>
    <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
    </body>
</html>
```

Now, if we try to spellcheck our modified file, we run into a problem. If we do it this way:

```
[me@linuxbox ~]$ aspell check foo.txt
```

we'll get this:

```
<html>
```

```
        <head>
                <title>Mispelled HTML file</title>
        </head>
        <body>
                <p>The quick brown fox jimped over the laxy dog.</p>
        </body>
</html>

1) HTML                                 4) Hamel
2) ht ml                                5) Hamil
3) ht-ml                                6) hotel
i) Ignore                               I) Ignore all
r) Replace                              R) Replace all
a) Add                                  l) Add Lower
b) Abort                                x) Exit
?
```

`aspell` will see the contents of the HTML tags as misspelled. This problem can be overcome by including the `-H` (HTML) checking-mode option, like this:

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

which will result in this:

```
<html>
        <head>
                <title>Mispelled HTML file</title>
        </head>
        <body>
                <p>The quick brown fox jimped over the laxy dog.</p>
        </body>
</html>

1) Mi spelled                           6) Misapplied
2) Mi-spelled                           7) Miscalled
3) Misspelled                           8) Respelled
4) Dispelled                            9) Misspell
5) Spelled                              0) Misled
```

```
i) Ignore                                I) Ignore all
r) Replace                               R) Replace all
a) Add                                   l) Add Lower
b) Abort                                 x) Exit
                                         
?
```

The HTML is ignored, and only the non-markup portions of the file are checked. In this mode, the contents of HTML tags are ignored and not checked for spelling. However, the contents of `ALT` tags, which benefit from checking, are checked in this mode.

> **Note:** By default, `aspell` will ignore URLs and email addresses in text. This behavior can be overridden with command line options. It is also possible to specify which markup tags are checked and skipped. See the `aspell` man page for details.

Summing Up

In this chapter, we looked at a few of the many command line tools that operate on text. In the next chapter, we will look at several more. Admittedly, it may not seem immediately obvious how or why you might use some of these tools on a day-to-day basis, though we have tried to show some practical examples of their use. We will find in later chapters that these tools form the basis of a tool set that is used to solve a host of practical problems. This will be particularly true when we get into shell scripting, where these tools will really show their worth.

Further Reading

The GNU Project website contains many online guides to the tools discussed in this chapter.

- From the Coreutils package:
  http://www.gnu.org/software/coreutils/manual/coreutils.html#Output-of-entire-files
  http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-sorted-files
  http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-fields
  http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-characters
- From the Diffutils package:

http://www.gnu.org/software/diffutils/manual/html_mono/diff.html

- `sed`:
  http://www.gnu.org/software/sed/manual/sed.html
- `aspell`:
  http://aspell.net/man-html/index.html
- There are many other online resources for `sed`, in particular:
  http://www.grymoire.com/Unix/Sed.html
  http://sed.sourceforge.net/sed1line.txt
- Also try googling “sed one liners”, “sed cheat sheets”

Extra Credit

There are a few more interesting text-manipulation commands worth investigating. Among these are `split` (split files into pieces), `csplit` (split files into pieces based on context), and `sdiff` (side-by-side merge of file differences).