

Join the discussion @ p2p.wrox.com



Wrox Programmer to Programmer™



Shell Scripting

Expert Recipes for Linux®, Bash, and More

Steve Parker

PART I

About the Ingredients

- ▶ **CHAPTER 1:** The History of Unix, GNU, and Linux
- ▶ **CHAPTER 2:** Getting Started
- ▶ **CHAPTER 3:** Variables
- ▶ **CHAPTER 4:** Wildcard Expansion
- ▶ **CHAPTER 5:** Conditional Execution
- ▶ **CHAPTER 6:** Flow Control Using Loops
- ▶ **CHAPTER 7:** Variables Continued
- ▶ **CHAPTER 8:** Functions and Libraries
- ▶ **CHAPTER 9:** Arrays
- ▶ **CHAPTER 10:** Processes
- ▶ **CHAPTER 11:** Choosing and Using Shells

1

The History of Unix, GNU, and Linux

The Unix tradition has a long history, and Linux comes from the Unix tradition, so to understand Linux one must understand Unix and to understand Unix one must understand its history. Before Unix, a developer would submit a stack of punched cards, each card representing a command, or part of a command. These cards would be read and executed sequentially by the computer. The developer would receive the generated output after the job had completed. This would often be a few days after the job had been submitted; if there was an error in the code, the output was just the error and the developer had to start again. Later, teletype and various forms of timesharing systems sped up the matter considerably, but the model was basically the same: a sequence of characters (punch cards, or keys on keyboards — it's still just a string of characters) submitted as a batch job to be run (or fail to run), and for the result to come back accordingly. This is significant today in that it is still how data is transmitted on any computerized system — it's all sequences of characters, transmitted in order. Whether a text file, a web page, a movie, or music, it is all just strings of ones and zeroes, same as it ever was. Anything that looks even slightly different is simply putting an interface over the top of a string of ones and zeroes.

Unix and various other interactive and timesharing systems came along in the mid-1960s. Unix and its conventions continue to be central to computing practices today; its influences can be seen in DOS, Linux, Mac OS X, and even Microsoft Windows.

UNIX

In 1965, Bell Labs and GE joined a Massachusetts Institute of Technology (MIT) project known as MULTICS, the Multiplexed Information and Computing System. Multics was intended to be a stable, timesharing OS. The “Multiplexed” aspect added unnecessary complexity, which eventually led Bell Labs to abandon the project in 1969. Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna retained some of the ideas behind it, took out a lot of the complexity, and came up with Unix (a play on the word MULTICS, as this was a simplified operating system inspired by MULTICS).

An early feature of Unix was the introduction of *pipes* — something that Doug McIlroy had been thinking about for a few years and was implemented in Unix by Ken Thompson. Again, it took the same notion of streamed serial data, but pipes introduced the idea of having `stdin` and `stdout`, through which the data would flow. Similar things had been done before, and the concept is fairly simple: One process creates output, which becomes input to another command. The Unix pipes method introduced a concept that dramatically affected the design of the rest of the system.

Most commands have a file argument as well, but existing commands were modified to default to read from their “Standard Input” (`stdin`) and “Standard Output” (`stdout`); the pipe can then “stream” the data from one tool to another. This was a novel concept, and one that strongly defines the Unix shell; it makes the whole system a set of generically useful tools, as opposed to monolithic, single-purpose applications. This has been summarized as “do one thing and do it well.” The GNU toolchain was written to replace Unix while maintaining compatibility with Unix tools. The developers on the GNU project often took the opportunity presented by rewriting the tool to include additional functionality, while still sticking to the “do one thing and do it well” philosophy.



The GNU project was started in 1983 by Richard Stallman, with the intention of replacing proprietary commercial Unices with Free Software alternatives. GNU had all but completed the task of replacing all of the userspace tools by the time the Linux kernel project started in 1991. In fact, the GNU tools generally perform the same task at least as well as their original Unix equivalents, often providing extra useful features borne of experience in the real world. Independent testing has shown that GNU tools can actually be more reliable than their traditional Unix equivalents (<http://www.gnu.org/software/reliability.html>).

For example, the `who` command lists who is logged in to the system, one line per logged-in session. The `wc` command counts characters, words, and lines. Therefore, the following code will tell you how many people are logged in:

```
who | wc -l
```

There is no need for the `who` tool to have an option to count the logged-in users because the generic `wc` tool can do that already. This saves some small effort in `who`, but when that is applied across the whole range of tools, including any new tools that might be written, a lot of effort and therefore complexity, which means a greater likelihood of the introduction of additional bugs, is avoided. When this is applied to more complicated tools, such as `grep` or even `more`, the flexibility of the system is increased with every added tool.



In the case of `more`, this is actually more tricky than it seems; first it has to find out how many columns and rows are available. Again, there is a set of tools that combine to provide this information. In this way, every tool in the chain can be used by the other tools.

Also this system means that you do not have to learn how each individual utility implements its “word count” feature. There are a few defacto standard switches; `-q` typically means Quiet, `-v` typically means Verbose, and so on, but if `who -c` meant “count the number of entries,” then `cut -c <n>`, which means “cut the first n characters,” would be inconsistent. It is better that each tool does its own job, and that `wc` do the counting for all of them.

For a more involved example, the `sort` utility just sorts text. It can sort alphabetically or numerically (the difference being that “10” comes before “9” alphabetically, but after it when sorted numerically), but it doesn’t search for content or display a page at a time. `grep` and `more` can be combined with `sort` to achieve this in a pipeline:

```
grep foo /path/to/file | sort -n -k 3 | more
```

This pipeline will search for `foo` in `/path/to/file`. The output (`stdout`) from that command will then be fed into the `stdin` of the `sort` command. Imagine a garden hose, taking the output from `grep` and attaching it to the input for `sort`. The `sort` utility takes the filtered list from `grep` and outputs the sorted results into the `stdin` of `more`, which reads the filtered and sorted data and paginates it.

It is useful to understand exactly what happens here; it is the opposite of what one might intuitively assume. First, the `more` tool is started. Its input is attached to a pipe. Then `sort` is started, and its output is attached to that pipe. A second pipe is created, and the `stdin` for `sort` is attached to that. `grep` is then run, with its `stdout` attached to the pipe that will link it to the `sort` process.

When `grep` begins running and outputting data, that data gets fed down the pipe into `sort`, which sorts its input and outputs down the pipe to `more`, which paginates the whole thing. This can affect what happens in case of an error; if you mistype “`more`,” then nothing will happen. If you mistype “`grep`,” then `more` and `sort` will have been started by the time the error is detected. In this example, that does not matter, but if commands further down the pipeline have some kind of permanent effect (say, if they create or modify a file), then the state of the system will have changed, even though the whole pipeline was never executed.

“Everything Is a File” and Pipelines

There are a few more key concepts that grew into Unix as well. One is the famous “everything is a file” design, whereby device drivers, directories, system configuration, kernel parameters, and processes are all represented as files on the filesystem. Everything, whether a plain-text file (for example, `/etc/hosts`), a block or character special device driver (for example, `/dev/sda`), or kernel state and configuration (for example, `/proc/cpuinfo`) is represented as a file.

The existence of pipes leads to a system whereby tools are written to assume that they will be handling streams of text, and indeed, most of the system configuration is in text form also. Configuration files can be sorted, searched, reformatted, even differentiated and recombined, all using existing tools.

The “everything is a file” concept and the four operations (`open`, `close`, `read`, `write`) that are available on the file mean that Unix provides a really clean, simple system design. Shell scripts themselves are another example of a system utility that is also text. It means that you can write programs like this:

```
#!/bin/sh
cat $0
echo "===="
tac $0
```

This code uses the `cat` facility, which simply outputs a file, and the `tac` tool, which does the same but reverses it. (The name is therefore quite a literal interpretation of what the tool does, and quite a typical example of Unix humor.) The variable `$0` is a special variable, defined by the system, and contains the name of the currently running program, as it was called.

So the output of this command is as follows:

```
#!/bin/sh
cat $0
echo "===="
tac $0
=====
tac $0
echo "===="
cat $0
#!/bin/sh
```

The first four lines are the result of `cat`, the fifth line is the result of the `echo` statement, and the final four lines are the output of `tac`.

BSD

AT&T/Bell Labs couldn't sell Unix because it was a telecommunications monopoly, and as such was barred from extending into other industries, such as computing. So instead, AT&T gave Unix away, particularly to universities, which were naturally keen to get an operating system at no cost. The fact that the schools could also get the source code was an extra benefit, particularly for administrators but also for the students. Not only could users and administrators run the OS, they could see (and modify) the code that made it work. Providing access to the source code was an easy choice for AT&T; they were not (at that stage) particularly interested in developing and supporting it themselves, and this way users could support themselves. The end result was that many university graduates came into the industry with Unix experience, so when they needed an OS for work, they suggested Unix. The use of Unix thus spread because of its popularity with users, who liked its clean design, and because of the way it happened to be distributed.

Although it was often given away at no cost or low cost and included the source code, Unix was not Free Software according to the Free Software Foundation's definition, which is about freedom, not cost. The Unix license prohibited redistribution of Unix to others, although many users developed their own patches, and some of those shared patches with fellow Unix licensees. (The patches would be useless to someone who didn't already have a Unix license from AT&T. The core software was still Unix; any patches were simply modifications to that.) Berkeley Software Distribution (BSD) of the University of California at Berkeley created and distributed many such patches, fixing bugs, adding features, and just generally improving Unix. The terms "Free Software" and "Open Source" would not exist for a long time to come, but all this was distributed on the understanding that if something is useful, then it may as well be shared. TCP/IP, the two core protocols of the Internet, came into Unix via BSD, as did BIND, the DNS (Domain Name System) server, and the Sendmail MTA (mail transport agent). Eventually, BSD developed so many patches to Unix that the project had replaced virtually all of the original Unix source code. After a lawsuit, AT&T and BSD made peace and agreed that the few remaining AT&T components of BSD would be rewritten or relicensed so that BSD was not the property of AT&T, and could be distributed in its own right. BSD has since forked into NetBSD, OpenBSD, FreeBSD, and other variants.

GNU

As mentioned previously, the GNU project was started in 1983 as a response to the closed source software that was by then being distributed by most computer manufacturers along with their hardware. Previously, there had generally been a community that would share source code among users, such that if anyone felt that an improvement could be made, they were free to fix the code to work as they would like. This hadn't been enshrined in any legally binding paperwork; it was simply the culture in which developers naturally operated. If someone expressed an interest in a piece of software, why would you not give him a copy of it (usually in source code form, so that he could modify it to work on his system? Very few installations at the time were sufficiently similar to assume that a binary compiled on one machine would run on another). As Stallman likes to point out, "Sharing of software...is as old as computers, just as sharing of recipes is as old as cooking."¹

Stallman had been working on the Incompatible Timesharing System (ITS) with other developers at MIT through the 1970s and early 1980s. As that generation of hardware died out, newer hardware came out, and — as the industry was developing and adding features — these new machines came with bespoke operating systems. Operating systems, at the time, were usually very hardware-specific, so ITS and CTSS died as the hardware they ran on were replaced by newer designs.



ITS was a pun on IBM's Compatible Time Sharing System (CTSS), which was also developed at MIT around the same time. The "C" in CTSS highlighted the fact that it was somewhat compatible with older IBM mainframes. By including "Incompatible" in its name, ITS gloried in its rebellious incompatibility.

Stallman's turning point occurred when he wanted to fix a printer driver, such that when the printer jammed (which it often did), it would alert the user who had submitted the job, so that she could fix the jam. The printer would then be available for everyone else to use. The user whose job had jammed the printer wouldn't get her output until the problem was fixed, but the users who had submitted subsequent jobs would have to wait even longer. The frustration of submitting a print job, then waiting a few hours (printers were much slower then), only to discover that the printer had already stalled before you had even submitted your own print job, was too much for the users at MIT, so Stallman wanted to fix the code. He didn't expect the original developers to work on this particular feature for him; he was happy to make the changes himself, so he asked the developers for a copy of the source code. He was refused, as the driver software contained proprietary information about how the printer worked, which could be valuable competitive information to other printer manufacturers.

What offended Stallman was not the feature itself, it was that one developer was refusing to share code with another developer. That attitude was foreign to Stallman, who had taken sharing of code for granted until that stage. The problem was that the software — in particular the printer driver — was not as free (it didn't convey the same freedoms) as previous operating systems that Stallman had worked with. This problem prevailed across the industry; it was not specific to one particular platform, so changing hardware would not fix the problem.

¹Free Software, Free Society, 2002, Chapter 1. ISBN 1-882114-98-1



GNU stands for “GNU’s Not Unix,” which is a recursive acronym; if you expand the acronym “IBM,” you get “International Business Machines,” and you’re done. If you expand “GNU,” you get “GNU’s Not Unix’s Not Unix.” Expand that, and you get “GNU’s Not Unix’s Not Unix’s Not Unix” and so on. This is an example of “hacker humor,” which is usually quite a dry sense of humor, with something a little bit clever or out of the ordinary about it. At the bottom of the grep manpage, under the section heading “NOTES” is a comment: “GNU’s not Unix, but Unix is a beast; its plural form is Unixen,” a friendly dig at Unix.

Richard Stallman is a strong-willed character (he has described himself as “borderline autistic”), with a very logical mind, and he determined to fix the problem in the only way he knew how: by making a new operating system that would maintain the old unwritten freedoms to allow equal access to the system, including the code that makes it run. As no such thing existed at the time, he would have to write it. So he did.

STALLMAN CHARGES AHEAD!

From CSvax:pur-ee:inuxc!ixn5c!ihnp4!houxm!mhuxi!eagle!mit-vax!mit-eddie!RMS@MIT-OZ

Newsgroups: net.unix-wizards,net.usoft

Organization: MIT AI Lab, Cambridge, MA

From: RMS%MIT-OZ@mit-eddie

Subject: new Unix implementation

Date: Tue, 27-Sep-83 12:35:59 EST

Free Unix!

Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (for Gnu’s Not Unix), and give it away free to everyone who can use it. Contributions of time, money, programs and equipment are greatly needed.

To begin with, GNU will be a kernel plus all the utilities needed to write and run C programs: editor, shell, C compiler, linker, assembler, and a few other things. After this we will add a text formatter, a YACC, an Empire game, a spreadsheet, and hundreds of other things. We hope to supply, eventually, everything useful that normally comes with a Unix system, and anything else useful, including on-line and hardcopy documentation.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer filenames, file version

numbers, a crashproof file system, filename completion perhaps, terminal-independent display support, and eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will have network software based on MIT's chaosnet protocol, far superior to UUCP. We may also have something compatible with UUCP.

Who Am I?

I am Richard Stallman, inventor of the original much-imitated EMACS editor, now at the Artificial Intelligence Lab at MIT. I have worked extensively on compilers, editors, debuggers, command interpreters, the Incompatible Timesharing System and the Lisp Machine operating system. I pioneered terminal-independent display support in ITS. In addition I have implemented one crashproof file system and two window systems for Lisp machines.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. I cannot in good conscience sign a nondisclosure agreement or a software license agreement.

So that I can continue to use computers without violating my principles, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One computer manufacturer has already offered to provide a machine. But we could use more. One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machine had better be able to operate in a residential area, and not require sophisticated cooling or power.

Individual programmers can contribute by writing a compatible duplicate of some Unix utility and giving it to me. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. Most interface specifications are fixed by Unix compatibility. If each contribution works with the rest of Unix, it will probably work with the rest of GNU.

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high, but I'm looking for people for whom knowing they are helping humanity is as important as money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

For more information, contact me.

Unix already existed, was quite mature, and was nicely modular. So the GNU project was started with the goal of replacing the userland tools of Unix with Free Software equivalents. The kernel was another part of the overall goal, although one can't have a kernel in isolation — the kernel needs an editor, a compiler, and a linker to be built, and some kind of initialization process in order to boot. So existing proprietary software systems were used to assemble a free ecosystem sufficient to further develop itself, and ultimately to compile a kernel. This subject had not been ignored; the Mach microkernel had been selected in line with the latest thinking on operating system kernel design, and the HURD kernel has been available for quite some time, although it has been overtaken by a newer upstart kernel, which was also developed under, and can also work with, the GNU tools.



HURD is “Hird of Unix-Replacing Daemons,” because its microkernel approach uses multiple userspace background processes (known as daemons in the Unix tradition) to achieve what the Unix kernel does in one monolithic kernel. HIRD in turn stands for “Hurd of Interfaces Representing Depth.” This is again a recursive acronym, like GNU (“GNU’s Not Unix”) but this time it is a pair of mutually recursive acronyms. It is also a play on the word “herd,” the collective noun for Gnu.

As the unwritten understandings had failed, Stallman would need to create a novel way to ensure that freely distributable software remained that way. The GNU General Public License (GPL) provided that in a typically intelligent style. The GPL uses copyright to ensure that the license itself cannot be changed; the rest of the license then states that the recipient has full right to the code, so long as he grants the same rights to anybody he distributes it to (whether modified or not) and the license does not change. In that way, all developers (and users) are on a level playing field, where the code is effectively owned by all involved, but no one can change the license, which ensures that equality. The creator of a piece of software may dual-license it, under the GPL and a more restrictive license; this has been done many times — for example, by the MySQL project.

One of the tasks taken on by the GNU project was — of course — to write a shell interpreter as free software. Brian Fox wrote the bash (Bourne Again SHell) shell — its name comes from the fact that the original /bin/sh was written by Steve Bourne, and is known as the Bourne Shell. As bash takes the features of the Bourne shell, and adds new features, too, bash is, obviously, the Bourne Again Shell. Brian also wrote the readline utility, which offers flexible editing of input lines of text before submitting them for parsing. This is probably the most significant feature to make bash a great interactive shell. Brian Fox was the first employee of the Free Software Foundation, the entity set up to coordinate the GNU project.



You’ve probably spotted the pattern by now; although bash isn’t a recursive acronym, its name is a play on the fact that it’s based on the Bourne shell. It also implies that bash is an improvement on the original Bourne shell, in having been “bourne again.”

LINUX

Linus Torvalds, a Finnish university student, was using Minix, a simple Unix clone written by Vrije Universiteit (Amsterdam) lecturer Andrew Tanenbaum, but Torvalds was frustrated by its lack of features and the fact that it did not make full use of the (still relatively new) Intel 80386 processor, and in particular its “protected mode,” which allows for much better separation between the kernel and userspace. Relatively quickly, he got a working shell, and then got GCC, the GNU C compiler (now known as the GNU Compiler Collection, as it has been extended to compile various flavors of C, Fortran, Java, and Ada) working. At that stage, the kernel plus shell plus compiler was enough to be able to “bootstrap” the system — it could be used to build a copy of itself.

TORVALDS' NEWSGROUP POST

On August 25, 1991, Torvalds posted the following to the MINIX newsgroup `comp.os.minix`:

From: torvalds@klaava.helsinki.fi (Linus Benedict Torvalds)
To: Newsgroups: `comp.os.minix`
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system

Hello everybody out there using minix-

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386 (486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-sytem due to practical reasons) among other things.

I've currently ported bash (1.08) an gcc (1.40), and things seem to work. This implies that i'll get something practical within a few months, and I'd like to know what features most people want.

Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus Torvalds `torvalds@kruuna.helsinki.fi`

What is interesting is that Torvalds took the GNU project's inevitable success for granted; it had been going for eight years, and had basically implemented most of its goals (bar the kernel). Torvalds also, after initially making the mistake of trying to write his own license (generally inadvisable for those of us who are not experts in the minutiae of international application of intellectual property law), licensed the kernel under the GNU GPL (version 2) as a natural license for the project.

In practice, this book is far more about shell scripting with Unix and GNU tools than specifically about shell scripting under the Linux kernel; in general, the majority of the tools referred to are GNU tools from the Free Software Foundation: `grep`, `ls`, `find`, `less`, `sed`, `awk`, `bash` itself of course, `diff`, `basename`, and `dirname`; most of the critical commands for shell scripting on Linux

are GNU tools. As such, some people prefer to use the phrase “GNU/Linux” to describe the combination of GNU userspace plus Linux kernel. For the purposes of this book, the goal is to be technically accurate while avoiding overly political zeal. RedHat Linux is what RedHat calls its distribution, so it is referred to as RedHat Linux. Debian GNU/Linux prefers to acknowledge the GNU content so we will, too, when referring specifically to Debian. When talking about the Linux kernel, we will say “Linux”; when talking about a GNU tool we will name it as such. Journalists desperate for headlines can occasionally dream up a far greater rift than actually exists in the community. Like any large family, it has its disagreements — often loudly and in public — but we will try not to stoke the fire here.



Unix was designed with the assumption that it would be operated by engineers; that if somebody wanted to achieve anything with it, he or she would be prepared to learn how the system works and how to manipulate it. The elegant simplicity of the overall design (“everything is a file,” “do one thing and do it well,” etc.) means that principles learned in one part of the system can be applied to other parts.

The rise in popularity of GNU/Linux systems, and in particular, their relatively widespread use on desktop PCs and laptop systems — not just servers humming away to themselves in dark datacenters — has brought a new generation to a set of tools built on this shared philosophy, but without necessarily bringing the context of history into the equation.

Microsoft Windows has a very different philosophy: The end users need not concern themselves with how the underlying system works, and as a result, should not expect it to be discernable, even to an experienced professional, because of the closed-source license of the software. This is not a difference in quality or even quantity; this is a different approach, which assumes a hierarchy whereby the developers know everything and the users need know nothing.

As a result, many experienced Windows users have reviewed a GNU/Linux distribution and found to their disappointment that to get something configured as it “obviously” should be done, they had to edit a text file by hand, or specify a certain parameter. This flexibility is actually a strength of the system, not a weakness. In the Windows model, the user does not have to learn because they are not allowed to make any decisions of importance: which kernel scheduler, which filesystem, which window manager. These decisions have all been made to a “one size fits most” level by the developers.

SUMMARY

Although it is quite possible to administer and write shell scripts for a GNU/Linux system without knowing any of the history behind it, a lot of apparent quirks will not make sense without some appreciation of how things came to be the way they are. There is a difference between scripting for a typical Linux distribution, such as RedHat, SuSE, or Ubuntu, and scripting for an embedded device, which is more likely to be running `busybox` than a full GNU set of tools. Scripting for commercial Unix is slightly different again, and much as a web developer has to take care to ensure that a website works

in multiple browsers on multiple platforms, a certain amount of testing is required to write solid cross-platform shell scripts.

Even when writing for a typical Linux distribution, it is useful to know what is where, and how it came to be there. Is there an `/etc/sysconfig`? Are init scripts in `/etc/rc.d/init.d` or `/etc/init.d`, or do they even exist in that way? What features can be identified to see what tradition is being followed by this particular distribution? Knowing the history of the system helps one to understand whether the syntax is `tar xzf` or `tar -xzf`; whether to use `/etc/fstab` or `/etc/vfstab`; whether running `killall httpd` will stop just your Apache processes (as it would under GNU/Linux) or halt the entire system (as it would on Solaris)!

The next chapter follows on from this checkered history to compare the variety of choices available when selecting a Unix or GNU/Linux environment.

2

Getting Started

Before you can work through and test the code in this book, you will need to get some kind of Unix-like environment running. Since you are reading this book, it is likely that you already have access to a Unix or Linux system, but this chapter provides an overview of some of the choices available, how to get them, and how to get up and running with your test environment. It might also be worth considering running a virtual machine, or at least creating a separate account on your existing system when working on the code in this book.

Although GNU/Linux and the Bash shell is probably the most common operating system and shell combination currently in use, and that combination is the main focus of this book, there are lots of other operating systems available, and a variety of shells, too. For shell scripting, the choice of operating system does not make a huge difference much of the time, so this chapter focuses more on operating system and editor choices.

CHOOSING AN OS

First of all, it is worth mentioning that Linux is not the only option available; other freely available operating systems include the BSDs (FreeBSD, NetBSD, OpenBSD), Solaris Express, Nexenta, and others. However, there are many GNU/Linux distributions available, and these generally have support for the widest range of hardware and software. Most of these distributions can be downloaded and used totally legally, even for production use. Of the Linux distributions mentioned here, RedHat Enterprise Linux (RHEL) and SuSE Linux Enterprise Server (SLES) have restricted availability and access to updates; Oracle Solaris is restricted to a 90-day trial period for production use.

GNU/Linux

RHEL is the commercial distribution based on Fedora. It is particularly popular in North America and much of Europe. Because the RHEL media includes RedHat trademarks and some non-Free Software (such as the RedHat Cluster), distribution of the media is restricted to licensed customers. However, the CentOS project rebuilds RHEL from source, removing

RedHat trademarks, providing a Linux distribution that is totally binary and source code-compatible with RHEL. This can be very useful as a lot of commercial software for Linux is tested and supported only on RHEL, but those vendors will often also support the application running on CentOS, even if they do not support the OS itself.

RHEL itself is available by paid subscription only. However, CentOS and Oracle Enterprise Linux are two clones built by stripping the RedHat trademarks from the source code and rebuilding in exactly the same way as the RedHat binaries are built. CentOS is available from <http://centos.org/>, and Oracle Enterprise Linux is available from <http://edelivery.oracle.com/linux>.

Fedora is the community-maintained distribution that feeds into RHEL. It has a highly active, generally very technical user base, and a lot of developments tested in Fedora first are then pushed upstream (to the relevant project, be it GNOME, KDE, the Linux kernel, and so on). Like Ubuntu, it has six-month releases, but a much shorter one-year support cycle. The technologies that have been proven in Fedora make their way into RedHat Enterprise Linux. As with Ubuntu, KDE, XFCE, and LXDE respins are available as well as the main GNOME-based desktop. DVD images can be obtained from <http://fedoraproject.org/>.

SLES is Novell's enterprise Linux. It is based on OpenSUSE, which is the community edition. SLES and OpenSUSE are particularly popular in Europe, partly due to SuSE's roots as a German company before Novell purchased it in 2004. SuSE's biggest differentiator from other Linux distributions is its YaST2 configuration tool. SLES has a fairly stable release cycle; with a new major release every 2–3 years, it is updated more frequently than RHEL but less frequently than most other Linux distributions.

SLES is available for evaluation purposes from <http://www.novell.com/products/server/>. Like RedHat Enterprise Linux, a support contract is required to use the full version.

OpenSUSE is to SLES as Fedora is to RHEL — a possibly less stable but more community-focused, cutting-edge version of its Enterprise relative. Test versions are available before the official release. OpenSUSE is available from <http://software.opensuse.org/>. The main OpenSUSE website is <http://www.opensuse.org/>.

Ubuntu is based on the Debian “testing” branch, with additional features and customizations. It is very easy to install and configure, has lots of Internet forums providing support, and is a polished GNU/Linux distribution. Ubuntu offers a Long-Term Support (LTS) release once every 2 years, which is supported for 2 years on the desktop and 5 years for servers. There are also regular releases every 6 months, which are numbered as YY-MM, so the 10-10 release (Lucid Lynx) was released in October 2010. Although widely known for its desktop OS, the server version, without the graphical features, is growing in popularity.

Ubuntu can be installed in many ways — from a CD/DVD, a USB stick, or even from within an existing Windows installation. Instructions and freely downloadable media and torrents are available from <http://ubuntu.com/>. Many rebuilds of Ubuntu are also available: Kubuntu with KDE instead of GNOME and Xubuntu with the XFCE window manager, as well Edubuntu, which includes educational software, and the Netbook Edition tailored for netbook computers.

Debian is one of the older GNU/Linux distributions in mainstream use. It has a team of over 1,000 Debian developers, providing over 30,000 packages. The stable branch is generally released every 5 years or so, so the current stable release can be rather old, although plans are to increase the frequency of stable releases. The testing branch is popular with many users, providing the latest

packages but without the unpredictability of the unstable branch. Debian CD/DVD images are available for direct download, or via BitTorrent, from www.debian.org/CD/.

Many hundreds of GNU/Linux distributions are available. The website <http://distrowatch.com/> is an excellent resource with information on just about every distribution that exists, as well as other Unix and Unix-like software. Some other popular distributions worth highlighting include Gentoo, Damn Small Linux, Knoppix, Slackware, and Mandriva.

The BSDs

Berkeley Software Distribution, or BSD, is one of the oldest Unix flavors. It has split into a number of different developments, the main three of which are listed here. Each flavor of BSD has a different focus which determines its development style.

FreeBSD is probably the most accessible of the BSDs, with support for a wider variety of hardware. OpenBSD is a fork of NetBSD and is generally regarded as the most secure Unix system available, and although its development is often slower, the resulting system is incredibly stable and secure. OpenBSD is widely used as a router or firewall. As for version 4.9 which was released in May 2011, only two remotely exploitable security holes have ever been found in a default install of OpenBSD. Some operating systems find that many in one month.

NetBSD is the most portable of the BSDs, running on PC, Alpha, and PowerPC, as well as ARM, HPPA, SPARC/SPARC64, Vax, and many others.

Proprietary Unix

Oracle Solaris traces its roots back to 1983, and is arguably the most feature-full and actively developed enterprise OS on the market today. SunOS was originally based on BSD, but with the move to Solaris switched to the System V flavor of Unix. Solaris today comes with the original Bourne shell as /bin/sh, as well as ksh93, bash, csh, tcsh, and zsh shells. Solaris is available for SPARC and x86 architectures.

Oracle Solaris is available for download from <http://www.oracle.com/technetwork/server-storage/solaris/downloads/index.html>, which can be used for free in nonproduction use, or on a 90-day trial basis. Solaris Express is a technical preview of the version of Solaris currently in development. There is also OpenIndiana, a fork of OpenSolaris available at <http://openindiana.org/>, and Nexenta, another fork with a GNU user space, at <http://nexenta.org/>.

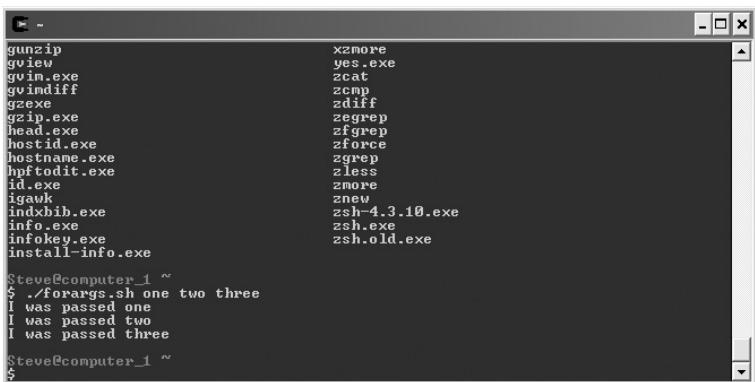
IBM AIX is IBM's Unix for the Power architecture, based on System V Unix. It is available in an Express edition (limited to four CPU cores and 8GB RAM), the Standard Edition (which does not have the scalability limitations), and the Enterprise Edition (which adds extra monitoring tools and features). At the time of this writing, the current version is AIX 7.1, released in September 2010.

HP-UX is HP's Unix offering, based on System V Unix. It runs on PA-RISC and Intel Itanium systems. At the time of this writing, the current version of HP-UX is 11iv3, released in April 2008.

Microsoft Windows

Cygwin is an environment that runs under Microsoft Windows, providing you with a fairly comprehensive GNU toolset. If you can't change to an OS that uses a shell natively, cygwin is a convenient way to get a fully functioning bash shell and the core utilities (ls, dd, cat — just about everything you would

expect in your GNU/Linux distribution) without leaving Windows. This means that you have the GNU tools such as grep, sed, awk, and sort working exactly as they do under Linux. Note that cygwin is not an emulator — it provides a Windows DLL (cygwin1.dll) and a set of (mainly GNU) utilities compiled as Microsoft Windows executables (.exe). These run natively under Windows; nothing is emulated. Figure 2-1 shows cygwin in use. Notice that some of the binaries are named with the .exe extension used by Microsoft DOS and Windows.



The screenshot shows a Cygwin terminal window with a dark background. At the top, there's a menu bar with options like 'File', 'Edit', 'View', 'Search', 'Help', and 'Exit'. Below the menu is a title bar with the text 'C:\'. The main area of the window displays a command-line session. The user has run the command 'dir' to list files in the current directory, which includes various Cygwin utilities like gunzip, gview, gzip, etc., and some executables like yes.exe, zcat, zcmp, zdiff, zgrep, zfgrep, zforce, zgrep, zless, zmore, znew, zsh-4.3.10.exe, zsh.exe, and zsh.old.exe. Below this, the user runs a shell script named 'forargs.sh' with three arguments: 'one', 'two', and 'three'. The script outputs the message 'I was passed one', 'I was passed two', and 'I was passed three' respectively. The prompt at the bottom of the window is '\$'.

```
gunzip          xzmore
gview          yes.exe
gzip.exe        zcat
gzmidiff      zcmp
gzexe         zdiff
gzip           zgrep
head.exe       zfgrep
hostid.exe     zforce
hostname.exe   zgrep
hpftodit.exe   zless
id.exe         zmore
igawk          znew
indxbib.exe    zsh-4.3.10.exe
info.exe       zsh.exe
infokey.exe    zsh.old.exe
install-info.exe

Steve@computer_1 ~
$ ./forargs.sh one two three
I was passed one
I was passed two
I was passed three
Steve@computer_1 ~
```

FIGURE 2-1

Cygwin is available from <http://www.cygwin.com/>.

CHOOSING AN EDITOR

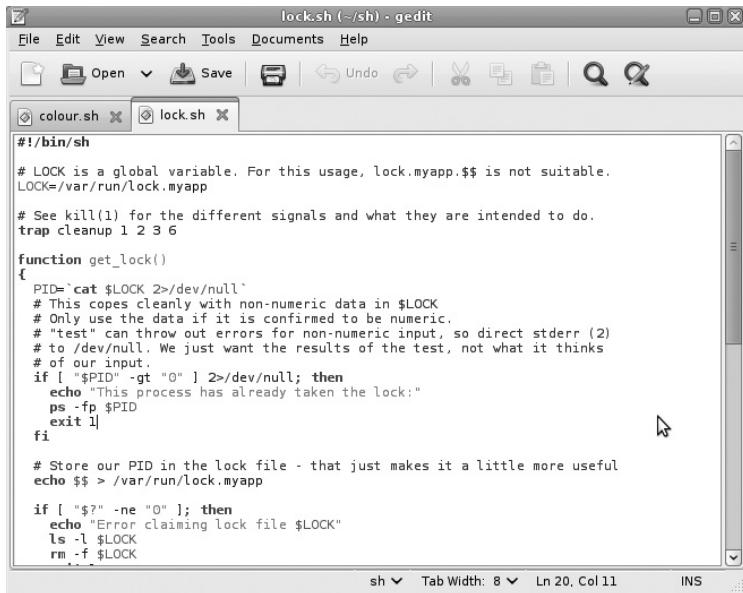
A variety of text editors are available in most of the OSs mentioned previously. Word-processing software, such as OpenOffice.org, Abiword, or Microsoft Word, is not particularly suitable for programming, as these programs often make changes to the text, such as spell-checking, capitalization, formatting, and so on, which can break the script in unexpected ways. It is far better to use a plain-text editor, the most common of which you will look at here. Just because they do not add formatting to the actual file does not mean that these editors are at all lacking in powerful features; most offer syntax highlighting, and many offer further useful features for editing shell scripts as well as other text files.

Graphical Text Editors

For a graphical environment, a GUI-based editor can be easier to use. It is still vital to know how to use a nongraphical editor for situations where a GUI is not available (broken X Window system configuration, remote ssh to the server, serial access to server, and so on). However, for day-to-day use, some people find the convenience of a graphical editor to be useful.

Gedit

The default GNOME text editor is gedit, normally to be found under Applications → Accessories → gedit Text Editor. Gedit offers basic syntax highlighting, which can be useful when checking for syntax errors in your script. It also has tabbed windows and support for different text file formats (Windows, Linux, Mac OS line breaks, and character encodings). Figure 2-2 shows gedit in action.



The screenshot shows a Gedit window titled "lock.sh (~/.sh) - gedit". The window has two tabs: "colour.sh" and "lock.sh". The "lock.sh" tab is active and contains the following shell script:

```

#!/bin/sh

# LOCK is a global variable. For this usage, lock.myapp.$$ is not suitable.
LOCK=/var/run/lock.myapp

# See kill(1) for the different signals and what they are intended to do.
trap cleanup 1 2 3 6

function get_lock()
{
    PID=`cat $LOCK 2>/dev/null`
    # This copes cleanly with non-numeric data in $LOCK
    # Only use the data if it is confirmed to be numeric.
    # "test" can throw out errors for non-numeric input, so direct stderr (2)
    # to /dev/null. We just want the results of the test, not what it thinks
    # of our input.
    if [ "$PID" -gt "0" ] 2>/dev/null; then
        echo "This process has already taken the lock."
        ps -fp $PID
        exit 1
    fi

    # Store our PID in the lock file - that just makes it a little more useful
    echo $$ > /var/run/lock.myapp

    if [ "$?" -ne "0" ]; then
        echo "Error claiming lock file $LOCK"
        ls -l $LOCK
        rm -f $LOCK
    fi
}

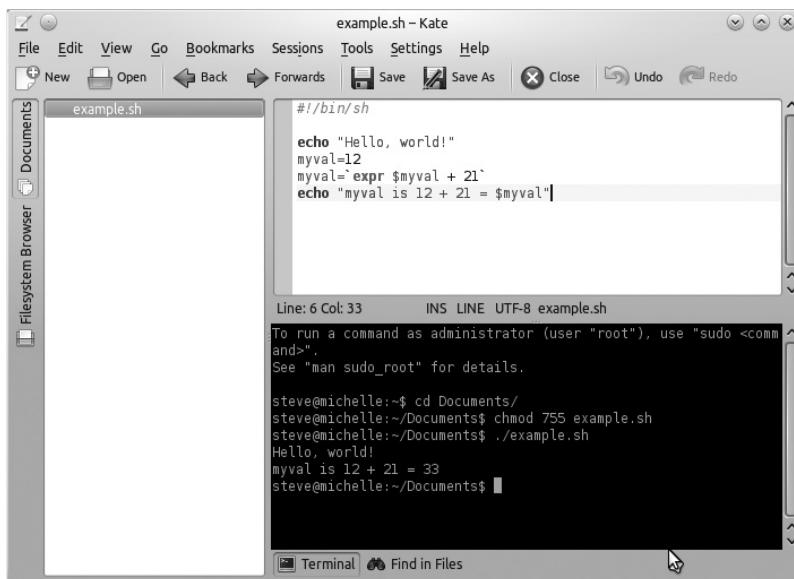
```

At the bottom of the window, there are status indicators: "sh", "Tab Width: 8", "Ln 20, Col 11", and "INS".

FIGURE 2-2

Kate

The default KDE text editor is kate. It offers syntax highlighting, multiple tabs, and so on, but also features a windowed shell so that you can edit your script and run it, all without leaving kate. Figure 2-3 shows kate running with a command window executing the shell script that is being edited by the editor.



The screenshot shows a Kate window titled "example.sh - Kate". The window has a sidebar labeled "Filesystem Browser" and a main area containing the following shell script:

```

#!/bin/sh

echo "Hello, world!"
myval=12
myval=$(expr $myval + 21)
echo "myval is 12 + 21 = $myval"

```

Below the script, a terminal window displays the output of the script and a command-line session:

```

Line: 6 Col: 33      INS LINE UTF-8 example.sh
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

steve@michelle:~$ cd Documents/
steve@michelle:~/Documents$ chmod 755 example.sh
steve@michelle:~/Documents$ ./example.sh
Hello, world!
myval is 12 + 21 = 33
steve@michelle:~/Documents$ 

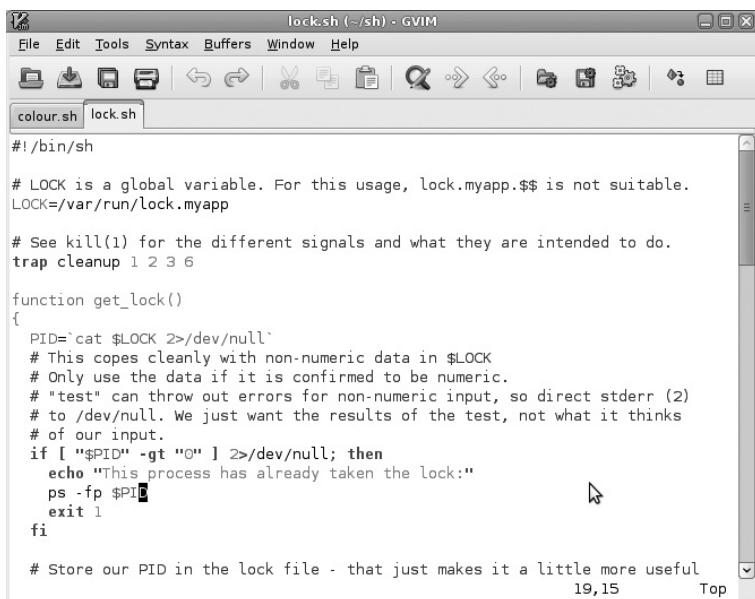
```

At the bottom of the window, there are buttons for "Terminal" and "Find in Files".

FIGURE 2-3

Kwrite is also available as part of KDE, although kwrite is more focused on writing short documents than writing code.

A graphical alternative to the hardcore commandline tool vi (which is provided in most Linux distributions as VIM [Vi IMproved] is gvim. This is a useful halfway house, providing some graphical features (it looks almost identical to gedit) while maintaining the familiar keystrokes of vi and vim. Figure 2-4 shows gvim in use, with two tabs editing two different scripts.



```

lock.sh (~/.sh) - GVIM
File Edit Tools Syntax Buffers Window Help
[Icons] colour.sh lock.sh
#!/bin/sh

# LOCK is a global variable. For this usage, lock.myapp.$$ is not suitable.
LOCK=/var/run/lock.myapp

# See kill(1) for the different signals and what they are intended to do.
trap cleanup 1 2 3 6

function get_lock()
{
    PID=`cat $LOCK 2>/dev/null`
    # This copes cleanly with non-numeric data in $LOCK
    # Only use the data if it is confirmed to be numeric.
    # "test" can throw out errors for non-numeric input, so direct stderr (2)
    # to /dev/null. We just want the results of the test, not what it thinks
    # of our input.
    if [ "$PID" -gt "0" ] >>/dev/null; then
        echo "This process has already taken the lock:"
        ps -fp $PID
        exit 1
    fi

    # Store our PID in the lock file - that just makes it a little more useful
19,15 Top

```

FIGURE 2-4

Vim is also available (in both vim and gvim incarnations) for Microsoft Windows from <http://www.vim.org/download.php#pc>.

Eclipse

Eclipse is a full IDE (Integrated Development Environment) by IBM. It is written with Java development in mind but can be used for shell scripting. It is overkill for most shell programming tasks, however.

Notepad++ for Windows

Notepad++ (<http://notepad-plus-plus.org/>) is a very powerful GPL (Free Software) editor for the Microsoft Windows environment. It offers syntax highlighting for many languages, powerful search options, and many additional features via the plugin infrastructure. It is very popular as a lightweight but full-featured text editor in the Windows environment. Figure 2-5 shows Notepad++ with its native Windows window decorations.

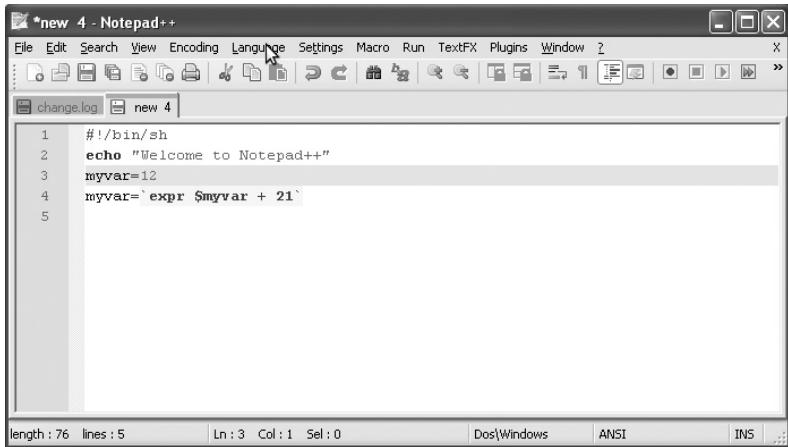


FIGURE 2-5

Terminal Emulation

GNOME has gnome-terminal; KDE has konsole. XFCE has a terminal emulator called simply “Terminal,” with a stated aim of being a worthy alternative to gnome-terminal without the GNOME dependencies. There is also xterm, rxvt, and others. There is also the native “linux” terminal emulation, which is what you get when you log in to a Linux system without a graphical session.

Gnome-terminal is the default terminal in the GNOME environment. It uses profiles so you can define different appearance settings for different purposes. It also uses tabs, which can be shuffled and even detached from the original window.

Konsole is the default, and very flexible, terminal emulator in the KDE environment. It is found under the System menu. Some particularly nice things about Konsole include the ability to get a popup alert from KDE when the terminal either stays quiet for 10 full seconds (for example, when a long-running job finishes writing data to the terminal) or when the silence ends (for example, when a long-running job ends its silence and starts writing data to the terminal).

Another standout feature is the capability, through the profile settings, to define what constitutes a “word” when you double-click on it. If you want to be able to select an entire e-mail address by double-clicking it, make sure that the at sign (@) and the period (.) are in the list; if you want to be able to double-click \$100 and select only the number, make sure that \$ is not in the list.

If you need to run the same command on a set of systems, you can log in to each server in a different tab, and then select **Edit** → **Copy Input To** → **All tabs in current window**. Don’t forget to deselect this as soon as you have finished.

The original terminal emulator for a graphical session is xterm. Although not as common any longer, it is well worth being familiar with xterm for those occasions when a more complete graphical environment is not available.

When you log in to a Linux system without graphical capabilities, or by pressing Ctrl+Alt+F1, you get the native Linux terminal emulator. This is the basic terminal emulator, which is part of the actual Linux OS. It is capable of color as well as highlighted and blinking text.

Nongraphical Text Editors

There are also a good number of command line-based text editors, each with different strengths.

Vi is by far the most widely used text editor among system administrators — it has quite a steep learning curve to start with, mainly because it can operate in two different modes — insert mode, where you can type text as normal in an editor, and command mode, where your keystrokes are interpreted as commands to perform on the text — and because it is difficult to tell which mode you are in at any given time. All that you really need to know about modes is to press Escape to enter command mode, and press `i` in command mode to enter Insert mode. Pressing Escape will always get you into command mode, so `Escape+i` will always get you into Insert mode. Once you have gotten the hang of that, and learned the first few of vi's many powerful commands, other editors will feel slow, awkward, and cumbersome by comparison. While vi is part of Unix, most GNU/Linux distributions include vim (Vi Improved), with vi as an alias to vim. Vim offers compatibility with vi, plus additional functionality, too. Vim comes with a `vimtutor` script, which walks you through tutorials using its many examples. Figure 2-6 shows the first page of `vimtutor`'s tutorial.

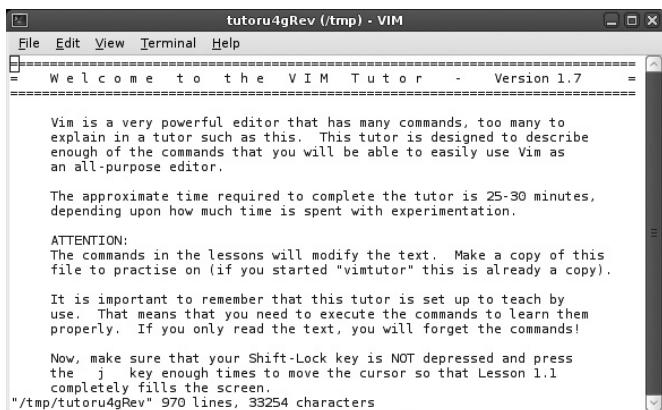


FIGURE 2-6

Emacs is another popular text editor, with an incredible amount of plugins. With a fully configured emacs setup, there is no need to ever go to the shell! It has been described as a “thermonuclear word processor.” Like vim, emacs started out as a console, nongraphical text editor, but now has graphical versions, too. Being cross-platform from the start, emacs does not make any assumptions about what keys will be available on your keyboard, so the PC Ctrl key is referred to as Control, and the Alt key is known as the Meta key. These are written out as `C-` and `M-` respectively, so `C-f`, that is, holding down Control and the `f` key, moves the cursor forward by one character, while `M-f`, or holding down Alt and the `f` key, moves the cursor forward by one word. Use `C-x C-s` to save, and `C-x C-c` to quit.

There is a long-running but generally light-hearted rivalry between vi and emacs; as long as nobody is forced to use the “other” editor, vi and emacs users can generally agree to disagree. Figure 2-7 shows a graphical Emacs session running under the KDE desktop environment.

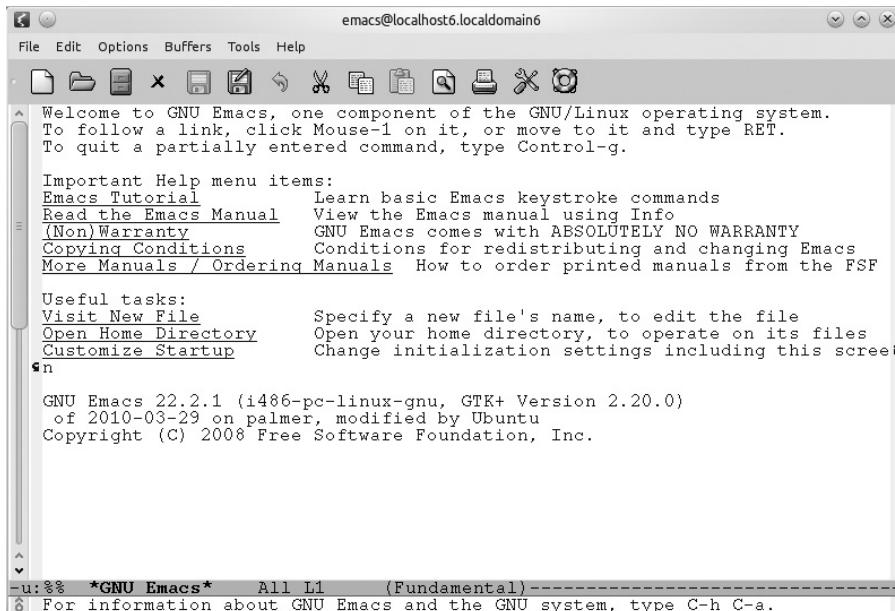


FIGURE 2-7

Pico and nano are rather more accessible text editors. Pico started as the editor for Washington University’s pine e-mail client; nano is the GNU clone of pico and is the editor normally suggested on Ubuntu forums. Much like emacs, commands are sent via the Control key (for example, **Ctrl-X** to exit), but unlike emacs, there is always a context-sensitive menu displayed on the bottom of the screen, making the available choices much more obvious. Figure 2-8 shows nano editing an `/etc/hosts` file.



FIGURE 2-8

SETTING UP THE ENVIRONMENT

Unix and Linux are very customizable systems. You can set the environment (settings and variables that define how the shell behaves) to your liking in a number of ways. If there is something that you find yourself repeatedly setting or changing, it is usually possible to have that automatically done for you by the system. Here some of the most useful are explored.

The Shell Profile

One of the main places for putting your personalized tweaks is the `~/.profile` (`$HOME/.profile`) file. When a new interactive shell is started, `/etc/profile`, followed by `/etc/bash.bashrc` (if a bash shell), `~/.profile`, and finally `~/.bashrc` are executed in that order. `~/.profile` is read by all shells so it is best to put generic settings in there, and then bash-specific settings in `~/.bashrc`. You can set variables and aliases here, and even run commands if you want to. Because the local (user-specific) versions of these files all reside in the home directory and begin with a period (.) so that a regular `ls` does not list them, they are often referred to as “dotfiles.” There are many examples of dotfiles around the net; <http://dotfiles.org/> is one useful repository.

Environment Variables

There are many environment variables that change the way the system works. You can set these interactively, or more usefully in your `~/.bashrc` file.

PS1 Prompt

`PS1` is the basic shell prompt; you can customize this. The default for bash is `\s-\v\$`, or “shell-version-dollar” — for example, `bash-4.1$`. Numerous settings are available — see the “Prompting” section of the bash man page for the full list. A common value for `PS1` is `\u@\h:\w$` — this displays the login name, the server name, and the current working directory. The following example:

```
steve@goldie:/var/log$
```

shows that you are logged in to the server “goldie” as the user “steve,” and are currently in the `/var/log` directory.

In Debian, the default `~/.bashrc` allows for color in the `PS1` prompt, but it also comments that “the focus in a terminal window should be on the output of commands, not on the prompt.” You can uncomment the `force_color_prompt=yes` line in that file if you really do want a color prompt.

PATH

You can set your `PATH` environment variable to tell the shell where to search for programs (and scripts) to be run. The main system commands are in `/bin`, `/usr/bin`, `/sbin`, and `/usr/sbin`, but you may have your own scripts in `$HOME/bin`, `$HOME/scripts`, `/usr/local/bin`, or elsewhere. Append these to the `PATH` so that they will be found by the shell even when you are not in that directory:

```
PATH=${PATH}:$(HOME)/bin
```

Without the PATH, you will need to provide an explicit path (either absolute or relative) to the command. For example:

```
$ myscript.sh
bash: myscript.sh: command not found
$ /home/steve/bin/myscript.sh
... or:
$ cd /home/steve/bin
$ ./myscript.sh
```

From a security perspective, it is very bad practice to put a dot (.) in your PATH, especially at the front of the PATH. If you change into a directory and run a command (maybe ls), any program in that directory called ls will be run, and not the system /bin/ls program. Avoid having a colon at the start or end of the PATH, or a pair of colons with nothing between them, as that will have the same effect as a dot (.). Also, it is better to keep the system directories such as /usr/bin and /bin at the start of the PATH so that local scripts do not override system default ones. Therefore, use the syntax

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

rather than:

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

Tool-Specific Variables

Many system tools have their own variables; less adds the value of \$LESS to its commands. ls adds \$LS_OPTIONS. Your profile can therefore define useful shortcuts by setting these environment variables.

```
# define tool-specific settings
export LS_OPTIONS='--color=yes'
# Tidy up the appearance of less
export LESS=' -X'
```

less also reads the \$LESS_TERMCAP_* variables, which tell it about your terminal's capabilities. This is a useful sequence, which means that the codes hidden inside man pages (which are formatted by less) are interpreted as color changes.



Available for
download on
Wrox.com

```
# man pages in color
export LESS_TERMCAP_mb=$'\E[01;31m'
export LESS_TERMCAP_md=$'\E[01;31m'
export LESS_TERMCAP_me=$'\E[0m'
export LESS_TERMCAP_se=$'\E[0m'
export LESS_TERMCAP_so=$'\E[01;44;33m'
export LESS_TERMCAP_ue=$'\E[0m'
export LESS_TERMCAP_us=$'\E[01;32m'
```

variables

There are also a few widely recognized variables that may be used by many other tools to allow the system to be flexible to your needs. You can specify which text editor you want to use, and

certain tools such as `mail` should use that value. You can also specify your preferred pagination tool — `less` and `more` are the two most common.

```
# define preferred tools
export EDITOR=vim
export PAGER=less
```

Your own scripts can make use of these variables to be more flexible to the user. Just use the `$(EDITOR:-vim)` syntax so that if `$EDITOR` is set then that command will be used, or if not set, you can provide a default for your application:



```
#!/bin/bash
${EDITOR:-vim} "$1"
echo "Thank you for editing the file. Here it is:"
${PAGER:-less} "$1"
```

[edit.sh](#)

This script will edit a file in your preferred `$EDITOR` and then display it back to you with your preferred `$PAGER`.

Aliases

Aliases provide mnemonics for aliases for frequently used, or hard-to-remember commands. Aliases can also be useful for specifying a default set of options where the command does not use a configuration file or environment variables for this. These can be put into your startup scripts to make everyday typing easier.

less

`less` has an `-x` option, which stops it from refreshing the screen after it has completed. This is very much a personal preference; if you wanted to `less` a file and then continue working with the file contents still visible in the terminal, you will want to use `-x` to stop the screen from being refreshed (much as if you had used `cat` on the file — its contents would be visible after the command has finished). However, if you want to be able to see what was displayed on the terminal before you invoked `less`, you would not want the `-x` option. Do try both and see which you prefer. If you want to use `-x`, you can set an alias in your `~/.bashrc` file.

```
alias less="less -X"
```

Because the `less` command takes parameters from the `$LESS` environment variable mentioned previously, you can set that variable instead.

cp, rm, and mv Aliases

Because they are binary-compatible clones of RedHat Enterprise Linux, some Linux distributions — in particular RedHat, and therefore CentOS and Oracle Enterprise Linux — define some very careful aliases for the `cp`, `rm`, and `mv` commands. These are all aliased to their `-i` option, which causes them in an interactive shell to prompt for confirmation before removing or overwriting a file. This

can be a very useful safety feature but quickly becomes irritating. If you find the defaults annoying, you can unset these aliases in `~/.bashrc`. The command `unalias rm` removes this aliasing, and similarly `unalias cp` and `unalias mv` reverts those commands, to their standard behavior, too.



If you know that a command (such as `rm`) is aliased, you can access the unaliased version in two ways. If you know the full path to the command is `/bin/rm`, you can type `/bin/rm`, which will bypass the alias definition. A simpler way to do this is to put a backslash before the command; `\rm` will call the unaliased `rm` command.

ls Aliases

Because it is such a common command, there are a few popular `ls` aliases, the two most common being `ll` for `ls -l` and `la` for `ls -a`. Your distribution might even set these for you. Some popular `ls` aliases include:

```
# save fingers!
alias l='ls'
# long listing of ls
alias ll='ls -l'
# colors and file types
alias lf='ls -CF'
# sort by filename extension
alias lx='ls -lXB'
# sort by size
alias lk='ls -lSr'
# show hidden files
alias la='ls -A'
# sort by date
alias lt='ls -ltr'
```

Other Command Shortcuts

There are many other commands that you might use frequently and want to define aliases for. In a graphical session, you can launch a web browser and direct it straight to a particular website.

```
# launch webpages from terminal
alias bbc='firefox http://www.bbc.co.uk/ &
alias sd='firefox http://slashdot.org/ &
alias www='firefox'
```

Another very frequently used command is `ssh`. Sometimes this is as simple as `ssh hostname`, but sometimes quite complicated command lines are used with `ssh`, in which case an alias again is very useful.



```
# ssh to common destinations by just typing their name
# log in to 'declan'
alias declan='ssh declan'
# log in to work using a non-standard port (222)
```

```
alias work='ssh work.example.com -p 222'
# log in to work and tunnel the internal proxy to localhost:80
alias workweb='ssh work.example.com -p 222 -L 80:proxy.example.com:8080'
```

aliases

Changing History

Another feature of the shell that can be changed in your personalized settings is the `history` command. This is affected by some environment variables and some shell options (`shopt`). When you have multiple shell windows open at once, or multiple sessions logged in for the same user from different systems, the way that the `history` feature logs commands can get a bit complicated, and some history events may be overwritten by newer ones. You can set the `histappend` option to prevent this from happening.

Another potential problem with history is that it can take up a lot of disk space if you do not have much disk quota for your personal files. The `HISTSIZE` variable defines how many entries a shell session should store in the history file; `HISTFILESIZE` defines the maximum total size of the history file.

`HISTIGNORE` is a colon-separated list of commands that should not be stored in the history; these are often common commands such as `ls`, which are not generally very interesting to audit. From an auditing perspective, it is more useful to keep commands such as `rm`, `ssh`, and `scp`. Additionally, `HISTCONTROL` can tell history to ignore leading spaces (so that these two commands are both stored as `rm` and not as “`rm`” (with the leading spaces before the command):

```
$ rm /etc/hosts
$ rm /etc/hosts
```

`HISTCONTROL` can also be told to ignore duplicates, so if one command was run multiple times, there may not be much point in storing that information in the history file. `HISTCONTROL` can be set to `ignorespace`, `ignoredups`, or `ignoreboth`. The history section of your `~/.bashrc` could look like this:



Available for
download on
Wrox.com

```
# append, don't overwrite the history
shopt -s histappend

# control the size of the history file
export HISTSIZE=100000
export HISTFILESIZE=409600

# ignore common commands
export HISTIGNORE=":pwd:id:uptime:resize:ls:clear:history:"

# ignore duplicate entries
export HISTCONTROL=ignoredups
```

history

~/.inputrc and /etc/inputrc

`/etc/inputrc` and `~/.inputrc` are used by GNU readline facility (used by bash and many other utilities to read a line of text from the terminal) to control how readline behaves. These configuration files are only used by shells that make use of the readline library (bash and dash, zsh) and are not used by any other shells — ksh, tcsh, and so on. This defines many of the handy things that bash gets credit for over Bourne shell, such as proper use of the cursor keys on today's PC keyboards. There is normally no need to edit this file, nor to create your own custom `~/.inputrc` (the global `/etc/inputrc` normally suffices). It is useful to know what it contains in order to understand how your shell interacts with your keyboard commands. `inputrc` also defines 8-bit features so you may need to use this if you are working heavily with 7-bit systems.

Another useful bash option to know is

```
set completion-ignore-case On
```

which means that when you type `cd foo` and press the Tab key, if there is no `foo*` directory, the shell will search without case, so that any directories named `Foo*`, `fOo*` or `foo*` will match.

Another bash option is to shut up the audible bell:

```
set bell-style visible
```

It is important to note that `inputrc` affects anything using the readline library, which is normally a good thing as it gives you consistency in the behavior of multiple different tools. I have never been aware of a situation where this caused a problem, but it is good to be aware of the impact of the changes.

~/.wgetrc and /etc/wgetrc

If you need to go via a proxy server, the `~/.wgetrc` file can be used to set proxy settings for the `wget` tool. For example:

```
http_proxy = http://proxyserver.intranet.example.com:8080/
https_proxy = https://proxyserver.intranet.example.com:8080/
proxy_user = steve
proxy_password = letmein
```

You can also set equivalent variables in the shell.

The `/etc/wgetrc` file will be processed first, but is overruled by the user's `~/.wgetrc` (if it exists).



You must use `chmod 0600 ~/.wgetrc` for `~/.wgetrc` to be processed — this is for your own protection; valid passwords should not be visible by anyone but yourself! If the permissions are any more open than 0600, wget will ignore the file.

Vi Mode

People coming from a Unix background may be more comfortable with the ksh, both for interactive use as well as for shell scripting. Interactively, ksh scrolls back through previous commands via the Esc-k key sequence and searches history with the Esc-/ sequence. These are roughly equivalent to bash's up arrow (or ^P) and Ctrl-R key sequences, respectively. To make bash (or indeed the Bourne shell under Unix) act more like ksh, set the -o vi option:

```
bash$ set -o vi
bash$
```

Vim Settings

The following useful commands can be set in `~/.vimrc` or manually from command mode. Note that vim uses the double quote ("") character to mark comments. These samples should be fairly self-explanatory; these can also be set interactively from within a vim session, so typing :syntax on or :syntax off will turn syntax highlighting on or off for the rest of the current session. It can be useful to have all of your favorite settings predefined in `~/.vimrc`.



```
$ cat ~/.vimrc
" This must be first, because it changes other options as a side effect.
set nocompatible

" show line numbers
set number

" display "-- INSERT --" when entering insert mode
set showmode

" incremental search
set incsearch
" highlight matching search terms
set hlsearch
" set ic means case-insensitive search; noic means case-sensitive.
set noic
" allow backspacing over any character in insert mode
set backspace=indent,eol,start
" do not wrap lines
set nowrap

" set the mouse to work in the console
set mouse=a
" keep 50 lines of command line history
set history=50
" show the cursor position
set ruler
" do incremental searching
set incsearch
" save a backup file
set backup

" the visual bell flashes the background instead of an audible bell.
```

```
set visualbell

" set sensible defaults for different types of text files.
au FileType c set cindent tw=79
au FileType sh set ai et sw=4 sts=4 noexpandtab
au FileType vim set ai et sw=2 sts=2 noexpandtab

" indent new lines to match the current indentation
set autoindent
" don't replace tabs with spaces
set noexpandtab
" use tabs at the start of a line, spaces elsewhere
set smarttab

" show syntax highlighting
syntax on

" show whitespace at the end of a line
highlight WhitespaceEOL ctermfg=blue guibg=blue
match WhitespaceEOL /\s\+$/
```

vimrc

SUMMARY

There are many operating systems, shells, and editors to choose from. In general, the choice of editor is a personal preference. The choice of operating system can be very significant in some ways, although for shell scripting purposes, many environments (all of the GNU/Linux distributions, Cygwin, and some proprietary Unixes, notably Solaris) today use GNU bash and the GNU implementations of standard Unix tools such as bc, grep, ls, diff, and so on. This book focuses on GNU/Linux, bash, and the GNU tools, but the vast majority also applies to their non-GNU equivalents.

I hope some of the customizations in the second part of the chapter will prove useful as you tweak the environment to customize the system to your personal preferences; the computer is there to make your life easier, and not the other way around, so if an alias means that you don't have to remember some complicated syntax, your mind is freed of distractions and you can concentrate on what you are actually trying to achieve, not on memorizing the exact syntax of some obscure command.

These first two introductory chapters should have prepared you to do some shell scripting; the rest of Part I covers the tools available and how to use them. The rest of the book builds on this introductory material with real-world recipes that you can use and build on, and so that you can be inspired to write your own scripts to perform real-world tasks to address situations that you face.

CONTENTS

INTRODUCTION

xxix

PART I: ABOUT THE INGREDIENTS

CHAPTER 1: THE HISTORY OF UNIX, GNU, AND LINUX	3
Unix	3
“Everything Is a File” and Pipelines	5
BSD	6
GNU	7
Linux	11
Summary	12
CHAPTER 2: GETTING STARTED	15
Choosing an OS	15
GNU/Linux	15
The BSDs	17
Proprietary Unix	17
Microsoft Windows	17
Choosing an Editor	18
Graphical Text Editors	18
Terminal Emulation	21
Nongraphical Text Editors	22
Setting Up the Environment	24
The Shell Profile	24
Aliases	26
vim Settings	30
Summary	31
CHAPTER 3: VARIABLES	33
Using Variables	33
Typing	34
Assigning Values to Variables	35
Positional Parameters	39
Return Codes	42
Unsetting Variables	45

Preset and Standard Variables	47
BASH_ENV	47
BASHOPTS	47
SHELLOPTS	48
BASH_COMMAND	50
BASH_SOURCE, FUNCNAME, LINENO, and BASH_LINENO	51
SHELL	55
HOSTNAME and HOSTTYPE	55
Working Directory	55
PIPESTATUS	55
TIMEFORMAT	56
PPID	57
RANDOM	58
REPLY	58
SECONDS	58
BASH_XTRACEFD	59
GLOBIGNORE	60
HOME	62
IFS	62
PATH	63
TMOUT	64
TMPDIR	65
User Identification Variables	65
Summary	66
<hr/> CHAPTER 4: WILDCARD EXPANSION	<hr/> 67
Filename Expansion (Globbing)	67
Bash Globbing Features	70
Shell Options	71
Regular Expressions and Quoting	75
Overview of Regular Expressions	76
Quoting	77
Summary	81
<hr/> CHAPTER 5: CONDITIONAL EXECUTION	<hr/> 83
If/Then	83
Else	85
elif	85
Test ()	87
Flags for Test	88
File Comparison Tests	95

String Comparison Tests	96
Regular Expression Tests	98
Numerical Tests	101
Combining Tests	103
Case	105
Summary	109
CHAPTER 6: FLOW CONTROL USING LOOPS	111
For Loops	111
When to Use for Loops	112
Imaginative Ways of Feeding “for” with Data	112
C-Style for Loops	118
while Loops	119
When to Use while Loops	119
Ways to Use while Loops	119
Nested Loops	125
Breaking and Continuing Loop Execution	126
while with Case	130
until Loops	131
select Loops	133
Summary	137
CHAPTER 7: VARIABLES CONTINUED	139
Using Variables	139
Variable Types	141
Length of Variables	142
Special String Operators	144
Stripping Variable Strings by Length	144
Stripping from the End of the String	146
Stripping Strings with Patterns	147
Searching Strings	151
Using Search and Replace	151
Replacing Patterns	153
Deleting Patterns	153
Changing Case	153
Providing Default Values	153
Indirection	157
Sourcing Variables	158
Summary	159

CHAPTER 8: FUNCTIONS AND LIBRARIES	161
Functions	161
Defining Functions	162
Function Output	162
Writing to a File	164
Redirecting the Output of an Entire Function	167
Functions with Trap	171
Recursive Functions	173
Variable Scope	177
Libraries	181
Creating and Accessing Libraries	183
Library Structures	183
Network Configuration Library	187
Use of Libraries	191
getopts	191
Handling Errors	194
getopts within Functions	195
Summary	197
CHAPTER 9: ARRAYS	199
Assigning Arrays	199
One at a Time	200
All at Once	200
By Index	201
All at Once from a Source	201
Read from Input	203
Accessing Arrays	205
Accessing by Index	205
Length of Arrays	206
Accessing by Variable Index	206
Selecting Items from an Array	209
Displaying the Entire Array	209
Associative Arrays	210
Manipulating Arrays	211
Copying an Array	211
Appending to an Array	213
Deleting from an Array	214
Advanced Techniques	216
Summary	217

CHAPTER 10: PROCESSES	219
The ps Command	219
ps Line Length	220
Parsing the Process Table Accurately	220
killall	223
The /proc pseudo-filesystem	225
prtstat	226
I/O Redirection	227
Appending Output to an Existing File	229
Permissions on Redirections	229
exec	229
Using exec to Replace the Existing Program	230
Using exec to Change Redirection	231
Pipelines	237
Background Processing	237
wait	238
Catching Hangups with nohup	239
Other Features of /proc and /sys	242
Version	242
SysRq	242
/proc/meminfo	245
/proc/cpuinfo	245
/sys	246
/sys/devices/system/node	251
sysctl	253
Summary	254
CHAPTER 11: CHOOSING AND USING SHELLS	255
The Bourne Shell	256
The KornShell	256
The C Shell	256
The Tenex C Shell	257
The Z Shell	257
The Bourne Again Shell	257
The Debian Almquist Shell	258
Dotfiles	258
Interactive Login Shells	259
Interactive Non-Login Shells	260
Non-Interactive Shells	261
Logout Scripts	262

Command Prompts	262
The PS1 Prompt	262
The PS2, PS3, and PS4 Prompts	264
Aliases	265
Timesavers	265
Modifying Behaviors	265
History	266
Recalling Commands	267
Searching History	267
Timestamps	268
Tab Completion	269
ksh	269
tcsh	270
zsh	270
bash	271
Foreground, Background, and Job Control	272
Backgrounding Processes	272
Job Control	273
nohup and disown	275
Summary	276

PART II: RECIPES FOR USING AND EXTENDING SYSTEM TOOLS

CHAPTER 12: FILE MANIPULATION	279
stat	279
cat	281
Numbering Lines	282
Dealing with Blank Lines	282
Non-Printing Characters	283
cat Backwards is tac	284
Redirection	285
Redirecting Output: The Single Greater-Than Arrow (>)	285
Appending: The Double Greater-Than Arrow (>>)	286
Input Redirection: The Single Less-Than Arrow (<)	288
Here Documents: The Double Less-Than Arrow (<< EOF)	290
dd	292
df	294
mktemp	295
join	297
install	298

grep	300
grep Flags	300
grep Regular Expressions	301
split	303
tee	304
touch	306
find	307
find-exec	310
Summary	313
CHAPTER 13: TEXT MANIPULATION	315
cut	315
echo	316
dial1	316
dial2	319
fmt	320
head and tail	323
Prizes	323
World Cup	324
od	328
paste	331
pr	334
printf	335
shuf	337
Dice Thrower	337
Card Dealer	338
Travel Planner	340
sort	341
Sorting on Keys	342
Sorting Log Files by Date and Time	344
Sorting Human-Readable Numbers	345
tr	346
uniq	350
wc	351
Summary	352
CHAPTER 14: TOOLS FOR SYSTEMS ADMINISTRATION	353
basename	353
date	355
Typical Uses of date	355
More Interesting Uses of date	359

dirname	360
factor	362
identity, groups, and getent	364
logger	367
md5sum	368
mkfifo	370
Master and Minions	371
Reversing the Order	373
Networking	375
telnet	376
netcat	376
ping	378
Scripting ssh and scp	381
OpenSSL	383
nohup	390
seq	391
Integer Sequences	391
Floating Point Sequences	393
sleep	394
timeout	394
Shutdown Script	396
Network Timeout	399
uname	400
uuencode	401
xargs	402
yes	405
Summary	406

PART III: RECIPES FOR SYSTEMS ADMINISTRATION

CHAPTER 15: SHELL FEATURES	409
 Recipe 15-1: Installing Init Scripts	409
Technologies Used	410
Concepts	410
Potential Pitfalls	410
Structure	410
Recipe	412
Invocation	414
Summary	414
 Recipe 15-2: RPM Report	414
Technologies Used	415
Concepts	415

Potential Pitfalls	415
Structure	415
Recipe	417
Invocation	419
Summary	420
Recipe 15-3: Postinstall Scripts	421
Technologies Used	421
Concepts	421
Potential Pitfalls	422
Structure	422
Recipe	423
Invocation	425
Summary	426
CHAPTER 16: SYSTEMS ADMINISTRATION	427
Recipe 16-1: init Scripts	427
Technologies Used	428
Concepts	428
Potential Pitfalls	429
Structure	430
Recipe	431
Invocation	432
Summary	433
Recipe 16-2: CGI Scripts	433
Technologies Used	433
Concepts	434
Potential Pitfalls	434
Structure	435
Recipe	438
Invocation	441
Summary	445
Recipe 16-3: Configuration Files	445
Technologies Used	445
Concepts	445
Potential Pitfalls	446
Structure	446
Recipe	446
Invocation	447
Summary	448
Recipe 16-4: Locks	448
Technologies Used	448
Concepts	448

Potential Pitfalls	449
Structure	450
Recipe	453
Invocation	455
Summary	458
CHAPTER 17: PRESENTATION	459
 Recipe 17-1: Space Game	459
Technologies Used	459
Concepts	460
Potential Pitfalls	462
Structure	462
Recipe	464
Invocation	469
Summary	470
CHAPTER 18: DATA STORAGE AND RETRIEVAL	471
 Recipe 18-1: Parsing HTML	471
Technologies Used	471
Concepts	472
Potential Pitfalls	472
Structure	472
Recipe	473
Invocation	474
Summary	476
 Recipe 18-2: CSV Formatting	476
Technologies Used	476
Concepts	476
Potential Pitfalls	477
Structure	477
Recipe	478
Invocation	480
Summary	481
CHAPTER 19: NUMBERS	483
 Recipe 19-1: The Fibonacci Sequence	483
Technologies Used	483
Concepts	484

Potential Pitfalls	484
Structure for Method 1	485
Recipe for Method 1	486
Invocation of Method 1	486
Structure for Method 2	487
Recipes for Method 2	488
Invocations of Method 2	489
Structure for Method 3	490
Recipe for Method 3	490
Invocation of Method 3	491
Summary	492
Recipe 19-2: PXE Booting	492
Technologies Used	492
Concepts	493
Potential Pitfalls	493
Structure	493
Recipe	494
Invocation	497
Summary	499
CHAPTER 20: PROCESSES	501
Recipe 20-1: Process Control	501
Technologies Used	501
Concepts	502
Potential Pitfalls	503
Structure	503
Recipe	506
Invocation	511
Summary	516
CHAPTER 21: INTERNATIONALIZATION	517
Recipe 21-1: Internationalization	517
Technologies Used	518
Concepts	518
Potential Pitfalls	519
Structure	520
Recipe	521
Invocation	525
Summary	526

PART IV: REFERENCE

APPENDIX: FURTHER READING	529
Shell Tutorials and Documentation	529
Arrays	530
Tools	530
Unix Flavors	531
Shell Services	531
GLOSSARY	533
INDEX	539

Tried-and-true recipes that can be immediately applied or easily adjusted to meet your needs

The shell is the primary way of communicating with Unix and Linux systems, providing a direct way to program by automating simple to intermediate tasks. In this invaluable resource, Unix, Linux, and shell scripting expert Steve Parker shares a collection of shell scripting recipes that can be used as provided or easily modified for a variety of environments and situations. The book begins with coverage of theory and principles, replete with insightful examples of each element discussed. You then move on to an in-depth discussion of shell programming, covering all Unix flavors but with a focus on Linux and the Bash shell. All the while, you explore credible, real-world recipes and the tools necessary to get started immediately.

Shell Scripting:

- Shares a compendium of helpful shell scripting recipes that can be used to address a variety of real-world challenges
- Includes recipes using file and text control as well as general systems administration tasks
- Provides a host of plug-and-play recipes ready for you to immediately apply and easily modify
- Examines variables, if/then conditionals, loops, functions, pipes, redirects, and more

Steve Parker is an IT consultant specializing in Solaris and GNU/Linux. He has been providing consultancy services for more than a decade. He is the author of the popular Bourne Shell Programming/Scripting Tutorial (<http://steve-parker.org/sh/sh.shtml>), which sees more than one million visitors a year.

Wrox guides are crafted to make learning programming languages and technologies easier than you think. Written by programmers for programmers, they provide a structured, tutorial format that will guide you through all the techniques involved.

Wrox
An Imprint of
 WILEY

Linux / UNIX



WROX.COM

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters, and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

