Jim's Ten Steps to Linux Survival

Jim Lehmer

Contents

By Jim Lehmer

Dedicated to my first three technical mentors - Jim Proffer, who taught me digging deeper was fun and let me do so (often in production). Jerry Wood, who taught me to stop and think. And Kim Manchak, who let me be more than he hired me to be. Thank you, gentlemen. I've tried to pay it forward. This book is part of that.

Figure 1: Merv sez, “Don’t panic.”

Chapter 1

Step -1. Introduction

> *"And you may ask yourself, 'Well, how did I* **get** *here?'"* - Talking Heads (*Once in a Lifetime*)

This is my little "Linux and Bash in 10 steps" guide. It's based around what I consider the essentials for floundering around acting like I know what I'm doing in Linux, BSD and *IX-flavored systems and looking impressive among people who have only worked on Windows in the GUI. Your "10 steps" may be different than mine and that's fine, but this list is mine.

I said ten things, but I lied, because history is really important, so we will start at step #0. And since this is before even that I guess that means this is a 12-step program...

Here is what we'll cover in the rest of this book:

0. **Some History** – UNIX vs. BSD, System V vs. BSD, Linux vs. BSD, POSIX, "UNIX-like," Cygwin, and why any of this matters now, "Why does this script off the internet work on this system and not on that one?"
1. **Come Out of Your Shell** – `sh` vs. `ash` vs. `bash` vs. everything else, "REPL", interactive vs. scripts, command history, tab expansion, environment variables and "A path! A path!"
2. **File Under "Directories"** – `ls`, `mv`, `cp`, `rm` (`-rf *`), `cat`, `chmod`/`chgrp`/`chown` and everyone's favorite, `touch`.
3. **Finding Meaning** – the `find` command in all its glory. Probably the single most useful command in *IX (I think).
4. **Grokking `grep`** – and probably gawking at `awk` while we are at it, which means regular expressions, too. Now we have two problems.

5. **"Just a Series of Pipes"** – `stdin`/`stdout`/`stderr`, redirects, piping between commands.
6. `vi` (had to be #6, if you think about it) – how to stay sane for 10 minutes in `vi`. Navigation, basic editing, find, change/change-all, cut and paste, undo, saving and canceling. Plus easier alternatives like `nano`, and why `vi` still matters.
7. **The Whole Wide World** – `curl`, `wget`, `ifconfig`, `ping`, `ssh`, `telnet`, `/etc/hosts` and email before Outlook.
8. **The Man Behind the Curtain** - `/proc`, `/dev`, `ps`, `/var/log`, `/tmp` and other things under the covers.
9. **How Do You Know What You Don't Know, `man`?** – `man`, `info`, `apropos`, Linux Documentation Project, Debian and Arch guides, StackOverflow and the dangers of searching for "`man find`" or "`man touch`" on the internet.
10. **And So On** - `/etc`, starting and stopping services, `apt-get`/`rpm`/`yum`, and more.

Plus some stuff at the end to tie the whole room together.

Batteries Not Included

It should be obvious that there is ***plenty*** that is not covered:

- **System initialization** - besides, the whole *IX world is in flux right now over system initialization architecture and the shift from "init" scripts to `systemd`.
- **Scripting logic** - scripting, logic constructs (`if`/`fi`, `while`/`done`, and the like).
- **Desktops** - X Windows and the plethora of desktop environments like GNOME, KDE, Cinnamon, Mate, Unity and on and on. This is where *IX systems get the farthest apart in terms of interoperability, settings and customization.
- **Servers** - setting up or configuring web servers like Apache or node, email servers like dovecot, Samba servers for file shares, and so on.
- **Security** - other than the simple basics of the file system security model.

Plus so much more. Again, this is not meant to be exhaustive, but to help someone whose system administration experience has been limited to "Next-Next-Next-Finish" installs and filling in text boxes in wizards on Windows.

Please, Give Generously

That said, if you find something amiss in here - a typo, a misconception or mistake, or a command or parameter you ***really, really, really*** think should be in here even though I said I am not trying to be exhaustive, feel free to send me a `git pull` request.

Why?

Because I work in a primarily Windows-oriented shop, and I seem to be "the guy" that everyone comes to when they need help on a Linux or related system. I don't count myself a Linux guru (***at all***), but I have been running it since 1996 (Slackware on a laptop with 8MB of memory!), and have worked on or run at home various ports and flavors and and versions and distros of "*IX" over the years, including:

- **AIX**
- **FreeBSD**
- **HP/UX**
- **Linux** - literally more distros than I can count or remember, but at least Debian, Fedora, Yellow Dog, Ubuntu/Kubuntu/Xubuntu, Mint, Raspbian, Gentoo, Red Hat and of course the venerable Slackware.
- **Solaris**
- **SunOS**

. . . on various machines and machine architectures from mighty Sun servers to generic "Intel" VMs down to Raspberry Pis (plus an original "wedge" iMac running as a kitchen kiosk long after its "Best by" date and OS/9's demise, thanks to Yellow Dog Linux).

All that while also working on MVS, VSE, OS/2, DOS since 3.x, Windows since 1.x, etc., etc. I don't think I am special when I list all that - there are lots of people with my level of experience ***and better***, especially in commercial software engineering. I am just one of them.

But for some reason there are many places, especially in small and medium business (SMB) environments, where the "stack" tends to be more purely Microsoft because it keeps things simpler and cheaper for the (smaller) staff. I work in such a place. The technical staff is quite competent, but when they bump up against systems whose primary "user interface" for system administration is a command prompt and some scripts, they panic.

This is my attempt to help my co-workers by saying:

"Don't panic." - Douglas Adams (*Hitchhiker's Guide to the Galaxy*)

It started out as a proposal I made a few weeks ago to develop a "lunch and learn" session of about 60-90 minutes of what I considered to be "a Linux survival guide." The list in the *Introduction* above is based on my original email proposal. The audience would be entirely technical, primarily "IT" (Windows/Cisco/VMWare/Exchange/SAN admins).

My goal is not to get into scripting or system setup and hardening or the thousand different ways to slice a file. Instead, the scenario I see in my head is for one of the participants in that "lunch and learn," armed with that discussion and having glanced through this book, to be better able to survive if dropped into the jungle with:

> *"The main www site is down, and all the people who know about it are out. It's running on some sort of Linux, I think, and the credentials and IP address are scrawled on this sticky note. Can you get in and poke around and see if you can figure it out?"* - your boss (next Tuesday morning)

Well, as I started to type out my notes of what I considered to be "essential," they just kept growing and growing. And now, some nights, weekends and lunch hours gone, this is what you see as the result. I figure the slides will be easier to prepare for that "lunch and learn," now that I have the "notes"!

Caveat Administrator

Even so, anything like this is incomplete. Anyone knowledgable of Linux will probably splutter their coffee into their neckbeard at least once a chapter because I don't mention a parameter on a command or an entire subject at all! And that's right - because this "survival guide" is already long enough.

This book is not meant to be an authoritative source, but instead a "fake book" for getting up and running ***quickly*** with the sheer basics, plus knowing where to go for help. It is not a replacement for reading the real documentation and doing research and testing, especially in production! But hopefully it will help get you through that "Can you get in and poke around and see if you can figure it out?" scenario, above. And if Linux should start becoming more of your job, maybe this will help as a gentle push toward "RTFM" along with thinking in "The UNIX Way."

Chapter 2

Step 0. Some History

UNIX vs. BSD, System V vs. BSD, Linux vs. BSD, POSIX, "UNIX-like," Cygwin, and why any of this matters now. "Why does this script off the internet work on this system and not on that one?"

> *"That men do not learn very much from the lessons of history is the most important of all the lessons of history."* - Aldous Huxley

UNIX and its successors such as Linux have a long history reaching into the depths of time:

- **Prehistory** - late 1960s, Nixon, Vietnam, Woodstock, Moon landing, Multics at MIT, GE and Bell Labs.
- **In the beginning** - early 1970s, Nixon drags on, Watergate, Bell Labs, Thompson & Ritchie, UNIX, blah blah blah...
- **More Trouble From Berkeley** - late 1970s, Carter, disco, Iran hostages, UC Berkely releases the Berkeley Software Distribution (BSD), a port based on the Bell Labs UNIX. Let the forking begin!
- **Goes commercial** - 1980s, Reagan, Iran Contra, *E.T.*, AT&T releases System V as first commercial UNIX. From the same background as Bell Labs UNIX, but evolved with subtle and not so subtle differences in approaches to command syntax, networking and much more. It is this release and AT&T's copyrights that are the basis of all the SCO-vs-Linux lawsuits 2-3 decades later.
- **Explosion of *IX** -late 1980s/early 1990s, Bush I, Berlin Wall falls, Gulf War I, proliferation of proprietary (and different) "UNIX" platforms:
  - **HP HP-UX**

  - **Sun SunOS** - BSD flavor.
  - **Sun Solaris** - System V flavor. Now Oracle Solaris.
  - **IBM AIX**
  - **SGI IRIX**
  - **. . . and many, many more!** - although mostly all that's left now is HP-UX, AIX and Solaris.

- **Linux** - 1991+, Clinton I, grunge, *Titanic*, Linus Torvalds releases a project called Linux based on MINIX (and hence why Linus says Linux is pronounced like "MINIX" and not like "Linus"). **Note:** By 1996 I was running an early Linux version (Slackware distro) on a laptop with 8MB of memory!

- **Proliferation of the BSDs** - mid-to-late 1990s, still Clinton I, Monicagate, Kosovo, various ports of BSD including NetBSD, FreeBSD and OpenBSD, all happen in the same time frame as Linux. Like Linux distros, each has its own focus and prejudices, some of which are distinctly "anti-Linux." The "big three" are all still in heavy use today, especially among ISPs. The perception is still out there among a generation of sysadmins that Linux is for the desktop and BSDs for servers, but that reality shifted a long time ago.

- **Ports of call** - 2000+, Bush II & Obama, Afghanistan & Gulf War 2, lots of cross-porting of everything open source. However, licenses matter, and there sure are a lot of them. While it has settled down some with the dismissal of the SCO lawsuit, intellectual property remains a problem area in open source, even as its use has exploded.

Q: So, what's Linux? Or BSD? Or even UNIX?

A: Depends on who you're asking and in what context!

To further muddy the waters, there have been multiple attempts to "standardize" whatever it is this thing is called:

- **POSIX** - a de jure set of standards created in the 1980s and 1990s to try to bring order to the chaos that was commercial UNIX-flavored operating systems of the time. It worked. Sorta. Especially once the US government started wanting systems to be "POSIX-compliant." **Note:** No system runs POSIX, they all are "similar but different." Even Windows can claim to be POSIX in some respects (and has an installable POSIX subsystem), but that doesn't mean POSIX-compliant code will run there unchanged.

- **GNU Project** - Richard Stallman founded the Free Software Foundation (FSF) and GNU project in the mid-1980s, ***long*** before Linux (GNU = "GNU's Not Unix"). The GNU project delivers a suite of programs and tools, many of which are used in both Linux and BSD variants as de facto standards.

- **Various Linux Efforts** - there have also been various movements over the years, some more successful than others, to "standardize" Linux or some part of it, such as the file system layout, the `init` system, documentation, and now even what is part of the most basic "core OS" for things like better containerization.

For the rest of this we will use "Linux" or "*IX" to stand in for a "generic" UNIX-flavored OS unless a difference is specifically called out.

Why Does This Matter?

Because there are various "flavors" of commands and tools, based on whether you're dealing with a System V (Linux) or BSD (Free/Net/Open) descendant. Some of the OS versions are strong in security, or networking, or as a desktop. Certain things are "built-in" to the operating system but most are installed as packages, and depending on the source of the package it may or may not work correctly on another "*IX" system without effort.

It is similar to the history and relationship between `COMMAND.EXE` in DOS and `CMD.EXE` in Windows 10, where this would work in both:

```
COPY A.TXT B.TXT
```

But only the later, network-and-NTFS-aware `CMD.EXE` could handle:

```
COPY "My 2015 Tax Returns.pdf" \\MyServer\Finances\.
```

In *IX-land over time these differences seem to be getting better, but there are still "gotchas," often involving the differences in open source licenses in the underlying code. There are fundamental differences and assumptions between the "GNU" and "GPL" licenses on the one side and "MIT" and "BSD" licenses on the other. I am not a lawyer, but I would summarize:

- **FSF/GNU/GPL** - mostly concerned with keeping open source "open," that is sharable and modifiable by all.
- **BSD & MIT** - more focused on letting anyone do anything to the code as long as the original author is acknowledged and liability released.

The best thing is to be vaguely aware of this history and licenses and if something isn't available on a certain platform or if a command isn't taking a specific parameter to search for variants.

For example, note the differences in command line parameters and output between showing all processes with the `ps` (*process*) command on a Linux system, in this case Linux Mint:

```
$ ps -AH
  PID TTY          TIME CMD
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00   ksoftirqd/0
    5 ?        00:00:00   kworker/0:0H
    7 ?        00:00:19   rcu_sched
    8 ?        00:00:04   rcuos/0
    9 ?        00:00:09   rcuos/1
   10 ?        00:00:07   rcuos/2
...and so on...
```

. . . versus on a FreeBSD system at my ISP, where `csh` is the default shell:

```
%ps -ax
  PID  TT  STAT      TIME COMMAND
73591  ??  S      0:00.03 sshd: myuser@ttyp1 (sshd)
79503  ??  S      0:00.07 dovecot/imap
80065  ??  S      0:00.05 dovecot/imap
73593  p1  Ss     0:00.02 -csh (csh)
90737  p1  RN+    0:00.00 ps -ax
```

To make things even more confusing, the Linux version of `ps` has been written to understand the BSD-style syntax and flags, too!

Panic at the Distro

Remember that "Linux," FreeBSD, OpenBSD and NetBSD are all really just OS kernels, boot loaders, drivers and enough functionality to get a computer up and running. Most functionality comes via other "packages." From almost the beginning there have been alternative approaches to both what packages should (and should not) be included, as well as how best to manage the installing, updating and removal of those packages.

In the BSD world each major port has its own approach. In the Linux world the job of deciding all this and putting it all together falls to distributions or "distros." These have evolved over time into a series of "families" based in large part around the package management tool predominantly used:

- **`apt-get`, `dpkg` and `.deb` files** - Debian flavors, such as Ubuntu and Mint (Mint is currently my desktop Linux of choice, Debian my preferred server OS, but both solely based on familiarity).
- **`pacman`** - Arch flavors.

- **`rpm` and `yum`** - Red Hat flavors, such as Fedora, Red Hat Enterprise and CentOS.
- **Source code** - Gentoo tends to be a "compile from scratch" environment, much like FreeBSD.
- **"Tar balls"** - source code or binaries delivered via archived and zipped directories. Common on Slackware, some others.

Get Embed With Me

A lot of firmware in embedded devices is based on some sort of *IX flavor. Networking gear at both the consumer and enterprise level, storage devices and so on all tend to run something that "looks like" UNIX at some level. Of course, as to what's actually available, who knows? If you can get a shell (command prompt) the best thing to do is see what works.

Cygwin

Cygwin is an interesting beast. It is a DLL for Windows that implements most of the POSIX and related UNIX-like "system API calls" for programming, and then is also a series of ported open source packages, including shells, utilities and even desktop environments, all ***recompiled*** to run on Windows as long as the Cygwin DLL is accessible. Like a Linux distro it has an installer that is a "package manager," and if a package isn't available, you can usually recompile the source code using Cygwin.

You cannot run Linux or BSD binaries on Cygwin without recompiling them first. **However**, you can often run ***scripts*** from a Linux environment on Cygwin with little or no tweaking. Which means you can then take advantage of a lot of excellent open source tools simply by installing their packages in Cygwin and running scripts against them.

Ultimately, though, Cygwin is of limited use, basically for getting to some open source tools on Windows without having to set up a Linux box. You can do a lot of amazing things with Cygwin with enough effort (including running X and a desktop environment like GNOME!), but at some point why not expend that effort in standing up a "real" Linux (virtual) machine anyway?

Chapter 3

Step 1. Come Out of Your Shell

`sh` vs. `ash` vs. `bash` vs. everything else, "REPL", interactive vs. scripts, command history, tab expansion, environment variables and "A path! A path!"

"If you hold a shell up to your ear, you can hear the OS." - me

To avoid getting all pedantic, I am just going to define a shell as an environment in which you can execute commands. People tend to think of a shell as a "command prompt," but you can run a shell without running a command prompt, but not vice versa - an interactive command prompt is an instance of a shell environment almost by definition.

Examples of shells:

- `CMD.EXE` - yes, Windows has a shell.
- `PowerShell.exe` - in fact, it has at least two!

In UNIX-land:

- `sh` - the "original" Bourne shell in UNIX, which spawned:
  - `ash` - Almquist shell.
    - `dash` - Debian Almquist shell (replaced `ash` in Debian)
  - `bash` - Bourne-again shell (get it?), the "standard" Linux shell.
  - `ksh` - Korn shell.

  - **zsh** - Z shell.

- **csh** - C shell, historically it is the default shell on BSD systems (although there are arguments on why you should ***never use it***).
- **. . . and many more!** - tons, really.

Most Linux distros use **bash**, but the BSDs are all over the place. We're going to assume **bash** for the rest of this tutorial. With few modifications, anything in the **sh** hierarchy above can usually run in the other members of the same tree.

bash Built-Ins

Every shell has some "built-in" commands that are implemented as part of the shell and not as an external command or program, and **bash** has its share, as shown by running the **help** command in a **bash** terminal:

```
$ help
GNU bash, version 4.3.42(4)-release (x86_64-unknown-cygwin)
These shell commands are defined internally.  Type ‘help’ to see this list.
Type ‘help name’ to find out more about the function ‘name’.
Use ‘info bash’ to find out more about the shell in general.
Use ‘man -k’ or ‘info’ to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

 job_spec [&]                            history [-c] [-d offset] [n] or hist>
 (( expression ))                        if COMMANDS; then COMMANDS; [ elif C>
 . filename [arguments]                  jobs [-lnprs] [jobspec ...] or jobs >
 :                                       kill [-s sigspec | -n signum | -sigs>
 [ arg... ]                              let arg [arg ...]
 [[ expression ]]                        local [option] name[=value] ...
 alias [-p] [name[=value] ... ]          logout [n]
 bg [job_spec ...]                       mapfile [-n count] [-O origin] [-s c>
 bind [-lpsvPSVX] [-m keymap] [-f file>  popd [-n] [+N | -N]
 break [n]                               printf [-v var] format [arguments]
 builtin [shell-builtin [arg ...]]       pushd [-n] [+N | -N | dir]
 caller [expr]                           pwd [-LP]
...and so on...
```

The above was run in **bash** under Cygwin, but identical output is shown when running **help** under **bash** on Linux, too.

Why does this matter? Because if you are in an environment and something as fundamental as **echo** isn't working, you may not be working in a shell that

is going to act like a "sh" shell. ***In general***, sh, ash, bash, dash and ksh all act similarly enough that you don't care, but sometimes you may have to care. Knowing if you are on a csh variant or even something more esoteric can be key.

Pay attention to the first line in script files, which will typically have a "shebang" line that looks like this:

```
#!/bin/bash
```

In this case we know the script is expecting to be executed by bash, and in fact should throw an error if /bin/bash doesn't exist. Note that on some systems:

```
#!/bin/sh
```

. . . is pointing to an alias of bash, and on some it is a different implementation of the original sh command, such as ash or dash. Now you know what to google if you hit problems as simple as an expected built-in command not being found.

Everything You Know is (Almost) Wrong

CMD.EXE has a lineage that is a mish-mash of CP/M and UNIX excreted through three decades of backwards compatibility via that devil spawn we call DOS. It has gotten even muddier over the years as Microsoft has added more commands, PowerShell, POSIX subsystems, etc.

But even so, there are some similarities. In both bash and CMD.EXE, the set command shows you all environment variables that have been set:

bash

[Under Cygwin]

```
$ set
ALLUSERSPROFILE='C:\ProgramData'
APPDATA='C:\Users\myuser\AppData\Roaming'
BASH=/bin/bash
BASHOPTS=cmdhist:complete_fullquote:expand_aliases:extglob:extquote:for...
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION=/etc/bash_completion
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_COMPLETION_DIR=/etc/bash_completion.d
BASH_LINENO=()
...and so on...
```

CMD.EXE

```
C:\> set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\myuser\AppData\Roaming
CLIENTNAME=JLEHMER650
CommandPromptType=Native
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=JCAPPDEV
ComSpec=C:\Windows\system32\cmd.exe
ExtensionSdkDir=C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0\Exten...
FP_NO_HOST_CHECK=NO
Framework35Version=v3.5
...and so on...
```

Similarly, the `echo` command can be used to show you the contents of an environment variable (among other things):

bash

```
$ echo $HOMEDRIVE
C:
```

CMD.EXE

```
C:\> echo %homedrive%
C:
```

This example shows some valuable differences between shells, though. Even though both have the concept of environment variables and echoing out their contents using the "same" command, note that:

1. The syntax for accessing an environment variable is `$variable` in `bash` and `%variable%` in `CMD.EXE`.
2. `bash` is case-sensitive and so `echo $HOMEDRIVE` works but `echo $homedrive` does not. `CMD.EXE` is ***not*** case-sensitive, so either `echo %homedrive%` or `echo %HOMEDRIVE%` (or `EcHo %hOmEdRiVe%`) would work.

You're a Product of Your Environment (Variables)

It is much more common to set up environment variables to control execution in Linux than in Windows. In fact, it is quite common to override a given environment variable for the single execution of a program, to the point that `bash` has built-in "one-line" support for it:

```
FOO=myval /home/lehmer/myscript
```

This sets the environment variable `FOO` to "myval" but only for the duration and scope of running `myscript`.

By convention, environment variables are named all uppercase, whereas all scripts and programs tend to be named all lowercase. Remember, almost without exception Linux and company are case-sensitive and Windows is not.

You can set or override multiple variables for a single command or script execution simply by separating them with spaces:

```
FOO=myval BAR=yourval BAZ=ourvals /home/lehmer/myscript
```

Note that passing in values in this way does not safeguard sensitive information from other users on the system who can see the values at least while the script is running using the `ps -x` command.

You can also set the value of environment variables to the output of a command using ‘:

```
$ filetype=‘file --print --mime-type --no-pad --print0 otschecker.csv‘

$ echo $filetype
otschecker.csv: text/plain
```

Paths (a Part of Any Balanced Shrubbery)

The concept of a "path" for finding executables is almost identical, and Windows lifted it from UNIX (or CP/M, which lifted it from UNIX). You can tell how similar they are by looking at the output of the `PATH` environment variable under `CMD.EXE` and `bash` running under Cygwin for the same user on the same machine:

CMD.EXE

[Formatted for readability]

```
C:\> echo %path%
C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\CommonEx...
C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\BIN\amd64;
C:\Windows\Microsoft.NET\Framework64\v4.0.30319;
C:\Windows\Microsoft.NET\Framework64\v3.5;
C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\VCPackages;
C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE;
C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\Tools;
...and so on...
```

bash

[Formatted for readability]

```
$ echo $PATH
/usr/local/bin:/usr/bin:/cygdrive/c/Windows/system32:/cygdrive/c/Windows...
/cygdrive/c/Windows/System32/Wbem:/cygdrive/c/Windows/System32/WindowsPo...
/cygdrive/c/Program Files/SourceGear/Common/DiffMerge:
/cygdrive/c/Program Files/TortoiseHg:
/cygdrive/c/Program Files (x86)/Microsoft SQL Server/100/Tools/Binn:
/cygdrive/c/Program Files/Microsoft SQL Server/100/Tools/Binn:
/cygdrive/c/Program Files/Microsoft SQL Server/100/DTS/Binn:
...and so on...
```

Note the differences and similarities. Both the paths are evaluated left to right. Both use separators between path components, a `;` for DOS and Windows, a `:` for Linux. Both delimit their directory names with slashes, with `\` for DOS and Windows and `/` for Linux. But Linux has no concept of a "drive letter" like `C:`, and instead everything is rooted in a single namespace hierarchy starting at the root `/`. We'll be talking more about directories in the next chapter.

Open Your Shell and Interact

The actual "command prompt" is when you bring up a shell in an "interactive session" in a terminal window. This might be from logging into the console of a Linux VM, or starting a terminal window in a X window manager like GNOME or KDE, or `ssh`'ing into an interactive session of a remote machine, or even running a Cygwin command prompt under Windows.

Command prompts allow you to work in a so-called "REPL" environment (Read, Evaluate, Print, Loop). You can run a series of commands once, or keep refining a command or commands until you get them working the way you want, then transfer their sequence to a script file to capture it.

Real shell wizards can often show off their magic in an incredible one-liner typed from memory with lots of obscure commands piped together and invoked with cryptic options.

I am not a real shell wizard. See chapter 9 for how you can fake it like I do.

Getting Lazy

Most modern interactive shells like `bash` and `CMD.EXE` allow for tab expansion and command history, at least for the current session of the shell.

Tab expansion is "auto-complete" for the command prompt. Let's say you have the following files in a directory:

```
$ ls -l
total 764
-rwxrwx---+ 1 myuser mygroup  18554 Oct  9 15:01 Agenda.md
drwxrwx---+ 1 myuser mygroup      0 Oct  9 08:50 Bad and Corrupted Test
Files
drwxrwx---+ 1 myuser mygroup      0 Sep 22 15:35 CheckMD5sLog
-rw-rwxr--+ 1 myuser mygroup   1431 Oct  9 14:58 CygwinPath.txt
-rwxrwx---+ 1 myuser mygroup  22461 Oct  7 14:19 Disabled Active Directory Accounts.xlsx
-rwxrwx---+ 1 myuser mygroup  55647 Sep 18 08:31 filtered.txt
drwxrwx---+ 1 myuser mygroup      0 Sep 15 15:59 FLOCK
-rwxrwx---+ 1 myuser mygroup  11185 Feb 24  2015 GitLab Upgrade Info.txt
...and so on...
```

Without tab expansion, typing out something like:

```
mv Disabled\ Active\ Directory\ Accounts.xlsx
```

. . . is painful. But with tab expansion, we can simply:

```
mv D^t
```

. . . where `^t` represents hitting the `Tab` key, and since there is only one file that starts with a "D" tab expansion will fill in the rest of the file name:

```
mv Disabled\ Active\ Directory\ Accounts.xlsx
```

. . . and we can go about our business of finishing our command.

One place the tab completion in `bash` is different than `CMD.EXE` is that in `bash` if you hit `Tab` and there are multiple candidates, it will expand as far as it can

and then show you a list of files that match up to that point and allow you to type in more characters and hit `Tab` again to complete it. Whereas in `CMD.EXE` it will "cycle" between the multiple candidates, showing you each one as the completion option in turn. Both are useful, but each is subtly different and can give you fits when moving between one environment and another.

Pro Tip: Remember, UNIX was built by people on slow, klunky teletypes and terminals, and they hated to type! Tab expansion is your friend and you should use it as often as possible. It gives at least three benefits:

1. Saves you typing.
2. Helps eliminate misspellings in a long file or command name.
3. Acts as an error checker, because if the tab doesn't expand, chances are you are specifying something else (the beginning path of the file) wrong.

The other thing to remember about the interactive shell is command history. Again, both `CMD.EXE` and `bash` give you command history, but `CMD.EXE` only remembers it for the session, while `bash` stores it in one of your hidden "profile" or "dot" files in your home directory called `.bash_history`:

```
$ ls -a
.               .bash_profile  .gitignore  .minttyrc  Dropbox      Sandbox
..              .bashrc        .inputrc    .profile   fast-export  Shared
.bash_history   .gitconfig     .lesshst    .ssh       myuser      Temp
```

Inside, `.bash_history` is just a text file, with the most recent commands at the bottom.

The `bash` shell supports a rich interactive environment for searching for, editing and saving command history. However, the biggest thing you needf to remember to fake it is simply that the up and down arrows work in the command prompt and bring back your recent commands so you can update them and re-execute them.

Note: If you start multiple sessions under the same account, the saved history will be of the last login to successfully write back out `.bash_history`.

Chapter 4

Step 2. File Under "Directories"

`ls`, `mv`, `cp`, `rm (-rf *)`, `cat`, `chmod/chgrp/chown` and everyone's favorite, `touch`.

> *"I'm in the phone book! I'm somebody now!"* - Navin Johnson (*The Jerk*)

Typically in Linux we are scripting and otherwise moving around files. The file system under the covers may be one of any number of supported formats, including:

- **ext2**
- **ext3**
- **ext4**,
- **ReiserFS**
- **. . . and so much more!** - NTFS, FAT, etc.

Each has its strengths and weaknesses. While Linux tends to treat the ext* file systems as preferred, it can write to a lot of file systems and can read even more.

As mentioned above, the biggest differences between Linux and Windows is that the Linux environments tend not to have a concept of "drive letters." Instead everything is "mounted" under a single hierarchy that starts at the "root directory" or `/`.

```
$ ls /
bin   etc         lib         media  proc  sbin     sys  var
boot  home        lib64       mnt    root  selinux  tmp  vmlinuz
dev   initrd.img  lost+found  opt    run   srv      usr
```

The root file system may be backed by a disk device, LUN, memory or even the network. It will have one or more directories under it. Multiple physical drives and network locations can be "mounted" virtually anywhere, under any directory or subdirectory in the hierarchy.

Note: Dynamically mounted devices like USB drives and DVDs are often mounted automatically under either a `/mnt` or `/media` directory.

Looking at Files

The command to *list* the contents of a directory is the `ls` command:

```
$ ls
Desktop    Downloads  FreeRDP     Music     Public  Temp       Videos
Documents  Dropbox    installrdp  Pictures  rdp     Templates
```

Remember, UNIX environments think of files that start with a `.` as "hidden." If you want to see all these "dotfiles", you can use `ls -a`:

```
$ ls -a
.              Desktop         .gksu.lock       .mozilla   .themes
..             .dmrc           .gnome2          Music      .thumbnails
.adobe         Documents       .gnome2_private  Pictures   .thunderbird
.atom          Downloads       .hugin           .pki       Videos
.bash_history  .dropbox        .ICEauthority    .profile   .wine
.bash_logout   Dropbox         .icons           .ptbt1     .Xauthority
.cache         .dropbox-dist   installrdp       Public     .xinputrc
.cinnamon      .face           .lastpass        rdp        .xsession-errors
.cmake         FreeRDP         .linuxmint       .sbd
.config        .gconf          .local           Temp
.dbus          .gimp-2.8       .macromedia      Templates
```

Wow! That's a lot of dotfiles!

If you want to see some details of each file, use `ls -l`:

```
$ ls -l
total 92
drwxr-xr-x  2 myuser mygroup      4096 Sep  7 04:16 Desktop
```

```
drwxr-xr-x  2 myuser mygroup      4096 Oct 13 10:02 Documents
drwxr-xr-x  2 myuser mygroup      4096 Oct 14 09:45 Downloads
drwx------  8 myuser mygroup      4096 Oct 16 19:58 Dropbox
drwxr-xr-x 19 myuser mygroup      4096 Oct 12 09:48 FreeRDP
-rwxr-x---  1 myuser sambashare    883 Oct 12 11:34 installrdp
drwxr-xr-x  5 myuser mygroup      4096 Oct 16 10:47 LightTable
drwxr-xr-x  2 myuser mygroup      4096 Sep  7 04:16 Music
drwxr-xr-x  3 myuser mygroup     36864 Oct 12 17:29 Pictures
drwxr-xr-x  2 myuser mygroup      4096 Sep  7 04:16 Public
-rwxr-xr-x  1 myuser mygroup       816 Oct 15 18:00 rdp
...and so on...
```

And of course parameters can be combined, as with the two above:

```
$ ls -al
total 344
drwxr-xr-x 40 myuser mygroup      4096 Oct 17 07:14 .
drwxr-xr-x  3 root   root         4096 Sep  7 04:09 ..
drwx------  3 myuser mygroup      4096 Sep  7 09:33 .adobe
drwxr-xr-x  5 myuser mygroup      4096 Oct 12 15:48 .atom
-rw-------  1 myuser mygroup      6428 Oct 17 06:11 .bash_history
-rw-r--r--  1 myuser mygroup       220 Sep  7 04:09 .bash_logout
drwx------ 18 myuser mygroup      4096 Oct 13 07:31 .cache
drwxr-xr-x  5 myuser mygroup      4096 Oct 16 19:57 .cinnamon
drwxr-xr-x  3 myuser mygroup      4096 Oct 12 09:45 .cmake
drwxr-xr-x 26 myuser mygroup      4096 Oct 15 10:23 .config
drwx------  3 myuser mygroup      4096 Sep  7 04:16 .dbus
...and so on...
```

A Brief Detour Around Parameters

In `bash` and many Linux commands in general, there are old, "short" (terse) parameter names, like `ls -a`, and newer, longer, descriptive parameter names like `ls --all` that mean the same thing. It is typically good to use the shorter version during interactive sessions and testing, but I prefer long parameter names in scripts, because when I come back and look at it in two years, I may not remember what `rm -rf *` means (in the *IX world it means you're toast if you run it in the wrong directory by mistake), thus `rm --recursive --force *` seems a bit more "intuitive."

> ***The behind you save in the future by describing things well today may well be your own.*** - me

The older style parameters are typically preceded by a single hyphen "switch" character:

```
ls -r
```

Or even no "switch" character at all, as with `xvf` (e***X***tract, ***V***erbose, input ***F***ile name) in the following:

```
tar xvf backup.tar
```

The newer "GNU-style" parameters are preceded by two hyphens and usually are quite "verbose":

```
ls --recursive --almost-all --ignore-backups
```

Again, it is ***highly recommended*** that you take the time to use the GNU-style parameters in scripts as self-documenting code.

More Poking at Files

If we suspect the file is a text file, we can echo it to the console with the `cat` (*concatenate*) command:

```
$ cat installrdp
#!/bin/bash
sudo apt-get -y install git
cd ~
git clone git://github.com/FreeRDP/FreeRDP.git
cd FreeRDP
sudo apt-get -y install build-essential git-core cmake libssl-dev \
  libx11-dev libxext-dev libxinerama-dev libxcursor-dev \
  libxdamage-dev libxv-dev libxkbfile-dev libasound2-dev libcups2-dev \
  libxml2 libxml2-dev libxrandr-dev libgstreamer0.10-dev \
  libgstreamer-plugins-base0.10-dev libxi-dev \
  libgstreamer-plugins-base1.0-dev
sudo apt-get -y install libavutil-dev libavcodec-dev
sudo apt-get -y install libcunit1-dev libdirectfb-dev xmlto doxygen \
  libxtst-dev
cmake -DCMAKE_BUILD_TYPE=Debug -DWITH_SSE2=ON .
make
sudo make install
sudo echo "/usr/local/lib/freerdp" > /etc/ld.so.conf.d/freerdp.conf
```

```
sudo echo "/usr/local/lib64/freerdp" >> /etc/ld.so.conf.d/freerdp.conf
sudo echo "/usr/local/lib" >> /etc/ld.so.conf.d/freerdp.conf
sudo ldconfig
which xfreerdp
xfreerdp --version
```

We can determine from the above that `installrdp` is a `bash` shell script that looks to install and configure FreeRDP on a Debian-style system:

1. **`apt-get`** - Debian-style package manager.
2. **`git clone`** - cloning package from GitHub.
3. **`cmake`** and **`make`** - configuring and building software from source.

A better way to display a longer file is to use the `less` command (which is a derivative of the original `more`, hence the name). `less` is a paginator, where the `Space`, `Page Down` or down arrow keys scroll down and the `Page Up` or up arrow keys scrolls up. `Q` quits.

Note: The `vi` search (`/`, `?`, `n` and `p`) and navigation (`G`, `0`) keys work within `less`, too. In general `less` is a great lightweight way to motor around in a text file without editing it.

We can also look at just the end or *tail* of a file (often the most interesting when looking at log files and troubleshooting a current problem) with the `tail` command. To show the last 10 lines of the kernel `dmesg` log:

```
# tail dmesg
[    2.774931] loop: module loaded
[    3.349880] eth0: intr type 3, mode 0, 3 vectors allocated
[    3.351331] eth0: NIC Link is Up 10000 Mbps
[    3.422647] RPC: Registered named UNIX socket transport module.
[    3.422649] RPC: Registered udp transport module.
[    3.422650] RPC: Registered tcp transport module.
[    3.422651] RPC: Registered tcp NFSv4.1 backchannel transport module.
[    3.432437] FS-Cache: Loaded
[    3.443980] FS-Cache: Netfs 'nfs' registered for caching
[    3.449794] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
```

To show the last 20 lines:

```
# tail -n 20 dmesg
[    2.317838] [drm] Fifo max 0x00040000 min 0x00001000 cap 0x0000077f
[    2.318843] [drm] Supports vblank timestamp caching Rev 1 (10.10.2010).
[    2.318845] [drm] No driver support for vblank timestamp query.
```

```
[    2.318914] [drm] Screen objects system initialized
[    2.318917] [drm] Detected no device 3D availability.
[    2.323011] [drm] Initialized vmwgfx 2.4.0 20120209 for 0000:00:0f.0 ...
[    2.486733] input: ImPS/2 Generic Wheel Mouse as /devices/platform/i8...
[    2.655694] Adding 4191228k swap on /dev/sda5.  Priority:-1 extents:1...
[    2.666714] EXT4-fs (sda1): re-mounted. Opts: (null)
[    2.754699] EXT4-fs (sda1): re-mounted. Opts: errors=remount-ro
[    2.774931] loop: module loaded
[    3.349880] eth0: intr type 3, mode 0, 3 vectors allocated
[    3.351331] eth0: NIC Link is Up 10000 Mbps
[    3.422647] RPC: Registered named UNIX socket transport module.
[    3.422649] RPC: Registered udp transport module.
[    3.422650] RPC: Registered tcp transport module.
[    3.422651] RPC: Registered tcp NFSv4.1 backchannel transport module.
[    3.432437] FS-Cache: Loaded
[    3.443980] FS-Cache: Netfs 'nfs' registered for caching
[    3.449794] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
```

You can also use `tail` to *follow* an open file and continuously display any new output at the end, which is useful for monitoring log files in real time:

```
# tail -f dmesg
[    2.774931] loop: module loaded
[    3.349880] eth0: intr type 3, mode 0, 3 vectors allocated
[    3.351331] eth0: NIC Link is Up 10000 Mbps
[    3.422647] RPC: Registered named UNIX socket transport module.
[    3.422649] RPC: Registered udp transport module.
[    3.422650] RPC: Registered tcp transport module.
[    3.422651] RPC: Registered tcp NFSv4.1 backchannel transport module.
[    3.432437] FS-Cache: Loaded
[    3.443980] FS-Cache: Netfs 'nfs' registered for caching
[    3.449794] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
...new lines will appear here over time...
```

If we know nothing about a file, we can use the `file` command to help us guess:

```
$ file installrdp
installrdp: Bourne-Again shell script, ASCII text executable
```

That's straightforward enough! The `file` command isn't always 100% accurate, but it is pretty good and uses an interesting set of heuristics and a text file "database" of "magic" number definitions to define how it figures out what type of file it is examining.

Remember: File extensions have no real meaning per se in Linux (although some are used, especially for media and document formats), so a file name with

no extension like `installrdp` is perfectly valid. Hence the utility of the `file` command.

Sorting Things Out

The `sort` command can be used to not just sort files, but also to merge them and remove duplicates.

Let's say we have three files:

```
$ ls
ElevatorTrucks  FarmCombines  FarmTractors
```

Here are the contents of each:

```
$ cat ElevatorTrucks
Truck   brakes  200
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100

$ cat FarmCombines
Combine motor   1500
Combine brakes  400
Combine tires   2500

$ cat FarmTractors
Tractor motor   1000
Tractor brakes  300
Tractor tires   2000
```

But what if we wanted to process all the lines in all the files in a single alphabetical order? Just redirecting the files into a program won't do it, because the file names will be sorted by the shell and the lines will be processed in file name order, not the ultimate sorted order of all the file contents:

```
$ cat *
Truck   brakes  200
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100
```

```
Combine motor   1500
Combine brakes  400
Combine tires   2500
Tractor motor   1000
Tractor brakes  300
Tractor tires   2000
```

The `sort` command to the rescue!

```
$ sort *
Combine brakes  400
Combine motor   1500
Combine tires   2500
Tractor brakes  300
Tractor motor   1000
Tractor tires   2000
Truck   brakes  200
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100
```

What if we want to sort by the parts column? Well, it is the second "key" field delimited by whitespace, so:

```
$ sort -k 2 *
Truck   brakes  200
Tractor brakes  300
Combine brakes  400
Tractor motor   1000
Combine motor   1500
Tractor tires   2000
Combine tires   2500
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100
```

What about by the third column, the amount?

```
$ sort -k 3 *
Truck   winch   100
Tractor motor   1000
Combine motor   1500
```

```
Truck   brakes  200
Tractor tires   2000
Combine tires   2500
Tractor brakes  300
Combine brakes  400
Truck   tires   400
Truck   tires   400
Truck   tires   400
```

That's not what we expected because it is sorting numbers alphabetically. Let's fix that by telling it to sort numerically:

```
$ sort -k 3 -n *
Truck   winch   100
Truck   brakes  200
Tractor brakes  300
Combine brakes  400
Truck   tires   400
Truck   tires   400
Truck   tires   400
Tractor motor   1000
Combine motor   1500
Tractor tires   2000
Combine tires   2500
```

Maybe we care about the top three most expensive items. We haven't talked about pipes yet, but check this out:

```
$ sort -k 3 -n * | tail -n 3
Combine motor   1500
Tractor tires   2000
Combine tires   2500
```

Finally, what if we want only unique rows?

```
$ sort -k 3 -n -u *
Truck   winch   100
Truck   brakes  200
Tractor brakes  300
Truck   tires   400
Tractor motor   1000
Combine motor   1500
Tractor tires   2000
Combine tires   2500
```

Just to reinforce long parameters, the last example is equivalent to:

```
$ sort --key 3 --numeric-sort --unique *
Truck   winch   100
Truck   brakes  200
Tractor brakes  300
Truck   tires   400
Tractor motor   1000
Combine motor   1500
Tractor tires   2000
Combine tires   2500
```

Rearranging Deck Chairs

We can copy, move or rename (same thing) and delete files and directories. To *copy*, simply use the `cp` command:

```
$ cp diary.txt diary.bak
```

You can copy entire directories recursively:

```
$ cp -r thisdir thatdir
```

Or, if we want to be self-documenting in a script, we can use those long parameter names:

```
$ cp --recursive thisdir thatdir
```

To *move* use `mv`:

```
$ mv thismonth.log lastmonth.log
```

Note: There is no semantic difference between "move" and "rename." However, there are some really cool renaming scenarios that the `rename` command can take care of beyond `mv`, like renaming all file extensions from `.htm` to `.html`.

Making Files Disappear

To delete or *remove* a file you use `rm`:

```
$ rm desktop.ini
```

Pro Tip: There is no "Are you sure?" prompt when removing a single file specified with no wildcards, or even all files with a wildcard, and there is no "Recycle Bin" or "Trash Can" when working from the command prompt, so be careful!

This kind of scenario can happen ***way*** too often, even to experienced system administrators (note the space between `*` and `.bak`):

```
$ cd MyDissertation

$ ls
Citations.bak  Citations.doc  Dissertation.bak  Dissertation.doc  Notes.doc

$ rm * .bak
rm: cannot remove '.bak': No such file or directory

$ ls
```

So, in order, our hapless user:

1. Changed to directory `MyDissertation`.
2. Listed the directory contents with `ls`, saw the combination of `.doc` and `.bak` files.
3. Decided to delete the `.bak` files with `rm`, but accidentally typed in a space between the wildcard `*` and the `.bak`. Note ominous warning message.
4. Presto! `ls` shows ***everything*** is gone, not just the backup files! Yay! The user's day's priorities just got rearranged as they go hunting for another backup of their dissertation.

So be careful out there! This is an example where tab completion can be an extra error check. Or a lot of times I use command history in these cases by changing the `ls` to look for just the files I want to delete:

```
$ ls *.bak
Citations.bak  Dissertation.bak
```

Then using the "up arrow" to bring back up the `ls` command and changing `ls` to `rm` and re-executing it. Safer that way.

Touch Me

We just learned how to make a file disappear. We can also make a file magically appear, just by `touch`:

```
$ touch NewEmptyDissertation.doc

$ ls -l
total 0
-rw-rwxr--+ 1 myuser mygroup 0 Oct 19 14:12 NewEmptyDissertation.doc
```

Notice the newly created file is zero bytes long.

Interestingly enough, we can also use touch just to update the "last modified date" of an existing file, as you can see in time change in the following listing after running `touch` on the same file again:

```
$ touch NewEmptyDissertation.doc

$ ls -l
total 0
-rw-rwxr--+ 1 myuser mygroup 0 Oct 19 14:14 NewEmptyDissertation.doc
```

It can be useful (but also distressing from a forensics point of view) to sometimes set the last modified date of a file to a specific date and time, which `touch` also allows you to do, in this case to the night before Christmas:

```
$ touch -t 201412242300 NewEmptyDissertation.doc

$ ls -l
total 0
-rw-rwxr--+ 1 myuser mygroup 0 Dec 24  2014 NewEmptyDissertation.doc
```

To *make a directory* you use `mkdir`:

```
$ mkdir Bar

$ ls
Bar
```

Typically you need to create all intervening directories before creating a "child" directory:

```
$ mkdir Xyzzy/Something
mkdir: cannot create directory ‘Xyzzy/Something’: No such file or directory
```

But of course you can override that behavior:

```
$ mkdir --parents Xyzzy/Something

$ ls
Bar  Xyzzy

$ ls Xyzzy
Something
```

Navigating Through Life

Ever notice that "life" is an anagram for "file"? Spooky, eh?

Given that the UNIX-style file systems are hierarchical in nature they are similar to navigate as with `CMD.EXE`. The biggest difference is the absense of drive letter and the direction of the slashes.

To *change directories*, simply use `cd` much like in Windows:

```
$ cd /etc

$ pwd
/etc
```

`pwd` simply *prints the working (current) directory.*

In Linux, users can have "home" directories (similar to Windows profiles), typically located under `/home/username` for normal users and `/root` for the "root" id. To change to a user's "home" directory, simply use `cd`:

```
$ cd

$ pwd
/home/jlehmer
```

The tilde (`~`) character is an alias for the current user's home directory. The following example is equivalent to above:

```
$ cd ~

$ pwd
/home/jlehmer
```

More useful is that the tilde can be combined with a user name to specify the home directory of ***another*** user:

```
$ cd ~git

$ pwd
/home/git
```

Note: The above assumes you have permissions to `cd` into `/home/git`. See the section on file permissions for more info.

In addition, you need to know the difference between "absolute" and "relative" paths:

- **Absolute path** - ***always*** "goes through" or specifies the "root" (`/`) directory, e.g. `cd /etc`.
- **Relative path** - does ***not*** specify the root directory, expects to start the navigation at the current directory with all path components existing from there, e.g., `cd Dissertations`.

Windows inherited the concept of `.` for the current directory and `..` for the parent directory directly from UNIX. Consider the following examples that combine all of the above about relative paths and see if it all makes sense:

```
$ mkdir Bar Baz

$ ls
Bar  Baz

$ cd Bar

$ touch a b c

$ ls
a  b  c

$ cd ../Baz

$ ls

$ touch d e f

$ ls
d  e  f
```

```
$ ls ..
Bar  Baz

$ ls ../Bar
a  b  c
```

Did you notice how both `mkdir` and `touch` allow for specifying multiple directory and file names in the same command?

May I?

Most UNIX-style file systems come with a set of nine permissions that can be thought of as a "grid" of 3x3 showing "who has what?" The "who" is "UGO":

- **User** - the user that is the "owner" of the file or directory.
- **Group** - the group that is the "owner" of the file or directory.
- **Other** - everyone else.

The "what" is:

- **Read**
- **Write**
- **Execute** - for files, for directories this means "navigate" or "list contents".

The combination of "who has what?" is usually shown in detailed directory listings by a set of ten characters, with the first one determining whether an entry is a directory or a file:

```
# ls -l /etc
total 844
drwxr-xr-x 3 root root    4096 Feb 25  2015 acpi
-rw-r--r-- 1 root root    2981 Apr 23  2014 adduser.conf
-rw-r--r-- 1 root root      45 Jul  9 08:46 adjtime
-rw-r--r-- 2 root root     621 May 22  2014 aliases
-rw-r--r-- 1 root root   12288 May 22  2014 aliases.db
drwxr-xr-x 2 root root   20480 Feb 25  2015 alternatives
-rw-r--r-- 1 root root    4185 Dec 28  2011 analog.cfg
drwxr-xr-x 7 root root    4096 Feb 25  2015 apache2
drwxr-xr-x 6 root root    4096 Feb 25  2015 apt
```

```
-rw-r----- 1 root daemon   144 Jun  9  2012 at.deny
-rw-r--r-- 1 root root    1895 Dec 29  2012 bash.bashrc
-rw-r--r-- 1 root root      45 Jun 17  2012 bash_completion
drwxr-xr-x 2 root root    4096 Feb 25  2015 bash_completion.d
...and so on...
```

In the above, for example, we can see that the user `root` owns the file `at.deny` while the `daemon` group is the primary group for it. `root` can both read and write the file (`rw-`) while any user in the `daemon` group can only reade it (`r--`). No other id will have any access to the file at all (`---`).

Similarly we see that `acpi` is a directory (`d`) that can be read, written (new files created) and listed by `root` (`rwx`), and read and listed by the group `root` and all other ids (`r-xr-x`).

If we look in `/etc/init.d` where many services store their startup scripts we see:

```
# ls -l /etc/init.d
total 332
-rwxr-xr-x 1 root root  2227 Apr 15  2013 acpid
-rwxr-xr-x 1 root root  7820 Jan 31  2014 apache2
-rwxr-xr-x 1 root root  1071 Jun 25  2011 atd
-rwxr-xr-x 1 root root  1276 Oct 15  2012 bootlogs
-rwxr-xr-x 1 root root  1281 Jul 14  2013 bootmisc.sh
-rwxr-xr-x 1 root root  3816 Jul 14  2013 checkfs.sh
-rwxr-xr-x 1 root root  1099 Jul 14  2013 checkroot-bootclean.sh
-rwxr-xr-x 1 root root  9673 Jul 14  2013 checkroot.sh
-rwxr-xr-x 1 root root  1379 Dec  8  2011 console-setup
-rwxr-xr-x 1 root root  3033 Jul  3  2012 cron
-rwxr-xr-x 1 root root  2813 Feb  5  2015 dbus
-rwxr-xr-x 1 root root  6435 Jan  2  2013 exim4
...and so on...
```

In this case all the scripts are readable, writable and executable (`rwx`) by the `root` user, and readable and executable by the `root` group and all other users (`r-xr-x`).

To *change* the *owning* user of a file or directory (assuming you have permissions to do so), use the `chown` command:

```
# ls -l
total 4
-rwxr--r-- 1 root root 17 Oct 20 10:07 foo

# chown git foo
```

```
# ls -l
total 4
-rwxr--r-- 1 git root 17 Oct 20 10:07 foo
```

To *change* the primary *group*, use the `chgrp` command:

```
# chgrp git foo

# ls -l
total 4
-rwxr--r-- 1 git git 17 Oct 20 10:07 foo
```

To *change* the various permissions or *mode* bits, you use the `chmod` command. It uses mnemonics of "ugo" for (owning) user, group and "other," respectively. It also uses mnemonics of "rwx" for read, write and execute, and `+` to add a permission and `-` to remove it. For example, to add execute permission for the group and remove read permission for "other":

```
# chmod g+x,o-r foo

# ls -l
total 4
-rwxr-x--- 1 git git 17 Oct 20 10:07 foo
```

Pro Tip: To look like an old hand UNIX hacker, you can also convert any set of "rwx" permissions into an octal number from 0 (no permissions) to 7 (all permissions). It helps to think of the three permissions as "binary places":

- **r** - 2^2 = 4
- **w** - 2^1 = 2
- **x** - 2^0 = 1
- **-** - 0

Some examples:

- — - 0 + 0 + 0 = 0
- **r**– - 2^2 + 0 + 0 = 4
- **r-x** - 2^2 + 0 + 2^0 = 5
- **rw-** - 2^2 + 2^1 + 0 = 6
- **rwx** - 2^2 + 2^1 + 2^0 = 7

Now to use octal with `chmod`, we think of the overall result we want for a file. For example, if we want the `foo` file to be readable, writable and executable by both its owning user and group, and not accessible at all by anyone else, we could use:

```
# chmod u+rwx,g+rwx,o- foo

# ls -l
total 4
-rwxrwx--- 1 git git 17 Oct 20 10:07 foo
```

Or we could simply convert those permissions into octal in our head and:

```
# chmod 770 foo

# ls -l
total 4
-rwxrwx--- 1 git git 17 Oct 20 10:07 foo
```

Now you know the answer to that "How will we ever use octal in real life?" question you asked in school!

Note: For a script or executable file to be allowed to run, it ***must*** be marked as executable for one of the user, group or other entries. The following should be insightful:

```
# echo "echo Hello world" > foo

# ls -l
total 4
-rw-r--r-- 1 root root 17 Oct 20 10:07 foo

# ./foo
-bash: ./foo: Permission denied

# chmod u+x foo

# ls -l
total 4
-rwxr--r-- 1 root root 17 Oct 20 10:07 foo

# ./foo
Hello world
```

"I'll Send You a Tar Ball"

In the Windows world, we are used to compressing and sending directories around as `.zip` files. In Linux you can also deal with `.zip` files, although they don't tend to be the most common, using the `zip` and `unzip` commands:

```
$ mkdir foo

$ cd foo

$ touch a b c

$ mkdir d

$ touch d/e

$ cd ..

$ zip -r foo foo
updating: foo/ (stored 0%)
  adding: foo/c (stored 0%)
  adding: foo/b (stored 0%)
  adding: foo/d/ (stored 0%)
  adding: foo/d/e (stored 0%)
  adding: foo/a (stored 0%)

$ ls -l foo.zip
-rw-r--r-- 1 myuser mygroup 854 Oct 24 15:56 foo.zip

$ unzip foo
Archive:  foo.zip
replace foo/c? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
 extracting: foo/c
 extracting: foo/b
 extracting: foo/d/e
 extracting: foo/a
```

Not too exciting, but you get the drift. There is typically support for other compression algorithms, too, using `bzip2` and `7z` (7-zip) commands.

However, the "native" way to "archive" a directory's contents in *IX is with `tar`, which is so old that `tar` stands for "tape archive." Its purpose is to take virtually any directory structure and create a single output "stream" or file of it. That is then typically ran through a compression command and the result is called a "tarball":

```
$ tar cvf foo.tar foo/*
foo/a
foo/b
foo/c
foo/d/
foo/d/e

$ ls -l foo.tar
-rw-r--r-- 1 myuser mygroup 10240 Oct 24 16:14 foo.tar

$ gzip foo.tar

$ ls -l foo.tar.gz
-rw-r--r-- 1 myuser mygroup 187 Oct 24 16:14 foo.tar.gz
```

In the `tar` command above, the parameters are `c` (create a new archive), `v` (turn on "verbose" output) and `f` followed by the file name of the new `.tar` file.

Note: `tar` supports POSIX-style parameters (`-c`), GNU-style (`--create`), and the old BSD-style (`c` with no hyphens at all), as shown in these examples. So both of the following are also equivalent to the above:

```
$ tar -c -v -f foo.tar foo/*

$ tar --create --verbose --file=foo.tar foo/*
```

The use of compression commands along with `tar` is so prevalent that they've been built into `tar` itself now as optional parameters:

```
$ tar cvzf foo.tgz foo
foo/
foo/c
foo/b
foo/d/
foo/d/e
foo/a

$ ls -l foo.tgz
-rw-r--r-- 1 myuser mygroup 191 Oct 24 16:19 foo.tgz
```

In this case the `z` parameter says to use `gzip` compression, and the `.tgz` file suffix means basically "tarred and gzipped", or the equivalent to `.tar.gz` in the first example.

`tar` is used to both create and read `.tar` files. So to extract something like the above, you can change the create (`c`) parameter to extract (`x`), like this:

‘bash $ tar xvf foo.tgz foo/ foo/c foo/b foo/d/ foo/d/e foo/a

Let's Link Up!

In Windows there are "shortcuts," which are simply special files that the OS knows to interpret as "go open this other file over there." There are also "hard links" that allow to different directory entries *in the same file system* to point to the same physical file.

UNIX file systems also have both these types of links (which isn't surprising, given that Microsoft got the ideas from UNIX). A "soft link" is equivalent to a Windows shortcut, and can point to a file or a directory, and can point to anything on any mounted file system:

```
$ ls -l
total 4
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 a
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 b
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 c
drwxr-xr-x 2 myuser mygroup 4096 Oct 24 16:00 d

$ cd d

$ pwd
/tmp/foo/d

$ cd ..

$ ln -s a MyThesis.doc

$ ln -s d Dee

$ ls -l
total 4
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 a
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 b
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 c
drwxr-xr-x 2 myuser mygroup 4096 Oct 24 16:00 d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 Dee -> d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 MyThesis.doc -> a

$ cd Dee

$ pwd
/tmp/foo/Dee
```

The things to notice about this example:

1. The `-s` parameter indicates "create a *soft* link."
2. Instead of a `-` or `d`, a soft link is shown in a `ls` listing as `l` regardless of whether the target is a file or directory. This is because a soft link doesn't "know" ***what*** the target is - it is just a file with a name in a directory pointing to another location. ***What*** that location is will be determine after the link is traversed.

A "hard link" is a bit different. It can only be made between files and the two files must be on the same file system. That is because hard links are actually directory entries (as opposed to files in directories) that point to the same "inode" on disk. From within a single directory it is impossible to tell if there are other directories with pointers to the same files (inodes) on disk.

```
$ ls
a  b  c  d  Dee  MyThesis.doc

$ ln b B

$ cd d

$ ln ../b .

$ ls -l
total 0
-rw-r--r-- 3 myuser mygroup 0 Oct 24 15:53 b
-rw-r--r-- 1 myuser mygroup 0 Oct 24 15:54 e

$ cd ..

$ ls -l
total 4
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 a
-rw-r--r-- 3 myuser mygroup    0 Oct 24 15:53 b
-rw-r--r-- 3 myuser mygroup    0 Oct 24 15:53 B
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 c
drwxr-xr-x 2 myuser mygroup 4096 Oct 24 16:49 d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 Dee -> d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 MyThesis.doc -> a
```

The "net net" of all the above is that now `b`, `B` and `d/b` all point to exactly the same inode, or disk location, i.e., the exact same physical file.

I Said "Go Away!", Dammit!

So what can possibly go wrong with links? With soft links the answer is easy - the "remote" location being pointed to goes away or is renamed:

```
$ ls -l
total 4
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 a
-rw-r--r-- 3 myuser mygroup    0 Oct 24 15:53 b
-rw-r--r-- 3 myuser mygroup    0 Oct 24 15:53 B
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 c
drwxr-xr-x 2 myuser mygroup 4096 Oct 24 16:49 d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 Dee -> d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 MyThesis.doc -> a

$ rm a

$ ls -l
total 4
-rw-r--r-- 3 myuser mygroup    0 Oct 24 15:53 b
-rw-r--r-- 3 myuser mygroup    0 Oct 24 15:53 B
-rw-r--r-- 1 myuser mygroup    0 Oct 24 15:53 c
drwxr-xr-x 2 myuser mygroup 4096 Oct 24 16:49 d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 Dee -> d
lrwxrwxrwx 1 myuser mygroup    1 Oct 24 16:40 MyThesis.doc -> a

$ cat MyThesis.doc
cat: MyThesis.doc: No such file or directory
```

So even though the soft link `MyThesis.doc` was still in the directory, the actual underlying file `a` is now gone, and trying to access it via the soft link leads to the somewhat confusing "No such file or directory" error message ("I can see it! It's right there!")

With hard links, it isn't so much a problem as just the nature of the beast. Because each hard link is a directory (metadata) entry pointing to an inode, deleting one simply deletes that directory entry. As long as the file has other hard links pointing to it, it "exists." Only when the last remaining hard link is removed has it been "deleted." Let's play:

```
$ echo "This is b." > b

$ cat b
This is b.
```

```
$ cat B
This is b.

$ cat d/b
This is b.
```

So, that makes sense. Above we had an original file `b` and created to hard links to it, `B` and `d/b`. When we edit `b` by placing "This is b." in it, we see that it has the same contents no matter how we access it, because it is pointing to the same inode.

Can you guess how many `rm` commands it will take to delete the file containing "This is b."?

```
$ rm b

$ cat b
cat: b: No such file or directory

$ cat B
This is b.

$ cat d/b
This is b.

$ rm B

$ cat d/b
This is b.

$ rm d/b

$ ls
c  d  Dee  MyThesis.doc
```

So, ultimately, it takes a `rm` for every hard link to permanently delete a file.

Mount It? I Don't Even Know It's Name!

With all this talk that a hard link can only be on the same file system, how do you know whether two directories are on the same file system? In Windows it's easy - that's exactly what the drive letters are telling you. But in Linux, where everything is "mounted" under a single hierarchy starting at `/`, how do I know that `/var/something` and `var/or/other` are on the same file system?

There are multiple ways to tell, actually. The easiest is with the `df` command:

```
$ df
Filesystem                1K-blocks     Used Available Use% Mounted on
/dev/mapper/mint--vg-root 118647068 28847464  83749608  26% /
none                              4        0         4   0% /sys/fs/cgroup
udev                        1965068        4   1965064   1% /dev
tmpfs                        396216     1568    394648   1% /run
none                           5120        0      5120   0% /run/lock
none                        1981068      840   1980228   1% /run/shm
none                         102400       24    102376   1% /run/user
/dev/sda1                    240972    50153    178378  22% /boot
```

The ones of interested are the `/dev` entries, and we see that everything mounted under `/` is on one file system, except for whatever happens to be on the file system mounted under `/boot`. So outside of `/boot`, on this system we could hard link away to our heart's content.

Note: - It is (barely) beyond the scope of this book to cover the `mount` command. I wanted to, really bad, but it can get so complex so fast that I decided not to. Maybe if you ask, real nice...

I'm Seeing Double

So, both hard and soft links can have some interesting side effects if you think about them, yes? For one, if you are backing things up, then you may get duplicates in your backup set. In fact, with hard links you will, by definition, unless the backup software is very smart and doing things like de-duplication.

But even with soft links if everything just blindly followed them you could also get duplicates where you didn't want them, or even circular references. Also, the pointers in the soft link files are not evaluated until the a command references them. Note that the following is perfectly legal with soft links, but may not give the results you expect - think about current working directory shown by `pwd` in the following, and what the effects of the relative paths shown are as the sample progresses:

```
$ pwd
/tmp/foo

$ rm -rf *

$ touch a b c

$ mkdir d

$ touch d/e
```

```
$ ln -s . d/f

$ ls d/f
e  f

$ ln -s .. d/g

$ ls d/g
a  b  c  d
```

Many commands that deal with files and file systems, like `find`, have parameters specifically telling the command whether to follow soft links or not (by default, `find` does not).

What's the `diff`?

Most people think of `diff` as a tool only programmers find useful, but that is short-sighted. The whole purpose of `diff` is to show differences between files. For example, I backed up this document (which is a text file) before starting this chapter, then typed this introduction to `diff`. This is what `diff` shows:

```
$ diff Agenda.bak Agenda.md
1285a1286,1291
> Most people think of [‘diff‘](http://linux.die.net/man/1/diff) as a tool
> only programmers find useful, but that is short-sighted. The whole purpose
> of ‘diff‘ is to show differences between files. For example, I backed up
> this document (which is a text file) before starting this chapter, then
> typed this introduction to ‘diff‘. This is what ‘diff‘ shows:
```

In other words, the "arrows" are pointing to the "new" file (by convention the file specified on the left is the "old" file and the file on the right is the "new" file), showing five lines were inserted, starting at line 1285. Pretty meta, but not real exciting.

Let's look at something else, say a configuration file for an application. We have an original file, `orig.conf`:

```
$ cat orig.conf
FOO=1

SOME=THINGS
STAY=THE
SAME=ALWAYS
```

```
BAR=Xyzzy
```

Then we have a new file, `new.conf`:

```
$ cat new.conf
FOO=2

SOME=THINGS
STAY=THE
SAME=ALWAYS
```

Now if we `diff` them:

```
$ diff orig.conf new.conf
1c1
< FOO=1
---
> FOO=2
7d6
< BAR=Xyzzy
```

Now we can more easily see that line #1 changed (`1c1`) from `FOO=1` on the "left" file to `FOO=2` on the "right," and that line #7 was deleted (`7d6`) from the "left" file to form the "right." Again, not too interesting, but imagine that both files were thousands of lines long, and there were only a few changes, and you were trying to detect and recover an accidentally-deleted line. Now you can see why `diff` can be handy, as long as you keep around a prior version either in a backup file or source code control to compare against.

`diff` is your friend. It really comes into play with a version control system like `git`, but again, that is beyond the scope of this book.

Chapter 5

Step 3. Finding Meaning

The `find` command in all its glory. Probably the single most useful command in* IX (I think).

> *"If we had bacon, we could have bacon and eggs, if we had eggs."* - old joke

Different people will have different answers to "What is the single most useful *IX command?" There certainly are many to consider. But I keep coming back to `find`. It can be intimidating to figure out from the documentation, especially at first, but once you start mastering it, you end up using it over and over again.

The main concepts of `find` is simple:

1. Starting at location X...
2. Recursively find all files or directories (or "file system entries" to be more precise) that successfully match one or more tests...
3. And for each match execute one or more actions.

The simplest example is "starting in the current directory, recursively list all files you find":

```
$ find
.
./Agenda.md
./Bad and Corrupted Test Files
./Bad and Corrupted Test Files/.DS_Store
./Bad and Corrupted Test Files/2008 Letter of Understanding.TIF
```

```
./Bad and Corrupted Test Files/3948175.dat
./Bad and Corrupted Test Files/3948176.dat
./Bad and Corrupted Test Files/3948178.dat
./Bad and Corrupted Test Files/3948180.dat
./Bad and Corrupted Test Files/3948182.dat
./Bad and Corrupted Test Files/3948186.dat
./Bad and Corrupted Test Files/3948190.dat
./Bad and Corrupted Test Files/3948193.dat
./Bad and Corrupted Test Files/3948195.dat
./Bad and Corrupted Test Files/3948197.dat
./Bad and Corrupted Test Files/3948259.dat
...and so on...
```

In this case `find` is just shorthand for `find . -true -print`.

That's not really that interesting. Let's poke around and "find" (pun intended) some better examples of using `find`. It is better to show than tell in this case. Let's dive into a semi-complicated one and pick it apart:

```
find //myserver/myshare/logs/000[4-9] -name \*.dat -newer logchecker.csv \
    -exec /home/myuser/Sandbox/FileCheckers/logchecker \{\} \;
```

How does this all work? Remembering the three steps at the beginning:

1. **Starting at location `//myserver/myshare/logs/000[4-9]`** - in this case a CIFS/SMB share. Note the regular expression (which we will cover later), in this case stating only to look in directories 0004-0009.
2. **Recursively find file system entries that match one or more tests** - the tests in this example are:
    a. ***All files that have a name that ends in `.dat`*** - the only thing to note here is the \ preceding the wildcard `*`. This prevents "shell expansion," which would allow the `bash` process interpreting the command to expand it to the list of files present in the current directory only, not recursively across all directories.
    b. ***That are newer (created or modified after) the file `logchecker.csv`*** - presumably this file gets created by running `logchecker` or some related process. This is an optimization condition check to only look at files that have been updated since the last time the script ran.
3. **For each match, execute `logchecker`** - and pass in the name of the currently found (matching) file.

What's With the Backslashes?

Reconsider this example:

```
find //myserver/myshare/logs/000[4-9] -name \*.dat -newer logchecker.csv \
    -exec /home/myuser/Sandbox/FileCheckers/logchecker \{\} \;
```

There are five (5) backslash (\) characters in the above. In each case, the backslash is preventing shell expansion:

1. `\*.dat` - preserves the `*` for `find` to use as it recursively searches through directories, instead of the shell expanding it to all files that end in `.dat` in the current directory.
2. `\` - the `\` at the end of the first line tells the shell that the command continues on the next line.
3. `\{\} \;` - these three prevent the shell from trying to expand the braces into an environment variable or the semicolon (which is meant to tell `find` when the command being ran via `-exec` and its parameters end), otherwise `;` is normally used to separate independent commands on the same line in the shell.

That last point bears repeating. Any time you `-exec` in a `find` command (which will be a lot), just get used to typing `\{\} \;` (the space between the ending brace and the `\;` is required).

Useful `find` Options

The `find` documentation gives a bewildering number of options. Here are the ones you may find the most useful:

- **`-executable`** - the file is executable or the directory is searchable (in other words, the file or directory's `x` mode bit is set true for user, group or other ("ugo"), per the file permissions discussion above), and the user executing the `find` command falls into one of the categories for which it is set.
- **`-group <gname>`** - file belongs to group *gname*.
- **`-iname <pattern>`** - case-insensitive name search. Any wildcard characters should be escaped.
- **`-name <pattern>`** - case-sensitive name search. Any wildcard characters should be escaped.

- **`-newer <file>`** - each file is tested to see if it is newer than *file*.
- **`-size <n>`** - file uses n units of space, which can be specified in various measures like 512-byte blocks (`b`) through gigabytes (`G`).
- **`-type <c>`** - file is of type c, with the two most common being `d` (directory) or `f` (file).
- **`-user <uname>`** - file is owned by *uname*.

Useful `find` Actions

Similarly, you are going to keep coming back to just a handful of `find` actions:

- **`-delete`** - deletes any files matched so far. Note that actions are also tests (predicates), so as the `find` documentation says, "Don't forget that the find command line is evaluated as an expression, so putting `-delete` first will make find try to delete everything below the starting points you specified."
- **`-exec` and `-execdir`** - executes a command or script, typically passing in the name of the file or directory found. You will use this ***all*** the time. The difference between the two is that `-execdir` changes the working directory to that of the file found before invoking the program or script, whereas `-exec` simply passes in the fully-qualified path of the found item.
- **`-print`** - prints the full path of the found file or directory. This is the default action.
- **`-printf`** - prints a formatted string, useful for reports.

The `-printf` action allows you to do some interesting things when producing output. For example, consider these three files:

```
$ touch a b c

$ ls -l
total 0
-rw-rwxr--+ 1 myuser mygroup 0 Oct 21 11:02 a
-rw-rwxr--+ 1 myuser mygroup 0 Oct 21 11:02 b
-rw-rwxr--+ 1 myuser mygroup 0 Oct 21 11:02 c
```

If for some reason we wanted a report where for each of those files we wanted three lines with the name, owner and created date and time in ISO 8601 format, all followed by a blank line, we could use the following `find` command:

```
$ find . -type f -printf "%p\n%u\n%TY-%Tm-%TdT%TT\n\n"
./a
myuser
2015-10-21T11:02:51.7014527000

./b
myuser
2015-10-21T11:02:51.7035423000

./c
myuser
2015-10-21T11:02:51.7048997000
```

The `-printf` format string `"%p\n%u\n%TY-%Tm-%TdT%TT\n\n"`breaks down as:

- `"` - prevent shell expansion on the format string.
- `%p` - file name.
- `\n` - new line.
- `%u` - owning user name.
- `\n` - new line.
- `%TY` - the last modification date of the file expressed as a year.
- `-` - a literal hyphen.
- `%Tm` - the last modification date of the file expressed as a month.
- `-` - a literal hyphen.
- `%Td` - the last modification date of the file expressed as a day.
- `T` - a literal 'T'.
- `%TT` - the time expressed in ***hh:mm:ss.hhhhhh*** format.
- `\n\n` - two new lines.
- `"` - prevent shell expansion on the format string.

Chapter 6

Step 4. Grokking grep

And probably gawking at `awk` while we are at it, which means regular expressions, too. Now we have two problems.

> *"Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems."* - Jamie Zawinski

If the `file` command is useful for finding file system entries based on their attributes, the `grep` command is good for finding files with contents that match a regular expression. You already know at least one regular expression, the wildcard `*` character from even the `CMD.EXE` prompt and Windows Explorer. It means "match zero or more characters." We'll cover more on regular expressions, or "regexes," in a moment.

First, an example of `grep`, showing all files in a directory with the pattern "is" in them:

```
$ touch a b c

$ echo This sequence of characters is called a \"string\". > d

$ cat d
This sequence of characters is called a "string".

$ ls
a  b  c  d

$ grep is *
d:This sequence of characters is called a "string".
```

Expressing Yourself Regularly

So what are "regular expressions?" Simply, they are patterns for matching "strings," which are sequences of "characters," e.g.:

```
This sequence of characters is called a "string".
```

That is a string. So is, "That is a string." And "That" and "T" and so on. ***In general*** (with many exceptions), the UNIX world view is that everything is composed of text (or "strings"), and that creating, changing, finding and passing around text is the primary mode of operation.

In the `grep` example, we can see a regular expression can be as simple as "is". It can also be as complicated as:

```
(?bhttp://[-A-Za-z0-9+&@#/%?=~_()|!:,.;]*[-A-Za-z0-9+&@#/%=~_()|]
```

That shows at least one attempt at being a very complete parser of valid HTTP URLs. Wow! What's all that? Now you see why you have two problems. Even if you get that all figured out, or if you actually sit and create something like that from scratch yourself (and it works!), imagine coming back six months later and trying to decipher it again.

There are literally whole web sites and books on just regular expressions. With variations they are used in all *IX shells, Perl, Python, Javascript, Java, C# and more. So obviously (a) they are really useful, and (b) we're not going to cover regexes all here.

There are so many things you can do, the only thing to it is to remember "regular expressions" when you think "I need to find things based on a pattern" and then research what it will take to define the pattern you want.

In the mean time, a few ***simple*** regex examples. Consider the following file `invoices`:

```
$ cat invoices
Combine brakes  400
Combine motor   1500
Combine tires   2500
Tractor brakes  300
Tractor motor   1000
Tractor tires   2000
Truck   brakes  200
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100
```

Let's find all lines with "tractor":

```
$ grep tractor invoices
```

Huh, nothing was found. But this is UNIX-land, so we know it is sensitive - about case anyway:

```
$ grep Tractor invoices
Tractor brakes  300
Tractor motor   1000
Tractor tires   2000
```

Or we could just tell `grep` we are insensitive (to case, anyway):

```
$ grep -i tractor invoices
Tractor brakes  300
Tractor motor   1000
Tractor tires   2000
```

And just to remind you about long-style parameters:

```
$ grep --ignore-case tractor invoices
Tractor brakes  300
Tractor motor   1000
Tractor tires   2000
```

But what ***lines*** are those on?

```
$ grep -i -n tractor invoices
1:Tractor       motor   1000
2:Tractor       brakes  300
3:Tractor       tires   2000
```

To get more complicated, we can pass the `-E` parameter (for *extended* regular expressions) and start doing some really fun stuff. Let's look for lines with either "Tractor" or "Truck":

```
$ grep -E "Tractor|Truck" invoices
Tractor brakes  300
Tractor motor   1000
Tractor tires   2000
Truck   brakes  200
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100
```

For me, the following keep coming up when using regular expressions:

- `one|other` - find `one` pattern or the `other`.
- `^` - pattern for the beginning of a line.
- `$` - pattern for the end of a line.
- `?` - match exactly one character.
- `*` - match zero or more characters.
- `+` - match one or more characters.
- `[A-Z]` - match any character in a range (such as in this case any uppercase Latin alphabetic character).
- `[n|y]` - match one character or another (such as `n` or `y` here).

For example, to find the lines that end in `400`:

```
$ grep  -E "^*400$" invoices
Combine brakes  400
Truck   tires   400
Truck   tires   400
Truck   tires   400
```

Groveling With `grep`

To recursively find all files that contain the string "pdfinfo":

```
$ grep -R -i pdfinfo *
./FileCheckers/otschecker:# pdfinfo, too. If pdfinfo thinks it's junk, ...
./FileCheckers/otschecker:        pdfinfo=`pdfinfo -opw foo "$1" 2>&1 1...
./FileCheckers/otschecker:        if [ $rc != 0 -a "$pdfinfo" != "Comma...
./FileCheckers/pdfchecker:        # pdfinfo, too. If pdfinfo thinks it'...
./FileCheckers/pdfchecker:                pdfinfo=`pdfinfo "$1" > /dev/...
./FileCheckers/pdfpwdchecker:# pdfinfo, too. If pdfinfo thinks it's jun...
./FileCheckers/pdfpwdchecker:        pdfinfo=`pdfinfo -opw foo "$1" 2>&...
./FileCheckers/pdfpwdchecker:        if [ $rc != 0 -a "$pdfinfo" = "Com...
./FileCheckers/README.md:* ***[pdfinfo(1)](http://linux.die.net/man/1/p...
```

The above is functionally equivalent but ***much*** quicker than:

```
$ find . -type f -exec grep -H -i pdfinfo \{\} \;
```

Note: In general, if a command has its own "recursive" option (such as `-R` with `grep`), it is quicker to use that rather than to invoke the command repeatedly using `find` instead.

However, sometimes you can use `find` to filter down files to be checked before having `grep` read through them, and have that result in much quicker results.

For example, if you only wanted to check files that contain "pdfinfo" that have been created or modified since the last time you checked, it could be quicker to run something like:

```
$ find . ! -name pdfinfo.log -newer pdfinfo.log -type f -exec grep -H \
-i pdfinfo \{\} \; > pdfinfo.log
```

This says to ignore files named "pdfinfo.log" (`! -name pdfinfo.log`) and otherwise look for files (`-type f`) containing "pdfinfo" (`-exec grep ...`) that haven't been checked since the last time "pdfinfo.log" was modified (`-newer pdfinfo.log`). In my tests the first run (which initially creates the "pdfinfo.log" file) ran in 30 seconds but subsequents runs took just a few seconds. This was because the number of files to be searched through all directories was big enough it paid to pre-filter the results before handing them to `grep`.

Gawking at `awk`

I don't have much to say about `awk` other than:

1. It is named after its three authors, Aho, Weinberger and Kernighan, all three of whom are computer science greats from Bell Labs. The GNU version is called `gawk`, of course!
2. It is a "data driven scripting language." That's a fancy way of saying it was written specifically with slicing and dicing text in mind.
3. It generally is broken out when the typical *IX commands and shell features like pipes and redirection aren't enough.
4. Usually, if I start thinking of `awk`, I start thinking of a way to program the answer in another language, or reframe the question to get an answer not requiring `awk`.

That said, it is a powerful knife in the tool belt, and you should be aware it exists.

To whet your taste, here is the type of "one-liner" for which `awk` is famous, in this case formatting and printing a report on user ids from `/etc/passwd`:

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3 }' /etc/passwd
username: root      uid:0
username: daemon        uid:1
```

```
username: bin       uid:2
username: sys       uid:3
username: sync      uid:4
username: games     uid:5
username: man       uid:6
username: lp        uid:7
username: mail      uid:8
username: news      uid:9
username: uucp      uid:10
...and so on...
```

Chapter 7

Step 5. "Just a Series of Pipes"

`stdin/stdout/stderr`, redirects and piping between commands.

> *"Ceci n'est pas une pipe."* - René Magritte

The "UNIX philosophy" tends to be to have a bunch of small programs that each do one thing very well, and then to combine them together in interesting ways. The "glue" for combining them together is often the "piping" or redirection of "streams" of data (typically text) between programs, each doing one small change to the stream until it is finally emitted on the console or saved to a file or sent over the Internet.

The first thing to note is there are three "file I/O streams" that are open by default in every *IX process:

- **stdin** - input, typically from the console in an interactive session. In the underlying C file system APIs, this is file descriptor 0.
- **stdout** - "normal" output, typically to the console in an interactive session. This is file descriptor 1.
- **stderr** - "error" output, typically to the console in an interactive session (so it can be hard to distinguish when intermingled with `stdout` output). This is file descriptor 2.

Note: The file descriptors will go from being trivia to important in just a bit.

When a program written in C calls `printf`, it is writing to `stdout`. When a `bash` script calls `echo`, it too is writing to `stdout`. When a command writes an

error message, it is writing to `stderr`. If a command or program accepts input from the console, it is reading from `stdin`.

In this example, `cat` is started with no file name, so it will read from `stdin` (a quite common *IX command convention), and echo each line to `stdout` until the "end of file," which in an interactive session can be emulated with `Ctrl-D`, in the example below shown as `^D` but not seen on the console in real life:

```
$ cat
This shows reading from stdin
This shows reading from stdin
and writing to stdout.
and writing to stdout.
^D
```

All Magic is Redirection

So one way to string things together in "the UNIX way" is with file redirection. This is a concept that works even in `CMD.EXE` and even with the same syntax.

Let's create a file with a single line of text in it. One way would be to `vi newfilename`, edit the file, save it, and exit `vi`. A quicker way is to simply use file redirection:

```
$ echo Hello, world > hw

$ ls -l
total 1
-rw-rwxr--+ 1 myuser mygroup 13 Oct 22 10:40 hw

$ cat hw
Hello, world
```

In this case the `> hw` tells `bash` to take the output that `echo` sends to `stdout` and send it to the file `hw` instead.

As mentioned above many *IX commands are set up to take one or more file names from the command line as parameters, and if there aren't any, to read from `stdin`. The `cat` command does that. While it doesn't save us anything over the above example, the following is illustrative of redirecting a file to `stdin` for a command or program:

```
$ cat < hw
Hello, world
```

Finally, we need to deal with `stderr`. By convention it is sent to the console just like `stdout`, and that can make output confusing:

```
$ echo This is a > a

$ echo This is b > b

$ echo This is c > c

$ mkdir d

$ echo This is e > d/e

$ find . -exec cat \{\} \;
cat: .: Is a directory
This is a
This is b
This is c
cat: ./d: Is a directory
This is e
```

In the above, between echoing the contents of the `a`, `b`, `c` and `e` files, we see two error messages from `cat` complaining that `.` and `d` are directories. These are being emitted on `stderr`. One way to get rid of them would be to change find to filter for only files:

```
$ find . -type f -exec cat \{\} \;
This is a
This is b
This is c
This is e
```

But let's say the example is not so trivial, and we want to capture and log the error messages separately for later analysis. While we've seen `<` used to represent redirecting `stdin` and `>` used for redirecting `stdout`, how do we tell the shell we want to redirect `stderr`? Remember the discussion about file handles above? That's where those esoteric numbers come in handy! Consider the original problem of `stderr` being intermingled with `stdout`:

```
$ find . -exec cat \{\} \;
cat: .: Is a directory
This is a
This is b
This is c
cat: ./d: Is a directory
This is e
```

To redirect `stderr` we recall it is ***always*** file descriptor 2, and then we can use:

```
$ find . -exec cat \{\} \; 2>/tmp/finderrors.log
This is a
This is b
This is c
This is e

$ cat /tmp/finderrors.log
cat: .: Is a directory
cat: ./d: Is a directory
```

The `2>/tmp/finderrors.log` is the magic that is redirecting file descriptor 2 (`stderr`) to the log file `/tmp/finderrors.log`.

A very common paradigm is to capture both `stdout` and `stderr` to the same file. Here is how that is done, again using file descriptors:

```
$ find . -exec cat \{\} \; >/tmp/find.log 2>&1

$ cat /tmp/find.log
cat: .: Is a directory
This is a
This is b
This is c
cat: ./d: Is a directory
This is e
```

Now we see `stdout` being redirected to `/tmp/find.log` with `>/tmp/find.log`, and `stderr` (file descriptor 2) being sent to the same place as `stdout` (file descriptor 1) with `2>&1`.

One final note is the difference between creating or re-writing a file and appending to it using redirection. The following creates a new `/tmp/find.log` file every time it runs (there is no need to `rm` it first):

```
$ find . -exec cat \{\} \; >/tmp/find.log
```

However, the next sample creates a new `/tmp/find.log` file if it doesn't exist, but otherwise appends to it:

```
$ find . -exec cat \{\} \; >>/tmp/find.log
```

Note: There is also a variation on input redirection using `<<`, but it is used mostly in scripting and is outside the scope of this book.

Everyone Line Up

So we can see that we could pass things between programs by redirecting `stdout` to a file and then redirecting that file to `stdin` on the next program, and so on. But *IX environments take it a bit further with the concept of a command "pipeline" that allows directly sending `stdout` from one program into `stdin` of another.

```
$ cat *.txt | tr '\\' '/' | while read line ; do ./mycmd "$line" ; done
```

This little one-liner starts showing off the usefulness of small programs, each doing one thing. In this case:

1. `cat` echos the contents of all `.txt` files to `stdout`, which is piped to...
2. `tr` translates any backslash characters (here "escaped" as `'\\'` because the backslash character is a special character) to forward slashes (`/`), before sending it into...
3. A `while` loop that reads each line into a variable called `$line` and then calls...
4. Some custom script or program called `./mycmd` passing in the value of each `$line`.

Think about the power of that. `cat` didn't know there were multiple `.txt` files or not - the shell expansion of the `*.txt` wildcard did that. It read all those files and echoed them to `stdout` which in this case was a pipeline sending each line in order to another command to transform the data, before sending each line to the custom code in `mycmd`, that only expects a single line or value each time it is run. It has no idea about the `.txt` files or the transformation or the pipeline!

That is the UNIX philosophy at work.

There are some nice performance benefits for this approach, too. In general Linux & Co. will overlap the processing by starting all the commands in the pipeline, with the ones on the right getting data from the ones further "upstream" to the left as soon as it is written, instead of using file redirection where one program would have to finish completely running and writing out to a file before the next program could start and read in that file as input.

Finally, if you want to capture something to a file ***and*** see it on the console at the same time, that is where the `tee` command comes in:

```
find . -name error.log | tee > errorlogs.txt
```

This would write the results of finding all files names `error.log` to the console and also to `errorlogs.txt`. This is useful when you are manually running things and want to see the results immediately, but also want a log of what you did.

Chapter 8

Step 6. `vi`

How to stay sane for 10 minutes in `vi`. Navigation, basic editing, find, change/change-all, cut and paste, undo, saving and canceling. Plus easier alternatives like `nano`, and why `vi` still matters.

> *"You're too young to know."* - Vi (*Grease*)

`vi` stands for *visual editor* (as well as the Roman numeral for 6, which is why it is this chapter), and once you use it you will understand what editing from the command line must've been like for `vi` to seem both "visual" and a step forward.

Many Linux clones don't use `vi` proper, but a port called `vim` ("**vi** **im**proved") "that can then be accessed via the alias `vi`. The differences tend to be minor, with `vim` being more customizable.

`vi` and a similar editor, `emacs`, both tend to trip up users from GUI operating systems such as Windows or OS X that have editors like Notepad that are always ready for user input.

Instead, `vi` typically starts in "command mode," where keystrokes execute various navigation and editing commands. To actually insert text requires a keystroke such as `i` while in command mode, which then causes `vi` to go into "insert mode." Insert mode is what most Windows users expect from an editor, i.e., when you type the line changes. The `ESC` key exits insert mode.

It is as hard to get used to as it sounds, and you ***will*** execute text you were meaning to insert as commands, and commands that you were meaning to execute as text you ***will*** insert as text, and sooner or later you ***will*** enter `vi` commands into Notepad. That will be the day you know you've become truly tainted.

This will not even begin to scratch the surface of `vi`, when there are many, many books and web sites just on wielding it to its full potential. In the hands of

someone who has mastered it, `vi` can do some really remarkable feats of editing way beyond the capability of most modern GUI programming environments.

Command Me

Again, when you first open `vi` it is in "command mode." That means any keystrokes you enter will "do something." The "something" to be done may be navigating around the file, inserting, deleting or changing text, manipulating lines, "undo", writing the changes to disk and the like.

What are commands? Well, for example `d` means "delete." We'll talk about how to specify ***what*** to delete next. `i` tells `vi` to enter "insert mode" at the point where the cursor is. `0` navigates to the start of the current line, and so on.

Commands can have ***modifiers*** preceding and following them. Consider the "delete" command, `d`. If we follow with `w` as in `dw` while in command mode, it will delete a whitespace-delimited "word" starting at where the cursor is through (including) the next whitespace character.

If the `|` in the following represents the cursor:

```
This is a wo|rd and so is this.
```

Then typing `dw` will delete from the cursor position the characters `r`, `d` and the space, leaving the following:

```
This is a wo|and so is this.
```

We can also specify a number of times we want to perform a command before the command. So now if we wanted to delete three words from the cursor position in the above, we'd use `3dw` and end up with:

```
This is a wo|this.
```

Again, in all these examples the `|` represents the cursor.

There is a little bit of nuance in using command modifiers. Consider the `r` (*replace*) command. It is typically used to change the single ***character*** under the cursor. You may be tempted to think you can do something like `rw` for "replace word," but it is actually going to simply replace the current character with a `w`, whereas the real command for doing that is `cw` ("change word"). In addition, you can use repeaters as above, just be sure you understand `r` means "replace a single character," so `3rx` executed on:

```
This is a wo|this.
```

. . . results in:

```
This is a woxx|xs.
```

To quit without saving enter `:q`. To write any file changes to disk use `:w`. To save and quit, type `:wq`.

Undo Me

`u` is the "undo" command. It "undoes" or reverts the last change. You can undo the last *n* changes just as you'd expect, e.g., `3u` undoes the last three changes.

If you want to just cancel out of the file without writing any changes to disk, use `:q!` (the `!` means to force the quit without saving).

If you want to protect yourself from inadvertent changes to a file you can always open it using `view`, the alias for `vi` invoked in read-only mode.

Circumnavigating `vi`

In modern implementations of `vi` (like `vim`) running under modern shells the arrow and page keys will work as you expect, ***in general***. However, you may want to be aware that when in insert mode, while the left and right arrows may work for navigation, often the up and down arrows can introduce "garbage" characters into the file (since you are in insert mode). This is because the keymappings for those keys aren't being interpreted correctly. I usually just swear, exit insert mode, hit `u` and try again.

As an example, under Cygwin I went into `vi`, went into insert mode after the first line, typed in "This is a new line" and then hit the up arrow five times, yielding this:

```
This is a word and so is this.
A
A
A
A
A
This is a new line
```

When in command mode, there are multiple ways to jump around in the file besides using the arrow and page keys:

- `0` - jumps to the beginning of the current line.
- `$` - jumps to the end of the current line.
- `w` - jumps forward a whitespace-delimited "word" on the current line (and of course `3w` would jump forward three "words").
- `b` - jumps back a whitespace-delimited "word" on the current line.
- `G` - jumps to end of file.
- `:0` - jumps to start of file.
- `/foo` - find "foo" going forward toward the end of the file.
- `?foo` - find "foo" going backward toward the front of the file.
- `n` - find the next instance of the search text specified by the last `/` or `?`.
- `p` - find the previous instance of the search text specified by the last `/` or `?`.

Insert Tab A Into Slot B

There are multiple ways to enter insert mode, but only one way to escape it (pun intended - `ESC`, get it?)

- `i` - enters insert mode at the current cursor position.
- `I` - enters insert mode at the beginning of the current line.
- `A` - enters insert mode (appends) at the end of the current line.
- `o` - inserts a new line under (lowercase `o` = "lower" or "below") the current line and puts the cursor on it in insert mode.
- `O` - inserts a new line over (uppercase `O` = "upper" or "above") the current line and puts the cursor on it in insert mode.

Ctrl-X, Ctrl-C, Ctrl-V

When you copy or cut/delete it, it goes into a "buffer." There are ways to access multiple buffers, but mostly you want the very last thing to be put in the buffer, especially for copying (or cutting) and pasting. Note that "cutting" and "deleting" are synonymous, since deleting puts the deleted text in the buffer.

Another thing to understand is that a command "doubled" or repeated typically means "the whole line." So `dd` means "delete the whole line the cursor is currently on."

So if deleting is synonymous with cutting, and the cursor is on the second line:

```
This is a word and so is this.
This is a new line.|
```

Then executing `dd` leaves:

```
|This is a word and so is this.
```

We know "This is a new line." went into the buffer. We can paste it back above the current line with `P`, which would result in:

```
|This is a new line.
This is a word and so is this.
```

Here are some more examples:

- `p` - paste the buffer into the current line starting after the cursor location.
- `3dd` - delete (cut) three lines into the buffer.
- `5yw` - "yank" (copy) five words starting at the current cursor position into the buffer.

Change Machine

The hardest thing to get down in `vi` is the *substitute* (change) command. Its syntax is esoteric, but once you've memorized it, it becomes intuitive.

The most common scenario is the "change all" command. Given the following file:

```
This is a new line
This is a word
and so is this
This and thus
This and this and this
```

Let's change all "this" to "that" by using:

```
:1,$s/this/that/
```

We'll get into the details in a bit, but the results are interesting, and not what we'd expect:

```
This is a new line
This is a word
and so is that
This and thus
This and that and this
```

It only changed the "that" at the end of the third line, and the middle "that" on the last. Why? Two reasons:

1. The substitute command is case sensitive, just like everything else in Linux, unless you tell it to be *insensitive*.
2. The substitute command only makes one change per line unless you tell it to change *globally*.

So let's hit `u` to reset (undo) the file, and try again with this:

```
:1,$s/this/that/i
```

Results in:

```
that is a new line
that is a word
and so is that
that and thus
that and this and this
```

That's better. There is at least one "that" on every line that had a "this," so passing the `i` ("insensitive") switch at the end of the `s` (substitute) command helped with that. But we still didn't get all the "this" words changed, as the last line shows. Hit `u` and try one more time with this:

```
:1,$s/this/that/gi
```

Results in:

```
that is a new line
that is a word
and so is that
that and thus
that and that and that
```

That's what we wanted! In general, if you are looking for a Windows Notepad-like, case insensitive "change all," the magic string to remember is:

```
:1,$s/from/to/gi
```

Picking that apart, we have:

- `:` - tells `vi` a special command is coming.
- `1,$` - specifies a line range, in this case from the first (`1`) to last (`$`) lines in the file. You can of course use other lines numbers to restrict the range, and there are other ways to create ranges as well.
- `s` - substitute (change) command.
- `/from` - "from" pattern (regular expression).
- `/to` - "to" (results).
- `/gi` - optional switches, `g` means "global" (change all instances on a line, not just the first one), `i` means (case) "insensitive."

Regular expressions you say! Now we have two problems! But consider:

```
that is a new line
that is a word
and so is that
that and thus
that and that and that
```

First, let's capitalize all `t` characters, but only where they are at the beginning of the line:

```
:1,$s/^t/T/
```

Yields:

```
That is a new line
That is a word
and so is that
That and thus
That and that and that
```

Now let's change all instances of "that" at the end of a line to be "that."

```
:1,$s/that$/that./
```

Ends up with:

```
That is a new line
That is a word
and so is that.
That and thus
That and that and that.
```

And finally as a fun exercise for the reader, using the full power of regular expressions see if you can figure out how this is adding commas to the end of lines that don't already have a period:

```
:1,$s/\([^.]$\)/\1,/
```

Like this:

```
That is a new line,
That is a word,
and so is that.
That and thus,
That and that and that.
```

The Unseen World

Any technical person knows that all the binary permutations of possible values for a byte aren't mapped to visible characters. Some are "control characters" that range back to the teletype days. For example, a tab character is hexadecimal 9 (`0x09`), but is often represented as `\t` in many programming languages, regular expressions and the like.

Similarly, the "end of line" is marked by a control character. Or in the case of Windows, ***two*** control characters. And this causes no end of problems when editing files that can be opened on both *IX systems and Windows.

On *IX, the line feed control character (`0x0a`, or `\n`) is all that marks the end of a line. For historical reasons (CP/M), Windows ends each line with two control characters, carriage return (`0x0d`, or `\r`) and line feed. The two together are often referred to as "CRLF."

This difference manifests in two ways:

1. If you've ever opened a file on Windows in Notepad and all the lines "flow" even though they're supposed to be individual lines, that means it is probably using *IX end-of-lines (`\n`) only. Use a line feed aware editor such as Notepad++ instead.

2. If you open a file in `vi` and it has a `^M` at the end of every line and/or at the bottom you see something like:

   ```
   "Agenda.md" [dos format] 16 lines, 1692 characters
   ```

   Either of those mean the file lines each end with "CRLF" (`\r\n`). To change it in `vi` you can use the following command (`ff` means "file format":

   ```
   :set ff=unix
   ```

Since regular expressions have syntax for expressing control codes in either shorthand (`\t`) or as hexadecimal, you can alter control codes in `vi` easily. For example, to change all tab characters to four spaces:

```
:1,$s/\t/    /g
```

Let's Get Small

So, `vi` is the best we can do? No. On many Linux systems an alternative terminal-based editor will be installed, often several. There may be `emacs`, which will make you yearn for the simplicity of `vi`.

Here are two jokes that are only funny once you've used `emacs`:

> *"'emacs' stands for 'escape', 'meta', 'alt', 'control', 'shift'."*

> *"'emacs' is a good operating system, but it could use an editor."*

If those are funny to you, then you have already been infected by `emacs`. May God have mercy on your soul.

But there may also others, notably `pico` and its successor, `nano`. You can see the difference the second you see a file open in `nano`:

```
  GNU nano 2.2.6                File: Title.md

|[author]: # (Jim Lehmer)
[title]: # (Jim's Ten Steps to Linux Survival)

# Jim's Ten Steps to Linux Survival

**By Jim Lehmer**
```

```
<a rel="license"
href="http://creativecommons.org/licenses/by-sa/4.0/"><img
alt="Creative Commons License" style="border-width:0"
src="https://i.creativecommons.org/l/by-sa/4.0/88x31.png"
/></a><br /><span xmlns:dct="http://purl.org/dc/terms/"
property="dct:title">Jim's Ten Steps to Linux Survival</span>
by <span xmlns:cc="http://creativecommons.org/ns#"
property="cc:attributionName">James
Lehmer</span> is licensed under a <a rel="license"
href="http://creativecommons.org/licenses/by-sa/4.0/">Creative Commons
Attribution-ShareAlike 4.0 International License</a>.
                                    [ Read 18 lines ]
^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

Two things to note about the above:

1. The cursor (represented above by `|`) is already in "insert mode" like you would expect in a "normal" editor like Notepad.
2. Those lines at the bottom are commands that can be invoked by shortcuts. For example, `^O` means `Ctrl-O` and stands for "WriteOut" or "Save." That's probably easier to remember than `:w` in `vi`, especially since it is reminding you of it right there on the screen!

So why not always use `nano`? Why does this book harp on and on `vi`? Why do I insist on keeping all this arcane `vi` nonsense loaded in my head (and I do!)? Because often, like in the nightmare scenario I posed in the *Introduction*, you may not have control over the system, no ability to install packages - you have to take what the system has. And it's a pretty sure bet it is going to have `vi`. So if you have `nano` (or `pico`), use it! But if you don't, grit your teeth, remember "insert mode" vs. "command mode", and use `vi`.

And if you have the opportunity to use `emacs`... don't.

Chapter 9

Step 7. The Whole Wide World

curl, wget, ifconfig, ping, ssh, telnet, /etc/hosts and email before Outlook.

> *"Gopher, Everett?"* - Delmar O'Donnell (*O Brother, Where Are Thou?*)

If Sun's motto, "The network is the computer" is correct, then of course Linux and similar systems must be able to access the network from the command line and scripts.

For example, our friend `ping` is there:

```
# ping www.yahoo.com
PING fd-fp3.wg1.b.yahoo.com (98.138.253.109) 56(84) bytes of data.
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=1 ttl=46 time=62.1 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=2 ttl=46 time=61.9 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=3 ttl=46 time=70.9 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=4 ttl=46 time=65.0 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=5 ttl=46 time=63.6 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=6 ttl=46 time=59.9 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=7 ttl=46 time=60.8 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=8 ttl=46 time=60.8 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=9 ttl=46 time=60.3 ms
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=10 ttl=46 time=60.1 ms
^C
--- fd-fp3.wg1.b.yahoo.com ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9004ms
rtt min/avg/max/mdev = 59.933/62.581/70.935/3.191 ms
```

One difference with `ping` is that by default in Linux `ping` doesn't stop until the user presses `Ctrl-C` (which sends the SIGINT interrupt to the program). In this way it acts more like `ping -t` in `CMD.EXE`. Also, be aware that on Cygwin `ping` is still the system (Windows) `ping`.

`traceroute` works, too (although for once its name is longer than the `CMD.EXE` counterpart).

```
# traceroute www.yahoo.com
traceroute to www.yahoo.com (98.138.252.30), 30 hops max, 60 byte packets
 1  10.208.3.254 (10.208.3.254)  0.720 ms  0.706 ms  0.693 ms
 2  10.208.6.53 (10.208.6.53)  0.808 ms  0.896 ms  0.943 ms
 3  10.208.6.46 (10.208.6.46)  2.632 ms  2.636 ms  2.634 ms
 4  kcm-priv-20.inet.qwest.net (63.159.159.185)  30.786 ms  30.852 ms  31.350 ms
 5  * * *
 6  67.134.114.230 (67.134.114.230)  30.441 ms  29.811 ms  30.372 ms
 7  67.130.10.174 (67.130.10.174)  32.267 ms  32.700 ms  32.789 ms
 8  67.130.10.103 (67.130.10.103)  32.416 ms  32.421 ms  32.420 ms
 9  min-edge-13.inet.qwest.net (67.130.30.21)  33.878 ms  31.719 ms  34.749 ms
10  chp-brdr-03.inet.qwest.net (67.14.8.194)  45.668 ms  55.177 ms  45.629 ms
11  63.146.27.18 (63.146.27.18)  46.371 ms  46.333 ms  47.234 ms
...and so on...
```

You can do some digging in DNS with `dig`:

```
# dig yahoo.com

; <<>> DiG 9.8.4-rpz2+rl005.12-P1 <<>> yahoo.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 18148
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;yahoo.com.                     IN      A

;; ANSWER SECTION:
yahoo.com.              1705    IN      A       206.190.36.45
yahoo.com.              1705    IN      A       98.139.183.24
yahoo.com.              1705    IN      A       98.138.253.109

;; Query time: 17 msec
;; SERVER: 10.208.2.4#53(10.208.2.4)
;; WHEN: Fri Oct 23 13:16:51 2015
;; MSG SIZE  rcvd: 75
```

And `whois`:

```
# whois yahoo.com

Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Server Name: YAHOO.COM.ACCUTAXSERVICES.COM
   IP Address: 98.136.43.32
   IP Address: 66.196.84.168
   Registrar: WILD WEST DOMAINS, LLC
   Whois Server: whois.wildwestdomains.com
   Referral URL: http://www.wildwestdomains.com
...and so on...
```

`sudo` Make Me a Sandwich

It may not be the best place to discuss it, but we've finally come to a point where your normal user account may not have access to these tools. On many systems network commands are considered "system" or privileged commands and are restricted.

One way to run restricted commands is to log in as a "elevated" or privileged user, such as `root`. But this is frowned on, and many distros today rely on the `sudo` command to act as a way for a normal user to signal they want to escalate their privileges temporarily (presuming they are allowed to do so, which is usually indicated by being a member of the `sudo` group or similar.

In a sense, `sudo` is similar to Windows User Access Control (UAC, or "Are you sure?") prompts. They ensure a human is in control, in the case of `sudo` by prompting for the user's password (if multiple commands are invoked by `sudo` within a short time period, you will not be reprompted for a password each time).

Here is a really common example on Debian-based systems:

```
$ apt-get update
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denied)
E: Unable to lock directory /var/lib/apt/lists/
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
```

The error message, especially the last line, is pretty clear. Let's try it again with `sudo`:

```
$ sudo apt-get update
[sudo] password for myuser:
Ign http://extra.linuxmint.com rafaela InRelease
Ign http://packages.linuxmint.com rafaela InRelease
Hit http://security.ubuntu.com trusty-security InRelease
Hit http://extra.linuxmint.com rafaela Release.gpg
Hit http://packages.linuxmint.com rafaela Release.gpg
Hit http://security.ubuntu.com trusty-security/main amd64 Packages
Ign http://archive.ubuntu.com trusty InRelease
Ign http://archive.canonical.com trusty InRelease
Hit http://security.ubuntu.com trusty-security/restricted amd64 Packages
Hit http://extra.linuxmint.com rafaela Release
...and so on...
```

Now you should get the punchline to this comic, and hence the title of this section.

Surfin' the Command Prompt

You can browse the web from the command prompt using something like `lynx`. A text-based browser isn't too exciting, but it can have its purposes (like quickly testing network access from a command prompt). For example, `lynx http://google.com` yields:

```
   Search Images Maps Play YouTube News Gmail Drive More »
   Web History | Settings | Sign in

   Google

                                                  ------------------------------------------
                                                  Google Search  I'm Feeling Lucky


   Advertising Programs     Business Solutions     +Google     About Google

                                                                    © 2015 - Privacy
```

There are two other commands that are used to pull down web resources and save them locally - `curl` and `wget`. Both support HTTP(S) and FTP, but `curl` supports even more protocols and options, while `curl` tends to be the simplest

to just "grab a file and go." You see both used often in install scripts that then download more bits from the internet:

```
wget -O - http://foocorp.com/installs/install.sh | bash
```

Or:

```
curl http://foocorp.com/installs/install.sh | bash
```

Note: As always, you should be cautious when downloading and executing arbitrary bits, and this technique doesn't lessen your responsibility there.

You've Got Mail

You can send and receive email from the command prompt. Reading email will be rare, but if the system has `pine` installed, that's probably the most intuitive from a non-UNIX perspective (although it is still obviously a command line program). Otherwise look for `mutt`.

Sending email is more interesting, especially from shell scripts. There are multiple ways, but `email` is straightforward enough:

```
email --blank-mail --subject "Possibly corrupted files found..." \
  --smtp-server smtp --attach badfiles.csv --from-name NoReply \
  --from-addr noreply@mycorp.com alert@mycorp.com
```

Let's Connect

There are two primary ways to get an interactive "shell" session on a remote machine. The first is the venerable `telnet` command. It isn't used very often for actual interactive sessions any more (for one, because it sends credentials in plain text on the wire). However, because you can specify the port number, it is still handy for testing and debugging text-based protocols such as SMTP or HTTP. In the following, after opening a `telnet` connection on port 80 to Google, I simply entered the HTTP protocol sequence `GET / HTTP/1.1` followed by a blank line to get Google to return its home page:

```
# telnet google.com 80
Trying 216.58.216.110...
Connected to google.com.
Escape character is '^]'.
```

```
GET / HTTP/1.1

HTTP/1.1 200 OK
Date: Fri, 23 Oct 2015 18:26:04 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See http://www.google.com/support/acc...
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: PREF=ID=111111111111111:FF=0:TM=1445624764:LM=1445624764:V=1:S=YV3fLcTe...
Set-Cookie: NID=72=HLgGubMnO1ThhvhOAmvehue96EKTh9D6F19zidZQU-E9AibEg2Op6...
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

8000
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"...
...and so on...
```

To get a modern, secure shell to a remote machine, use `ssh`, passing in the userid and server like this:

```
ssh myuser@remoteserver
```

You will be prompted for credentials (or you can use certificates, but that is ***way*** beyond this text's goals).

You can also use the `SSH` protocol to securely transfer files between systems with the `scp` command. It works like this for a recursive directory copy:

```
scp -r myfiles/* myuser@remoteserver:/home/myuser/myfiles/.
```

In this case we are copying the files in `myfiles` and its subdirectories to `/home/myuser/myfiles/` on `remoteserver` logged in as `myuser`.

Note: The first time you log into a remote server with `ssh` or `scp` you will be asked to accept the remote server's "fingerprint." You can usually just say "yes":

```
~# ssh myuser@remotehost
The authenticity of host '[remotehost] ([10.0.2.3]:22)' can't be established.
ECDSA key fingerprint is 98:70:17:38:db:d0:16:ee:b2:93:08:3e:30:25:14:70.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[remotehost],[10.0.2.3]:22' (ECDSA) to the list of known hosts.
```

```
myuser@remotehost's password:
Linux remotehost 3.2.0-4-amd64 #1 SMP Debian 3.2.65-1+deb7u2 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Oct 20 09:37:10 2015 from otherhost
$
```

Network Configuration

We won't dive too deep into configuring a network, but there are a few things you should know about right away. The first is the `ifconfig`. While you can use `ifconfig` to alter your networking settings, it is most commonly used to get a quick display of them:

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:56:a3:35:fe
          inet addr:10.0.2.3  Bcast:10.0.2.255  Mask:255.255.252.0
          inet6 addr: fe80::255:56ff:fea3:35fe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:364565022 errors:0 dropped:386406 overruns:0 frame:0
          TX packets:35097654 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:34727642861 (32.3 GiB)  TX bytes:195032017498 (181.6 GiB)

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:111207 errors:0 dropped:0 overruns:0 frame:0
          TX packets:111207 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6839306 (6.5 MiB)  TX bytes:6839306 (6.5 MiB)
```

To see what DNS servers the system is using:

```
# cat /etc/resolv.conf
domain mydomain.com
search mydomain.com
```

```
nameserver 10.0.2.1
nameserver 10.0.2.2
```

And to see any local overrides of network names or aliases:

```
# cat /etc/hosts
127.0.0.1       localhost
```

Note: The UNIX `hosts` file is the basis for the Windows version (`C:\Windows\System32\drivers\etc\hosts`) and has similar syntax.

Chapter 10

Step 8. The Man Behind the Curtain

/proc, /dev, ps, /var/log, /tmp and other things under the covers.

> *"As always, should any member of your team be caught or killed, the Secretary will disavow all knowledge of your actions."* - voice on tape (*Mission: Impossible*)

This section will cover some "background" techniques that are valuable for system monitoring, problem determination and the like. Depending on your role and access levels, some of these commands may not be available to you, or may require `sudo` access.

All Part of the Process

To see what *processes* you are running, use `ps`:

```
# ps
  PID TTY          TIME CMD
14691 pts/0    00:00:00 bash
25530 pts/0    00:00:00 ps
```

To show processes from *all* users in a process *hierarchy* (child processes indented under parents), use `ps -AH`:

```
ps -AH
# ps -AH
  PID TTY          TIME CMD
    2 ?        00:00:00 kthreadd
    3 ?        00:05:00   ksoftirqd/0
    5 ?        00:00:00   kworker/u:0
    6 ?        00:02:38   migration/0
    7 ?        00:01:06   watchdog/0
    8 ?        00:02:37   migration/1
   10 ?        00:05:02   ksoftirqd/1
   12 ?        00:00:59   watchdog/1
   13 ?        00:00:00   cpuset
   14 ?        00:00:00   khelper
   15 ?        00:00:00   kdevtmpfs
...and so on...
```

You can *kill* a process using the `kill` command, which takes a process id and optionally a "signal". Here is an example looking for any running instance of `vi` and sending it a `kill` command:

```
ps -A | grep vi | kill `cut -f2 -d" "`
```

That's:

- **`ps -A`** - list all running processes.
- **`|`** - pipe `stdout` from `ps` to next command.
- **`grep vi`** - find all instances of `vi` (be careful, because that would include `view` and anything else containing the string `vi`, too).
- **`|`** - pipe `stdout` from `grep` to next command.
- **`kill`** - send a `SIGINT` signal to a process specified by:
- **`` `cut -f2 -d" "` ``** - execute the `cut` command and take the second space-delimited field (in this case the process id - the first "field" is just leading spaces), and place the results of the command execution as the parameter to the `kill` command.

To monitor the ongoing CPU, memory and other resource utilization of the *top* processes, you use the `top` command, which unlike most in this book updates dynamically every second by default:

```
top - 14:11:26 up 106 days,  5:24,  2 users,  load average: 0.11, 0.05, 0.05
Tasks:  95 total,   1 running,  94 sleeping,   0 stopped,   0 zombie
```

```
%Cpu(s):  0.2 us,  0.8 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2061136 total,  1909468 used,   151668 free,   151632 buffers
KiB Swap:  4191228 total,   287620 used,  3903608 free,   654900 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 9715 git       20   0  525m 230m 4376 S   0.7 11.4  10:11.44 ruby
 9171 git       20   0  520m 229m 4672 S   0.3 11.4  10:27.97 ruby
22899 root      20   0     0    0    0 S   0.3  0.0   0:30.16 kworker/1:0
    1 root      20   0 10648  584  560 S   0.0  0.0   1:02.60 init
    2 root      20   0     0    0    0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S   0.0  0.0   0:38.05 ksoftirqd/0
    5 root      20   0     0    0    0 S   0.0  0.0   0:00.00 kworker/u:0
    6 root      rt   0     0    0    0 S   0.0  0.0   0:12.23 migration/0
    7 root      rt   0     0    0    0 S   0.0  0.0   0:24.83 watchdog/0
    8 root      rt   0     0    0    0 S   0.0  0.0   0:13.01 migration/1
   10 root      20   0     0    0    0 S   0.0  0.0   0:34.55 ksoftirqd/1
   12 root      rt   0     0    0    0 S   0.0  0.0   0:21.38 watchdog/1
   13 root       0 -20     0    0    0 S   0.0  0.0   0:00.00 cpuset
...and so on...
```

Note: Use `Q` or `Ctrl-C` to exit `top`.

When All You Have is a Hammer

Remember that one of the primary UNIX philosophies is that everything is a file or can be made to look like a file, including network streams, device output and the like. This is a really powerful concept, because it allows you to access things with tools that have ***no idea*** what they are working on, as long as it "looks like" a file (or stream of text).

One of the places this has become really handy is in the `/proc` "file system." On modern Linux systems, there is typically a `/proc` directory that looks like directories and files:

```
# ls /proc
1     1776  2     2244  2308  2415  2599  2693  5     9171        cmdline      fb
10    178   20    2269  2311  2416  26    3     5030  9174        consoles     filesyste
12    1781  2052  2287  2333  2417  2611  3120  5032  9715        cpuinfo      fs
13    1783  21    22899 2338  2418  2612  31651 560   9718        crypto       interrupt
130   1790  211   2297  2367  2422  2613  3197  570   99          devices      iomem
14    18    212   23    23835 2432  2614  32502 5991  acpi        diskstats    ioports
15    180   2165  2304  23841 24426 2615  355   6     asound      dma          irq
16    181   2191  2305  2395  25    2616  4691  7     buddyinfo   dri          kallsyms
17    182   22    2306  24    2550  26735 479   8     bus         driver       kcore
1713  19    2225  2307  2414  2556  26736 480   88    cgroups     execdomains  keys
```

What is all that? Well if we look a little closer:

```
# ls -l /proc
total 0
dr-xr-xr-x  8 root       root                    0 Sep 18 11:17 1
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 10
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 12
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 13
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 130
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 14
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 15
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 16
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 17
dr-xr-xr-x  8 root       root                    0 Oct 23 13:55 1713
dr-xr-xr-x  8 statd      nogroup                 0 Oct 23 13:55 1776
...and so on...
```

. . . we can see that the entries with numeric names are directories. Let's look in one of those directories:

```
# ls -l /proc/1
total 0
dr-xr-xr-x 2 root root 0 Oct 23 14:23 attr
-rw-r--r-- 1 root root 0 Oct 23 14:23 autogroup
-r-------- 1 root root 0 Oct 23 14:23 auxv
-r--r--r-- 1 root root 0 Oct 23 14:23 cgroup
--w------- 1 root root 0 Oct 23 14:23 clear_refs
-r--r--r-- 1 root root 0 Oct 23 14:23 cmdline
-rw-r--r-- 1 root root 0 Oct 23 14:23 comm
-rw-r--r-- 1 root root 0 Oct 23 14:23 coredump_filter
-r--r--r-- 1 root root 0 Oct 23 14:23 cpuset
lrwxrwxrwx 1 root root 0 Oct 23 14:23 cwd -> /
-r-------- 1 root root 0 Oct 23 14:23 environ
lrwxrwxrwx 1 root root 0 Oct 23 14:23 exe -> /sbin/init
...and so on...
```

This contains a lot of information on the process with process id (PID) #1. If the directory listing shows the entry as a file, it can be examined and holds ***current*** statistics for whatever the file name implies. If it is a directory it will hold other entries (files or directories) with yet more statistics.

In addition, there are system-wide statistics, such as `/proc/cpuinfo`:

```
# cat /proc/cpuinfo
processor       : 0
```

```
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Xeon(R) CPU           X5690  @ 3.47GHz
stepping        : 1
microcode       : 0x15
cpu MHz         : 3458.000
cache size      : 12288 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 11
...and so on...
```

Sawing Logs

Many Linux components and subsystems log to `/var/log`. Here is a pretty standard directory listing for it on a Debian system:

```
# ls /var/log
alternatives.log       auth.log.2.gz    debug       dmesg.4.gz     kern.log       mail.info
alternatives.log.1     auth.log.3.gz    debug.1     dpkg.log       kern.log.1     mail.info.
alternatives.log.2.gz  auth.log.4.gz    debug.2.gz  dpkg.log.1     kern.log.2.gz  mail.info.
alternatives.log.3.gz  btmp             debug.3.gz  dpkg.log.2.gz  kern.log.3.gz  mail.info.
apache2                btmp.1           debug.4.gz  dpkg.log.3.gz  kern.log.4.gz  mail.info.
apt                    daemon.log       dmesg       dpkg.log.4.gz  lastlog        mail.log
aptitude               daemon.log.1     dmesg.0     exim4          lpr.log        mail.log.1
aptitude.1.gz          daemon.log.2.gz  dmesg.1.gz  faillog        mail.err       mail.log.2
auth.log               daemon.log.3.gz  dmesg.2.gz  fsck           mail.err.1     mail.log.3
auth.log.1             daemon.log.4.gz  dmesg.3.gz  installer      mail.err.2.gz  mail.log.4
```

Some, like `samba` are their own subdirectories with log files under that. Others are log files that get "rotated" from the most current (no suffix) through ever older ones (increasing suffix number, e.g., `mail.log.2`).

If you are pursuing a problem with a specific subsystem (like `samba`), it is good to start in its log files. The two files of general interest are `dmesg`, which holds kernel-level debug messages and usually is useful for debugging things like device driver issues. The other is `messages`, which holds more general "system" messages.

Let's look for kernel errors when booting:

```
# cat dmesg | grep -i error
[    2.310161] Error: Driver 'pcspkr' is already registered, aborting...
[    2.754699] EXT4-fs (sda1): re-mounted. Opts: errors=remount-ro
```

It's All Temporary

By convention, temporary files are written to `/tmp`. You can place your own temporary or "work" files there, too. It's a great place to unzip install bits, for example. Just note that the temporariness is enforced in that when the system reboots, `/tmp` is reset to empty.

Chapter 11

Step 9. How Do You Know What You Don't Know, `man`?

`man`, `info`, `apropos`, Linux Documentation Project, Debian and Arch guides, StackOverflow and the dangers of searching for "`man find`" or "`man touch`" on the internet.

> *"You're soaking in it."* - Palmolive commercial

The biggest issue with bootstrapping into *IX is not the lack of documentation but almost the surplus of it, coupled with a severe "RTFM" attitude by most old-timers toward most newbies. Besides the typical "Google" and "StackOverflow" answers, there are actually lots of very reliable places to turn to for information:

`man`, is that `info apropos`?

There are three commands that are the basis for reading *IX documentation within* IX itself - `man`, `info` and `apropos`.

`man` is short for *manual pages*, and is used to display the main help for most *IX commands. For example, `man ls` shows:

```
LS(1)                                                    User Commands

NAME
       ls - list directory contents
```

```
SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by
       default).  Sort entries alphabetically if none of -cftuvSUX nor
       --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..
...and so on...
```

Note: `man` uses `less` as a paginator, with all that means, including the same navigation and search keys, and most important to remember - `Q` to quit. How do I know this? Because of course you can `man man`!

Notice the `LS(1)` part. The UNIX manual was originally divided into multiple sections by AT&T. Section 1 is normal user commands. Section 5 is file formats (like for config files), and section 8 is for system administration commands. You usually don't care, and can `man ls` or `man ifconfig` to your heart's content.

But sometimes there are duplicate names in the different sections. For example, there is both a `passwd` command and a `passwd` file format (for `/etc/passwd`). By default, `man passwd` will show you the documentation from the lowest numbered section with a match, in this case section 1, usually referred to as `passwd(1)` to disambiguate which thing we're talking about:

```
PASSWD(1)                                                    User Commands

NAME
       passwd - change user password

SYNOPSIS
       passwd [options] [LOGIN]

DESCRIPTION
       The passwd command changes passwords for user accounts. A normal user may only change
       superuser may change the password for any account.  passwd also changes the account c

   Password Changes
```

```
       The user is first prompted for his/her old password, if one is present. This passwor
...and so on...
```

To see the `man` page for the `passwd` file format, we have to explicitly specify the section, in this case by using `man 5 passwd`:

```
PASSWD(5)                                                File Formats and Conversions

NAME
       passwd - the password file

DESCRIPTION
       /etc/passwd contains one line for each user account, with seven fields delimited by c

       ·   login name

       ·   optional encrypted password

       ·   numerical user ID

       ·   numerical group ID
...and so on...
```

Besides `man`, many GNU tools come with help in `info` format, which is from `emacs`. While `info` is much better at enabling complex help files with navigation I am not a fan because I tend not to hold all the keystrokes in my head. The biggest thing to remember if you do something like `info vi` is that `q` quits the `info` command.

Finally, what if you don't know the name of the command? Well, each "man page" has a title and brief description, e.g., "passwd - change user password" in the `man passwd` output above. The `apropos` command can simply search those titles and descriptions for a word or phrase and show you all the results:

```
# apropos edit
dpatch-edit-patch (1) - maintain dpatch patches for a Debian source package
edit (1)              - execute programs via entries in the mailcap file
rediff (1)            - fix offsets and counts of a hand-edited diff
editor (1)            - Nano's ANOther editor, an enhanced free Pico clone
elfedit (1)           - Update the ELF header of ELF files.
ex (1)                - Vi IMproved, a programmers text editor
grub-editenv (1)      - edit GRUB environment block
msgfilter (1)         - edit translations of message catalog
nano (1)              - Nano's ANOther editor, an enhanced free Pico clone
pdbedit (8)           - manage the SAM database (Database of Samba Users)
```

```
pico (1)              - Nano's ANOther editor, an enhanced free Pico clone
psed (1)              - a stream editor
readline (3readline) - get a line from a user with editing
rnano (1)             - Restricted mode for Nano's ANOther editor, an enh...
rview (1)             - Vi IMproved, a programmers text editor
rvim (1)              - Vi IMproved, a programmers text editor
s2p (1)               - a stream editor
sed (1)               - stream editor for filtering and transforming text
sensible-browser (1) - sensible editing, paging, and web browsing
sensible-editor (1)  - sensible editing, paging, and web browsing
sensible-pager (1)   - sensible editing, paging, and web browsing
sudoedit (8)          - execute a command as another user
vi (1)                - Vi IMproved, a programmers text editor
view (1)              - Vi IMproved, a programmers text editor
vigr (8)              - edit the password, group, shadow-password or shad...
vim (1)               - Vi IMproved, a programmers text editor
vimdiff (1)           - edit two, three or four versions of a file with V...
vipw (8)              - edit the password, group, shadow-password or shad...
visudo (8)            - edit the sudoers file
```

Note the `man` section numbers after each command name. Also note that `apropos` is not sophisticated - it is simply searching for the exact string you give it in the very limited "brief descriptions" from the `man` pages. That's all. But a lot of time that's all you need to remember, "Ah, yes, `nano` is the other editor I was thinking about and like better than `vi`."

Note: `man`, `info` and `apropos` are just normal *IX commands like all the others, so while they may default to displaying with a paginator on an interactive terminal, you can run their output through other commands, just like any other. For example, maybe we remember only that the command had something with "edit" and was a system administration ("section 8") command:

```
$ apropos edit | grep "(8)"
jfs_debugfs (8)      - shell-type JFS file system editor
pdbedit (8)          - manage the SAM database (Database of Samba Users)
samba-regedit (8)    - ncurses based tool to manage the Samba registry
sudoedit (8)         - execute a command as another user
vigr (8)             - edit the password, group, shadow-password or shadow-group file
vipw (8)             - edit the password, group, shadow-password or shadow-group file
visudo (8)           - edit the sudoers file
```

Or maybe you can't remember whether it's `-r`, `-R` or `--recursive` to copy subdirectories recursively with `cp`:

```
$ man cp | grep -i "recurs"
              copy contents of special files when recursive
```

```
-R, -r, --recursive
       copy directories recursively
```

Whaddya know. It can be any of the three.

And yes, you can `man man`, `man info`, `info info` and `info man`, for that matter!

How Do You Google, `man`?

You can often search the internet for *IX documentation, and the `man` pages have long been online. A site I like (and link to a lot here) is http://linux.die.net/man/. Often, though, you can just google "man ls" and the top hits will be what you want.

However, there are times you need to be careful. Googling for either `man touch` or `man tail`, for example, will probably not give you the results you seek and may set off filters at work, so be careful out there and remember to bookmark a couple of actual `man` page sites so that you can go there directly and look up a command.

Books and Stuff

There are several consistently high-quality free sources of information on various parts of Linux and related systems on the internet.

- **The Linux Documentation Project (LDP)** - has fallen a bit behind over the years, but still has two of the best `bash` scripting books out there, *Bash Guide for Beginners* and *Advanced Bash-Scripting Guide*. I continue to use the latter all the time.
- **Arch Linux Wiki** - you may not think this would be useful if you are running Debian or Fedora or something else, but remember most *IX systems are all very similar, and often the best documentation on a package or setting something up in Linux is in the Arch wiki.
- **Debian documentation** - again, even if you are not running a Debian-based distro, this can be handy because it describes how to administer Linux in a way that often transcends distro specifics (and at least explains how Debian approaches the differences). The best books in the series are *The Debian Administrator's Handbook* and the *Debian Reference*, which is a lot more formal attempt at the same type of territory this guide covers.

Ubuntu, Mint and some other distros have quite active message fora, and of course StackOverflow and its family are also very useful.

Besides the above, if you are dealing with a package that is not part of the "core" OS, such as Samba for setting up CIFS shares on Linux, you should always look at the package site's documentation as well as any specific info you can find about the distro you are running.

Chapter 12

Step 10. And So On

/etc, starting and stopping services, apt-get/rpm/yum, and more.

> *"Et cetera, et cetera, et cetera!"* - The King (*The King and I*)

This step is a grab bag of stuff that didn't seem to directly belong anywhere before, but I still think needs to be known, or at least brushed up against.

One-Stop Shopping

In UNIX-like systems, most (not all) system configuration is stored in directories and text files under `/etc`.

Note: In Linux almost universally `/etc` is pronounced "slash-et-see," ***not*** "forward slash et cetera."

```
# ls -l /etc
total 844
drwxr-xr-x 3 root root    4096 Feb 25  2015 acpi
-rw-r--r-- 1 root root    2981 Apr 23  2014 adduser.conf
-rw-r--r-- 1 root root      45 Jul  9 08:46 adjtime
-rw-r--r-- 2 root root     621 May 22  2014 aliases
-rw-r--r-- 1 root root   12288 May 22  2014 aliases.db
drwxr-xr-x 2 root root   20480 Feb 25  2015 alternatives
-rw-r--r-- 1 root root    4185 Dec 28  2011 analog.cfg
drwxr-xr-x 7 root root    4096 Feb 25  2015 apache2
drwxr-xr-x 6 root root    4096 Feb 25  2015 apt
-rw-r----- 1 root daemon   144 Jun  9  2012 at.deny
-rw-r--r-- 1 root root    1895 Dec 29  2012 bash.bashrc
...and so on...
```

Depending on what you are trying to configure, you may be in one or many files in `/etc`. This is a ***very short*** list of files and directories you may need to examine there:

- **`fstab`** - a listing of the file systems currently mounted and their types.
- **`group`** - the security groups on the system.
- **`hosts`** - network aliases (overrides DNS, takes effect immediately).
- **`init.d`** - startup and shutdown scripts for "services."
- **`mtab`** - list of current "mounts."
- **`passwd`** - "shadow" file containing all the user accounts on the system.
- **`resolv.conf`** - DNS settings.
- **`samba`** - file sharing settings for CIFS-style shares.

There are lots of other interesting files under `/etc`, but I keep returning to the above again and again. On most of them you can run the `man` command against section 5 to see their format and documentation, e.g., `man 5 hosts`.

Service Station

We are going to ignore system initialization and "stages," and assume most of the time you are running on a well-functioning system. Even so sometimes you want to restart a specific system service without rebooting the whole system, often to force re-reading changed configuration files. If the service has a script in `/etc/init.d`:

```
# ls /etc/init.d
acpid                   console-setup  kbd                   mountkernfs.sh         nginx
apache2                 cron           keyboard-setup        mountnfs-bootclean.sh  openbsd
atd                     dbus           killprocs             mountnfs.sh            postfix
bootlogs                exim4          kmod                  mpt-statusd            postgre
bootmisc.sh             gitlab         motd                  mtab.sh                procps
checkfs.sh              halt           mountall-bootclean.sh networking             rc
checkroot-bootclean.sh  hostname.sh    mountall.sh           nfs-common             rc.loca
checkroot.sh            hwclock.sh     mountdevsubfs.sh      nfs-kernel-server      rcS
```

. . . then chances are it will respond to a fairly standard set of commands, such as the following samples with `samba`:

```
# /etc/init.d/samba stop
[ ok ] Stopping Samba daemons: nmbd smbd.

# /etc/init.d/samba start
[ ok ] Starting Samba daemons: nmbd smbd.

# /etc/init.d/samba restart
[ ok ] Stopping Samba daemons: nmbd smbd.
[ ok ] Starting Samba daemons: nmbd smbd.
```

Note: The above examples were run as `root`, otherwise they would probably have required execution using `sudo`.

Package Management

Almost all Linux distros have the concept of "packages" which are used to install, update and uninstall software. There are different package managers, including `dpkg` and `apt-get` on Debian-based distros, `rpm` on Fedora descendants, etc. For the rest of this section we will use Debian tools, but in general the concepts and problems are similar for the other toolsets.

One of the nicest things about Linux-style package managers (as opposed to traditional Windows installers) is that they can satisfy all a packages "dependencies" (other packages that are required for a package to run) and automatically detect and install those, too. See Chocolately for an attempt to build a similar ecosystem in Windows.

One thing Linux distros do is define the "repositories" (servers and file structures) that serve the various packages. In addition, there are usually multiple versions of packages, typically matching different releases of the distro. We won't go into setting up a system to point to these here.

In Debian flavors, `apt-get` is usually the tool of choice for package management.

There are three common `apt-get` commands that get used over and over. The first downloads and *updates* the local metadata cache for the repositories:

```
$ sudo apt-get update
[sudo] password for myuser:
Ign http://packages.linuxmint.com rafaela InRelease
Hit http://packages.linuxmint.com rafaela Release.gpg
Ign http://extra.linuxmint.com rafaela InRelease
Ign http://archive.ubuntu.com trusty InRelease
Hit http://security.ubuntu.com trusty-security InRelease
Hit http://packages.linuxmint.com rafaela Release
Hit http://extra.linuxmint.com rafaela Release.gpg
```

```
Hit http://archive.ubuntu.com trusty-updates InRelease
Hit http://extra.linuxmint.com rafaela Release
...and so on...
```

Note: `apt-get` is an administrative command and usually requires `sudo`.

The second common command *upgrades* all the packages in the system to the latest release in the repository (which may not be the latest and greatest release of the package):

```
$ sudo apt-get dist-upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

In this case there was nothing to upgrade. And the final common command is obviously to install a package:

```
$ sudo apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  curl
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 123 kB of archives.
After this operation, 314 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu/ trusty-updates/main curl amd64 7.35.0-1ubuntu2.5 [12
Fetched 123 kB in 0s (312 kB/s)
Selecting previously unselected package curl.
(Reading database ... 182823 files and directories currently installed.)
Preparing to unpack .../curl_7.35.0-1ubuntu2.5_amd64.deb ...
Unpacking curl (7.35.0-1ubuntu2.5) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up curl (7.35.0-1ubuntu2.5) ...
```

You can also `remove` packages.

This all looks very convenient, and it is. The problems arise because some distros are better at tracking current versions of packages in their repositories than others. In fact, some distros purposefully stay behind cutting edge for system stability purposes.

Other Sources

Besides the distribution's repositories, you can install packages and other software from a variety of places. It may be an "official" site for the package, GitHub, or whatever. The package may be in a binary installable format (`.deb` files for Debian systems), in source format requiring it to be built, in a zipped "tarball," and more.

If you want the latest and greatest version of a package you often have to go to its "official" site or GitHub repository. There, you may find a `.deb` file, in which case you could install it with `dpkg`:

```
sudo dpkg -i somesoftware.deb
```

There is, however, a problem. You now have to remember that you installed that package by hand and keep it up to date by hand (or not). `apt-get upgrade` isn't going to help you here. This is true no matter what way you get the alternative package - `.deb` file, tarball, source code, or whatever.

The final problem with package managers is that they're such a good idea that everybody has them now. Not just the operating systems like Linux, but languages like Python have `pip` and execution environments like node have `npm`. So now you end up with having to keep track of what you have installed on a system across two or three or more package managers at different levels of abstraction. It can be a mess!

Add into this that many of these language and environment package managers allow setting up "global" (system-wide) or "local" (current directory) versions of a package to allow different versions of the same package to exist on the same system, where different applications may be relying on the different versions to work.

Which `which` is Which?

Now that we've seen that we can have multiple versions of the same command or executable on the system, an interesting question arises. *Which* `foo` command am I going to call if I just type `foo` at the command prompt? In other words, after taking the `$PATH` variable into consideration and searching for the program through that from left to right, which version in which directory is going to be called?

Luckily we have the `which` command for just that!

```
$ which curl
/usr/bin/curl
```

How can you tell if you have multiple versions of something installed? One way is with the `locate` command:

```
locate md5
/boot/grub/i386-pc/gcry_md5.mod
/lib/modules/3.16.0-38-generic/kernel/drivers/usb/gadget/amd5536udc.ko
/usr/bin/md5pass
/usr/bin/md5sum
/usr/bin/md5sum.textutils
/usr/include/libavutil/md5.h
/usr/include/openssl/md5.h
/usr/lib/casper/casper-md5check
/usr/lib/grub/i386-pc/gcry_md5.mod
/usr/lib/i386-linux-gnu/sasl2/libcrammd5.so
...and so on...
```

The `locate` command, if installed, is basically a database of all of the file names on the system (collected periodically - not real time). You are simply searching the database for a pattern.

One final note on which thing gets executed. Unlike in Windows, UNIX environments do not consider the local directory (the current directory you are sitting at the command prompt, i.e., what `pwd` shows) as part of the path unless `.` is explicitly listed in `$PATH`. This is for security purposes. So it can be a bit unnerving to try and execute `foo` in the current directory and get:

```
$ ls -l foo
-rwxrwx--- 1 myuser mygroup 16 Oct 23 19:03 foo

$ foo
No command 'foo' found, did you mean:
 Command 'fgo' from package 'fgo' (universe)
 Command 'fop' from package 'fop' (main)
 Command 'fog' from package 'ruby-fog' (universe)
 Command 'fox' from package 'objcryst-fox' (universe)
 Command 'fio' from package 'fio' (universe)
 Command 'zoo' from package 'zoo' (universe)
 Command 'xoo' from package 'xoo' (universe)
 Command 'goo' from package 'goo' (universe)
foo: command not found
```

Instead, to invoke `foo`, you can either fully qualify the path as shown by `pwd`:

```
$ /home/myuser/foo
```

Or you can prepend the `./` relative path to it, to indicate "the `foo` in the current directory (`.`)":

```
$ ./foo
```

Over and Over and Over

The function of scheduled tasks in Windows is performed by `cron`. It reads in the various `crontab(5)` files on the system and executes the commands in them at the specified times. You use the `crontab(1)` command to view and edit the `crontab` files for you and other users (if you have admin privileges).

The sample given in the comments of the `crontab` when initially opened using `crontab -e` give a fine example of the syntax of the `crontab` file:

```
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
```

If you have `sudo` privileges you can edit the `crontab` file for another user with:

```
$ sudo crontab -e -u otheruser
```

This can be useful to do things like run backup jobs as the user that is running the web server, say, so it has access rights to all the necessary files to back up the web server installation by definition.

The only other thing I have to add about `cron` is when it runs the commands from each `crontab`, they are typically not invoked with that particular user's environment settings, so it is best to fully specify the paths to files both in the `crontab` file itself and in any scripts or parameters to scripts it calls. Depending on the system and whether `$PATH` is set at all when a "`cron` job" runs, you may have to specify the full paths to binaries in installed packages or even what you would consider "system" libraries! The `which` command comes in handy here.

Start Me Up

If you need to reboot the system the quickest way is with the `reboot` command:

```
$ sudo reboot
```

You can also use the `shutdown` command with the `-r` option, but why? The handy use for `shutdown` is to tell a system to halt and power off after shutting down:

```
$ sudo shutdown -h now
```

Turn on Your Signals

One of the basic concepts in UNIX program is that of "signals". You are probably already familiar with one way to send signals to a program, which is via `Ctrl-C` at the command prompt, which sends the `SIGINT` ("interrupt") signal to the program. Typically this will cause a program to terminate.

However, most signals can be "caught" by a program and coded around. There is one "uninterruptable" signal, however, which is `SIGKILL`. We can send `SIGKILL` to a process and cause it to terminate immediately with:

```
kill -s 9 14302
```

The `-s 9` is for signal #9, which is the `SIGKILL` signal (it is the tenth signal in the signal list, which is 0-relative, hence #9).

You can also use the following "shorthand" for `SIGKILL`:

```
kill -9 14302
```

Or if you want to get all verbose:

```
kill -s SIGKILL 14302
```

Note: `SIGKILL` should be used as a last resort, because a program is not allowed to catch it or be notified of it and hence can perform no closing logic or cleanup and may lead to data corruption. It is for getting rid of "hung" processes when nothing else will work. Always try to stop a program with a more "normal" method, which can include sending `SIGINT` to it first.

Exit, Smiling

Sometimes a command runs and there isn't a good way to tell if it worked or not. UNIX programs are supposed to set an "exit status" when they end that by convention is `0` if the program exited successfully and a non-zero, typically positive number if there was an error. The exit status for the last executed command or program can be shown at the command line using the `$?` environment variable. Consider if the file `foo` exists and `bar` does not:

```
$ ls foo
foo

$ echo $?
0

$ ls bar
ls: cannot access bar: No such file or directory

$ echo $?
2
```

Note: In many cases the exit codes come from the ANSI Standard C library's `errno.h` file. All of this is much handier when handling errors in scripts, but we're not going to go into script logic here.

However, sometimes even at the command line we want to be able to conditionally control a sequence of commands, and continue (or not continue) based on the success (or failure) of a previous command. In `bash` we have `&&` and `||` to the rescue!

- `a && b` - execute `a` ***and*** `b`, i.e., execute `b` only if `a` is successful.
- `a || b` - execute `a` ***or*** `b`, that is execute `b` whether ***or*** not `a` is successful.

Our example of file `foo` (which exists) and file `bar` (which does not) and the effect on the exit code of `ls` can be illustrative here, too:

```
$ ls foo && ls bar
foo
ls: cannot access bar: No such file or directory

$ echo $?
2
```

Both `ls` commands execute because the first successfully found `foo`, but the second emits its error and sets the exit code to `2` (failure).

```
$ ls foo || ls bar
foo

$ echo $?
0
```

Note in this case the second `ls` didn't execute because the logical "or" condition was already satisfied by the successful execution of the first `ls`. The exit code is obviously `0` (success).

```
$ ls bar && ls foo
ls: cannot access bar: No such file or directory

$ echo $?
2
```

Obviously if the first command fails, the "and" condition as a whole fails and the expression exits with a code of `2`.

```
$ ls bar || ls foo
ls: cannot access bar: No such file or directory
foo

$ echo $?
0
```

And finally, while the first command failed the second still can execute because of the "or", and the whole expression returns `0`.

Note: There is actually a `true` command whose purpose is to, "do nothing, successfully." All it does is return a `0` (success) exit code. This can be useful in scripting and also sometimes when building "and" and "or" clauses like above.

And yes, of course, that means there is also a `false` command to "do nothing, unsuccessfully!"

```
$ true

$ echo $?
0

$ false

$ echo $?
1
```

The End

Now you know what I know. Or at least what I keep loaded in my head vs. what I simply search for when I need to know it, and you know how to do that searching, too.

Good luck, citizen!

Chapter 13

Appendices

> *"That rug really tied the room together, did it not?"* - Walter Sobchak (*The Big Lebowski*)

Cheat Sheet

This list outlines all the commands, files and other UNIX items of interest brought up in this book. Use `man` or other methods outlined in the book to find more information on them.

Environment Variables

- **`$?`** - the exit code for the last command or program executed.
- **`$PATH`** - the execution search path.

Conditional Execution

See "logical operators."

- **`&&`** - execute the second command only if the first command succeeds.
- **`||`** - execute the secon command even if the first command fails.

Redirection

See "I/O Redirection."

- **stderr** - file descriptor 2, always open for writing from a process, defaults to the screen on a terminal session.
- **stdin** - file descriptor 0, always open for reading in a process, defaults to the keyboard input on a terminal session.
- **stdout** - file descriptor 1, always open for writing from a process, defaults to the screen on a terminal session.
- **<** - redirect a file to **stdin**.
- **>** - redirect **stdout** to a file.
- **2>** - redirect **stderr** to a file.
- **|** - pipe **stdout** from one process into **stdin** in another process.

Special Files and Directories

- **~** - shortcut for current user's home directory.
- **.bash_history** - history of commands entered at the command prompt (also a nice example of a hidden "dotfile").

System Directories

See Important System Directories.

- **/etc** - configuration files location.
- **/home** - "home" or user profile directories.
- **/proc** - system run-time information.
- **/root** - "home" directory for "root" user (system admin).
- **/tmp** - temporary files location.
- **/var/log** - log files location.

Commands

These are "section 1" commands, i.e., normal user commands that typically don't require any special privileges beyond permissions to access files and the like.

- [**7z**]*http://linux.die.net/man/1/7z) - compress and uncompress files and directories using the 7-zip algorithm.
- **apropos** - search for help on commands by pattern.
- **awk** - language for processing streams of data.
- **bash** - the Bourne-again shell.
- **bzip2** - compress and uncompress files using the **bzip2** algorithm.
- **cat** - concatenate the input files to **stdout**.
- **cd** - change the current directory.

- `chgrp` - change the primary group of a file or directory.
- `chmod` - change the permissions (mode bits) of a file or directory.
- `chown` - change the owner of a file or directory.
- `cp` - copy files or directories.
- `crontab` - display or edit tasks to be run by `cron`.
- `curl` - download files from the internet.
- `df` - show space utilization by file system.
- `diff` - show the differences between files.
- `dig` - look up DNS info on an address.
- `dpkg` - package manager for Debian flavors.
- `echo` - display passed parameters to `stdout`.
- `email` - send email.
- `false` - do nothing, unsuccessfully.
- `file` - give best guess as to type of file.
- `find` - find files based on various conditions and execute actions against the results.
- `grep` - search for a pattern (regular expression) in files.
- `help` - help for built-in commands in `bash`.
- `info` - an alternative for `man`, especially for GNU programs. Remember `q` quits.
- `less` - display the file one page at a time on `stdout`.
- `ln` - create hard or soft (shortcut) links.
- `ls` - list directory contents.
- `man` - display manual pages. Remember `q` quits.
- `mkdir` - make a new directory.
- `mutt` - email client.
- `mv` - move files or directories.
- `pine` - email client.
- `ps` - list running processes.
- `pwd` - print the current (working) directory name.
- `rm` - delete (remove) files or directories.
- `set` - set an environment variable, or display all environment variables.
- `sort` - sort `stdin` or a file to `stdout`.
- `ssh` - secure shell termina progam and protocol.
- `tail` - display the last lines of a file.
- `tar` - "tape archive", a way to combine directories into a single flat file.
- `tee` - write to a file and `stdout` at the same time.
- `telnet` - ancient terminal program and protocol.
- `touch` - create an empty file or change the last-modified time of an existing file.
- `true` - do nothing, successfully.
- `uname` - print system info.

- `unzip` - uncompress `.zip` files.
- `vi` - "visual" editor, a file editor.
- `wget` - download files from the internet.
- `whois` - look up DNS ownership info on an address.
- `zip` - compress files and directories using the PKZip algorithm.

System Commands

These are "section 8" commands, and ***may*** require special privileges such as `sudo` to run, depending on the system. Yes, some systems restrict the use of `ping`!

- `apt-get` - package manager for Debian flavors.
- `cron` - system for running "scheduled tasks."
- `dmesg` - display kernel log messages.
- `ifconfig` - display network (interface) configuration.
- `mount` - mount a file system to a specific location.
- `ping` - test for network connectivity to an IP address.
- `reboot` - restart the system.
- `rpm` - package manager for Fedora flavors.
- `shutdown` - shutdown or restart the system.
- `sudo` - execute a command with elevated privileges.
- `traceroute` - trace the route to an IP address.

Examples

Chapter 14

Colophon

> *"I can't come back, I don't know how it works! Good-bye, folks!"* - The Wizard of Oz

This document was produced in the environments it discusses, including (with their `uname -rv` results):

- **Cygwin** - `2.2.1(0.289/5/3) 2015-08-20 11:42`
- **Debian** - `3.2.0-4-amd64 #1 SMP Debian 3.2.65-1+deb7u2`
- **FreeBSD** - `7.3-RELEASE-p2 FreeBSD 7.3-RELEASE-p2 #0:  Tue Nov 4 22:08:52 EST 2014`
- **Linux Mint** - `3.16.0-38-generic #52~14.04.1-Ubuntu SMP Fri May 8 09:43:57 UTC 2015`

I could have done something with my Raspberry Pi, too, but that would just be showing off.

Written in CommonMark using `vi` and ReText, among others.

Output produced using `pandoc`, `pdflatex` and `make`, based on the [@evangoer's work](https://github.com/evangoer/pandoc-ebook-template).

Source code control is provided by `git`.

The cover photo is of our dog, Merv, who is reminding you, "Don't panic!"