Ten Steps to Linux Survival

Bash for Windows People

Jim Lehmer

2015

Steps

List of Figures

Merv sez, ”Don’t panic.”

By James Lehmer

v0.7

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 once called me an ”inveterate toolmaker” in a review, a badge I still wear with pride.
- Kim Manchak, who allowed me to be more than he hired me to be, and continues to be a great chess opponent.

Thank you, gentlemen. I’ve tried to pay it forward. This book is part of that.

[1] http://creativecommons.org/licenses/by-sa/4.0/

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 on what I consider the essentials for ~~floundering around~~ acting like I know what I'm doing in Linux, BSD and "UNIX-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 "UNIX" (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.

The most current release of the book should always be available for download in different formats on GitHub[1].

Batteries Not Included

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

- **System initialization** - besides, the whole "UNIX" world is in flux right now over system initialization architecture and the shift from "init"[2] scripts to `systemd`[3].
- **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 "UNIX" systems get the farthest apart in terms of interoperability, settings and customization.

[1] https://github.com/dullroar/ten-steps-to-linux-survival/releases
[2] https://en.wikipedia.org/wiki/Init
[3] https://en.wikipedia.org/wiki/Systemd

- **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 Windows.

Please, Give (Suggestions) 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 clone it from GitHub[4], make your changes and send me a `git pull` request. Or you can try to file it as an issue[5] and I'll see how I feel that day.

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 "UNIX" 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**

[4] https://github.com/dullroar/ten-steps-to-linux-survival
[5] https://github.com/dullroar/ten-steps-to-linux-survival/issues

All 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 `bash` 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 while 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 is 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)

As I started to type out my notes of what I considered to be ”essential,” they just kept growing and growing. Many nights, weekends and lunch hours later, this is the result. The slides were much easier to prepare now that I have the ”notes”!

Note: - The slides are included in the same GitHub repository as this book[6].

[6] https://github.com/dullroar/ten-steps-to-linux-survival/releases

Caveat Administrator

Even so, anything like this is incomplete. Anyone truly knowledgable of Linux will splutter their coffee into their neckbeard[7] 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"[8] for getting up and running ***quickly*** with the sheer basics, plus knowing where to go for help. I modeled it explicitly after "short and opinionated" tech books such as Douglas Crockford's *Javascript: The Good Parts*[9] and especially those licensed under Creative Commons[10], such as the books from Green Tea Press[11]. If you like those big tech books that are priced by the kilogram, this is not the book for you.

It is also 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."

WARNING: *Many of the commands in this book can alter your system and possibly damage it.*

Obvious candidates include the file system commands like `rm` , the `vi` editor (obviously), and some of the "system admin" commands mentioned later, including system and service restarts. Use your common sense plus the various resources for documentation mentioned in this book to make sure you aren't doing anything destructive to your system, especially in production.

YOU HAVE BEEN WARNED!

Conventions

If a command, file name or other "computer code" is shown in-line in a sentence, it will appear in a fixed-width font, e.g., `ls --recursive *.txt`.

If a command and its output, script code or something else is shown in a block, it will appear like this:

[7] Stereotype intentional.
[8] https://en.wikipedia.org/wiki/Fake_book
[9] http://shop.oreilly.com/product/9780596517748.do
[10] http://creativecommons.org/licenses/by-sa/4.0/
[11] http://greenteapress.com/

Figure -1.1: Sample command

```
~ $ 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 ?        00:00:06   rcu_sched
    8 ?        00:00:02   rcuos/0
    9 ?        00:00:01   rcuos/1
   10 ?        00:00:03   rcuos/2
   11 ?        00:00:01   rcuos/3
   12 ?        00:00:00   rcuos/4
   13 ?        00:00:00   rcuos/5
   14 ?        00:00:00   rcuos/6
   15 ?        00:00:00   rcuos/7
   16 ?        00:00:00   rcu_bh
   17 ?        00:00:00   rcuob/0
   18 ?        00:00:00   rcuob/1
   19 ?        00:00:00   rcuob/2
   20 ?        00:00:00   rcuob/3
   21 ?        00:00:00   rcuob/4
   22 ?        00:00:00   rcuob/5
   23 ?        00:00:00   rcuob/6
   24 ?        00:00:00   rcuob/7
...and so on...
```

All such blocks have been normalized to only show a maximum of 80x24 characters. This is intentional. While most modern ”UNIX” systems and terminal windows like `ssh` can handle any geometry, there are still systems and situations where you get the same terminal size that your grandfather would’ve used. It is best to learn how to deal with these by using `less`, redirection and the like.

The examples in this book typically show something like `~ $` before the command, or `~ #` (when logged in as root) or `%` (when running under `csh`). These ”command prompts” are set in `bash` via the `PS1` environment variable[12] and are not meant to be typed in as part of the command.

In the few places where a ”UNIX” command is shown in comparison to a ”DOS” command run under `CMD.EXE`, the latter is shown in all uppercase to help distinguish it from

[12] https://www.linux.com/learn/docs/ldp/443-bash-prompt-howto

the ”UNIX” equivalent, even though `CMD.EXE` is case-insensitive. In other words, `set` will be shown for `bash` and `SET` for `CMD.EXE`.

Acknowledgments

Thanks to Ken Astl for reading an early draft of this book. Thanks to my wife Leslie for putting up with me while I obsessed over it.

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[1] at MIT, GE and Bell Labs.
- **In the beginning** - early 1970s, Nixon drags on, Watergate, Bell Labs, Thompson[2] & Ritchie[3], UNIX[4] is born.
- **More trouble from Berkeley** - late 1970s, Carter, disco, Iran hostages, UC Berkely releases the Berkeley Software Distribution[5] (BSD), a port based on the Bell Labs UNIX. Let the forking begin!

[1] https://en.wikipedia.org/wiki/Multics
[2] https://en.wikipedia.org/wiki/Ken_Thompson
[3] https://en.wikipedia.org/wiki/Dennis_Ritchie
[4] https://en.wikipedia.org/wiki/History_of_Unix
[5] https://en.wikipedia.org/wiki/Berkeley_Software_Distribution

- **UNIX goes commercial** - 1980s, Reagan, Iran Contra, *E.T.*, AT&T releases System V[6] as first commercial UNIX. From the same background as Bell Labs UNIX, System V 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 "UNIX"** -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[7] releases a project called Linux[8] based on MINIX[9] (and hence why Linus says Linux is pronounced like "MINIX" and not like "Linus").

- **Proliferation of the BSDs** - mid-to-late 1990s, still Clinton I, Monicagate, Kosovo, various ports of BSD including NetBSD[10], FreeBSD[11] and OpenBSD[12]. 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 II, lots of cross-porting of everything open source. However, licenses matter[13], and there sure are a lot of them[14]. While things have settled down some with the dismissal of the SCO lawsuit, intellectual property remains a problem area in open source, even as the use of open source software (OSS) has exploded.

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

[6] https://en.wikipedia.org/wiki/UNIX_System_V
[7] https://en.wikipedia.org/wiki/Linus_Torvalds
[8] https://en.wikipedia.org/wiki/Linux
[9] https://en.wikipedia.org/wiki/MINIX
[10] https://en.wikipedia.org/wiki/NetBSD
[11] https://en.wikipedia.org/wiki/FreeBSD
[12] https://en.wikipedia.org/wiki/OpenBSD
[13] https://en.wikipedia.org/wiki/Open-source_license
[14] https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses

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

Hence, for the rest of this text I will tend to talk somewhat interchangeably about "Linux" and "UNIX" and the like. When it matters, I will mention which OS I am discussing by name, but often I will use "UNIX" (in quotes) to mean anything in the "family tree" of the original Bell Labs offspring, or that "acts like," well, UNIX.

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

- **POSIX**[15] - 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. All POSIX-compliant system are "similar but different." Even Windows can claim to be POSIX-compliant in some respects (and has an installable POSIX subsystem), but that doesn't mean POSIX-compliant code will run there unchanged.

- **GNU Project**[16] - Richard Stallman[17] founded the Free Software Foundation[18] (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[19], 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.

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

[15] https://en.wikipedia.org/wiki/POSIX
[16] https://en.wikipedia.org/wiki/GNU_Project
[17] https://en.wikipedia.org/wiki/Richard_Stallman
[18] https://en.wikipedia.org/wiki/Free_Software_Foundation
[19] https://www.gnu.org/software/software.html

the source of the package it may or may not work correctly on another ”UNIX” 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, long file name and network-aware `CMD.EXE` could handle:

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

In UNIX-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 difference in output between showing all processes with the `ps`[20] (*process*) command on a Linux system, in this case Linux Mint under `bash`:

Figure 0.1: `ps` on Linux in `bash`

```
~ $ ps -a
  PID TTY          TIME CMD
 4508 pts/3    00:00:00 su
 4516 pts/3    00:00:00 bash
 4594 pts/3    00:00:00 ps
```

[20] http://linux.die.net/man/1/ps

Versus the ”same” command on a FreeBSD system at my ISP, where `csh` is the default shell:

Figure 0.2: `ps` on FreeBSD in `csh`

```
%ps -a
  PID  TT  STAT      TIME COMMAND
 5073  p0  Ss     0:00.02 -csh (csh)
 5115  p0  RN+    0:00.00 ps -a
```

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 to best 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”[21] based in large part around the package management tool[22] predominantly used:

- **`apt-get`, `dpkg` and `.deb` files** - Debian[23] flavors, such as Ubuntu[24] and Mint[25]. Mint is currently my desktop Linux of choice and Debian my preferred server OS, both based on familiarity.
- **`pacman`** - Arch[26] flavors.
- **`rpm` and `yum`** - Red Hat flavors,such as Fedora[27], Red Hat Enterprise[28] and CentOS[29].

[21] https://en.wikipedia.org/wiki/Linux_distribution#Popular_distributions
[22] https://en.wikipedia.org/wiki/Package_manager
[23] https://en.wikipedia.org/wiki/Debian
[24] https://en.wikipedia.org/wiki/Ubuntu_%28operating_system%29
[25] https://en.wikipedia.org/wiki/Linux_Mint
[26] https://en.wikipedia.org/wiki/Arch_Linux
[27] https://en.wikipedia.org/wiki/Fedora_%28operating_system%29
[28] https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux
[29] https://en.wikipedia.org/wiki/CentOS

- **Source code** - Gentoo[30] tends to be a ”compile from scratch” environment, much like FreeBSD[31].
- **”Tar balls”** - source code or binaries delivered via archived and zipped directories. Common on Slackware[32], some others.

Get Embed With Me

A lot of firmware in embedded devices is based on some sort of ”UNIX” 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. BusyBox[33] is a good example of a ”UNIX-like” shell (command prompt) used by many embedded systems. Of course, as to what’s actually available, who knows? If you can get shell open, the best thing to do is see what works.

Cygwin

Cygwin[34] 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?

[30] https://en.wikipedia.org/wiki/Gentoo_Linux
[31] https://en.wikipedia.org/wiki/FreeBSD_Ports
[32] https://en.wikipedia.org/wiki/Slackware
[33] https://en.wikipedia.org/wiki/BusyBox
[34] http://cygwin.com/

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`**[1] - yes, Windows has a shell.
- **`PowerShell.exe`**[2] - in fact, it has at least two!

In UNIX-land:

- **`sh`**[3] - the "original" Bourne shell in UNIX, which spawned:
  - **`ash`**[4] - Almquist shell.

[1] https://technet.microsoft.com/en-us/library/cc754340.aspx
[2] https://technet.microsoft.com/en-us/library/ms714469%28v=VS.85%29.aspx
[3] https://en.wikipedia.org/wiki/Bourne_shell
[4] https://en.wikipedia.org/wiki/Almquist_shell

    * **dash** - Debian Almquist shell (replaced ash in Debian)
  - **bash**[5] - Bourne-again shell (get it?), the ”standard” Linux shell (as much as anything is standard across Linux distros).
  - **ksh**[6] - Korn shell.
  - **zsh**[7] - Z shell.

- **csh**[8] - C shell, historically it is the default shell on BSD systems (although there are arguments on why you should ***never use it***[9]).
- **...and many more!** - tons, really[10].

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[11] command in a bash terminal:

Figure 1.1: Built-in commands in bash

```
~ $ help
GNU bash, version 4.3.11(1)-release (x86_64-pc-linux-gnu)
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.
```

[5] https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29
[6] https://en.wikipedia.org/wiki/Korn_shell
[7] https://en.wikipedia.org/wiki/Z_shell
[8] https://en.wikipedia.org/wiki/C_shell
[9] http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/
[10] https://en.wikipedia.org/wiki/Unix_shell#Shell_categories
[11] http://linux.die.net/man/1/help

```
 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]
 case WORD in [PATTERN [| PATTERN]...)>  read [-ers] [-a array] [-d delim] [->
 cd [-L|[-P [-e]] [-@]] [dir]            readarray [-n count] [-O origin] [-s>
...and so on...
```

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"[12] line that looks like this:

Figure 1.2: `bash` "shebang"

```
#!/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. For example, on the FreeBSD system I have access to, `dash` is not installed. So consider the following `hello.sh` script:

Figure 1.3: Script with 'dash' "shebang"

```
#!/bin/dash
echo Hello, World!
```

[12] https://en.wikipedia.org/wiki/Shebang_%28Unix%29

When I try to run it on FreeBSD, I get:

Figure 1.4: ”Shebang” error

```
% ./hello.sh
./hello.sh: Command not found.
```

This is confusing, because it seems to be saying that `hello.sh` is not found! But in reality it is complaining about missing `dash`. If I change the script to point to `bash` (which is installed on that FreeBSD system), it works as expected:

Figure 1.5: Hello, World!

```
% ./hello.sh
Hello, World!
```

Note that on some systems `#!/bin/sh` points 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 search for 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 to that devil’s 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 between `CMD.EXE` and a Linux shell like `bash`. In both `bash` and `CMD.EXE` the `set`[13] command shows you all environment variables that have been set. Here’s `bash`:

Figure 1.6: `set` command in `bash`

[13] http://linux.die.net/man/1/set

```
~ $ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote
:force_fignore:histappend:interactive_comments:login_shell:progcomp:promptvars:s
ourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="3" [2]="11" [3]="1" [4]="release" [5]="x86_64-pc-lin
ux-gnu")
BASH_VERSION='4.3.11(1)-release'
COLORTERM=gnome-terminal
COLUMNS=80
DIRSTACK=()
DISPLAY=:0
EUID=1003
GROUPS=()
HISTCONTROL=ignoreboth
HISTFILE=/home/myuser/.bash_history
...and so on...
```

And `CMD.EXE`:

Figure 1.7: `SET` command in `CMD.EXE`

```
C:\Users\myuser>SET
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\myuser\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=JLEHMER650
ComSpec=C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK=NO
HOMEDRIVE=C:
HOMEPATH=\Users\myuser
```

```
LOCALAPPDATA=C:\Users\myuser\AppData\Local
LOGONSERVER=\\JLEHMER650
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\system32
\config\systemprofile\.dnx\bin;C:\Program Files\Microsoft DNX\Dnvm\;C:\Program F
iles (x86)\nodejs\;C:\Program Files\Microsoft\Web Platform Installer\;C:\Program
 Files\Microsoft SQL Server\130\Tools\Binn\;C:\Program Files (x86)\Microsoft SQL
 Server\130\DTS\Binn\;C:\Program Files\Microsoft SQL Server\120\Tools\Binn\;C:\P
rogram Files (x86)\Microsoft SDKs\Azure\CLI\wbin;C:\Windows\System32\WindowsPowe
rShell\v1.0\
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
...and so on...
```

Similarly, the `echo`[14] command can be used to show you the contents of an environment variable like `HOME` on `bash`:

Figure 1.8: `echo` the `HOME` environment variable in `bash`

```
~ $ echo $HOME
/home/myuser
```

Versus the `HOMEPATH` variable under `CMD.EXE`:

Figure 1.9: `echo` the `HOMEPATH` environment variable in `CMD.EXE`

```
C:\> ECHO %HOMEPATH%
\Users\myuser
```

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

1. The syntax for accessing an environment variable is `$variable` in `bash` and `%variable%` in `CMD.EXE`.

[14] http://linux.die.net/man/1/echo

2. `bash` is case-sensitive and so `echo $HOME` works but `echo $home` does not. `CMD.EXE` is ***not*** case-sensitive, so either `echo %homedrive%` or `echo %HOMEDRIVE%` (or `EcHo %hOmEdRiVe%`) would work.

One final note of caution. You can set up command aliases in `bash` and other shells that allow you to define a `CMD.EXE`-style `dir` command as a substitute for the `ls` command in `bash`, or `copy` for `cp`, `del` for `rm`, and so on. I recommend you don't do this for at least two reasons:

1. It is difficult to get these right in terms of being able to map all the various parameters from the `bash` command to the appropriate parameters for a `CMD.EXE`-style command. Most people don't go that far, which means you then end up with a "toy" substitute for the `CMD.EXE` command, and have to fall back to the native commands anyway.
2. It simply delays you actually learning about the "UNIX" environment. You end up relying on a crutch that then must be replicated on every system you touch. In my opinion it is better to just learn the native commands, because then you are instantly productive at any shell window.

You're a Product of Your Environment (Variables)

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

Figure 1.10: Assign `FOO` environment variable before executing script

```
~ $ FOO=myval /home/myuser/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 "UNIX" is case-sensitive and Windows is not.

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

Figure 1.11: Set multiple environment variables at once

```
~ $ FOO=myval BAR=yourval BAZ=ourvals /home/myuser/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 another command by surrounding it with paired ‘ (”back ticks”, or ”grave accents”):

Figure 1.12: Set environment variable to output from a command

```
~ $ FILETYPE=`file --brief --mime-type header.tex`
~ $ echo $FILETYPE
text/plain
```

Who Am I?

When writing scripts that can be run by any user, it may be helpful to know their user name at run-time. There are at least two different ways to determine that. The first is via the `USER` environment variable:

Figure 1.13: `USER` environment variable

```
~ $ echo $USER
myuser
```

The second is with a command with one of the best names, ever - `whoami`[15]:

Figure 1.14: `whoami` command

[15] http://linux.die.net/man/1/whoami

```
~ $ whoami
myuser
```

Some environments set the `USER` environment variable, some set a `USERNAME` variable, and some like Mint set both. I think it is better to use `whoami`, which tends to be on almost all systems.

Paths (a Part of Any Balanced Shrubbery)

The concept of a ”path” for finding executables is almost identical between ”UNIX” and Windows, and Windows lifted it from UNIX (or CP/M, which lifted it from UNIX). Look at the output of the `PATH` environment variable under `bash`:

Figure 1.15: `PATH` environment variable in `bash`

```
~ $ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Echoing the `PATH` environment variable under `CMD.EXE` works, too:

Figure 1.16: `PATH` environment variable in `CMD.EXE`

```
C:\Users\myuser>ECHO %PATH%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\system32\conf
ig\systemprofile\.dnx\bin;C:\Program Files\Microsoft DNX\Dnvm\;C:\Program Files
(x86)\nodejs\;C:\Program Files\Microsoft\Web Platform Installer\;C:\Program File
s\Microsoft SQL Server\130\Tools\Binn\;C:\Program Files (x86)\Microsoft SQL Serv
er\130\DTS\Binn\;C:\Program Files\Microsoft SQL Server\120\Tools\Binn\;C:\Progra
m Files (x86)\Microsoft SDKs\Azure\CLI\wbin;C:\Windows\System32\WindowsPowerShel
l\v1.0\
```

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:` in Windows, and instead everything is mounted in a single namespace hierarchy starting at the root `/`. We’ll be talking more about directories, paths and file systems in the next chapter.

Just to muddy the waters further, notice how Cygwin under Windows shows the `PATH` environment variable with `bash` syntax but a combination of both Cygwin and Windows directories, and Windows drive letters like `C:` mapped to `/cygdrive/c`:

Figure 1.17: `PATH` environment variable in Cygwin

```
$ echo $PATH
/usr/local/bin:/usr/bin:/cygdrive/c/Windows/system32:/cygdrive/c/Windows:/cygdri
ve/c/Windows/System32/Wbem:/cygdrive/c/Windows/system32/config/systemprofile/.dn
x/bin:/cygdrive/c/Program Files/Microsoft DNX/Dnvm:/cygdrive/c/Program Files (x8
6)/nodejs:/cygdrive/c/Program Files/Microsoft/Web Platform Installer:/cygdrive/c
/Program Files/Microsoft SQL Server/130/Tools/Binn:/cygdrive/c/Program Files (x8
6)/Microsoft SQL Server/130/DTS/Binn:/cygdrive/c/Program Files/Microsoft SQL Ser
ver/120/Tools/Binn:/cygdrive/c/Program Files (x86)/Microsoft SDKs/Azure/CLI/wbin
:/cygdrive/c/Windows/System32/WindowsPowerShell/v1.0
```

Open Your Shell and Interact

The actual ”command prompt” is when you run 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 wizards at using the shell can often show off their magic with an incredible one-liner typed rom 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 some files in a directory:

Figure 1.18: List some files

```
~/Documents $ ls
Disabled User Accounts.csv  elsewhere  LOLcatz.jpg  MyResume.md
```

Without tab expansion, typing out something like this is painful:

Figure 1.19: Lots of typing and escape characters

```
~/Documents $ mv Disabled\ User\ Accounts.csv elsewhere/.
```

But with tab expansion, we can simply type `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 for us:

Figure 1.20: Tab expansion magic

```
~/Documents $ mv Disabled\ User\ Accounts.csv
```

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

One place 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 part of the file name) 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`, which you can display with `ls -a`:

Figure 1.21: `ls` command showing hidden files

```
~ $ ls -a
.              .config    .gconf           .mozilla  Templates
..             .dbus      .gnome2          Music     Videos
.bash_history  Desktop    .gnome2_private  Pictures  .xsession-errors
.bash_logout   .dmrc      .ICEauthority    .profile
.cache         Documents  .linuxmint       Public
.cinnamon      Downloads  .local           .ssh
```

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 need 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`.

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:

- **btrfs**[1]
- **ext2**[2]
- **ext3**[3]
- **ext4**[4],
- **ReiserFS**[5]
- **ZFS**[6]
- **...and so many more!** - NTFS, FAT, CDFS, etc.

[1] https://en.wikipedia.org/wiki/Btrfs
[2] https://en.wikipedia.org/wiki/Ext2
[3] https://en.wikipedia.org/wiki/Ext3
[4] https://en.wikipedia.org/wiki/Ext4
[5] https://en.wikipedia.org/wiki/ReiserFS
[6] https://en.wikipedia.org/wiki/ZFS

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 before, the biggest differences between Linux and Windows is that the Linux environments do not have a concept of ”drive letters.” Instead everything is ”mounted” under a single hierarchy that starts at the ”root directory” or `/`:

Figure 2.1: Listing of the root directory

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

The root file system may be backed by a disk device, 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

As we’ve already seen, the command to *list* the contents of a directory is `ls`[7] :

Figure 2.2: Listing directory contents

```
~ $ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

Remember, ”UNIX” environments think of files that start with a `.` as ”hidden.” If you want to see all these ”dotfiles”[8] , you can use `ls -a`, in this case on an average ”home” directory:

Figure 2.3: Listing a home directory showing hidden ”dotfiles”

[7] http://linux.die.net/man/1/ls
[8] https://en.wikipedia.org/wiki/Hidden_file_and_hidden_directory#Unix_and_Unix-like_environments

```
~ $ ls -a
.              .config    .gconf           .mozilla  Templates
..             .dbus      .gnome2          Music     Videos
.bash_history  Desktop    .gnome2_private  Pictures  .xsession-errors
.bash_logout   .dmrc      .ICEauthority    .profile
.cache         Documents  .linuxmint       Public
.cinnamon      Downloads  .local           .ssh
```

Wow! That’s a lot of dotfiles!

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

Figure 2.4: Detailed listing of home directory

```
~ $ ls -l
total 32
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Desktop
drwxr-xr-x 3 myuser mygroup 4096 Dec 13 18:22 Documents
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Downloads
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Music
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Pictures
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Public
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Templates
drwxr-xr-x 2 myuser mygroup 4096 Dec 13 18:18 Videos
```

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

Figure 2.5: Detailed listing of home directory with ”dotfiles”

```
~ $ ls -al
total 112
drwxr-xr-x 21 myuser mygroup 4096 Dec 13 18:19 .
drwxr-xr-x  6 root   root    4096 Dec 13 14:24 ..
-rw-------  1 myuser mygroup  287 Dec 13 18:19 .bash_history
-rw-r--r--  1 myuser mygroup  220 Dec 13 14:24 .bash_logout
drwx------  5 myuser mygroup 4096 Dec 13 18:18 .cache
drwxr-xr-x  3 myuser mygroup 4096 Dec 13 18:18 .cinnamon
drwxr-xr-x 12 myuser mygroup 4096 Dec 13 18:18 .config
drwx------  3 myuser mygroup 4096 Dec 13 18:18 .dbus
```

```
drwxr-xr-x  2 myuser mygroup 4096 Dec 13 18:18 Desktop
-rw-------  1 myuser mygroup   29 Dec 13 18:18 .dmrc
drwxr-xr-x  3 myuser mygroup 4096 Dec 13 18:22 Documents
drwxr-xr-x  2 myuser mygroup 4096 Dec 13 18:18 Downloads
drwx------  3 myuser mygroup 4096 Dec 13 18:18 .gconf
drwx------  3 myuser mygroup 4096 Dec 13 18:18 .gnome2
drwx------  2 myuser mygroup 4096 Dec 13 18:18 .gnome2_private
-rw-------  1 myuser mygroup  668 Dec 13 18:18 .ICEauthority
drwxr-xr-x  3 myuser mygroup 4096 Dec 13 18:18 .linuxmint
drwxr-xr-x  3 myuser mygroup 4096 Dec 13 18:18 .local
drwxr-xr-x  4 myuser mygroup 4096 Dec 13 18:18 .mozilla
drwxr-xr-x  2 myuser mygroup 4096 Dec 13 18:18 Music
drwxr-xr-x  2 myuser mygroup 4096 Dec 13 18:18 Pictures
...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 ”UNIX” world it means you’re toast if you run it 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 or ”switch” character:

Figure 2.6: Short parameter

```
~ $ ls -r
```

Some commands support parameters with no ”switch” character at all, as with `xvf` (e***X***tract, ***V***erbose, input ***F***ile name) in the following `tar` example:

Figure 2.7: Alternate short parameter syntax

```
~ $ tar xvf backup.tar
```

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

Figure 2.8: Long parameters

```
~ $ 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`[9] (*concatenate*) command:

Figure 2.9: `cat` 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 lib
xext-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
```

[9] http://linux.die.net/man/1/cat

```
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
```

In this example when we `cat installrdp` we can determine it is a `bash` shell script (because the "shebang" is pointing to `bash`) that looks to install and configure FreeRDP[10] on a Debian-style system:

1. **`apt-get`** - Debian-style package manager.
2. **`git clone`** - cloning package from GitHub[11].
3. **`cmake`** and **`make`** - configuring and building software from source.

A better way to display a longer file is to use the `less`[12] command (which is a derivative of the original `more`[13] , 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`[14] command. The next example shows the last 10 lines of the kernel `dmesg` log:

Figure 2.10: `tail` command

[10] https://github.com/FreeRDP/FreeRDP
[11] http://github.com
[12] http://linux.die.net/man/1/less
[13] http://linux.die.net/man/1/more
[14] http://linux.die.net/man/1/tail

```
/var/log $ tail dmesg
[    3.913318] Bluetooth: BNEP socket layer initialized
[    3.914888] Bluetooth: RFCOMM TTY layer initialized
[    3.914895] Bluetooth: RFCOMM socket layer initialized
[    3.914900] Bluetooth: RFCOMM ver 1.11
[    3.935772] init: failsafe main process (732) killed by TERM signal
[    4.046700] init: cups main process (896) killed by HUP signal
[    4.046710] init: cups main process ended, respawning
[    4.186239] init: samba-ad-dc main process (919) terminated with status 1
[    4.328999] r8169 0000:02:00.0 eth0: link down
[    4.329037] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
```

To show a specific number of lines use the `-n` parameter with `tail`:

Figure 2.11: Display last 15 lines of a file with `tail -n`

```
/var/log $ tail -n 15 dmesg
[    3.899169] Bluetooth: HCI socket layer initialized
[    3.899170] Bluetooth: L2CAP socket layer initialized
[    3.899179] Bluetooth: SCO socket layer initialized
[    3.913306] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[    3.913309] Bluetooth: BNEP filters: protocol multicast
[    3.913318] Bluetooth: BNEP socket layer initialized
[    3.914888] Bluetooth: RFCOMM TTY layer initialized
[    3.914895] Bluetooth: RFCOMM socket layer initialized
[    3.914900] Bluetooth: RFCOMM ver 1.11
[    3.935772] init: failsafe main process (732) killed by TERM signal
[    4.046700] init: cups main process (896) killed by HUP signal
[    4.046710] init: cups main process ended, respawning
[    4.186239] init: samba-ad-dc main process (919) terminated with status 1
[    4.328999] r8169 0000:02:00.0 eth0: link down
[    4.329037] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
```

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

Figure 2.12: `tail -f` command

```
/var/log $ tail -f syslog
Dec 13 19:23:40 MtLindsey dhclient: DHCPACK of 192.168.0.8 from 192.168.0.1
Dec 13 19:23:40 MtLindsey dhclient: bound to 192.168.0.8 -- renewal in 1423 seco
nds.
Dec 13 19:23:40 MtLindsey NetworkManager[960]: <info> (eth0): DHCPv4 state chang
ed renew -> renew
Dec 13 19:23:40 MtLindsey NetworkManager[960]: <info>   address 192.168.0.8
Dec 13 19:23:40 MtLindsey NetworkManager[960]: <info>   prefix 24 (255.255.255.0
)
Dec 13 19:23:40 MtLindsey NetworkManager[960]: <info>   gateway 192.168.0.1
Dec 13 19:23:40 MtLindsey NetworkManager[960]: <info>   nameserver '97.64.168.12
'
Dec 13 19:23:40 MtLindsey NetworkManager[960]: <info>   nameserver '192.119.194.
131'
Dec 13 19:23:40 MtLindsey dbus[689]: [system] Activating service name='org.freed
esktop.nm_dispatcher' (using servicehelper)
Dec 13 19:23:40 MtLindsey dbus[689]: [system] Successfully activated service 'or
g.freedesktop.nm_dispatcher'
```

Use `Ctrl-C` to cancel following the file.

If we know nothing about a *file*, we can use the `file`[15] command to help us guess:

Figure 2.13: `file` command

```
~ $ 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[16] 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.

[15] http://linux.die.net/man/1/file
[16] http://linux.die.net/man/5/magic

Sorting Things Out

Let's say we have three files, and want to display the contents of one of them. We know we can do that with `cat`:

Figure 2.14: Show contents of one file

```
~ $ cd Invoices/
~/Invoices $ ls
ElevatorTrucks  FarmCombines  FarmTractors
~/Invoices $ cat ElevatorTrucks
Truck   brakes  200
Truck   tires   400
Truck   tires   400
Truck   tires   400
Truck   winch   100
```

But what if we wanted to process all the lines in all the files in alphabetical order? Just directing 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.

Figure 2.15: Show contents of all three files

```
~/Invoices $ 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`[17] command to the rescue! We will see that the `sort` command can be used to not just *sort* files, but also to merge them and remove duplicates.

[17] http://linux.die.net/man/1/sort

Figure 2.16: `sort` command

```
~/Invoices $ 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:

Figure 2.17: Sort by the second ”key” column

```
~/Invoices $ 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?

Figure 2.18: Sort by the third column

```
~/Invoices $ 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:

Figure 2.19: Sort by third column, numerically

```
~/Invoices $ 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:

Figure 2.20: Top three most expensive items

```
~ $ sort -k 3 -n * | tail -n 3
Combine motor   1500
```

```
Tractor tires   2000
Combine tires   2500
```

Finally, what if we want only unique rows?

Figure 2.21: Sort and show only unique rows

```
~/Invoices $ 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:

Figure 2.22: Sort unique rows using long parameter names

```
~/Invoices $ 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
```

If you read that command in a script file, there would be little confusion as to what it was doing.

Rearranging Deck Chairs

We can copy, move (or rename - same thing) and delete files and directories. To *copy*, simply use the `cp`[18] command:

Figure 2.23: `cp` command

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

You can copy entire directories recursively:

Figure 2.24: Copying directories recursively

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

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

Figure 2.25: `cp` command with long parameter names

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

To *move* use `mv`[19] :

Figure 2.26: `mv` command

```
~ $ 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`[20] command can take care of beyond `mv`, like renaming all file extensions from `.htm` to `.html`.

[18] http://linux.die.net/man/1/cp
[19] http://linux.die.net/man/1/mv
[20] http://linux.die.net/man/1/rename

Making Files Disappear

To delete or *remove* a file you use `rm`[21] :

Figure 2.27: `rm` command

```
~ $ 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!***

The following scenario can happen ***way*** too often, even to experienced system administrators. Note the accidental space between `*` and `.bak` on the `rm` command:

Figure 2.28: Oops!

```
~ $ cd MyDissertation/
~/MyDissertation $ ls
Bibliography.bak  Bibliography.doc  Dissertation.bak  Dissertation.doc
~/MyDissertation $ rm * .bak
rm: cannot remove ‘.bak’: No such file or directory
~/MyDissertation $ ls
~/MyDissertation $
```

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! The user’s priorities just got rearranged as they go hunting for another copy of their dissertation.

[21] http://linux.die.net/man/1/rm

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

Figure 2.29: First make sure we are dealing with the right files

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

Then I use the "up arrow" to bring back the `ls` command and change `ls` to `rm` before running 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`[22] :

Figure 2.30: `touch` command

```
~ $ 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:

Figure 2.31: A second `touch`

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

[22] http://linux.die.net/man/1/touch

It can be useful (but also distressing from a forensics point of view) to 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:

Figure 2.32: Set file modified date to a specific date and time

```
~ $ 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`[23] :

Figure 2.33: `mkdir` command

```
~ $ cd Foo
~/Foo $ ls -l
total 4
-rw-r--r-- 1 myuser mygroup    0 Dec 14 05:49 a
-rw-r--r-- 1 myuser mygroup    0 Dec 14 05:49 b
-rw-r--r-- 1 myuser mygroup    0 Dec 14 05:49 c
drwxr-xr-x 2 myuser mygroup 4096 Dec 14 05:49 d
~/Foo $ mkdir Bar
~/Foo $ ls -l
total 8
-rw-r--r-- 1 myuser mygroup    0 Dec 14 05:49 a
-rw-r--r-- 1 myuser mygroup    0 Dec 14 05:49 b
drwxr-xr-x 2 myuser mygroup 4096 Dec 14 14:49 Bar
-rw-r--r-- 1 myuser mygroup    0 Dec 14 05:49 c
drwxr-xr-x 2 myuser mygroup 4096 Dec 14 05:49 d
```

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

Figure 2.34: `mkdir` error

[23] http://linux.die.net/man/1/mkdir

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

But of course you can override that behavior:

Figure 2.35: Make multiple intervening directories at once

```
~/Foo $ mkdir --parents Xyzzy/Something
~/Foo $ ls
a  b  Bar  c  d  Xyzzy
~/Foo $ 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 letters and the direction of the slashes.

To *change directories*, simply use `cd`[24] much like in Windows:

Figure 2.36: `cd` command

```
~ $ cd /etc
~ $ pwd
/etc
```

`pwd`[25] simply *prints the working (current) directory*. If `whoami` tells you who you are, `pwd` tells you ***where*** you are.

In Linux, users can have ”home” directories (similar to Windows profiles), typically located under `/home/<username>` for normal users, and `/root` for the ”root” (admin) id. To change to a user’s ”home” directory, simply use `cd` with no parameters:

[24] http://linux.die.net/man/1/cd
[25] http://linux.die.net/man/1/pwd

Figure 2.37: Change to home directory

```
/etc $ cd
~ $ pwd
/home/myuser
```

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

Figure 2.38: Alternative way to change to home directory

```
/etc $ cd ~
~ $ pwd
/home/myuser
```

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

Figure 2.39: Change to the home directory of another user

```
~ # cd ~myuser
myuser # pwd
/home/myuser
```

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

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

- **Absolute path**/index{Files and Directories!Absolute path} - ***always*** "goes through" or specifies the "root" (/) directory, e.g. regardless of the current working directory, `cd /etc` will change it to `/etc`.
- **Relative path**/index{Files and Directories!Relative path} - does ***not*** specify the root directory, and expects to start the navigation at the current directory with all path components traversed from there, e.g., `cd Dissertations` changes the current directory to a subdirectory called `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 makes sense:

Figure 2.40: Relative paths exercise

```
~/Foo $ ls
~/Foo $ mkdir Bar Baz
~/Foo $ ls
Bar  Baz
~/Foo $ cd Bar
~/Foo/Bar $ touch a b c
~/Foo/Bar $ ls
a  b  c
~/Foo/Bar $ cd ../Baz
~/Foo/Baz $ touch d e f
~/Foo/Baz $ ls
d  e  f
~/Foo/Baz $ ls ..
Bar  Baz
~/Foo/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” 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 known as ”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 (`d`) or a file (`-`):

Figure 2.41: Another `ls -l` example, this time on FreeBSD

```
% ls -l /etc
total 1876
drwxr-xr-x   2 root  wheel        512 Jan 15  2009 X11
-rw-r--r--   1 root  wheel          0 Sep  3  2013 aliases
-rw-r--r--   1 root  wheel      16384 Sep  3  2013 aliases.db
-rw-r--r--   1 root  wheel        210 May  6  2009 amd.map
-r--r--r--   1 root  wheel        233 Feb 15  2007 amd.map.snap
-rw-r--r--   1 root  wheel       1234 May  6  2009 apmd.conf
-rw-r--r--   1 root  wheel        231 May  6  2009 auth.conf
drwxr-xr-x   2 root  wheel        512 May  6  2009 bluetooth
-rw-r--r--   1 root  wheel        737 Mar 19  2015 crontab
-rw-r--r--   1 root  wheel        108 May  6  2009 csh.cshrc
-rw-r--r--   1 root  wheel        617 Apr 15  2009 csh.login
-rw-r--r--   1 root  wheel        110 May  6  2009 csh.logout
-rw-r--r--   1 root  wheel        565 May  6  2009 ddb.conf
drwxr-xr-x   2 root  wheel        512 May  6  2009 defaults
-rw-r--r--   1 root  wheel       9779 May  6  2009 devd.conf
-rw-r--r--   1 root  wheel       2071 May  6  2009 devfs.conf
-rw-r--r--   1 root  wheel        267 May  6  2009 dhclient.conf
-rw-r--r--   1 root  wheel       5704 May  6  2009 disktab
-rw-rw-r--   1 root  operator       0 Nov  3  2005 dumpdates
drwxr-xr-x   6 root  staff        512 Nov 12  2014 fail2ban
-rw-r--r--   1 root  wheel        142 May  6  2009 fbtab
...and so on...
```

In the above, for example, we can see that the user `root` owns the file `aliases` while the `wheel` group is the primary group for it. `root` can both read and write the file (`rw-`) while any user in the `wheel` group can only read it (`r--`). Any other id will also have read access (`r--`).

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

Back on Linux, if we look in `/etc/init.d` where many services store their startup scripts we see:

Figure 2.42: Listing the `/etc/init.d` directory

```
~ $ ls -l /etc/init.d
total 276
-rwxr-xr-x 1 root root 2243 Apr  3  2014 acpid
-rwxr-xr-x 1 root root 2014 Feb 19  2014 anacron
-rwxr-xr-x 1 root root 4596 Apr 24  2015 apparmor
-rwxr-xr-x 1 root root 2401 Dec 30  2013 avahi-daemon
-rwxr-xr-x 1 root root 1322 Mar 30  2014 binfmt-support
-rwxr-xr-x 1 root root 4474 Sep  4  2014 bluetooth
-rwxr-xr-x 1 root root 2125 Mar 13  2014 brltty
-rwxr-xr-x 1 root root 4651 Apr  9  2014 casper
-rwxr-xr-x 1 root root  425 Jun 26 09:11 cinnamon
-rwxr-xr-x 1 root root 1919 Jan 18  2011 console-setup
-rwxr-xr-x 1 root root 2489 May  6  2012 cpufrequtils
lrwxrwxrwx 1 root root   21 Sep  7 04:00 cron -> /lib/init/upstart-job
-rwxr-xr-x 1 root root  938 Nov  1  2013 cryptdisks
-rwxr-xr-x 1 root root  896 Nov  1  2013 cryptdisks-early
-rwxr-xr-x 1 root root 3184 Apr  3  2014 cups
-rwxr-xr-x 1 root root 1961 Apr  7  2014 cups-browsed
-rwxr-xr-x 1 root root 2813 Nov 25  2014 dbus
-rwxr-xr-x 1 root root 1217 Mar  7  2013 dns-clean
lrwxrwxrwx 1 root root   21 Sep  7 04:00 friendly-recovery -> /lib/init/upstart-
job
-rwxr-xr-x 1 root root 1105 May 13  2015 grub-common
...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`). Later on I will explain linked files (those that start with an `l` instead of a `-` in the detailed listing above).

To *change* the *owning* user of a file or directory (assuming you have permissions to do so), use the `chown`[26] command:

[26] http://linux.die.net/man/1/chown

Figure 2.43: Change file ownership

```
# 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`[27] command:

Figure 2.44: `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`[28] command. It uses mnemonics of "ugo" for 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 the execute permission for the group and remove read permission for "other":

Figure 2.45: `chmod` command

```
# 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":

[27] http://linux.die.net/man/1/chgrp
[28] http://linux.die.net/man/1/chmod

- **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:

Figure 2.46: `chmod` with lots of typing

```
# 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:

Figure 2.47: `chmod` with octal like a boss

```
# 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!

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:

Figure 2.48: Marking a file as executable

```
# 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`[29] and `unzip`[30] commands:

Figure 2.49: `zip` command

```
~ $ cd Foo
~/Foo $ touch a b c
~/Foo $ mkdir d
~/Foo $ touch d/e
~/Foo $ cd ..
~ $ zip -r Foo Foo
  adding: 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%)
```

[29] http://linux.die.net/man/1/zip
[30] http://linux.die.net/man/1/unzip

```
~ $ ls -l Foo.zip
-rw-r--r-- 1 myuser mygroup 854 Dec 14 15:31 Foo.zip
```

Figure 2.50: `unzip` command

```
~ $ 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, such as the `gzip`[31], `bzip2`[32] and `7z`[33] (7-zip) commands.

However, the "native" way to "archive" a directory's contents in "UNIX" is with `tar`[34], 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":

Figure 2.51: Creating 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 Dec 19 07:52 Foo.tar
~ $ gzip Foo.tar
~ $ ls -l Foo.tar.gz
-rw-r--r-- 1 myuser mygroup 193 Dec 19 07:52 Foo.tar.gz
```

[31] http://linux.die.net/man/1/gzip
[32] http://linux.die.net/man/1/bzip2
[33] http://linux.die.net/man/1/7z
[34] http://linux.die.net/man/1/tar

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`), as well as the older style (`c` with no hyphens at all), as shown in these examples. So both of the following are also equivalent to the above:

Figure 2.52: `tar` parameter styles

```
~ $ 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:

Figure 2.53: One-step tarball

```
~ $ 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 197 Dec 19 07:54 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:

Figure 2.54: Extracting a tarball

```
~ $ 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 for 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). Both are created with the `ln`[35] command. 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:

Figure 2.55: Soft links example

```
~ $ 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
```

[35] http://linux.die.net/man/1/ln

```
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 in a directory pointing to another location. ***What*** that location is will be determined 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”[36] 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.

Figure 2.56: Hard links example

```
~ $ 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
```

[36] https://en.wikipedia.org/wiki/Inode

```
-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:

Figure 2.57: Broken soft links example

```
~ $ 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` is 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 (*splutter* ”I can see it! ***It’s right there!***”)

With hard links, it isn’t so much a problem because of the nature of the beast. Since 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:

Figure 2.58: Many hard linkes, one inode

```
~ $ 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. We created an original file `b` by placing ”This is b.” in it, and then created two hard links to it, `B` and `d/b`. 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.”?

Figure 2.59: Deleting a file with many hard links

```
~ $ 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
```

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`[37] command:

Figure 2.60: `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 interest 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`[38] command. I wanted to, really bad, but with all the different file systems and device types and all the options for both 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. 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.

[37] http://linux.die.net/man/1/df
[38] http://linux.die.net/man/8/mount

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 a command references them. Note that the following is perfectly legal with soft links, but may not give the results you expect - think about the current working directory shown by `pwd` in the following, and the effects of the ***relative paths*** as the sample progresses:

Figure 2.61: Soft links and relative paths

```
~ $ cd Foo
~/Foo $ rm -rf *
~/Foo $ cd ..
~ $ cd Foo
~/Foo $ pwd
/home/myuser/Foo
~/Foo $ rm -rf *
~/Foo $ mkdir d
~/Foo $ touch a b c d/e
~/Foo $ ln -s . d/f
~/Foo $ ls d/f
e  f
~/Foo $ ln -s .. d/g
~/Foo $ 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 - see the next chapter for more).

What's the `diff`?

Most people think of `diff`[39] as a tool only programmers find useful, but that is shortsighted. 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 section, then typed this introduction to `diff`. This is what `diff` showed after I added the new paragraph:

Figure 2.62: `diff` example

[39] http://linux.die.net/man/1/diff

```
~ $ diff Step02.bak Step02.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`:

Figure 2.63: `orig.conf` file

```
~ $ cat orig.conf
FOO=1

SOME=THINGS
STAY=THE
SAME=ALWAYS

BAR=Xyzzy
```

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

Figure 2.64: `new.conf` file

```
~ $ cat new.conf
FOO=2

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

Now if we `diff` them:

Figure 2.65: Using `diff` on config files

```
~ $ 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 version control system to compare against.

`diff` is your friend. It really comes into play with a version control system like `git`[40], but again, that is beyond the scope of this book.

[40] http://linux.die.net/man/1/git

Step 3

Finding Meaning

The `find` command in all its glory. Probably the single most useful command in "UNIX" (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 "UNIX" command?" There certainly are many to consider. But I keep coming back to `find`[1] . 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` are 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":

Figure 3.1: Simplest `find` example

[1] http://linux.die.net/man/1/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:

Figure 3.2: More complicated `find` example

```
~ $ 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 running under Cygwin[2] (this won't work on Linux). Note the regular expression (which we will cover later), in this case saying to look only in directories `0004` through `0009`.

2. **Recursively find file system entries that match one or more tests** - the tests in this example are:

[2] In fact, `find` is one of the main reasons I use Cygwin on Windows.

   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:

Figure 3.3: More complicated `find` example, explained

```
~ $ 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[3]:

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***).

[3] http://www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_04.html

Useful `find` Options

The `find`[4] 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." In other words, placing `-delete` too early in the expression is going to yield behavior distressingly similar to `rm -r *`.

[4] http://linux.die.net/man/1/find

- **`-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 item 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, if for some reason we wanted a report where for each file 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:

Figure 3.4: Using `find` as a simple reporting tool

```
~ $ 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
~ $ 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
```

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

- **`"`** - 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.

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`[1] command is good for finding files whose ***contents*** match a regular expression[2]. You already know at least one regular expression, the wildcard `*` character from 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:

Figure 4.1: `grep` example

```
~ $ 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
```

[1] http://linux.die.net/man/1/grep
[2] https://en.wikipedia.org/wiki/Regular_expression

```
~ $ 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.:

Figure 4.2: A string

```
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:

Figure 4.3: Complex regular expression

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

That shows at least one attempt at being a very complete parser of valid HTTP URLs[3]. Wow! What ***is*** 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[4] and books just on regular expressions. With variations they are used in all ”UNIX” shells, Perl, Python, Javascript, Java, C# and more. So obviously (a) they are really useful, and (b) we’re not going to cover all of regexes here.

[3] http://blog.codinghorror.com/the-problem-with-urls/
[4] http://www.regular-expressions.info/

There are so many things you can do, the only thing to remember is ”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, following are a few ***simple*** regex examples. Consider the file `invoices`:

Figure 4.4: Invoices file

```
~ $ 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”:

Figure 4.5: Trying to find tractors

```
~ $ grep tractor invoices
```

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

Figure 4.6: Trying to find tractors, part two

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

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

Figure 4.7: Let's be insensitive

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

And just to remind you about long-style parameters:

Figure 4.8: Spelling out our insensitivity

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

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

Figure 4.9: Print the line numbers of matches

```
~ $ 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":

Figure 4.10: Extended regular expressions

```
~ $ 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 (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`:

Figure 4.11: Find lines ending with 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”:

Figure 4.12: Recursive `grep`

```
~ $ 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:

Figure 4.13: Recursive `grep` is faster than `find ... -exec grep`

```
~ $ 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:

Figure 4.14: A better example of when to use `find ... -exec grep`

```
~ $ 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 -H -i pdfinfo`) 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 with `find` before handing them to `grep`.

Gawking at awk

I don't have much to say about awk[5] other than:

1. It is named after its three authors, Aho, Weinberger and Kernighan[6], 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 "UNIX" 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 such as Python, 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. If you are searching the internet and find an answer using awk that you can ***quickly*** adapt to your needs, use it.

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[7] from /etc/passwd:

Figure 4.15: awk example

```
~ $ 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
```

[5] http://linux.die.net/man/1/awk
[6] https://en.wikipedia.org/wiki/AWK
[7] http://www.ibm.com/developerworks/library/l-awk1/

```
username: uucp      uid:10
...and so on...
```

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”[1] 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 ”UNIX” 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: Those numeric file descriptors will go from being trivia to important in just a bit.

[1] https://en.wikipedia.org/wiki/Unix_philosophy

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 "UNIX" command convention), and echo each line typed by the user to `stdout` until the "end of file," which in an interactive session can be emulated with `Ctrl-D`, shown as `^D` in the example below but not seen on the console in real life:

Figure 5.1: *stdin* and *stdout*

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

So in the above I typed in "This shows reading from stdin" and hit `Enter` (which send a linefeed and hence marks the "end of the line") and `cat` echoed that line to `stdout`. Then I typed "and writing to stdout." and hit `Enter` and that line was echoed to `stdout` as well. Finally I hit `Ctrl-D`, which ended the process.

All Magic is Redirection

So one way to string things together in "the UNIX way" is with file redirection. This is a concept that also works 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:

Figure 5.2: Hello, world

```
~ $ 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 "UNIX" 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 example using `<` is illustrative of redirecting a file to `stdin` for a command or program:

Figure 5.3: Redundant redirection

```
~ $ 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:

Figure 5.4: Default *stderr* behavior

```
~ $ 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`, but there is no good way of visually telling that. One way to get rid of them would be to change `find` to filter for only files:

Figure 5.5: Get rid of the errors in the first place

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

Figure 5.6: Redirecting *stderr*

```
~ $ 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:

Figure 5.7: Redirecting both *stdout* and *stderr* to a file

```
~ $ 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`. Note that this works in `CMD.EXE`]drshl{CMD.EXE}, too!

If we want to send `stdout` to one file and `stderr` to another, you can do it like this:

Figure 5.8: Redirecting *stdout* one way *stderr* another

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

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

Figure 5.9: Overwriting a file with redirection

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

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

Figure 5.10: Appending to a file with redirection

```
~ $ 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 "UNIX" 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 using the "pipe" (`|`):

Figure 5.11: Piping output between programs

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

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

1. `cat` echos the contents of all `.txt` files in alphabetical order by their file name to `stdout`, which is piped to...
2. `tr`[2] "translates" (replaces) 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 ransformation or the pipeline!

That is the "UNIX philosophy" at work.

[2] http://linux.die.net/man/1/tr

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`[3] 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.

[3] http://linux.die.net/man/1/tee

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`[1] 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`[2] ("**vi** i**m**proved"), that is then accessed via the alias `vi`. The differences tend to be minor, with `vim` being more customizable.

`vi` and a similar editor, `emacs`[3], 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`xit insert mode} key exits insert mode.

[1] http://linux.die.net/man/1/vi
[2] http://www.vim.org/
[3] http://linux.die.net/man/1/emacs

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 you ***will*** insert as text, and sooner or later you ***will*** enter `vi` commands into Notepad. I guarantee it. That will be the day you know you've become truly tainted.

We will not even begin to scratch the surface of `vi`, when there are 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, "undoing" or "redoing," 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` (zero) 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:

Figure 6.1: Deleting a "word"

```
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:

Figure 6.2: After deleting the "word"

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

We can also specify the number of times we want to perform a command by prefixing it to 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:

Figure 6.3: Deleting multiple words

```
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:

Figure 6.4: Replace three characters with "x"

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

...results in:

Figure 6.5: Three "x" characters

```
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`](http://linux.die.net/man/1/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:

Figure 6.6: Garbage characters

```
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 the file.
- **`:0`** - jumps to start of the file (note the preceding `:`).

- **`/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 `?`.

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:

Figure 6.7: Deleting a line

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

Then executing `dd` leaves:

Figure 6.8: After the line is gone

```
|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`(uppercase `P` = ”upper” or ”above”), which would result in:

Figure 6.9: After pasting the line above the current line

```
|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 or replace) command, `:s`. 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:

Figure 6.10: Sample text 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:

Figure 6.11: Changing "this" to "that"

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

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

Figure 6.12: What happened?

```
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 change, and try again with this:

Figure 6.13: Changing "this" to "that", redux

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

Results in:

Figure 6.14: Closer, but not quite

```
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:

Figure 6.15: Changing "this" to "that", one more time!

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

Results in:

Figure 6.16: Finally!

```
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! Well, sort of. If we wanted to keep the capitalization we'd have more work to do. See below.

In general, if you are looking for a case insensitive "change all" like in Windows Notepad, the magic string to remember is:

Figure 6.17: Memorize this - No, really

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

Picking that apart, we have:

- **`:`** - tells `vi` a special command is coming.
- **`0,$`** - specifies a line range, in this case from the first (`0` - zero-relative) line to last (`$`) line 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 (see about marking lines, below).
- **`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 where we left off:

Figure 6.18: But what about capitalization?

```
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:

Figure 6.19: Regular expression for the start of a line

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

Yields:

Figure 6.20: Voila! Capitals!

```
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."

Figure 6.21: Regular expression for the end of a line

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

Ends up with:

Figure 6.22: That with a full stop

```
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:

Figure 6.23: Say what?

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

Renders this:

Figure 6.24: Nicely punctuated

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

Hint: While trying to figure that out, search the Internet for regular expression ”capturing groups.”

”X” Marks the Spot

You can ”mark” lines in `vi` for use in ”ranges” like the ”substitute” (change) command above. Let’s say you have a file like the following:

Figure 6.25: Simple file

```
This is a line
This is also a line
This, too
This is next
This is last
```

Maybe we want to change the ”This” on the first three lines to ”That,” but not the last two (imagine this is a much more complex example). We could do it by hand with the `r` command, but that’s tedious and error prone. Instead, we can ”mark” a range.

1. Place the cursor on the first line and use the `m` command followed by a one-character ”label” like `x` (I typically use `m` so I don’t have to move my fingers, e.g., `mm`).

2. Place the cursor on the third line and again use the `m` command, but ***with a different label character*** (I usually use `n` so my fingers don’t travel far, so `mn`).

3. Now you can use the `'` character followed by a label to denote the beginning and end of the range in all kinds of `vi` commands. In our case we want to change ”This” on the first three lines, so:

```
:'m,'ns/This/That/
```

Try doing that in Notepad!

Note: We could have done that with line ranges, too (`:0,2s/This/That/`), but figuring out the beginning and ending lines in a large range is a pain. It is much easier to just mark them and go.

Executing External Commands

Sometimes in `vi` it would be great to run the contents of the file through an external command (`sort` is a favorite) without saving and exiting the file, sorting it, and then re-editing it. We can do that with `!` , which works a lot like the ”substitute” (change) command.

To sort the whole file in place:

Figure 6.26: Sort a whole file in vi

```
:0,$!sort
```

To sort a marked range:

Figure 6.27: Sorting a range

```
:'m,'n!sort
```

Another handy command to check out for this kind of thing, especially for formatting written text, is the `fmt`[4] command.

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”[5] that range

[4] http://linux.die.net/man/1/fmt
[5] https://en.wikipedia.org/wiki/Control_character

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 ”UNIX” systems and Windows.

On ”UNIX,” 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 ”UNIX” end-of-lines (`\n`) only. Use a line feed aware editor such as Notepad++[6] 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 override the `ff` (file format) setting:

   ```
   :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:

Figure 6.28: Change all tabs to four spaces as God meant them to be

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

[6] https://notepad-plus-plus.org/

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`[7] , 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`. The prognosis is grim.

But there may also others, notably `pico`[8] and its successor, `nano`[9] . You can see the difference the second you see a file open in `nano` - in this case, the generated Github-flavored Markdown of this document:

Figure 6.29: Editing a file in `nano`

```
    GNU nano 2.2.6        File: TenStepsToLinuxSurvival.md

|![Merv sez, "Don't panic."](./images/Merv.jpg "Merv sez, 'Don't panic.'")
Merv sez, "Don't panic."

By James Lehmer

v0.7

![](./images/cc-by-sa.png "Creative Commons Attribution-ShareAlike 4.0 Internat$
*Jim's Ten Steps to Linux Survival* by James Lehmer is licensed under a [Creati$

**Dedicated to my first three technical mentors** - Jim Proffer, who taught me $

Introduction
============
```

[7] http://linux.die.net/man/1/emacs
[8] http://linux.die.net/man/1/pico
[9] http://linux.die.net/man/1/nano

```
> *"And you may ask yourself, 'Well, how did I get here?'"* - Talking Heads (*O$

This is my little "Linux and Bash in 10 steps" guide. It's based around what I $

                              [ Read 3627 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 such as 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! You can find out simply by typing in `nano <filename>` on what you want to edit and see if it works. But if `nano` or `pico` aren't installed, grit your teeth, remember "insert mode" vs. "command mode", and use `vi`.

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

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`[1] is there:

Figure 7.1: ping command

```
# 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=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=2 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=3 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=4 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=5 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=6 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=7 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=8 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=9 ttl=...
64 bytes from ir1.fp.vip.ne1.yahoo.com (98.138.253.109): icmp_req=10 ttl...
```

[1] http://linux.die.net/man/8/ping

```
^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[2] 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`[3] works, too (although for once its name is longer than the `CMD.EXE` counterpart).

Figure 7.2: traceroute command

```
~ $ traceroute google.com
traceroute to google.com (216.58.216.78), 30 hops max, 60 byte packets
 1  192.168.0.1 (192.168.0.1)  3.623 ms  3.978 ms  7.231 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  72.14.215.212 (72.14.215.212)  26.205 ms  27.502 ms  27.648 ms
 8  209.85.242.133 (209.85.242.133)  31.547 ms  31.550 ms  31.548 ms
 9  72.14.237.231 (72.14.237.231)  29.516 ms  29.556 ms  29.657 ms
10  ord30s21-in-f78.1e100.net (216.58.216.78)  30.313 ms  33.138 ms  28.092 ms
```

You can do some digging in DNS with `dig`[4]:

Figure 7.3: dig command

```
~ $ dig yahoo.com

; <<>> DiG 9.9.5-3ubuntu0.6-Ubuntu <<>> yahoo.com
;; global options: +cmd
```

[2] https://en.wikipedia.org/wiki/Unix_signal
[3] http://linux.die.net/man/8/traceroute
[4] http://linux.die.net/man/1/dig

```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46478
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;yahoo.com.         IN  A

;; ANSWER SECTION:
yahoo.com.      605 IN  A   98.138.253.109
yahoo.com.      605 IN  A   206.190.36.45
yahoo.com.      605 IN  A   98.139.183.24

;; Query time: 23 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Tue Dec 22 09:46:26 CST 2015
;; MSG SIZE  rcvd: 86
```

And `whois`[5]:

```
~ $ 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

   Server Name: YAHOO.COM.ANGRYPIRATES.COM
   IP Address: 8.8.8.8
   Registrar: NAME.COM, INC.
   Whois Server: whois.name.com
```

[5] http://linux.die.net/man/1/whois

```
   Referral URL: http://www.name.com

   Server Name: YAHOO.COM.AU
   Registrar: WILD WEST DOMAINS, LLC
...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`[6] 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) 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, (unlike UAC).

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

Figure 7.4: Make me a sandwich

```
~ $ apt-get update
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denie
d)
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`:

Figure 7.5: sudo Make me a sandwich

[6] http://linux.die.net/man/8/sudo

```
~ $ sudo apt-get update
Ign http://packages.linuxmint.com rafaela InRelease
Ign http://extra.linuxmint.com rafaela InRelease
Hit http://extra.linuxmint.com rafaela Release.gpg
Hit http://packages.linuxmint.com rafaela Release.gpg
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
Hit http://archive.ubuntu.com trusty-updates InRelease
Hit http://security.ubuntu.com trusty-security/main amd64 Packages
Hit http://packages.linuxmint.com rafaela/main amd64 Packages
Hit http://extra.linuxmint.com rafaela/main amd64 Packages
Ign http://archive.canonical.com trusty InRelease
Hit http://archive.ubuntu.com trusty Release.gpg
Hit http://security.ubuntu.com trusty-security/restricted amd64 Packages
Hit http://extra.linuxmint.com rafaela/main i386 Packages
Hit http://packages.linuxmint.com rafaela/upstream amd64 Packages
Hit http://security.ubuntu.com trusty-security/universe amd64 Packages
Hit http://archive.ubuntu.com trusty-updates/main amd64 Packages
Hit http://packages.linuxmint.com rafaela/import amd64 Packages
Hit http://security.ubuntu.com trusty-security/multiverse amd64 Packages
Hit http://archive.canonical.com trusty Release.gpg
...and so on...
```

Now you should get the punchline to this comic[7], and hence the title of this section.

Surfin' the Command Prompt

You can browse the web from the command prompt using something like `lynx`[8]. 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:

Figure 7.6: Browsing like it's 1994

[7] https://xkcd.com/149/
[8] http://linux.die.net/man/1/lynx

```
                                                                    Google

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

   Google

     ____________________________________________________
     Google Search  I'm Feeling Lucky                      Advanced search
                                                           Language tools

   Advertising Programs     Business Solutions     +Google     About
   Google

                         © 2015 - Privacy - Terms




(NORMAL LINK) Use right-arrow or <return> to activate.
  Arrow keys: Up and Down to move.  Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

There are two other commands that are used to pull down web resources and save them locally - `curl`[9] and `wget`[10] drcmd{wget}. Both support HTTP(S) and FTP, but `curl` supports even more protocols and options and tends to be the simplest to just "grab a file and go." You see both used often in install scripts that download bits from the internet and then execute them:

Figure 7.7: `wget` in an install script

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

Or:

Figure 7.8: `curl` in an install script

[9] http://linux.die.net/man/1/curl
[10] http://linux.die.net/man/1/wget

```
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. It is often better to use something like `curl` to download the script but instead of piping it to `bash` to be executed, redirect it to a file and look at what the script is doinng first:

Figure 7.9: Check out what that script is doing first!

```
~ $ curl http://foocorp.com/installs/install.sh > install.sh
~ $ cat install.sh
#!/bin/bash
# I'm a script from a bad guy, check out the next line!
rm -rf /*
```

Now, aren't you glad you didn't just execute that without checking?

But if the script looks right, then you can `chmod` it and run it:

Figure 7.10: Got a good script, so execute it

```
~ $ curl http://foocorp.com/installs/install.sh > install.sh
~ $ cat install.sh
#!/bin/bash
# I'm a script from a good guy.
# Do some stuff...
~ $ chmod 770 install.sh
~ $ ./install.sh
```

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`[11] installed, that's probably the most intuitive from a non-UNIX perspective (although it is still obviously a terminal program). Otherwise look for `mutt`[12].

[11] http://linux.die.net/man/1/pine
[12] http://linux.die.net/man/1/mutt

Sending email is more interesting, especially from shell scripts. There are multiple ways, but `email`[13] is straightforward enough:

Figure 7.11: Sending email from the command line

```
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`[14] 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:

Figure 7.12: Using `telnet` to diagnose HTTP

```
~ $ telnet google.com 80
Trying 216.58.216.78...
Connected to google.com.
Escape character is '^]'.
GET / HTTP/1.1

HTTP/1.1 200 OK
Date: Tue, 22 Dec 2015 15:58:47 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 https://www.google.com/support/accounts/a
nswer/151657?hl=en for more info."
Server: gws
```

[13] http://linux.die.net/man/1/email
[14] http://linux.die.net/man/1/telnet

```
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=74=nqD9y_pSQudbaw6obB94Ngw6lsn4t_S8Z3NbZcUJ5HB4qUXCpu988A5QG3EQD
kwqgOdGapsUSmsi91yHAa9_LU9JeP4pKop-1p5w7LlrdMyGrGojwoaX58ML6PSH5nGLsdZV0Z5vBqNTh
A; expires=Wed, 22-Jun-2016 15:58:47 GMT; path=/; domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

...and so on...
```

To get a modern, secure shell to a remote machine, use `ssh`[15] , passing in the userid and server like this:

Figure 7.13: `ssh` command

```
ssh myuser@remoteserver
```

You will be prompted for credentials (or you can use certificates, but that is ***way*** beyond this text's goals). Once logged in, you will be presented with a command prompt to the remote system.

You can also use the `SSH` protocol to securely transfer files between systems with the `scp`[16] command. It works like this for a recursive directory copy:

Figure 7.14: `scp` command

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

In this case we are copying the files in `myfiles` and its subdirectories to `/home/myuser/` 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":

Figure 7.15: Sample `ssh` session

[15] http://linux.die.net/man/1/ssh
[16] http://linux.die.net/man/1/scp

```
~# ssh myuser@remotehost
The authenticity of host '[remotehost] ([10.0.2.3]:22)' can't be established.
ECDSA key fingerprint is 98:bb:17:38:ee:d0:16:ee:b2:93:08:4e: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`[17] command (in some ways is similar to `ipconfig` in `CMD.EXE`. While you can use `ifconfig` to alter your networking settings, it is most commonly used to get a quick display of them:

Figure 7.16: `ifconfig` command

```
# 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)
```

[17] http://linux.die.net/man/8/ifconfig

```
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, you can look in `/etc/resolv.conf` :

Figure 7.17: DNS servers in `resolv.conf`

```
# 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, look in `/etc/hosts` :

Figure 7.18: `hosts` file

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

Note: The UNIX `/etc/hosts` file is the basis for the Windows version located at `C:\Windows\System32\drivers\etc\hosts`, and has similar syntax.

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`[1]:

Figure 8.1: `ps` command

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

[1] http://linux.die.net/man/1/ps

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

Figure 8.2: Showing all processes

```
~ $ 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 ?        00:00:06   rcu_sched
    8 ?        00:00:02   rcuos/0
    9 ?        00:00:01   rcuos/1
   10 ?        00:00:03   rcuos/2
   11 ?        00:00:01   rcuos/3
   12 ?        00:00:00   rcuos/4
   13 ?        00:00:00   rcuos/5
   14 ?        00:00:00   rcuos/6
   15 ?        00:00:00   rcuos/7
   16 ?        00:00:00   rcu_bh
   17 ?        00:00:00   rcuob/0
   18 ?        00:00:00   rcuob/1
   19 ?        00:00:00   rcuob/2
   20 ?        00:00:00   rcuob/3
   21 ?        00:00:00   rcuob/4
   22 ?        00:00:00   rcuob/5
   23 ?        00:00:00   rcuob/6
   24 ?        00:00:00   rcuob/7
...and so on...
```

You can *kill* a process using the `kill`[2] 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:

Figure 8.3: Hunting down and killing `vi` sessions

[2] http://linux.die.net/man/1/kill

```
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`[3] 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`[4] command, which unlike most in this book updates dynamically every second by default:

Figure 8.4: `top` command

```
top - 14:11:26 up 106 days,  5:24,  2 users,  load average: 0.11, 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, ...
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
```

[3] http://linux.die.net/man/1/cut
[4] http://linux.die.net/man/1/top

```
    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:

Figure 8.5: `/proc` file system

```
~ $ ls /proc
1     1566  2607  299   4549  53    75         cmdline      mtrr
10    1587  2617  3     4579  54    754        consoles     net
100   16    2627  300   4589  55    760        cpuinfo      pagetypeinfo
1022  17    2629  301   46    56    762        crypto       partitions
1030  18    2699  3029  4602  575   764        devices      sched_debug
1035  1803  27    31    4612  61    77         diskstats    schedstat
1038  19    2712  3111  47    6146  79         dma          scsi
11    2     2799  3112  48    6153  8          driver       self
12    20    28    3116  49    6199  8167       execdomains  slabinfo
1295  2073  2802  3117  4955  62    8168       fb           softirqs
1297  2077  2811  3150  4958  6212  8200       filesystems  stat
13    21    2815  32    4960  63    822        fs           swaps
1304  22    2820  324   4976  640   8296       interrupts   sys
1305  23    2823  326   5     642   9          iomem        sysrq-trigger
1306  2324  2825  329   50    645   9266       ioports      sysvipc
1308  2349  2829  33    5005  6463  927        irq          timer_list
```

```
1311  2356  2831  330   5012  647   939        kallsyms     timer_stats
14    24    2836  34    5033  649   9465       kcore        tty
1408  2494  2846  36    5045  661   9613       keys         uptime
1468  25    2847  37    51    665   9796       key-users    version
147   2507  2848  3713  511   676   9850       kmsg         version_signature
148   2518  2850  374   5122  686   99         kpagecount   vmallocinfo
...and so on...
```

What is all that? Well, look a little closer:

Figure 8.6: Detailed listing of the `/proc` file system

```
~ $ ls -l /proc
total 0
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 10
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 100
dr-xr-xr-x  9 myuser     mygroup                  0 Dec 22 10:17 10035
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1022
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1030
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1035
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1038
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 11
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 12
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1295
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1297
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 13
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1304
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1305
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1306
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1308
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1311
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 14
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1408
dr-xr-xr-x  9 root       root                     0 Dec 22 06:06 1468
...and so on...
```

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

Figure 8.7: Looking inside one of the `/proc` process directories

```
~ # ls -l /proc/1
total 0
dr-xr-xr-x 2 root root 0 Dec 22 10:18 attr
-rw-r--r-- 1 root root 0 Dec 22 10:18 autogroup
-r-------- 1 root root 0 Dec 22 10:18 auxv
-r--r--r-- 1 root root 0 Dec 22 06:06 cgroup
--w------- 1 root root 0 Dec 22 10:18 clear_refs
-r--r--r-- 1 root root 0 Dec 22 06:06 cmdline
-rw-r--r-- 1 root root 0 Dec 22 10:18 comm
-rw-r--r-- 1 root root 0 Dec 22 10:18 coredump_filter
-r--r--r-- 1 root root 0 Dec 22 10:18 cpuset
lrwxrwxrwx 1 root root 0 Dec 22 10:18 cwd -> /
-r-------- 1 root root 0 Dec 22 06:06 environ
lrwxrwxrwx 1 root root 0 Dec 22 06:06 exe -> /sbin/init
dr-x------ 2 root root 0 Dec 22 10:18 fd
dr-x------ 2 root root 0 Dec 22 10:18 fdinfo
-rw-r--r-- 1 root root 0 Dec 22 10:18 gid_map
-r-------- 1 root root 0 Dec 22 10:18 io
-r--r--r-- 1 root root 0 Dec 22 06:06 limits
-rw-r--r-- 1 root root 0 Dec 22 10:18 loginuid
dr-x------ 2 root root 0 Dec 22 10:18 map_files
-r--r--r-- 1 root root 0 Dec 22 10:18 maps
-rw------- 1 root root 0 Dec 22 10:18 mem
...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:

Figure 8.8: How much I/O has process 1 done?

```
~ # cat /proc/1/io
rchar: 803882767
wchar: 152731542
syscr: 201510
syscw: 57855
read_bytes: 663872512
```

```
write_bytes: 113012736
cancelled_write_bytes: 3072000
```

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`:

Figure 8.9: Looking at CPU info in `/proc/cpuinfo`

```
~ # cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 69
model name  : Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz
stepping    : 1
microcode   : 0x14
cpu MHz     : 895.023
cache size  : 3072 KB
physical id : 0
siblings    : 4
core id     : 0
cpu cores   : 2
apicid      : 0
initial apicid  : 0
fpu     : yes
fpu_exception   : yes
cpuid level : 13
wp      : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pd
pe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopol
ogy nonstop_tsc ap
...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 Linux Mint system:

Figure 8.10: Looking at logs

```
~ # ls /var/log
alternatives.log       dmesg.4.gz             pm-suspend.log.1
alternatives.log.1     dpkg.log               pm-suspend.log.2.gz
alternatives.log.2.gz  dpkg.log.1             pm-suspend.log.3.gz
alternatives.log.3.gz  dpkg.log.2.gz          pycentral.log
apt                    dpkg.log.3.gz          samba
aptitude               faillog                speech-dispatcher
aptitude.1.gz          fontconfig.log         syslog
aptitude.2.gz          fsck                   syslog.1
auth.log               gpu-manager.log        syslog.2.gz
auth.log.1             hp                     syslog.3.gz
auth.log.2.gz          installer              syslog.4.gz
auth.log.3.gz          kern.log               syslog.5.gz
auth.log.4.gz          kern.log.1             syslog.6.gz
boot.log               kern.log.2.gz          syslog.7.gz
bootstrap.log          kern.log.3.gz          udev
btmp                   kern.log.4.gz          unattended-upgrades
btmp.1                 lastlog                upstart
ConsoleKit             mdm                    wtmp
cups                   mintsystem.log         wtmp.1
dmesg                  pm-powersave.log       Xorg.0.log
dmesg.0                pm-powersave.log.1     Xorg.0.log.old
dmesg.1.gz             pm-powersave.log.2.gz  Xorg.20.log
...and so on...
```

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 log 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:

Figure 8.11: Kernel errors when booting

```
~ # cat /var/log/dmesg | grep -i error
[   15.828463] EXT4-fs (dm-1): 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.

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 "UNIX" is not the lack of documentation but almost the surplus of it, coupled with a severe "RTFM" attitude by most old-timers toward newbies. Besides the typical "Google" and "StackOverflow"[1] 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 "UNIX" documentation within "UNIX" itself - `man`[2] , `info`[3] and `apropos`[4] .

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

[1] http://unix.stackexchange.com/
[2] http://linux.die.net/man/1/man
[3] http://linux.die.net/man/1/info
[4] http://linux.die.net/man/1/apropos

Figure 9.1: man command

```
LS(1)                          User Commands                          LS(1)

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  speci□
       fied.

       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 ..

       --author
 Manual page ls(1) line 1 (press h for help or q to quit)
```

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:

Figure 9.2: Ambiguous man commands default to lowest documentation section

```
PASSWD(1)                       User Commands                       PASSWD(1)

NAME
       passwd - change user password

SYNOPSIS
       passwd [options] [LOGIN]

DESCRIPTION
       The passwd command changes passwords for user accounts. A normal user
       may only change the password for his/her own account, while the
       superuser may change the password for any account.  passwd also changes
       the account or associated password validity period.

   Password Changes
       The user is first prompted for his/her old password, if one is present.
       This password is then encrypted and compared against the stored
       password. The user has only one chance to enter the correct password.
       The superuser is permitted to bypass this step so that forgotten
       passwords may be changed.

       After the password has been entered, password aging information is
       checked to see if the user is permitted to change the password at this
 Manual page passwd(1) line 1 (press h for help or q to quit)
```

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`:

Figure 9.3: Specifying a specific man section

```
PASSWD(5)               File Formats and Conversions               PASSWD(5)

NAME
       passwd - the password file

DESCRIPTION
       /etc/passwd contains one line for each user account, with seven fields
```

```
       delimited by colons (“:”). These fields are:

       ·   login name

       ·   optional encrypted password

       ·   numerical user ID

       ·   numerical group ID

       ·   user name or comment field

       ·   user home directory

       ·   optional user command interpreter

 Manual page passwd(5) line 1 (press h for help or q to quit)
```

Besides `man`, many GNU tools come with help in `info` format, which is originally from `emacs`. Here are the results of `info find`:

Figure 9.4: Running info on the find command

```
File: find.info,  Node: Invoking find,  Next: Invoking locate,  Up: Reference

7.1 Invoking 'find'
===================

     find [-H] [-L] [-P] [-D DEBUGOPTIONS] [-OLEVEL] [FILE...] [EXPRESSION]

   'find' searches the directory tree rooted at each file name FILE by
evaluating the EXPRESSION on each file it finds in the tree.

   The command line may begin with the '-H', '-L', '-P', '-D' and '-O'
options.  These are followed by a list of files or directories that
should be searched.  If no files to search are specified, the current
directory ('.') is used.

   This list of files to search is followed by a list of expressions
describing the files we wish to search for.  The first part of the
```

```
expression is recognised by the fact that it begins with '-' followed by
some other letters (for example '-print'), or is either '(' or '!'.  Any
arguments after it are the rest of the expression.

   If no expression is given, the expression '-print' is used.
--zz-Info: (find.info.gz)Invoking find, 44 lines --Top-------------------------
Welcome to Info version 5.2. Type h for help, m for menu item.
```

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 find` 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:

Figure 9.5: apropos command

```
~ $ apropos edit
atobm (1)            - bitmap editor and converter utilities for the X Window...
bitmap (1)           - bitmap editor and converter utilities for the X Window...
bmtoa (1)            - bitmap editor and converter utilities for the X Window...
cinnamon-menu-editor (1) - Editor for the panel menu
desktop-file-edit (1) - Installation and edition of desktop files
desktop-file-install (1) - Installation and edition of desktop files
ed (1)               - line-oriented text editor
edit (1)             - execute programs via entries in the mailcap file
editdiff (1)         - fix offsets and counts of a hand-edited diff
editkeep (8)         - frontend for deborphan
editor (1)           - Nano's ANOther editor, an enhanced free Pico clone
editres (1)          - a dynamic resource editor for X Toolkit applications
elfedit (1)          - Update the ELF header of ELF files.
ex (1)               - Vi IMproved, a programmers text editor
fix-qdf (1)          - repair PDF files in QDF form after editing
gedit (1)            - text editor for the GNOME Desktop
gnome-desktop-item-edit (1) - tool to edit .desktop file
gnome-text-editor (1) - text editor for the GNOME Desktop
Gnome2::DateEdit (3pm) - wrapper for GnomeDateEdit
grub-editenv (1)     - edit GRUB environment block
```

```
jfs_debugfs (8)      - shell-type JFS file system editor
mintsources (1)      - Software Sources List editor
...and so on...
```

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 "UNIX" 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:

Figure 9.6: Refining output from `apropos`

```
~ $ apropos edit | grep "(8)"
editkeep (8)         - frontend for deborphan
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-gr...
vipw (8)             - edit the password, group, shadow-password or shadow-gr...
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`:

Figure 9.7: Looking for specific parameter names in a man page

```
$ 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 "UNIX" 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"[5] 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)**[6] - has fallen a bit behind over the years, but still has two of the best bash scripting books out there, *Bash Guide for Beginners*[7] and *Advanced Bash-Scripting Guide*[8] . I continue to use the latter all the time.

- **Arch Linux Wiki**[9] - you may not think this would be useful if you are running Debian or Fedora or something else, but remember most "UNIX" 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**[10] - 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*[11] and the *Debian Reference*[12] , which is a lot more formal attempt at the same type of territory this guide covers.

[5] https://www.google.com/#q=man+ls
[6] http://www.tldp.org/guides.html
[7] http://www.tldp.org/LDP/Bash-Beginners-Guide/html/index.html
[8] http://www.tldp.org/LDP/abs/html/index.html
[9] https://wiki.archlinux.org/
[10] https://www.debian.org/doc/
[11] https://www.debian.org/doc/manuals/debian-handbook/
[12] https://www.debian.org/doc/manuals/debian-reference/

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[13] for setting up CIFS shares on Linux, you should always look at the package site’s documentation[14] as well as any specific info you can find about the distro you are running.

[13] https://www.samba.org/samba/
[14] https://www.samba.org/samba/docs/

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 else, 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."

Figure 10.1: /etc directory

```
~ $ ls /etc
acpi                    hosts                pki
adduser.conf            hosts.allow          pm
adjtime                 hosts.deny           pnm2ppa.conf
alternatives            hp                   polkit-1
anacrontab              icedtea-web          ppp
```

```
apg.conf                ifplugd              printcap
apm                     ImageMagick          profile
apparmor                init                 profile.d
apparmor.d              init.d               protocols
apport                  initramfs-tools      pulse
apt                     inputrc              purple
at-spi2                 insserv              python
avahi                   insserv.conf         python2.7
bash.bashrc             insserv.conf.d       python3
bash_completion         inxi.conf            python3.4
bash_completion.d       iproute2             rc0.d
bindresvport.blacklist  issue                rc1.d
blkid.conf              issue.dpkg-old       rc2.d
blkid.tab               issue.net            rc3.d
bluetooth               issue.net.dpkg-old   rc4.d
bonobo-activation       java-7-openjdk       rc5.d
brlapi.key              kbd                  rc6.d
...and so on...
```

Depending on what you are trying to configure, you may need to 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`:

Figure 10.2: init.d directory

```
~ $ ls /etc/init.d
acpid             dbus              ondemand     single
anacron           dns-clean         pppd-dns     skeleton
apparmor          friendly-recovery procps       smbd
avahi-daemon      grub-common       pulseaudio   speech-dispatcher
binfmt-support    halt              rc           sudo
bluetooth         hddtemp           rc.local     udev
brltty            irqbalance        rcS          umountfs
casper            kerneloops        README       umountnfs.sh
cinnamon          killprocs         reboot       umountroot
console-setup     kmod              resolvconf   unattended-upgrades
cpufrequtils      lm-sensors        rsync        urandom
cron              loadcpufreq       rsyslog      virtualbox-guest-utils
cryptdisks        mdm               samba        virtualbox-guest-x11
cryptdisks-early  mintsystem        samba-ad-dc  x11-common
cups              networking        saned
cups-browsed      nmbd              sendsigs
```

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

Figure 10.3: Stopping and starting services

```
~ # /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[1] 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`[2] is usually the tool of choice for package management. Another option is `aptitude`[3].

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:

Figure 10.4: apt-get update

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

[1] https://chocolatey.org/
[2] http://linux.die.net/man/8/apt-get
[3] http://linux.die.net/man/8/aptitude

```
Hit http://security.ubuntu.com trusty-security/main amd64 Packages
Hit http://packages.linuxmint.com rafaela/main amd64 Packages
Hit http://extra.linuxmint.com rafaela/main amd64 Packages
Hit http://security.ubuntu.com trusty-security/restricted amd64 Packages
Hit http://extra.linuxmint.com rafaela/main i386 Packages
Hit http://packages.linuxmint.com rafaela/upstream amd64 Packages
Ign http://archive.canonical.com trusty InRelease
Ign http://archive.ubuntu.com trusty InRelease
Hit http://security.ubuntu.com trusty-security/universe amd64 Packages
Hit http://packages.linuxmint.com rafaela/import amd64 Packages
Hit http://security.ubuntu.com trusty-security/multiverse amd64 Packages
Hit http://packages.linuxmint.com rafaela/main i386 Packages
Hit http://archive.canonical.com trusty Release.gpg
Hit http://archive.ubuntu.com trusty-updates InRelease
...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):

Figure 10.5: Upgrading installed packages

```
~ $ 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:

Figure 10.6: Installing a package

```
~ $ sudo apt-get install traceroute
Reading package lists... Done
Building dependency tree
```

```
Reading state information... Done
The following NEW packages will be installed:
  traceroute
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/45.0 kB of archives.
After this operation, 176 kB of additional disk space will be used.
Selecting previously unselected package traceroute.
(Reading database ... 307895 files and directories currently installed.)
Preparing to unpack .../traceroute_1%3a2.0.20-0ubuntu0.1_amd64.deb ...
Unpacking traceroute (1:2.0.20-0ubuntu0.1) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up traceroute (1:2.0.20-0ubuntu0.1) ...
update-alternatives: using /usr/bin/traceroute.db to provide /usr/bin/traceroute
 (traceroute) in auto mode
update-alternatives: using /usr/bin/lft.db to provide /usr/bin/lft (lft) in auto
 mode
update-alternatives: using /usr/bin/traceproto.db to provide /usr/bin/traceproto
 (traceproto) in auto mode
update-alternatives: using /usr/sbin/tcptraceroute.db to provide /usr/sbin/tcptr
aceroute (tcptraceroute) in auto mode
```

You can also `apt-get remove` or `apt-get purge` packages. See the man page for details.

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. Debian itself is a good example of this, as are many ”LTS” (long term support) releases in other distros.

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`:

Figure 10.7: Installing a package 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`[4] and execution environments like node have `npm`[5]. 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`[6] command for just that!

Figure 10.8: which command

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

[4] https://pypi.python.org/pypi/pip/
[5] https://www.npmjs.com/
[6] http://linux.die.net/man/1/which

How can you tell if you have multiple versions of something installed? One way is with the `locate`[7] command:

Figure 10.9: 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
/usr/lib/i386-linux-gnu/sasl2/libcrammd5.so.2
/usr/lib/i386-linux-gnu/sasl2/libcrammd5.so.2.0.25
/usr/lib/i386-linux-gnu/sasl2/libdigestmd5.so
/usr/lib/i386-linux-gnu/sasl2/libdigestmd5.so.2
/usr/lib/i386-linux-gnu/sasl2/libdigestmd5.so.2.0.25
/usr/lib/python2.7/md5.py
/usr/lib/python2.7/md5.pyc
/usr/lib/ruby/1.9.1/x86_64-linux/digest/md5.so
/usr/lib/x86_64-linux-gnu/sasl2/libcrammd5.so
/usr/lib/x86_64-linux-gnu/sasl2/libcrammd5.so.2
/usr/lib/x86_64-linux-gnu/sasl2/libcrammd5.so.2.0.25
/usr/lib/x86_64-linux-gnu/sasl2/libdigestmd5.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 in real time). You are simply searching the database for a pattern. It is a quicker way to look than `find / -name \*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`[8] 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:

[7] http://linux.die.net/man/1/locate
[8] http://linux.die.net/man/1/pwd

Figure 10.10: Command not found - but it's right there!

```
~ $ 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`:

Figure 10.11: Using a fully qualified path to execute a command

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

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

Figure 10.12: Specifying the command in the current directory

```
~ $ ./foo
```

Over and Over and Over

The function of scheduled tasks in Windows is performed by `cron`[9] in Linux. It reads in the various `crontab(5)`[10] files on the system and executes the commands in them at the

[9] http://linux.die.net/man/8/cron
[10] http://linux.die.net/man/5/crontab

specified times. You use the `crontab(1)`[11] command to view and edit the `crontab` files for your user (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:

Figure 10.13: Looking at default 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:

Figure 10.14: Editing another user's crontab file

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

This can be useful to do things like run backup jobs as the user that is running the web

[11] http://linux.die.net/man/1/crontab

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`[12] command:

Figure 10.15: reboot command

```
$ sudo reboot
```

You can also use the `shutdown`[13] command with the `-r` option, but why? The handier use for `shutdown` is to tell a system to halt and power off after shutting down:

Figure 10.16: Shutdown and power off

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

Turn on Your Signals

One of the basic concepts in UNIX program is that of "signals"[14]. 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.

[12] http://linux.die.net/man/8/reboot
[13] http://linux.die.net/man/8/shutdown
[14] https://en.wikipedia.org/wiki/Unix_signal

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

Figure 10.17: Terminating a process with extreme prejudice

```
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`:

Figure 10.18: Even shorter way to kill the process

```
kill -9 14302
```

Or if you want to get all verbose:

Figure 10.19: A more verbose killer

```
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 that 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:

Figure 10.20: Examining exit codes

```
~ $ 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[15]. 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) 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:

Figure 10.21: Using ampersands to chain commands together

```
~ $ 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).

Figure 10.22: Using || to execute the first and possibly the second command

[15] http://mazack.org/unix/errno.php

```
~ $ ls foo || ls bar
foo
~ $ echo $?
0
```

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

Figure 10.23: The second command won't execute if the first fails

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

Figure 10.24: One more example with ||

```
~ $ ls bar || ls foo
ls: cannot access bar: No such file or directory
foo
~ $ echo $?
0
```

Note: There is actually a `true`[16] 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`[17] command to "do nothing, unsuccessfully!"

Figure 10.25: true and false commands

[16] http://linux.die.net/man/1/true
[17] http://linux.die.net/man/1/false

```
~ $ 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!

A

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

- **`$?`**[1] - the exit code for the last command or program executed.
- **`$PATH`**[2] - the execution search path.

Conditional Execution

See "logical operators."[3]

- **`&&`** - execute the second command only if the first command succeeds.

[1] http://linux.die.net/abs-guide/exit-status.html
[2] http://linux.die.net/Bash-Beginners-Guide/sect_03_02.html
[3] http://linux.die.net/abs-guide/ops.html

- **||** - execute the secon command even if the first command fails.

Redirection

See ”I/O Redirection.”[4]

- **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

- **~**[5] - 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.[6]

- **/etc** - configuration files location.
- **/home** - ”home” or user profile directories.
- **/proc** - system run-time information.

[4] http://linux.die.net/abs-guide/io-redirection.html
[5] http://linux.die.net/Bash-Beginners-Guide/sect_03_04.html
[6] http://linux.die.net/abs-guide/systemdirs.html

- **/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**[7] - compress and uncompress files and directories using the 7-zip algorithm.
- **apropos**[8] - search for help on commands by pattern.
- **awk**[9] - language for processing streams of data.
- **bash**[10] - the Bourne-again shell.
- **bzip2**[11] - compress and uncompress files using the `bzip2` algorithm.
- **cat**[12] - concatenate the input files to `stdout`.
- **cd**[13] - change the current directory.
- **chgrp**[14] - change the primary group of a file or directory.
- **chmod**[15] - change the permissions (mode bits) of a file or directory.
- **chown**[16] - change the owner of a file or directory.
- **cp**[17] - copy files or directories.
- **crontab**[18] - display or edit tasks to be run by `cron`.

[7] http://linux.die.net/man/1/7z
[8] http://linux.die.net/man/1/apropos
[9] http://linux.die.net/man/1/awk
[10] http://linux.die.net/man/1/bash
[11] http://linux.die.net/man/1/bzip2
[12] http://linux.die.net/man/1/cat
[13] http://linux.die.net/man/1/cd
[14] http://linux.die.net/man/1/chgrp
[15] http://linux.die.net/man/1/chmod
[16] http://linux.die.net/man/1/chown
[17] http://linux.die.net/man/1/cp
[18] http://linux.die.net/man/1/crontab

- **curl**[19] - download files from the internet.
- **df**[20] - show space utilization by file system.
- **diff**[21] - show the differences between files.
- **dig**[22] - look up DNS info on an address.
- **dpkg**[23] - package manager for Debian flavors.
- **echo**[24] - display passed parameters to `stdout`.
- **email**[25] - send email.
- **false**[26] - do nothing, unsuccessfully.
- **file**[27] - give best guess as to type of file.
- **find**[28] - find files based on various conditions and execute actions against the results.
- **grep**[29] - search for a pattern (regular expression) in files.
- **help**[30] - help for built-in commands in `bash`.
- **info**[31] - an alternative for `man`, especially for GNU programs. Remember `q` quits.
- **less**[32] - display the file one page at a time on `stdout`.
- **ln**[33] - create hard or soft (shortcut) links.
- **ls**[34] - list directory contents.
- **man**[35] - display manual pages. Remember `q` quits.

[19] http://linux.die.net/man/1/curl
[20] http://linux.die.net/man/1/df
[21] http://linux.die.net/man/1/diff
[22] http://linux.die.net/man/1/dig
[23] http://linux.die.net/man/1/dpkg
[24] http://linux.die.net/man/1/echo
[25] http://linux.die.net/man/1/email
[26] http://linux.die.net/man/1/false
[27] http://linux.die.net/man/1/file
[28] http://linux.die.net/man/1/find
[29] http://linux.die.net/man/1/grep
[30] http://linux.die.net/man/1/help
[31] http://linux.die.net/man/1/info
[32] http://linux.die.net/man/1/less
[33] http://linux.die.net/man/1/ln
[34] http://linux.die.net/man/1/ls
[35] http://linux.die.net/man/1/man

- **mkdir**[36] - make a new directory.
- **mutt**[37] - email client.
- **mv**[38] - move files or directories.
- **pine**[39] - email client.
- **ps**[40] - list running processes.
- **pwd**[41] - print the current (working) directory name.
- **rename**[42] - rename files in more complex ways than `mv` can.
- **rm**[43] - delete (remove) files or directories.
- **set**[44] - set an environment variable, or display all environment variables.
- **sort**[45] - sort `stdin` or a file to `stdout`.
- **ssh**[46] - secure shell termina progam and protocol.
- **tail**[47] - display the last lines of a file.
- **tar**[48] - ”tape archive”, a way to combine directories into a single flat file.
- **tee**[49] - write to a file and `stdout` at the same time.
- **telnet**[50] - ancient terminal program and protocol.
- **touch**[51] - create an empty file or change the last-modified time of an existing file.
- **true**[52] - do nothing, successfully.

[36] http://linux.die.net/man/1/mkdir
[37] http://linux.die.net/man/1/mutt
[38] http://linux.die.net/man/1/mv
[39] http://linux.die.net/man/1/pine
[40] http://linux.die.net/man/1/ps
[41] http://linux.die.net/man/1/pwd
[42] http://linux.die.net/man/1/rename
[43] http://linux.die.net/man/1/rm
[44] http://linux.die.net/man/1/set
[45] http://linux.die.net/man/1/sort
[46] http://linux.die.net/man/1/ssh
[47] http://linux.die.net/man/1/tail
[48] http://linux.die.net/man/1/tar
[49] http://linux.die.net/man/1/tee
[50] http://linux.die.net/man/1/telnet
[51] http://linux.die.net/man/1/touch
[52] http://linux.die.net/man/1/true

- **uname**[53] - print system info.
- **unzip**[54] - uncompress `.zip` files.
- **vi**[55] - ”visual” editor, a file editor.
- **wget**[56] - download files from the internet.
- **whoami**[57] - the answer to life’s most existential question.
- **whois**[58] - look up DNS ownership info on an address.
- **zip**[59] - 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**[60] - package manager for Debian flavors.
- **aptitude**[61] - package manager for Debian flavors.
- **cron**[62] - system for running ”scheduled tasks.”
- **dmesg**[63] - display kernel log messages.
- **ifconfig**[64] - display network (interface) configuration.
- **mount**[65] - mount a file system to a specific location.
- **ping**[66] - test for network connectivity to an IP address.

[53] http://linux.die.net/man/1/uname
[54] http://linux.die.net/man/1/unzip
[55] http://linux.die.net/man/1/vi
[56] http://linux.die.net/man/1/wget
[57] http://linux.die.net/man/1/whoami
[58] http://linux.die.net/man/1/whois
[59] http://linux.die.net/man/1/zip
[60] http://linux.die.net/man/8/apt-get
[61] http://linux.die.net/man/8/aptitude
[62] http://linux.die.net/man/8/cron
[63] http://linux.die.net/man/8/dmesg
[64] http://linux.die.net/man/8/ifconfig
[65] http://linux.die.net/man/8/mount
[66] http://linux.die.net/man/8/ping

- **reboot**[67] - restart the system.
- **rpm**[68] - package manager for Fedora flavors.
- **shutdown**[69] - shutdown or restart the system.
- **sudo**[70] - execute a command with elevated privileges.
- **traceroute**[71] - trace the route to an IP address.

Examples

The following are meant to be simple, mostly ”one-liner” type samples to reinforce the concepts here and continue to show you ”the UNIX philosophy” of approaching solutions in multiple small steps.

Keep It Simple, Stupid

Here’s a good example. During the debugging of this book I kept having problems with internal links to other parts of the generated PDF not working. Some did, some didn’t. And they ***all*** worked in epub and HTML outputs.

I had a suspicion it was because I was wrapping links from one line to the next in Markdown (trying to keep below a certain column count), so I wanted to find all lines that had an opening square bracket but ***not*** a closing one, e.g., I wanted to catch the first line in the following:

Figure A.1: Some Markdown

```
See [Important System
Directories.](http://linux.die.net/abs-guide/systemdirs.html)
```

Now you could spend a long time with regular expressions trying to figure out how to do negative matching on that closing `]`. Good luck!

Or you could do something as simple as this:

[67] http://linux.die.net/man/8/reboot
[68] http://linux.die.net/man/8/rpm
[69] http://linux.die.net/man/8/shutdown
[70] http://linux.die.net/man/8/sudo
[71] http://linux.die.net/man/8/traceroute

Figure A.2: Searching through the Markdown for mismatched brackets

```
$ grep '\[' *.md | grep -v ']'
Step01.md: (( expression ))           if COMMANDS; then COMMANDS; [ elif C>
Step01.md: :                          kill [-s sigspec | -n signum | -sigs>
Step04.md:./FileCheckers/otschecker:  [ $rc != 0 -a "$pdfinfo" != "Comma...
Step04.md:./FileCheckers/pdfpwdchecker: if [ $rc != 0 -a "$pdfinfo" = "C...
```

What makes this simple? Finding `[` with the first `grep` and then simply piping it to a second `grep` and inverting the match logic (`-v`) on `]`.

Chain Gangs

Remembering that `&&` only executes the next command if the prior one is successful, we can do things like set up a sample directory and (empty) files for playing around with files and directories in one fell swoop:

Figure A.3: Make a bunch of files and directories at once

```
~ $ mkdir -p /tmp/foo/d && cd /tmp/foo && touch a b c d/e
~ $ ls
a  b  c  d
```

That is roughly equivalent to:

Figure A.4: Make a bunch of files the long way

```
~ $ cd /tmp
~ $ mkdir -p foo
~ $ cd foo
~ $ mkdir -p d
~ $ touch a b c d/e
~ $ ls
a  b  c  d
```

Simple Scripts

I said I wasn't going to cover scripting, especially logical constructs like `if/fi`. But simple scripts that just "do things" in a certain order are within scope, and the following, which installs `freerdp`[72], is a good example of simply taking the guesswork out of doing something repetitive across multiple machines. I keep this `installrdp` script in Dropbox so I can run it quickly and easily on any new machine I set up (once I get Dropbox set up on the machine!)

Figure A.5: A simple install script

```
#!/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 libavutil-dev libavcodec-dev \
  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
```

You should be able to understand all of the above now, or know where to look to figure it out. The only nuance we may not have covered is that at the shell prompt and in scripts both you can put a `\` at the end of a line and it will "escape" the newline (`\r`) so you can continue the same command on the next line. This is useful because some interactive terminals don't "wrap" well, and it makes more readable script files, too.

[72] https://github.com/freerdp/freerdp

And yes, in the section on package management I talked about the dangers of installing packages directly from source. In this case, though, `freerdp` in the Mint repositories is lagging far enough behind the new RDP protocol version 8 support that I want to use the latest and greatest `freerdp` from GitHub for performance reasons. But now it's up to me to track and update the software (if I care).

C

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`[1] results):

- **Cygwin**[2] - `2.2.1(0.289/5/3) 2015-08-20 11:42`
- **Debian**[3] - `3.2.0-4-amd64 #1 SMP Debian 3.2.65-1+deb7u2`
- **FreeBSD**[4] - `7.3-RELEASE-p2 FreeBSD 7.3-RELEASE-p2 #0: Tue Nov 4 22:08:52 EST 2014`
- **Linux Mint**[5] - `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 `pandoc`-flavored Markdown[6] using `vi`[7] and Visual Studio Code[8], among others.

[1] http://linux.die.net/man/1/uname
[2] https://cygwin.com/
[3] http://www.debian.org/
[4] http://www.freebsd.org/
[5] http://linuxmint.com/
[6] http://pandoc.org/README.html#pandocs-markdown
[7] http://linux.die.net/man/1/vi
[8] https://github.com/Microsoft/vscode

Output produced using `pandoc`[9], TeX Live[10], `pdflatex`[11], `make`[12], originally based on the @evangoer's pandoc ebook template[13] but long since modified so don't blame him.

Source code control is provided by `git`[14]. You can view the files used to create this book[15] on GitHub.

The fonts used are DejaVu[16] Serif for the body text, DejaVu Sans for headers, and Ubuntu Mono[17] for code (because it is nicely condensed).

The cover photo is of our dog, Merv, who is reminding you, "Don't panic!" Photo by Gloria Anderson, used with permission.

About the Author

Jim is son to Barb and Lou; husband to Leslie; father to Meghann (and Jeremy), Morgann, Erin, Gloria and Jon; grandfather to Ryan, Lindsay, Logan and Hannah; and alpha wolf to Merv. He has been "in computers" since 1980. His hobbies include reading, running, hiking, climbing and apparently writing books.

[9] http://pandoc.org/
[10] http://www.tug.org/texlive/
[11] http://linux.die.net/man/1/pdflatex
[12] http://linux.die.net/man/1/make
[13] https://github.com/evangoer/pandoc-ebook-template
[14] http://linux.die.net/man/1/git
[15] https://github.com/dullroar/ten-steps-to-linux-survival
[16] https://en.wikipedia.org/wiki/DejaVu_fonts
[17] https://en.wikipedia.org/wiki/Ubuntu_%28typeface%29

Index