UNIX and Linux

Jeremy Doolin

Contents

Chapter 0

Introduction

0.1 What is UNIX?

Welcome to the world of UNIX. But what is it?

I'll start by saying what UNIX *used* to be. At one time, UNIX was a single operating system developed by Bell Labs, the research division at AT&T. It was ahead of its time, popular and influential. But today, the official "UNIX Operating System" no longer exists. UNIX has survived as a very large family of operating systems that have the same basic design as the original UNIX. While the name "UNIX" is still technically trademarked, it is used in the technology world as a catch-all term to describe any of those modern operating systems that carry on its design. We'll talk more about UNIX history later in this chapter.

Today there are many operating systems that could be called a "UNIX system"[1]. We'll be talking about some of these systems, but the most common representative today is Linux. But Linux isn't even really an operating system, it's just a kernel (the operating system core). So why do we refer to Linux as if it were an operating system? Well mostly because it's easier that way. But we'll get into those details later.

[1]Remember the scene in *Jurassic Park* when Lex Murphy sits down to the computer and says "It's UNIX system, I know this."? It really was a UNIX system. It was the "Irix" system by Silicon Graphics

0.2 Why Learn UNIX?

If you've ever seen the command prompt for a UNIX system, it seems like some archaic, dated software used only by hardcore computer geeks or old sysadmins with beards. Where is the Graphical User Interface? You certainly can't play many of the latest PC games on UNIX machines[2]. This is a world dominated by Microsoft. Windows, Office, Exchange, DirectX and ASP.NET are products by one of the biggest software companies in the world. Windows absolutely dominates the desktop market.

So why bother with UNIX?

To put it simply, UNIX is *everywhere*. UNIX had some days of glory in the late 70's to the late 80's as a popular operating system for mainframe systems, but with the rise of the microcomputer and home PC market, software giant Microsoft began to get a foothold across a broad spectrum of systems, not just the desktop. UNIX fell out of favor for a while during a time when Windows NT was becoming a more attractive choice as a server system for more inexpensive hardware. UNIX and its systems were rather expensive and Microsoft was seen as a more cost-effective and viable option, especially for smaller businesses.

But UNIX has made a big comeback. It's no longer just a hobbyist system, or something old sysadmins use. UNIX skills are no longer just something a few people here and there know or need to use. It is so common now and growing so quickly, that it is an obligation for anyone seeking a career in IT, software development or technology. It is a skill in demand and one that can give you an edge against the competition in the job market.

In addition to being a very useful and valuable skill, UNIX also offers a computer user more power than alternative successful operating systems. It gives you the power to do very stupid and bad things to your computer, but at the same time gives you the power to do brilliant and excellent things. By learning UNIX, you will gain a much deeper understanding of the inner workings of a computer and of operating systems in general.

Finally, many UNIX systems are known for their stability and security[3]. One of the UNIX system administrator's bragging rights is *uptime*: how

[2]Actually, in many cases you already are.

[3]Not all UNIX systems are inherently secure out of the box. UNIX systems must still be updated, patched and configured properly for best security

long a system remains available without a reboot. UNIX machines are known for uptimes as long as years or even a decade. High uptime and availability is very attractive for system administrators of high availability systems and networks.

0.3 Where is UNIX Used?

Where does one even start? While Microsoft dominates the desktop, UNIX dominates just about every other computing domain.

First and foremost, UNIX is the backbone and central nervous system of the Internet and always has been. TCP/IP networking and the Internet itself was born on UNIX machines and continues to be used by many of the biggest internet companies and servers worldwide. Google, Wikipedia and Amazon use Linux for most of their services. Yahoo! and Netflix use FreeBSD. DNS, WWW, email, chat, streaming video and audio, web hosting, file hosting, cloud storage and a wide variety of other services run primarily on Linux, BSD and other UNIX systems.

Scientific institutions such as NASA and CERN use Linux. The laptops on the International Space Station were recently converted to Ubuntu Linux. CERN uses Linux to run the Large Hadron Collider and 20,000 of their internal servers. Companies like IBM, Novell, Peugeot, the New York and London Stock Exchanges all use Linux. Most (by far) of the world's top 500 fastest supercomputers are Linux clusters. UNIX is on many computer desktops, not just from computer geeks who prefer Linux or FreeBSD, but in the form of Apple's own macOS (formerly OSX). Yes, macOS is UNIX system.

Apple's iPhone is a UNIX system (based on OSX) and Android runs a modified version of Linux. These two factors alone make UNIX the most successful operating system in history, at least in terms of numbers of users or installations.

UNIX and Linux can be found in embedded systems, robotics, control systems, simulators, video game systems (the PS4 OS is based on FreeBSD), space probes and satellites, wireless routers, network appliances, firewalls, set top boxes and DVRs, power generation systems, 911 dispatch, navigation and radar systems, HVAC systems, casino gaming systems, medical and surgical systems and even cars and automobiles. UNIX is also popular for softare development on integrated systems and small boad computers.

So while Windows is undisputed champion of the home, office and gamer arena, UNIX maintains a firm hold in almost every other computing domain. This is why UNIX skills are practically a necessity for today's technology professionals.

0.4 A Bit of UNIX History

To best understand the role of UNIX today, I believe it is best to understand how it got here in the first place. The history of UNIX also serves to explain some other important topics for today's technology world, including Free and Open Source Software and versus proprietary and commercial software and licensing. These are important to understand for system administrators.

0.4.1 Computing in the Mid-60's

In the mid 50's into the mid 60's the computing world was a very different place. There were many vendors building very expensive machines. IBM, Digital Equipment Corporation (DEC), Honeywell, Interdata, Data General, Apollo, Prime and others were building desk and refrigerator sized computers. Each company designed its own hardware and CPU architectures, which meant that software from one company would not operate on another company's hardware. In fact, this was true even for different computer models produced by the same company.

In the 60's, operating systems were new and numerous. Every company had a different OS, or multiple OS'es, that again did not even work for all models made by the same company. For example, DEC's OS/8 operating system for their PDP-8 would not work on PDP-7 or PDP-10 models.

What this meant was that if a company purchased a computer (a significant financial investment in those days, often in the several tens of thousands to hundreds of thousands of dollars) and a few years later it was outdated, they had to re-learn a completely new architecture and operating system and enter all of their data once again. It required enormous amounts of time and resources (and thus money).

Another problem was that these computers ran primarily batch systems that allowed for only one program to run at a time, meaning only one person could use the system at any given moment. At universities this led to lines of programmers waiting hours for their turn at the computer and terribly inefficient use of expensive hardware.

In the mid 60's, three groups combined efforts to fix these issues. Bell Labs of AT&T, the Massachusetts Institute of Technolog (MIT) and General Electric (GE) set out to create a **timesharing** operating system called **MULTICS**. Timesharing basically means the system would allow multiple individuals to use the computer at the same time using separate terminals.

By 1969, some of the MULTICS developers began to feel that the project was getting out of hand. Ken Thompson of Bell Labs described it as "overdesigned, overbuilt and overeverything". This would lead Bell Labs to pull out of the MULTICS project.

MULTICS would go on to see some success. There were around 80 installations worldwide, with the last one being shutdown in the year 2000. The source code of MULTICS is now open source.

0.4.2 The Birth of UNIX

Ken Thompson and Dennis Ritchie of Bell Labs decided to take some of the good ideas from the MULTICS project and build a new operating system that was smaller, simpler and able to run on the hardware common at the time. In the early days of the project, it was suggested they name it UNICS as a pun on MULTICS. The name stuck but the spelling was eventually changed to UNIX.

Thompson and Ritchie built UNIX for the very popular PDP-11 system by Digital Equipment Corporation. This made UNIX a popular choice for any university or business that owned one. Dennis Ritchie also developed the C programming language at the same time, for the express purpose of developing UNIX into a portable operating system, meaning it could be built and used on multiple hardware platforms. The C language would go on to be arguably the most popular and important programming language of all time and it is still very heavily used today, especially for development of operating systems.

0.4.3 UNIX Branches Out

Those who received license to use UNIX also received it's source code as part of the license. This meant that anyone who had the license to use it could also modify and add to it. The University of California, Berkely was very interested in UNIX and began extending UNIX, creating a set of patches and additional software that significantly improved the system. AT&T even added some of these back to their base system. Berkeley's set of patches and add-ons were so well known and highly regarded that it became known as the Berkely Software Distribution, or BSD.

The BSD project was a very early branch of the UNIX system and perhaps the most successful in the 80's. Several other vendors developed their own UNIX systems as well, including Microsoft's Xenix, Sun Microsystem's

SunOS and eventually IBM's AIX, Hewlett Packard's HP-UX, DEC's Ultrix and Tru64 UNIX, Sun's Solaris and SCO UNIX. Some of these are based on AT&T's UNIX, while others were based on BSD.

By this time UNIX had branched into many competing and expensive proprietary products. In 1988, AT&T released its last version of the original UNIX system. From that point on, AT&T UNIX was gone and replaced by a dozen competing products, all of which were expensive and required expensive hardware to run.

0.4.4 The Home PC Revolution

In the early 70's, Intel built the very first **microprocessor**, the Intel 4004, a 4-bit CPU. While this chip would not be destined for any popular home PCs, it ushered in a new era of inexpensive microprocessor chips. Intel would later follow the 4004 with the 8080, an 8-bit microprocessor that would be the CPU for the MITS Altair 8800 microcomputer. The Altair 8800 is arguably the machine that begain the microcomputer and thus, the home PC revolution.

Other technology companies began to manufacture 8-bit microprocessors, such as Motorola with the 6800, MOS Technology's 6502, Zilog's Z80 and Fairchild's F8. Some of these would be the basis for many new home PCs that would far outsell the Altair 8800. The Apple and Apple2, Commodore PET, VIC-20 and 64, Atari 400/800 series and Tandy/Radioshack's TRS-80 were the primary competitors in the late 70's and early 80's home PC industry. The computers themselves were inexpensive, had little memory (the 64 kilobytes in a Commodore 64 was considered impressive) and not nearly as powerful as the industrial competitors like the VAX (a popular UNIX platform) or even the Motorola 68k.

Because of this lack of power, UNIX developers were not attracted to these platforms. They simply weren't capable of running an operating system as large and complex as UNIX. But one company was quite willing to work with this hardware: Micro-Soft (as it was originally written). Microsoft wrote the BASIC programming language interpreter that was the default system for nearly all of these early 8-bit home computers, starting with the MITS Altair 8800 itself. This was how Microsoft got their start.

In time the 8-bit PC would give way to the 16-bit PC, primarily with Intel's new 8086 architecture. IBM, Intel and Microsoft then began a partnership that would change the home PC landscape permanently. This partnership resulted in the IBM PC, based on Intel's 8088 (a more cost effective 8086) and Microsoft's new MS-DOS, an operating system that they had purchased from another developer [4].

The business world and home users now had an inexpensive option for home PCs. By the time Intel had released the 32-bit 386 CPU, home computer enthusiasts were building their own computers from affordable and easily obtainable parts, just as they do today. But it was not a platform that interested proprietary UNIX developers at the time. So basically, if you owned a 386 based computer (and there was a good chance you did, unless you used a Macintosh or an Amiga) and you were a UNIX enthusiast, you were out of luck.

Interesting to note here was that Apple did develop and release their own UNIX called A/UX for a few higher end models of Macintosh, but it was not well known or ultimately successful, being discontinued in 1995.

0.4.5 Enter GNU and Linux

In 1983, Richard Stallman of the MIT AI laboratory announced the development of a new UNIX system, completely independent from any proprietary product and developed from scratch. Stallman believed that software and its users should be free: free of cost and free to modify, extend, improve and share. Stallman dubbed the project GNU (a recursive acronym that stands for "GNU's Not UNIX").

Stallman's GNU Project successfully wrote many of their own UNIX utilities and programs that could replace the proprietary UNIX versions, yet had not developed a complete operating system. It was missing a kernel.

About Kernels

Before going further, let's talk about kernels, as it will help understand a few things later. The **kernel** is essentially the core of any operating system. It acts as the main software interface to the computer's hardware, controls memory management, process management, file and storage management, device allocation and management, network management, user and access management, and much more. If you've taken a college operating systems class, these terms will be quite familiar and you may have an understanding of what is involved with these various tasks. Needless to say, the kernal is a vital component of any operating system. Open Source operating systems even allow you to customize the kernel in many ways and build your own from the source[5].

[4]The original name of MS-DOS was "QDOS", which stood for Quick and Dirty Operating System.

[5]Building custom kernels is a UNIX right of passage

As important as the kernel is, it is useless on its own. In order for the kernel to run programs, manage processes, allocate memory and make our devices useful, we need a way of interacting with the kernel and telling it what to do. We need a way to create users and allow them to log in, a way for the user to create files and folders, make copies of them, rename them and delete them. We need programs to print documents, scan photos and filter text. We need utilities to partition and format storage devices and set up network connections. All of these programs are known as **utilities**, and sometimes referred to as **userland**, since this software requires the interaction of users.

Operating systems need both a kernel and utilities in order to be useful, and this playes an important role in the story of the GNU Project. It was tremendously successful building new, free and open source UNIX utilities, but had not yet developed a usable kernel.

0.4.6 Linux Completes the System

In the early 90's there were projects under way to port the very popular BSD UNIX to the highly popular Intel 386 platform that was dominating the home and office market. By this time, BSD UNIX had very little of AT&T's original code and was very nearly its own completely separate UNIX system and efforts were under way to port it to Intel's popular platform.

Meanwhile, another UNIX-like project, MINIX, was being developed by Andrew Tannenbaum as an educational system. It was its own standalone operating system created by Tannenbaum and his team. However it was not intended for use outside of the academic context. A Finnish student named **Linus Torvalds** was inspired by MINIX design and frustrated by the "educational use only" MINIX license. Like many others at the time, Torvalds was also frustrated at the lack of a good, general purpose UNIX system for the Intel 386 system. Torvalds decided to write his own operating system kernel for the popular platform.

Torvalds called his kernel "Linux" and eventually paired it with the UNIX utilities from the incomplete GNU system. This pairing of the Linux kernel by Torvalds and the GNU utilites by Stallman and the Free Software Foundation, would become what we collectively know of as "Linux", though some would insist it be called "GNU/Linux". At last, in 1991, there was a free, open source, general purpose UNIX system available for the Intel 386 platform.

An interesting side note: Torvalds has said that had the 386 port of BSD been released earlier, Linux may never have happened, since his primary motivation had been to have a UNIX system on his 386, which 386BSD would have provided.

Another interesting side note: The GNU Project very nearly adopted a BSD kernel instead of writing their own. If they had, the world could have been a different place.

0.4.7 BSD Lives On

The 386BSD project did succeed, only it was just a little too late. Linux had already been released to the world first.

Despite the meteoric success of Linux, BSD UNIX lived on and kept progressing. 386BSD forked into two separate projects: FreeBSD and NetBSD. DragonflyBSD forked from FreeBSD in 2003. NetBSD itself forked again, which resulted in OpenBSD. FreeBSD, NetBSD, OpenBSD and DragonflyBSD are all free and open source, modern BSD UNIX systems.

Even modern day macOS is part FreeBSD.

FreeBSD

FreeBSD is the most popular of the four major BSD systems available today. Its focus is advanced features, high performance, especially with networking, and the support of server, desktop and embedded systems.

Among FreeBSD's more interesting features are lightweight virtual machines called **jails**, the advanced storage filesystem **ZFS**, Linux emulation, advanced firewalls, and a built-in virtual machine system called **bhyve**.

NetBSD

NetBSD targets portability, which means it is designed so that it is easily built for many different hardware platforms and CPUS. The running joke is that NetBSD will even run on a toaster. NetBSD will run on major platforms like 32 and 64 bit Intel (x86/amd64), ARM (such as the Raspberry Pi), sparc, PowerPC, MIPS, alpha, the dreamcast, and many more.

OpenBSD

The focus of OpenBSD is security.

0.5 What is the difference between UNIX and Linux?

0.5.1 First, where is UNIX now?

UNIX, the operating system, doesn't really exist any more. You can't go to unix.com and purchase a license or visit UNIX headquarters. AT&T would eventually sell the rights to UNIX to a company called Novell, previously a big player in the network operating system business. This was effectively the end of the original UNIX.

Novell merged UNIX with their Netware software to create UNIXWare. Eventually Novell sold the rights to the Santa Cruz Operation (SCO), who already had their own version of UNIX.

SCO is the current rights holder to UNIX and they have two products based on AT&T UNIX, UNIXWare and OpenServer.

The closest OS you can easily obtain and install on modern commodity hardware would be FreeBSD.

Other systems, regardless of their lineage, may also claim to be an "official UNIX system", but only by means of the Single UNIX Specification.

0.5.2 The Single UNIX Specification and POSIX

During the major branching of UNIX in the 80's there were some who attempted to establish standards that would define what was required for an operating system to have the right to call itself UNIX. This resulted in two standards: the Single UNIX Specification and POSIX.

Any operating system may be POSIX compliant (even Windows). It is simply a list of technical requirements that an OS must meet, such that developers may know that they can expect certain things on any POSIX operating system. However, to make it official, the developers of a system must pay a fee, which for some can be rather expensive. Even so, a POSIX system could theoretically behave very little like classic UNIX.

The Single UNIX Specification is a similar concept, but covers more area than POSIX. It is the same in that the owners of an operating system must pay a fee to be an official "UNIX". Some examples are Sun's Solaris, Apple's OSX, Silicon Graphics' Irix, IBM's AIX and HP's HP-UX. Linux, the BSD systems and other open source systems typically do not bother with paying to be an official UNIX system as there is very little reason to do so.

0.5.3 So What About Linux? Is it UNIX?

Linux is unique. It has no connection to the original AT&T UNIX the way the BSD systems do. Neither is it certified as POSIX compliant or under the Single UNIX Specification. It was an independently developed kernel, and only a kernel, inspired by the design and implementation of UNIX. Since Linux isn't a complete OS as macOS or FreeBSD are, it technically doesn't even qualify to be considered under either standard. This means it would be up to the Linux Distributions, which provide all the other software that uses the kernel, to become POSIX or SUS certified. The problem is that there is little motivation to pay the high fee just for a certification that isn't really needed.

So to put it simply, no, Linux is not UNIX. It's not even *a* UNIX system. The best way to think of Linux, is that when it is paired with the GNU programs and utilities it becomes a UNIX system in spirit and design, just not in any official capacity. Or, you could think of Linux as a UNIX clone.

0.6 Free and Open Source Software

Linux became a usable and stable UNIX-like system, available at no cost and usable on very inexpensive hardware. GNU and Linux had started a powerful movement. Open Source software began to grow and mature. Hundreds, thousands and eventually tens of thousands of developers worldwide began to combine efforts to improve Linux, the BSD systems and an ever growing list of open source projects.

Programmers the world over began releasing their projects as open source, available to all for free and allowing anyone to help improve their software. These projects included web browsers, multimedia applications, encryption algorithms, filesystems and drivers, programming languages like PHP, Perl and Python, programming language compilers, emulators, complex server software such as the Apache web server, MySQL database server and complete desktop environments.

This also enabled engineers, designers and researchers to use Linux or a BSD as a development and testing platform. This means that many of the technology world's cutting edge technologies are being developed on Linux or BSD *first*. It also means they are often adopted in the Linux system before other proprietary systems.

0.6.1 The GNU Public License

GNU and Linux are released under a special license called the GNU Public License (or GPL for short). The GPL essentially states that software released under its license must be open source, and that if the source or software is used in another product, that it must be credited and the software must remain open source and also be released under the GPL.

This does not mean that you are not allowed to sell GPL licensed software. For example, if you were to build your own customized Linux distribution, you would be free to sell it to customers at any price you wish. The condition, however, is that you must also supply the original source code and any derivative works must be under the GPL.

0.6.2 BSD License

The BSD license used by FreeBSD (and other open source BSD's) is also used by a variety of non-operating system projects. It differs from the GPL in that BSD licensed code may be modified and redistributed in any way. It may continue to be open source, or someone may develop a derivative work under a completely proprietary license. For example, you could modify the FreeBSD operating system and redistribute and sell it as your own. In a way, the BSD license is even more free than the GNU Public License.

Summary

UNIX has a long history dating back to 1969. It brought many innovations and benefits, such as portability to other systems, a powerful new programming language, multi-user and timesharing capabilities, pipes and more. It was developed into a stable and powerful system used by universities, research facilities, governments and businesses.

UNIX did not follow the home and commodity PC revolution until the early 90's, allowing companies like Microsoft to gain a foothold in a market segment most visible to the average user. While UNIX has never held the home market, it remained a mainstay at Universities and some larger businesses, even while Microsoft, Novell and other competitors gained popularity among server systems. At this time it seemed UNIX was a system on its way to obsolescence.

With the rise of GNU, Linux, and the open source movement, UNIX regained its popularity in areas where it had previously lost ground and became prominent in new areas such as mobile devices and supercomputing. Even desktops are being offered with Linux or UNIX systems, including all of Apple's OSX products.

Knowledge of UNIX and Linux systems isn't just an option for the hardcore computer geeks and tinkerers of world. It is now an important part of technology and is indispensable for network and system administrators, as well as those serious about software development. While Microsoft holds the desktop market, almost everything else runs UNIX.

Chapter 1

Installing Unix/Linux

1.1 Requirements

First you need a computer. Any computer.

With few exceptions, there is a Unix or Linux distribution that will work on any computer you can scrape together or purchase. You can install it on anything from old Intel 386 machines that originally ran DOS to a modern laptop or server system. This is one of the great things about Unix and Linux systems: the ability to use it on hardware that most people would consider worthless.

The Raspberry Pi is another great option for someone interested in learning about Linux. At about 35 dollars, it is inexpensive, runs the OS from an SD card, is lightweight, and surprisingly capable, with current versions supporting built-in Bluetooth and Wifi[1]. The Pi also has built-in camera and LCD screen connectors as well as General Purpose IO pins that allow a programmer to turn the Pi into a sort of microcontroller. The possibilities are limited only by your imagination.

A third option is to dual boot your current laptop or desktop so that when you turn on the machine, you can select either Windows or Linux. This will be discussed in the Dual Booting section.

Finally, and perhaps most flexible of all, you can install Linux or other UNIX systems on a virtual machine using software such as VMWare Player

[1]Current Raspberry Pi as of this writing is the Raspberry Pi 3

or VirtualBox. This allows you to test, experiment and learn without having to install the OS to your hard disk. It also allows you to test various distributions.

This means that there is no reason not to have a Linux or BSD machine at your disposal for learning and practicing. Old hardware can be had for free or nearly free and this makes it a perfect candidate for learning, and the most effective way to learn Unix is to use it on a regular basis.

1.2 Linux Distributions

Unlike Apple or Microsoft, there is no Linux corporation. There is, however, a group of programmers who are in control of what goes into the Linux kernel, a benevolent dictator who has the final say, and another group of programmers that are in charge of the GNU utilities that are commonly paired with the Linux kernel. Literally anyone (yes, even you) can contribute code, ideas or innovations, but there is a core group that approves these inclusions or changes. The same is true for Mozilla Firefox, Libre Office, the GIMP image editor, Apache web server, MySQL database, PHP programming language and a multitude of other open source projects. Each group maintains their own projects and the same is true for Linux, which is just the kernel.

1.2.1 What is a "kernel"?

The **kernel** is the core of an operating system that provides access to the computer's hardware. It is always running and all other software must work with the kernel. It handles memory management, process management and scheduling, file and storage management, network management, security, and numerous other core tasks. This is obviously a very important part of an operating system, yet it would be useless on its own.

Yet that's all Linux really is. It's just a kernel. Think of it like a high performance race car engine that has no wheels, chassis, tires, or steering mechanism. In order for a kernel to be of any use, it requires **utilities** and **applications**, such as those built by the GNU project.

Since Linux is only a kernel, that means **there is no "official" Linux operating system that you can download and install**. Anyone is free to build an operating system based on the Linux kernel, packaged with whichever utilities and applications they desire. They can design the system differently than other systems, focusing on different aspects or principles. A collection of the Linux kernel and accompanying third-party software is called a Linux **distribution**. The number of Linux distributions as of this writing probably numbers in the thousands. It's difficult to enumerate simply because literally *anyone* can create a Linux distribution and many do, including businesses, research facilities and hobbyists[2].

Distributions (or "distros") are usually designed with a particular purpose or focus in mind. Some are designed to be a stable server environment, while others are aimed at a flexible desktop system able to support as many different hardware configurations as possible. Some are designed to run on distributed supercomputing systems while others target very small devices with limited resources. Some distros are concerned with packaging only completely free software, free of any proprietary drivers or utilities while others aren't concerned about this fact.

Here is a review of some popular (and a few not so popular) Linux distributions.

1.2.2 Ubuntu

Ubuntu and its derivatives (Kubuntu, Xubuntu, Lubuntu and others) focus primarily on providing a desktop system that is easy to install and use. They also have a version that targets server systems and are even getting into the mobile device systems.

Ubuntu proved to be very popular and spawned a few derivatives. Kubuntu is the same distro but with the KDE Desktop environment as its default (as opposed to Ubuntu's Unity desktop). Xubuntu is the same but with the more lightweight XFCE desktop. Lubuntu is a more lightweight version that includes software that requires less RAM, processing power and hard disk space. It also supports older processors that the newer versions of Ubuntu do not. This makes Lubuntu a good option for older computers.

Ubuntu itself is derived from another popular distribution called Debian.

[2]Yes, even you can create your own Linux distribution

1.2.3 Debian

Debian is a fairly early distribution focused on providing a complete, stable opearating system consisting of free, open source software. Debian developed a remarkable software package system that allowed users to install software from the internet just by typing a simple command. Think of it as Ubuntu but without the extra systems and software specific to Ubuntu. Or you could think of Debian as the older, wiser and more stable father of the younger, more cutting edge and flashy (yet resource intensive) Ubuntu.

1.2.4 Raspbian

Raspbian is a special derivative of Debian that is tailor made to run on the Raspbery Pi computer. It contains the exact drivers for its hardware, has special configuration utilities and even has smaller, more efficient programs as well as software written specifically for the Raspberry Pi, including Minecraft Pi, a stripped down Minecraft that you are able to program yourself with Python.

Raspbian isn't the only Linux system available for the Raspberry Pi, nor even the only Unix system. There is a version of Fedora (see below), Arch (see below), Ubuntu and even FreeBSD.

1.2.5 Red Hat/Fedora

Red Hat was previously a very popular free distribution somewhat akin to what Ubuntu is now. However their focus has changed to providing and supporting enterprise level Linux installations with their new distribution called Red Hat Enterprise Linux (RHEL). RHEL is *not* available as a free download.

There are, however, two Red Hat based distributions that pick up where the former free Red Hat left off. **Fedora**, sponsored by Red Hat, provides users a free Red Hat based distribution for installation on desktops. Fedora basically occupies the role previously held be Red Hat Linux. **CentOS** aims to provide a free enterprise level Red Hat distribution for use on servers.

Red Hat has its own popular package system called RPM, and has an associated package manager called yum.

1.2.6 Arch Linux

Arch Linux takes a polar opposite direction from Ubuntu. While Ubuntu aims for ease of installation and use, Arch targets Linux and Unix enthusiasts who are more experienced and like to know the internal workings of the operating system. Arch aims for simplicity, efficiency and a minimalist environment. Arch also incorporates more bleeding edge software that other distributions that are more concerned about stability. To put it simply, Arch is not for the feint of heart.

1.2.7 Puppy Linux

Puppy is a unique distribution focused on being lightweight and easy to use. Puppy can be run entirely within RAM, so no hard disk is needed. It is common to run Puppy from CD, DVD or USB drives, though Puppy can also be installed to a hard disk. It is well known for running very well on much older hardware, such as Pentium II systems.

1.2.8 Damn Small Linux

DSL attempts to provide a Linux system *with a graphical interface* that is as small as possible (small enough to fit on a business card CDR). It's current install size is 50 megabytes and, like Puppy, can be run from USB drives, SD cards and other portable media.

1.2.9 tomsrtbt

tomsrtbt and other distributions like it take small Linux distributions to the extreme. tomsrtbt fits on a single, 1.44MB floppy disk. While it contains as many Unix/Linux utilities as possible, it's primary goal is data rescue and recovery utilities. It's just enough to fix installations or to grab critical data when a system will not boot from the hard disk.

1.2.10 Open SUSE

OpenSUSE aims to be a complete, general purpose operating system that is stable, easy to use and good for a variety of purposes such as server, desktop or laptop. This makes it very similar to Ubuntu, Debian and Fedora, but SUSE aims to provide server possibilities in the same distro. It also has its own utilities different from other distros that some users prefer. If it's one thing the Open Source community offers, it's choice. It is also known for its all-in-one configuration utility, YaST.

1.3 The BSD Systems

While Linux is only a kernel, there are other open source UNIX systems available that descend from the BSD UNIX systems.

1.3.1 FreeBSD

FreeBSD is **not** a Linux distribution. FreeBSD is a complete operating system, including its own kernel and core utilities all developed together. This is different from Linux where the kernel is developed independently from all other software, including the GNU utilities. While FreeBSD has its own core utilities, it is also possible to install many of thousands of non-FreeBSD software packages, such as Firefox, VLC, MPlayer and many others.

FreeBSD originates from the 4.3 BSD UNIX system developed at Berkeley, which itself was derived from the original AT&T Unix, making FreeBSD a direct descendent of the original AT&T UNIX.

FreeBSD aims for stability, efficiency and speed. It is a very popular server system (Yahoo! and Netflix both use FreeBSD) and many Unix enthusiasts use it on their desktop. It also runs well on older hardware.

1.3.2 NetBSD

NetBSD forked from the same project FreeBSD did. NetBSD's claim and purpose is to provide a modern Unix system to as many hardware platforms as possible. This includes Intel/x86, sparc, alpha, powerpc, MIPS, ARM and older platforms like the VAX.

1.3.3 OpenBSD

OpenBSD is a NetBSD fork focused entirely on security. Its claim to fame is that there have been only two remote exploits in the default install since its inception in 1996 (25 years as of this writing). It remains among the most secure operating systems available.

1.4 Download Install Disk Images

The first step to installation is to acquire the installation media. For most Linux and BSD systems this means downloading a CD/DVD or USB image from the internet and burning it to a disk or writing the image to a USB drive or SD card.

Before downloading, make note of what PC hardware you have. Most users will have a 64-bit Intel based system (either AMD or Intel CPU), or if it's an older system, perhaps a 32-bit system. If you have a computer with a 32 bit processor, you will need to download the 32-bit version, also often referred to as x86 or i386. If you have a 64 bit capable processor you will need the 64 bit version, also known as amd64. If you happen to be installing on a machine with an architecture other than Intel/AMD, such as an ARM CPU (Raspberry Pi), an old Macintosh or Sun workstation, make sure you get the version for your architecture (ppc, sparc, mips, etc).

The best resource for downloading the images is the official website for whichever system you intend to install. Here is a small list:

- **Ubuntu** - http://www.ubuntu.com/download
- **Fedora** - http://fedoraproject.org/get-fedora
- **CentOS** - http://wiki.centos.org/Download
- **Debian** - http://www.debian.org/CD/live/
- **Raspbian** - https://www.raspberrypi.org/downloads/raspbian/
- **Arch** - https://www.archlinux.org/download/
- **OpenSUSE** - http://software.opensuse.org/123/en
- **FreeBSD** - http://www.freebsd.org/where.html
- **NetBSD** - http://www.netbsd.org/releases/
- **OpenBSD** - http://www.openbsd.org/ftp.html

1.5 Create Install Media

After you have downloaded the images, you will need to create the install media. There are serveral possibilities for creating it.

1.5.1 Burning CDs/DVDs

This used to be the most common method, but is now more useful for older systems that cannot boot to a USB drive. You will download an ISO (.iso) image that must be burned directly to disk and not as a file. Your CD/DVD burning software will normally handle this properly.

1.5.2 USB Installers

Most systems now offer USB thumb drive installer systems. To create these you will need to follow instructions on the distribution website, but it usually consists of directly copying files to a thumb drive or writing an image similarly to burning a CD image. Occasionally you may need to download a CD image first, boot from it and use a utility to create a thumb drive installer.

Utilities such as Balena Etcher, Win32 Disk Imager or the 'dd' utility on UNIX systems can be used to write these installer images to the drives.

1.5.3 Floppy Images

On particularly old hardware it may be necessary to create boot floppy images. Any distribution you may be considering for this task will normally have a floppy image on the CD or DVD. This file must be written directly to the floppy using a utility such as 'dd' in Unix or rawrite.exe in Windows. There will be instructions on the website.

1.5.4 SD Card Images

The Raspberry Pi runs the OS from an SD card. To "install" a Raspberry Pi distribution, you must download the SD card image and write the image directly to the SD card. This is very much like burning an ISO to cd. You don't want to copy the image file into the SD card's folder structure, you want to copy the image *directly to* the SD card, using software like Win32DiskImager or the **dd** command in Linux/Unix or macOS.

OpenBSD has a more unusual process on the Raspberry Pi. (SERIAL/UEFI/SET TTY)

1.6 Boot PC From Install Media

Once you have the media created, you must boot from it. This means you need to either use a Boot Menu (often accessed by pressing an F key, such as F12) or configure the BIOS so that the boot order checks the CD/DVD, USB or floppy drives first before attempting to boot from the hard disk. If this is done properly, the computer will boot from the install media instead of the hard drive.

1.6.1 Live Install Systems

Many distributions now provide live systems for installing the operating system. When you boot from the media (such as a DVD) you can select to try the system before installing. This will take you into a live environment of the operating system that you can actually use. If things go well, there will be a link or menu item for installing the system from within the live environment.

It is also worth noting that these live CD and USB systems are **immeasurably useful** for tasks other than installing the OS. They can be used for PC diagnosis and troubleshooting, data recovery and many other miscellaneous tasks. Always be sure to keep a Linux live CD or thumb drive around for emergencies.

1.7 Typical Install Process

While every operating system's install process will be slightly different, most of them will have many things in common. For details, consult the installation manual for your operating system of choice.

1.7.1 Select Language

This one is pretty simple. The installer asks for a language to be used by the remainder of the install system. Because open source software is so popular worldwide, one of its most appealing aspects is that it supports many languages. In the old Red Hat days there was even an option for a language called "redneck".

1.7.2 Partition Disk(s)

Before an OS can be installed to a hard disk, the disk must be prepared. This involves creating a partition on the hard drive specifically for the OS and its files. It is definitely easier to dedicate an entire disk to the OS, but multi-booting is a very popular option as well.

On Linux and Unix systems, the process usually involves optionally deleting other partitions on the disk (if they exist) and creating new ones. When you create a new one you will specify the partition size (20 GB, for example), which filesystem to use and where to mount the disk.

The filesystem type will depend on the OS you are installing. Linux supports many filesystems, including ext2, ext3, ext4, btrfs, reiser4, XFS and others. The most recent ext system (currently ext4) is usually the most popular for Linux systems, though there are reasons for using the others, which we will discuss in the Hard Disk chapter. Other systems will have their own supported filesystems. FreeBSD, for example, offers UFS and ZFS support.

The mountpoint is which directory will be mapped to the hard disk. In Windows systems, every disk partition is usually mapped to a drive letter, such as C:. This is not true of Unix systems. In Unix there is one filesystem root called /. Any other hard disk is simply accessed as a folder beneath that root partition.

For example, you may have system with two disks. One disk contains all of the user files and folders while the other contains the operating system. The system disk will be mounted on /, while the user folders will be mounted on /home.

Things get a little more complex with dual boot systems, but that's another section.

1.7.3 Choose Packages

After partitioning the disk, the installer may ask you to select software packages. Sometimes this is more of a general selection such as "games", "software development", "web hosting", "graphical interfaces/X11", etc. Other systems allow you to select to install or not install individual packages. Still others will install their default software without your intervention at all.

Note: some systems do allow you to install a system without any graphical user interface and some do not include them by default, such as Ubuntu Server and FreeBSD. They can be installed later, however.

1.7.4 Install System and Packages

At this point you can go have a cup of coffee or read a book while the software actually installs the system and packages. Some distros, such as Ubuntu, will offer a preview of the features of their system.

1.7.5 Set Time Zone

Very simple. You will choose a time zone very close to your location.

1.7.6 Set root Password

Not all Linux distributions require you to set the root password these days. Ubuntu only sets up a normal user account. Traditionally, however, Unix systems require a root password to be set. root is the administrator account on Unix systems. root has ultimate control of the system and machine, and can make system-wide changes, install or remove software, create other accounts, start services and many other tasks. If you do set a root password, make it a good one and do not forget it.

1.7.7 Create User Account

On any computer system it is not a good idea to perform every day tasks as the administrator or root account. This is a significant security risk. Therefore it is good practice to set up a normal, unprivileged user account.

In this step you will create an account for yourself by supplying a username, password and possibly full name and other information. This is the account you will use for your every day task, and only use the root account for making changes to the system. This is covered in more detail in the Users and Groups chapter.

1.7.8 Configure Network

Computer networks are ubiquitous today. Whether it's wired or wireless, nearly every computer is connected to a network. This part of the install phase allows you to configure the network settings.

There are typically two options: dynamic and static. Dynamic or automatic setup means that your computer will be set to automatically retrieve its settings and an IP address from a server on the network (such as a wireless router at home). There will be no further configuration necessary.

Static or manual configuration requires you to know your network settings and an available IP address you will assign to your network interface. This will be discussed in more detail in the Linux/Unix Networking chapter.

1.7.9 Reboot

After the post-install tasks are completed, it is time to reboot, remove the install media and attempt to boot to the new system. Assuming all went well, you should see a login prompt. This will either be a graphical login display or a text based prompt.

1.8 Dual/Multi Booting

Dual booting is very common today. Many users require Windows for specific software that may not have a Linux alternative (though this is becoming much less common) while others may be avid PC gamers. Some people like to experiment with multiple operating systems. Many simply like to keep Windows around while learning the Linux system.

Regardless of the reason for dual booting, there is a little bit of planning and setup involved. How you go about this depends on one of two scenarios.

1.8.1 Multiple Disks

By far the easiest way to set up a dual boot system is to use a single hard disk for each operating system. For example, you could have a hard drive dedicated to Windows and one dedicated to Linux. For that matter, you could have triple, quadruple or many-boot systems. This is a clean solution that isolates files, disk access and even boot code depending on how you set it up.

The procedure for setting up a dual boot system with multiple disks is as follows:

1. Install Windows on one disk first (if using Windows, of course). This is because Windows will completely overwrite the Master Boot Record of the primary boot drive when you install it. Installing Windows first allows us a nice choice later

2. Install Linux, BSD, etc. on the other disk after installing Windows. This will overwrite the Master Boot Record that Windows installed

previously. But that's ok. We can now configure the Unix boot system to recognize both operating systems and allow the choice of OS at boot time

It is also possible to install Windows with **only that disk in the computer**. That way Windows and its MBR are installed only on that disk. Then you will install Linux on the other disk, also with **only that disk in the computer**. Then add the Windows drive back to the system and configure the Linux boot loader to see the Windows system and add it as a boot option.

The advantage to this setup is that you can completely remove the Unix disk and Windows is still capable of booting from its own disk.

1.8.2 Single Disk

This one is a little more tricky because it requires partitioning. This is complicated if you have already been using a Windows install and would like to add Linux to the system.

It is easier to start from scratch, which involves the following steps:

1. Use a disk partitioning utility such as a GParted boot CD. You can also use many Linux Live CDs (such as Ubuntu). These often include the GParted utility.
2. Create two partitions, one formatted NTFS for Windows and the other ext4 (or your filesystem of choice) for Linux
3. Reboot and remove the GParted or Linux CD.
4. Boot from the Windows install disk and proceed to install Windows on its NTFS partition. This will also configure the Master Boot Record to boot into Windows.
5. After installing and configuring Windows, boot from the Linux install disk and install Linux to the other partition. Depending on the distribution it may recognize the Windows system and offer to allow booting to it. This will then overwrite the MBR that Windows installed, which is what we want.
6. Reboot and test. The Linux boot loader (which these days is usually a program called GRUB) will then prompt you to select which operating system to use.

The other scenario is if you already have Windows installed and would like to add Linux. In this case you must resize/shrink the Windows NTFS partition and create a new one for Linux. This is the procedure:

1. Defragment the hard drive if using Windows prior to Vista. This is often necessary because fragmented systems can have files all across the partition, which makes resizing impossible. Defragmenting places all files at the beginning of the partition, creating all free space on the rest of the drive

2. Resize the NTFS partition. This can be done with a GParted disk, some Linux Live CDs or with the Windows Disk Management utility. You will need to shrink the volume to an appropriate size. Make sure to leave enough for Windows, and give enough to Linux (how much you need depends a lot on the distribution). After resizing, reboot Windows *twice*.

3. Install Linux in the newly created free space. You will use its partitioning tool (probably GParted) to create a new partition in the unused space. This will then overwrite the MBR installed by Windows and configure the boot menu to allow a choice of OS.

Summary

One of the great things about Unix/Linux is that it can run and even be useful on almost any hardware, old or new. This means anybody can have a working Unix system, even with no money invested.

Once you have the hardware, the install process is as follows:

1. Download or otherwise acquire installation media, such as a CD or DVD image

2. Burn the CD/DVD or write installer to USB drive

3. Boot from the install media. It may be necessary to configure the BIOS to allow this

4. Follow the instructions or prompts for the installer. The key steps are:

    - Partition hard drive
    - Select packages (may be optional)

- Wait for packages to install
- Set root password
- Create a user account
- Configure network

5. If you wish to dual boot you have two options

   - Use a hard disk for each OS
   - Create multiple partitions on a single hard disk. If Windows is already installed, this will mean shrinking the NTFS partition to allow space for creating a Linux partition

Chapter 2

Command Line Basics

What? Command line already? We just got Unix installed. This isn't 1985 any more, we're way beyond DOS. Why are we bothering with the command line when we have all those easy to use Graphical User Interface (GUI, often pronounced *"GOO-ee"*) utilities and programs?

I'm going to answer that question very directly:

Learning the command line will make you a better computer user. It is more powerful and eventually faster (usually) than GUI tools. Like Unix itself, command line is *everywhere*, including Windows. Not only is the DOS-like Command Prompt useful in Windows, Microsoft's PowerShell command line system is becoming a necessity on its recent server operating systems. By "necessity", I mean that there are things in Windows Server 2012 that you can *only do via command line*.

Many systems are available only with command line remote access. Unix systems very often do not even have GUI environments installed, as it would require more memory and processing power to run them. As such, all management and administration is performed with the command line. If you are an experienced Linux command line user, your skills will be useful if you're ever working for a bank with an IBM AIX server that has no GUI installed.

So embrace the command line. Use it and use it a lot. It may be slow going at first, but before you know it, you'll be managing your files faster than you ever could with a GUI. You will be a better computer user for it and if you're pursuing a career in IT or software development, it will be one of the most useful skills you can have.

Command line isn't difficult to use. In fact, the basic idea is very simple: you type a command with the keyboard and the computer executes the task or program. It is, however, not as easy to *learn* as GUI tools. But with time and experience, it becomes second nature. You may even grow to prefer command prompt.

Finally, an example of the power of the command line. Say you have a folder full of one thousand photos and images of varying sizes and types. You need to convert all of them to PNG images no wider than 640 pixels. This would be a nightmare using GUI utilities. It would take days, or many people to complete the task. Admittedly, there are GUI tools that would do this for you in much less time than using Paint or Photoshop. But the idea is to show that it can be done easily via command line:

```
$ for img in *; \
do convert -resize "640x>" ${I} ${img%.*}.png; \
done
```

I'll be the first to admit that that command looks a bit cryptic in places. Certainly not something that could be learned on the first day, but an example of a single line command that can accomplish a task that would require specially installed GUI software, or a lot of time and effort. It would also be the only way to do it if the system was a server or machine without a GUI installed or accessible only via remote command line.

2.1 The Shell

The command line environment is provided by a program called a *shell*. The shell accepts and interprets your commands, executes them and displays output if necessary. In Windows, this program is called Command Prompt (or cmd.exe). There are and have been many Unix shells in the past, such as the Bourne Shell, csh (c-shell), ksh (korn-shell) and the Bourne Again Shell, or **bash**. bash is by far the most popular shell on Linux systems, while BSD systems offer their own versions of csh and ksh and require that bash be installed separately. Multiple shells can be installed at the same time, and each user can use whichever they prefer.

2.2 Command Structure

Most Unix commands have a particular structure, and this was by design. Unfortunately there are exceptions, but for the most part it is the same.

There are two types of commands: internal and external. **Internal** commands are part of the shell program itself (bash, csh, ksh, etc). When you type the command it is interpreted and executed by the shell software. Examples are cd, read, eval and echo. **External** commands are actually programs installed on the system are are thus external to the shell program. The shell must search for and execute the programs. Examples are tar, passwd, unzip and rm.

By far, most commands you use will be external. This means the name of the command corresponds to the name of a program file located somewhere on your hard drive (or other storage medium). If that program does not exist, the shell will give you an error if you try to run it.

Historically, a command prompt ends with "$", so that is how I will indicate the prompt itself. The prompt is *not* a part of the command itself.

Unix commands are composed of one or more parts, usually *separated by spaces*. The first part is the command itself. You will always need at least that part. The next parts are called the command **arguments**. We shall cover this in more detail shortly.

The most basic form of a command is to type only the command itself. This is an exmaple of a command that will list the contents of your present working directory (more on that in a few pages):

```
$ ls
programs/ websites.txt song.mp3
$
```

The output shows a folder called "programs" (denoted by the trailing / in the name), a text file called "websites.txt", and an mp3 file.

In graphical environments, a command prompt window can be used to launch common programs. For example, to start the firefox web browser, you can do the following in a command prompt window:

```
$ firefox
```

The next command form involves command line arguments. The basic form is as follows:

```
$ command_name argument
```

First type the command itself, then one or more spaces, followed by an argument. This argument could be many things, such as a file name, a web address or a string of characters. Examples:

To download a file:

```
$ wget http://unix.wvncc.edu/cit220/StudentDataFiles.zip
```

To unzip a file:

```
$ unzip StudentDataFiles.zip
```

To change to a new directory that's in your current one:

```
$ cd StudentDataFiles
```

To list only mp3 files:

```
$ ls *.mp3
song.mp3
$
```

By far one of the most common mistakes among those who are new to Unix commands is forgetting the space between the command and any following arguments. Always remember to include the space or the computer will think you are trying to run a different command.

Here is an important one you may find yourself using very frequently:

```
$ nano filename.txt
```

This command will run the **nano** text editor (like Notepad in Unix) and open a file called filename.txt. If you ever need to make changes to a text file, you will use a command just like this. If filename.txt does not exist, this will allow you to create it.

Another type of argument is a **command option** or **flag**. These are usually denoted with a dash, followed by a character or word. Command options are usually intended to change the behavior of a program or to invoke a particular feature that doesn't happen by default, or to supply some necessary bit of information the program needs to run properly.

For example, to get a long list of the files in your current directory:

```
$ ls -l
total 78808
drwxrwxr-x 2 jdoolin jdoolin     4096 Oct 19 16:03 programs
-rw-rw-r-- 1 jdoolin jdoolin 80690400 Oct 19 16:06 song.mp3
-rw-rw-r-- 1 jdoolin jdoolin     2679 Oct 19 16:05 websites.txt
```

The option passed to the 'ls' command is '-l'. The 'ls' command all by itself only gives a list of the filenames in a directory. However, this option changes the behavior of 'ls'. The 'l' stands for 'long' and instructs the ls command to show file details, including permissions, user and group ownership, file size, last access time and the name. These types of arguments to commands are VERY common and you should make it a point to become familiar with using them.

Commands can also take multiple arguments. For example, we can combine two previous examples to get a long list of all mp3s in the current directory:

```
$ ls -l *.mp3
-rw-rw-r-- 1 jdoolin jdoolin 80690400 Oct 19 16:06 song.mp3
```

Multiple flags/options can often be passed as a single argument. The following command adds the -h option to display the file size in 'human' readable form (in Kilo/Mega/Giga/Terabytes instead of number of bytes):

```
$ ls -lh *.mp3
-rw-rw-r-- 1 jdoolin jdoolin 77M Oct 19 16:06 song.mp3
```

Some command options require another argument. The following command downloads the same file from the earlier 'wget' command, but saves it under a different filename. The -O argument tells wget to use a different filename, but it *must be followed by the filename* you wish you use, otherwise wget will give an error:

```
$ wget http://unix.wvncc.edu/cit220/StudentDataFiles.zip -O sdf.zip
```

So just like 'ls -l', this command is changing the behavior of the 'wget' program. It normally downloads the file under the same name it gets from the web site (StudentDataFiles.zip in this example). But using the -O command tells wget to save the downloaded file under a different filename (sdf.zip). However, if you didn't supply the new name (sdf.zip), wget would give you an error.

These various command structures can be used together, sometimes resulting in rather long commands, but with the proper understanding and practice, they will make complete sense. The more you use the command line, the more it will make sense and the more comfortable you will be.

2.3 man Pages

Before going on any further, it must be mentioned that most Unix systems have a built-in documentation system. It is similar to the Windows Help system. This documentation system is called **man pages**. ′man′ stands for ′manual′ and can be used to view the documentation for any command. For example, this will show you the manual, or ″man page″ for the wget command:

```
$ man wget
```

man pages display a short and long description of what the command or program does, how to use it, what its arguments, flags and options are, relevant files, related commands, bugs and sometimes even examples. When all else fails, try ′man′ to see how to use a command.

2.4 Where Am I?

One of the concepts that many people new to the command line struggle with is the idea of the **Present Working Directory**.

When you log into a Unix system it ″places″ you in your personal home directory. This is always your starting point when loggin in. All users on a Unix system have their own isolated directory where they work and keep their files. On most Unix systems, this is located in /home/username, where ″username″ is whatever the user′s system username is. For example, my home directory is /home/jdoolin. On Mac OSX systems it is /Users/jdoolin. More on home directories in Users and Groups.

The present working directory can be thought of as your current operating context. It is the directory you are ″in″. If you are coming from Windows, you can think of it as the folder you are currently viewing in the File Explorer. If you type ″ls″ by itself, it will list the contents of your present working directory. There is a command to see what your present working directory is. Not surprisingly:

```
$ pwd
/home/jdoolin
$
```

So in the above case, my present working directory, or pwd, is /home/jdoolin. If I enter the ls command, I will see all the files in my home directory. If I have a subdirectory (technical term for sub-folder) in my home directory called 'StudentDataFiles', I can move or "change" to this directory with the cd command. Get it? cd? **c**-hange **d**-irectory? This is analogous to double clicking on a sub-folder in the Windows File Explorer.

```
$ cd StudentDataFiles/
$ pwd
/home/jdoolin/StudentDataFiles
```

I have now changed my present working directory. First, the 'cd StudentDataFiles/' command changed my present working directory. If that directory did not exist, the shell would have reported an error. Then the 'pwd' command shows my new present working directory. Note that if you use the cd command without any arguments, it takes you back to your home directory:

```
$ pwd
/home/jdoolin/StudentDataFiles
$ cd
$ pwd
/home/jdoolin
```

The first 'pwd' command reports that I am currently in the /home/jdoolin/StudentData directory. Running 'cd' by itself always takes you to your home directory, so my new pwd is /home/jdoolin.

It is very important to note that with many commands *there will be no output when the command is successful*. This is something that confuses many who are new to the Unix command line. A lot of people assume that a command will give output regardless of success or failure. However many programs only show output if there is an error or if it fails. cd doesn't show you any output unless you try to change to a directory that doesn't exist, such as in the following example:

```
$ cd StudentDataFiles/
$ cd NonExistent/
-bash: cd: NonExistent: No such file or directory
$ pwd
/home/jdoolin/StudentDataFiles
```

2.5 Paths

It is important in any operating system to understand the concept of paths. A path is a way of expressing where a file or folder is located within a filesystem. In the Windows world, paths look something like this:

```
C:\Users\jdoolin\My Documents\file.txt
```

This indicates that the file is located on the C: drive. To navigate to the file in Explorer, you would double click on C:, then the Users folder, then the jdoolin folder, then My Documents. Then to open the file you would double click on file.txt. In the path, each directory name is separated by a backslash (\). This is called the **path delimiter**.

In the Unix world there is no concept of drive letters. There is only a single root filesystem under which all other drives or shares are accessed. This root directory is called / (forward slash). More on drives and mount-points in another chapter.

A similarly placed file on a Unix system would look like this:

```
/home/jdoolin/Documents/file.txt
```

First take note that the path delimiter on Unix systems is a forward slash (/) as opposed to the backslash (\) in Windows paths. You could open this file in one of several ways. But to understand how they work, you must understand the two different types of paths.

2.5.1 Absolute Paths

An **absolute path** (or **full path**) is one like the example above. It describes in full detail, from the root of the system to the absolute location, where a file or directory is located. Examples:

```
/ <-- the root partition or top level directory of the entire system
/home/ <-- the home directory, which will contain all user directories
/home/jdoolin/ <-- my home directory
/home/jdoolin/Documents/file.txt <-- the file located in a Documents fol
```

The 'nano' command has already been mentioned as a way to create or make changes to a text file. It is often the case that on Unix systems you need to view the contents of a file, but you have no need to change it. To only view (but not edit) the contents of the file in the example above, you can use the *less* command, followed by the full path/absolute path to the file:

```
$ less /home/jdoolin/Documents/file.txt
```

This would successfully open the file no matter what your present working directory is, since you are supplying the full path. Full paths work everywhere.

Another approach would be to navigate to the directory containing the file then open it. This is technically the approach you must use to open a file in Windows Explorer, only instead of double-clicking on drive letters and folder names or icons, you would use the *cd* command to change to the directory containing the file, then open it with *less*. You can give the absolute path to the cd command:

```
$ cd /home/jdoolin/Documents/
$ less file.txt
```

This will change your present working directory to /home/jdoolin/Documents. Because you are now in that directory, you can just use the filename as the single argument to *less*.

As you can see, there are already multiple ways of performing the same task on the command line. We just looked at two different ways of opening a text file to view its contents. First, we used the full path to the less command. The second way was to use the cd command with the full path to the folder containing the file, then using the less command to view it using only the filename.

This brings up another common stumbling block for those new to command line. To access a file using *only the filename by itself*, you must be **in** the directory containing it. In other words, the directory containing the file must be your present working directory. Otherwise, you must use the path to the filename.

2.5.2 Relative Paths

Using absolute paths all the time would eventually get tiresome. In fact, for some folder structures it would be a complete mess. Fortunately it isn't necessary to always use full paths. There is another type of path called a **relative path**. Instead of using the full path, you can use a path that's relative to your present working directory. In fact, we have already used relative paths without realizing that's what we were doing. But first, there are two special directories that need to be discussed.

Every folder on Unix systems contains two special directories. The first one is called "." (just a single dot/period) and the other is ".." (two dots/periods). These aren't real directories. They are *virtual* directories that exists in *all* Unix directories. Recall our earlier example when we used the *ls* command. We can pass *ls* an argument that will show all files, including those two special directories. Appropriately, the option is *-a*.

```
$ ls
programs/  song.mp3  websites.txt
$ ls -a
.  ..  programs/  song.mp3  websites.txt
```

You can see . and .. listed among the other files and directories. But what are they?

The single dot refers to your present working directory. You can treat it like any other directory. For example:

```
$ pwd
/home/jdoolin
$ ls
programs/  song.mp3  websites.txt
$ cd .
$ pwd
/home/jdoolin
$ ls .
programs/  song.mp3  websites.txt
$ less ./websites.txt
```

In that example I executed a few commands that did essentially the same thing. My pwd is /home/jdoolin. The third command, *cd .*, told the shell to change directory to single dot. Since single dot refers to whatever our present working directory is, it doesn't really change directory. That is evident by the next *pwd* command. Passing single dot to the *ls* command is equivalent to using *ls* by itself. You can even view a text file using less and prepending ./ to the filename.

Double dot is a more interesting virtual directory. It refers to the directory that *contains* your present working directory. This is also known as the ″parent″ directory, or directory ″above″ your present working directory. For example, if your pwd is /home/jdoolin/Documents, .. refers to /home/jdoolin:

```
$ pwd
/home/jdoolin/Documents
$ cd ..
$ pwd
/home/jdoolin
$ cd ..
$ pwd
/home
$ cd ..
$ pwd
/
$
```

By passing .. to *cd*, I changed to the directory that contains the Documents directory, /home/jdoolin. Doing the same puts me into the directory above the jdoolin directory, which is /home. Doing it a third time puts me in /, or the root of the entire system. Using the double-dot is a lot like clicking the Back button in Windows File Explorer (though not quite).

With double-dot in our toolbox, we are now equipped to use relative paths for most of our needs. Remember first that relative means relative to your present working directory. So you don't necessarily need to know the full path to a file, but merely where it is relative to your present working directory.

The first and easiest way to use relative paths is one we've already been using, and that's by referring to files in the present working directory with only their filename:

```
$ pwd
/home/jdoolin/StudentDataFiles
$ less websites.txt
```

As mentioned before, we can refer to websites.txt directly since it is in our present working directory. Let's change the scenario and say our present working directory is simply /home/jdoolin. We can still use a relative path to open websites.txt:

```
$ pwd
/home/jdoolin
$ less StudentDataFiles/websites.txt
```

This works because StudentDataFiles/ is also in our present working directory and we can refer to it directly. We could also refer to it more explicitly by using the "dot" virtual directory (although this isn't very common):

```
$ less ./StudentDataFiles/websites.txt
```

This is functionally the same thing as the previous example. In the same way, "double dot" can also be used in relative paths. Remember that '..' refers to one directory above our present working directory. Assume that we are still in /home/jdoolin:

```
$ cd StudentDataFiles/programs/
$ pwd
/home/jdoolin/StudentDataFiles/programs
$ less ../websites.txt
```

Let's take a look at that closely. Since we are in /home/jdoolin already, we can pass a relative path to *cd* that takes us into StudentDataFiles/programs. *pwd* then shows us where we are. We can then view websites.txt by using '..', which refers to one directory above our pwd, or in this case, StudentDataFiles. Since the StudentDataFiles directory contains the file, the command works.

'..' can even be combined like so:

```
$ pwd
/home/jdoolin/StudentDataFiles/programs
$ cd ../../
$ pwd
/home/jdoolin
```

This command has the effect of taking us two directories above. The first '..' takes us back into the StudentDataFiles directory and the second one takes us back to the jdoolin directory. This would be similar to using Windows File Explorer and clicking on the Back button twice to leave the programs folder, then th eStudentDataFiles folder.

There is one final relative path shortcut that must be mentioned, and that is ' ˜ '. The character is called a *tilde* (pronounced "TILL-duh") and is usually to the left of the 1 key on a US keyboard and must be accessed with the Shift key. This character is basically shorthand for *your home directory*. For example:

```
$ cd ~
$ pwd
/home/jdoolin
$ less ~/StudentDataFiles/websites.txt
```

In this case, ' ~ ' was used as an argument to cd (silly, I know, since we could have just typed 'cd' without any arguments) and as a way of opening the websites.txt file again. The nice thing about ' ~ ' is that no matter what your present working directory is, you can refer to files or directories in your home directory just by using ' ~ '.

2.6 File Management

One of the primary tasks any computer user needs to perform is file management. This refers to where we create files, how we organize them, naming and renaming, removing unwanted files, moving, copying and opening them. Users coming from the Windows world are accustomed to doing this with a graphical tool such as File Explorer, Windows Explorer or if you're old enough, File Manager. Many Windows users aren't even very familiar with Explorer and just use default directories and libraries without really knowing where their data is saved on the hard drive. You can also manage files using the Windows Command Prompt, as one would have in the DOS days. In fact, if you suspect your working career may involve Windows system administration, I suggest getting at least a little familiar with the Windows Command Prompt.

File management using the command line may seem slow and cumbersome at first, but as you internalize the commands and commit them to muscle memory[1], it becomes much faster and more efficient.

2.6.1 Creating and Removing Files

Most files you work with will be created by applications and programs, files you download or that are transferred from other devices. However, there is a simple command that allows you to create a completely empty file: **touch**.

[1]Eventually you won't have to think about which command you use to move, copy or delete a file, your fingers will just type it automatically

```
$ touch notes.txt
```

It is occasionally useful to create empty files using touch, but this is not the usual way you will create the files you use on a regular basis.

Removing a file requires the **rm** command, which can take one or more filenames to be removed.

```
$ rm notes.txt
$ touch todo.txt changes.txt
$ rm todo.txt changes.txt
```

2.6.2 Creating and Removing Directories

Creating a directory, or folder as it is otherwise known, requires the **mkdir** command. You can use absolute or relative paths to make a directory.

```
$ mkdir cit222
$ mkdir cit222/scripts
$ mkdir /home/jdoolin/cit222/pdfs
$ ls
cit222  StudentDataFiles
$ cd cit222/
$ ls
pdfs  scripts
$ cd
```

This will create three directories. The first is a 'cit222' directory in my home directory. The second creates a 'scripts' directory within my new 'cit222' folder. The third uses an absolute path to create another directory called 'scripts' within the new 'cit222' directory.

There is a command option, -p, that can be passed to mkdir that will create directories within directories at once. For example, the above command could have been simplified:

```
$ mkdir -p cit222/scripts
$ mkdir cit222/pdfs
```

The -p option allows mkdir to create both the 'cit222' directory and the 'scripts' directory within it. You can also create multiple directories with one command simply by supplying more directory names to mkdir:

```
$ mkdir music videos photos
```

This will create three new folders inside your present working directory: music, videos and photos.

There are two ways to remove a directory. The first is the **rmdir** command. rmdir, however, requires that the directory you are removing to be empty.

```
$ mkdir photos/vacation
$ rmdir music
$ rmdir videos
$ rmdir photos
rmdir: failed to remove 'photos': Directory not empty
```

In that example, the first two rmdir commands were successful because those directories are empty. The third fails, however, because it is not empty. The first command created a directory inside the photos directory, meaning it is not empty.

The other way to remove directories is to use the 'rm' command, but with the -rf option passed to it. This tells 'rm' to remove the directory and all files and subdirectories beneath it.

```
$ mkdir photos/vacation
$ rm -rf photos/
```

Be careful about using 'rm -rf'. *It is not reversible* and it removes the entire folder and its contents. Be especially careful when using this command as the root user. Imagine the destruction of an accidental 'rm -rf /'.

2.7 Copying a File

The command to copy a file is **cp**. This command can be used to create a copy of a file in one location to either another directory or to the same directory with a new filename. The cp command requires two arguments: a source file (the file to be copied) and a destination directory or filename. Here are a few examples:

```
$ cp grandCanyon.jpg photos/vacation
$ cp photos/vacation/meteorCrater.jpg /tmp
$ cp paintedDesert.jpg paintedDesert2.jpg
$ cp /home/jdoolin/reminders.txt /tmp
```

This shows four different ways of using the copy command. Each one works as follows:

The first creates a copy of the file grandCanyon.jpg in the vacation directory inside the photos directory. There will now be a file in /home/jdoolin/photos/vacation named grandCanyon.jpg in addition to the original file.

The second command copies the meteorCrater.jpg file from a relative path to the /tmp directory. This is an example of how you can copy a file that isn't in your present working directory to another location.

The third example makes a copy of paintedDesert.jpg in the present working directory, but since it is being copied in the same directory, it must have a different filename (paintedDesert2.jpg). So there would now be two copies of the same photo but with different filenames.

The fourth is simply an example of copying a file using two full paths, showing how you can really make a copy of any file in any location from any present working directory.

2.7.1 Copying Multiple Files

You can also use the cp command to copy multiple files to a single destination folder. The format is as follows:

```
cp file1 file2 file3 file4 ... destinationFolder/
```

In other words, you can copy any number of files, which will be listed first, to a single destiantion folder, which will be the last argument in the command.

2.7.2 Copying a Directory

In order to copy a directory, you must supply the **-r** argument to the cp command as follows:

```
$ cp -r photos /tmp
```

This will create a copy of the entire photos directory to the /tmp directory.

2.8 Moving a File

Moving a file requires the **mv** command. The arguments are the same as the cp command, but instead of leaving the original file, it will be removed. For example:

```
$ mv grandCanyon.jpg photos/vacation
$ mv photos/vacation/meteorCrater.jpg /tmp
$ mv paintedDesert.jpg paintedDesert2.jpg
$ mv /home/jdoolin/reminders.txt /tmp
```

These are exactly the same examples listed above, but with cp replaced with mv. The first will move the grandCanyon.jpg file from the present working directory to photos/vacation. The second will move the meteorCrater.jpg file to /tmp. The third one will have the effect of renaming paintedDesert.jpg to paintedDesert2.jpg. The fourth will move a file using a full path to /tmp.

2.9 Renaming a File

There is not a separate command for renaming a file. There are two ways to rename a file: using mv (move) or cp (copy). The first was just illustrated in the third of the mv examples. The mv command is the primary method of renaming a file. The other method is during a file copy. When you use the cp command to copy a file, instead of supplying a directory name for the destination, you can also supply a filename. Here are a couple of examples:

```
$ cp petForest.jpg photos/vacation/petrifiedForest.jpg
$ cp /tmp/reminders.txt /home/jdoolin/myReminders.txt
```

In both cases, the file is copied but it is also renamed at the same time. But by far, the most common method of renaming a file is simply to use the 'mv' command:

```
$ mv IMG_311834.JPG sunset.jpg
```

This mv command renames a file, such as one you may transfer from a smartphone or digital camera, to a more descriptive name.

2.10 Reading and Editing Files

One area (among many) in which the Windows and Unix worlds differ significantly is in the storage of configuration and settings data and information. While Windows mostly uses its labyrinth of cryptic and mysterious keys in the form of its registry, most of the Unix world uses simple text files for the bulk of its configuration, many of which reside in the /etc directory or a subdirectory within /etc. Most individual user configuration files are also plain text. Most Unix logs are also plain text files, though there are exceptions.

What this means is that being able to read or edit text files is one of the most important and frequent tasks you will do on Unix systems. It is therefore imperative to learn how to view and edit these files.

While there are many graphical text editors available that put you in the mind of Microsoft's Notepad, you aren't necessarily guaranteed to have a graphical interface available to you for a Unix system. You know what that means. Yes, we're going to look at command line/console text editors.

The two most common editors found on Unix and Linux systems are **vi** and **nano**. vi is almost *guaranteed* to exist on any Unix system you ever use, so it's certainly a good choice if you think you may move beyond the Linux world. nano is very common nowadays on most major Linux distributions. It is lightweight, easy to use and even has syntax highlighting and coloring. However it is not guaranteed to be on any random Unix system (such as an AIX server or a FreeBSD box). There are many other editors out there, such as vim (vi Improved), emacs, jed, joe and pico, and a lot of graphical editors like jedit and gedit. But since vi and nano are so common, we will discuss these two before moving on.

Of the two, nano is more intuitive to use. To start nano, the command is simply:

```
$ nano file.txt
```

This will open a file called *file.txt*. If the file already exists, it will open the current file for editing. If the file does not exist nano will create the file once you save it.

Once nano is open, you can enter text normally and navigate with the arrow keys, home, end, pageup and pagedown. When you have finished editing, do the following steps:

1. Press Ctrl+x to tell nano to exit the editor, however the file is not yet saved. You will be asked if you wish to save.
2. Press 'y' to tell nano that you would like to save the file. You will then be prompted with a filename.
3. Press Enter to accept the filename -OR- if you wish to save as a different filename, change the name then press Enter.

That's pretty much all there is to nano. vi, on the other hand, is a little more complex, yet more powerful. Opening a file with vi is just as easy as it is with nano:

```
$ vi file.txt
```

However, things change as soon as the file is open. If you try to enter text it may not work. This is because vi opens in **command mode**. vi has two modes: command mode and **insert mode**.

It's easier to say what command mode is not. Command mode is not the mode you use for entering text normally. Command mode is for entering, well... commands. These commands range from copy and paste and search and replace to sorting, capitalizing sections of text, formatting paragraphs, grabbing output from the command line, building software, debugging code and many other tasks. It is also used for saving and exiting vi.

Insert mode is what you want for entering text normally. But since you start in command mode you must get into insert mode by pressing the 'i' key. Technically there are other keys as well, but we'll just focus on 'i' for insert. Once you're in insert mode, you can type normally.

Once you've made your changes or additions, you'll have to save and probably quit as well. Now we need command mode. To go from insert to command mode, you press the Esc (escape) key. Yep, that one all the way at the upper left. You can always tell when someone is doing some heavy editing in vi (or vim) when you see them repeatedly going for the Esc key.

Once you've pressed Esc to get into command mode, you save by typing ':w'. You then quite by pressing ':q'. If you wish to save *and* quit in the same step, you can type ':wq'.

There is a lot more to both editors, especially vi and it's more powerful cousin vim, but this will be enough to get you by.

2.11 I/O Redirection

Wouldn't it be great if we could send the output of a command to a file? For example, there is a command you can use to see how much disk space your files and folders are using. It would be very handy to send this output to a file so you could review it without scrolling up and down the terminal. Or maybe it would be handy to run a series of commands that collects some useful information about your Unix system, such as which OS and version you're using, disk space, network information, etc, and send it all to a file.

The command line offers a way to do this called I/O redirection. It allows you to send the output of any command to a file instead of the screen. However, it must be noted that **you will not see the output of the command**. The special character used for redirecting output is '>'. For example:

```
$ ls
programs/ song.mp3 websites.txt
$ ls > output.txt
$ ls
output.txt programs/ song.mp3 websites.txt
```

The first command is a simple ls. The second one redirects the output of the 'ls' command, so you don't see its output. Instead the output is sent to a file. You can see this new file by entering another 'ls'. Just FYI, you can do the same thing in the Windows command prompt if the need was ever there.

If you redirect output to the same file twice, **it will overwrite the previous file** with the new output. So be sure that this is what you want. You may find the need to append (add) the output of a program to a file that already exists, rather than overwrite it. This is done in the same matter, but instead of a single '>' character, you simply use two, '>>'.

```
$ ls >> output2.txt
$ ls >> output2.txt
$ cat output.txt
```

```
output.txt programs/ song.mp3 websites.txt
output.txt output2.txt programs/ song.mp3 websites.txt
```

The first 'ls' command sends its output to the file 'output2.txt' using the append symbol. Because the file does not yet exist, this will create the file. The second 'ls' command is exactly the same, but its output will be different because of the presence of the new file 'output2.txt' that was created previously. The 'cat' command simply dumps the contents of a file to the screen, so we can see both lines of output printed by the two 'ls' commands.

Occasionally you may try output redirection that seems as if it doesn't work. You still see output to the screen and nothing goes into the file. When this happens it is usually because the program is sending its output to **standard error**. It is basically a separate output stream even though it normally just gets printed to the screen. This allows a program to have normal output and error output that can be processed separately. In order to catch error output, the command becomes a little more complex:

```
$ wget http://unix.local/cit220/index.html 2> wget_output.txt
$
```

wget, as we saw in an earlier chapter, is a program that can be used to download a file from a web site. The output of the command shows connection information and a progress bar, as well as results of the download attempt. This is all sent to standard error instead of standard output, so if you just use '>', you will still see the output and there will be nothing in the output file. However, using '2>' tells the shell to redirect standard error[2].

It is also possible to redirect input. This is valuable for interactive programs that prompt for information. You can store your answers in a file, then simply use that file as the input to the program.

```
$ sort < websites.txt
```

This command will use the websites.txt file as the input for the sort command and will then send sort's output to the screen. However, there is a good chance you would like to also send that output to another file, so you could have both the sorted and unsorted files. This is possible by combining input and output redirection:

[2]The number 2 refers to file descripter number 2. File descriptors are basically the numerical assignments of any opened file, given by the operating system. Two reserved file descriptors are 1, which is assigned to standard output, and 2, which is assigned to standard error.

```
$ sort < websites.txt > websites_sorted.txt
$
```

2.12 Pipes

One of the most powerful command line tools is the **pipe**. Pipes were one of the features that made Unix stand out from other operating systems of its time, and it remains incredibly useful today. Pipes are basically a really slick form of I/O redirection. The idea of the pipe is that the output of one program becomes the input of another. But before getting into details, let's talk about the Unix Philosphy.

2.12.1 The Unix Philosophy

During the early development years of Unix at Bell Labs, an approach to computing began to develop that would eventually be referred to as the Unix Philosophy. This is best summarized by Doug McIlroy of Bell Labs:

> ″This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.″

The original Unix developers strove for simplicity. One way in which they achieved this was to write small programs that do only one thing, but do it very well. This is opposed to writing large programs that do many things. Smaller programs are easier to maintain.

These small programs were also designed to work well together. This was achieved using the third part of the philosophy, by handling text streams. We've already seen text streams any time a program on the command line produces output. This stream of output is sent to the screen. We can redirect that stream to a file. But we can also redirect it to another program, and that is a Unix pipe.

2.12.2 Output Becomes Input

Rather than sending the output of a program to the screen or a file, a pipe allows the output to be sent to another program. For example, the *cat* program, as mentioned before, reads the contents of a file and sends it to the screen. This is all well and good for a small file, but what about a very large file? A pipe can be used to send the output of cat to a program called a **pager**. There are two common pager programs in the Unix world. The earlier and more common pager is called **more**. This allows you to page through a file or the output of a program page by page instead of just dumping it to the screen.

Never without a sense of humor, Unix programmers created another pager called **less**. This is ironic, of course, since less actually does more than more. less allows you to page backward, forward and even to the side, rather than just straight through the output once. less, however, is not on as many Unix systems as more. So be aware of both, but use less when you can.

You can pipe the output of any command to less. Remember our sort command from earlier? Instead of sending its output to a file, we could also pipe it to less. In fact, instead of redirecting input, we can pipe the output of cat to sort. The pipe character is '|' (the vertical bar accessed by pressing Shift+\).

```
$ cat websites.txt | sort | less
```

First, the cat command dumps the contents of websites.txt, but since we're using a pipe, the output is not sent to the screen. Instead it becomes the input for the sort command, which then sends its output to less. This is the Unix Philosophy at work. There are three programs that are designed for one purpose each, but work well together using text streams.

Not surprisingly, pipes and I/O redirection can be combined.

2.13 Command Sequences

It is also possible to type a sequence of commands on the same line. There are two common ways to do this. The first is to separate each command with a ';' character. Using this method, each command will be executed, regardless of success or failure. For example:

```
$ cd StudentDataFiles; ls
```

This will work just fine. It will change to the StudentDataFiles directory then list its contents. This may be exactly what you want, however what if you tried to change to a directory that doesn't exist? You may not want the ls command to execute in the event that the directory doesn't exist. In this case, you use '&&' to separate the commands. If the first command fails, the second will **not** execute.

```
$ cd DoesNotExist && ls
```

The cd command will fail because the DoesNotExist directory isn't there. Since the cd command fails, the ls command will not even be attempted. This is appropriate for many situations where one command depends on the first being successful.

2.14 Environment Variables

When you use the command line, it is dependent on a variety settings that determine your operating environment. These are called **environment variables**. There can be many environment variables, depending on the system. Some shell programs use different variables than others do. We will be going over some common environment variables used by the bash shell.

Environment variables are referred to by name. A few examples are USER (your username), UID (your numeric user id), SHELL (the path to your shell's binary), TERM (the terminal emulator used by your command prompt) and PATH. Each of these variable names also has an associated value. To see its value, you use the **echo** command, followed by the variable name, but with a dollar sign ('$') preceding it. For example:

```
$ echo $USER
jdoolin
$ echo $SHELL
/bin/bash
$ echo $TERM
rxvt
```

These variables are used by the shell itself and can even be accessed by programs that are started within the shell.

Many variables are set, changed or updated automatically, such as USER, PWD (present working directory) and SHELL. Others, however, are useful to modify. To modify a variable, you use the **export** command as follows:

```
$ export VARIABLE=value
```

One useful environment variable that you can modify is your shell prompt. The variable name you need to modify is 'PS1'. The prompt this book has used thus far has been a simple $ followed by a space. If you wanted to change this to say, for example, "command>", you would use export to change the PS1 variable:

```
$ export PS1="command> "
command> echo $USER
jdoolin
command>
```

After the export command, the shell prompt changes, and it will remain this way until you log out of the shell.[3] Also notice that when you *set* a variable, you do not need the '$' before the variable name.

Possibly the most useful environment variable to know about, however, is the PATH environment variable. When you enter a command in the shell, it is often the name of a program file (also known as a "binary") located somewhere on the hard drive. A good example is the ls command. To see where the ls command is found, you can use the **which** command.

```
$ which ls
/usr/bin/ls
```

This tells us that the binary executable file for ls is located in the /usr/bin directory. But how did the shell know that? How did it know where to find the ls program? The answer is in the PATH environment variable. First, examine the contents of the PATH variable (yours will not likely be the same):

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:
/opt/java/bin:/opt/java/jre/bin:/opt/android-sdk/platform-tools
```

The PATH variable contains a list of directories separated by a colon. Each one of these directories contains binaries for programs. Any time you type a command that corresponds to a program, the shell searches each of these directories in order until it finds one with the name you typed. If it does not find one, it will display a "command not found" error.

As you will see in a future chapter, some of these directories are Unix standards for the location of binaries, such as /bin, /sbin and /usr/bin. A few in my PATH, however, aren't standard. A good example is /opt/android-sdk/platform-tools. This is where Android development programs are

[3]There is a way to make this permanent. You must add the same export command to the file named .bashrc located in your home directory. Note that the filename begins with a dot.

stored on my hard drive, and I had to put this directory in my PATH variable myself. If you want to add a directory to the PATH variable, it is slightly different than just setting a variable:

```
$ export PATH=$PATH:/opt/android-sdk/platform-tools
```

This command is essentially instructing the shell to set the PATH variable to whatever it currently is ($PATH) plus the /opt/android-sdk/platform-tools directory. You can add any directory to the PATH variable, including those in your own home directory. One bad practice, however, is to have '.' in your current PATH. Some beginners do this so that their Unix shell behaves more like Windows or so they can execute binaries that are in their current directory, whatever it happens to be. There are some security implications with doing this. For example, I could write my own custom program named 'ls' that does something devious. If you were to change to that directory and type 'ls', there is a chance that MY ls command will be executed instead of the real one.

Summary

- The command line environment is provided by a program called a *shell*, the most common of which is *bash*.
- Typical command structure is *command-name* (space) *options* (space) *arguments*, however there is a lot of variation.
- To see how to use a particular command, you can usually use the *man* command, also known as *man page*.
- Your *present working directory*, or pwd, is the directory in which you are currently operating. You change directories with the *cd* command.
- A *path* describes the location of a file or directory on the filesystem. Paths may be absolute, which displays the full path from the root all the way down to the file or directory, or relative, which is relative to your present working directory.
- *rm* removes a file, *mkdir* makes a directory, *rmdir* removes a directory, *cp* copies files and directories and *mv* is used to remove or rename a file.
- Editing text files is a very common task on Unix systems. Two popular Unix editors are *vi* and *nano*.

- The output of a command can be redirected to a file with the >character. To append/add the output to a file that already exists, use >>.
- The output of one command can be used as the input to another command by using a pipe, denoted by the |character between the commands.
- Multiple commands can be executed on one line using either a semicolon (;) or the double ampersand (&&), which requires that the first command be successful for the next command to execute.
- Environment variables are important settings for the shell and other programs. They can be accessed with the *echo* command and can be set with the *export* command.

2.15 Command aliasing

2.16 .bashrc

Chapter 3

Users and Groups

Those coming from a Windows world by now are familiar with the concepts of users and possibly even groups if you've worked with Active Directory. However users and groups work differently in the Unix world, especially in terms of one of the original purposes of Unix: timesharing.

3.1 Timesharing and Normal Users

In the 1960's, computers were largely single user batch systems. In other words, only one person at a time could use the computer and often could only run one program at a time. These machines were tremendously expensive and in high demand in the growing computer industry. It wasn't unusual for programmers to wait for hours for their time with the computer, nor was it unusual for people to be charged money for this time.

One of the goals of MULTICS and later Unix (as well as other newer systems of the time) was *timesharing*. This would allow multiple users to interface with a computer at the same time using terminals that could be situated around a room or even a building. Not only would the users be able to run their jobs concurrently with others using the system, but it should be transparent, such that each person felt as though they were the only one using the system.

This introduces the concept of **users** and **groups**. Each person that uses the computer has a unique user account. Each account is identified by a username and in Unix, a numeric User ID. These accounts are used for file and directory ownership so that each user's files can be protected from other users (permissions will be covered next chapter) and even isolates each user's processes (running programs). On systems that have mail servers, a user account may also then have an email address.

So far this isn't all that different from the Windows world. But what makes Unix different is that any of its users may login and use the system concurrently. This can be done with multiple terminals or even remote connections over the network. While Windows allows remote connection, only one user may be on the system at any time.

There are two primary types of user accounts: normal (unprivileged) users and root. This is equivalent to the normal and Administrator accounts on Windows machines, however there can be only one root user on Unix machines. On Unix systems, this user's name is nearly always **root**.

Normal users have full access to their files and read access to the programs and files needed for normal operation. They do not, however, have access to system configuration files.

3.2 Bow Before Me, For I Am root

root can do anything root wants on the system, up to and including removing every important file on the system until it is no longer operational. root can install and uninstall software, enable/disable or start/stop services and daemons, reboot or shut down the system and make critical changes to the system. root has access to all files, including those of all normal users. root can terminate any process, mount filesystems, partition hard disks and many other tasks that a normal user can not.

Because root has so much power, there is a security risk in performing operations as root. Not only could uninentional mistakes be made, but if you were to browse the web as root, for example, and your browser was compromised by a malicious web site, the attack would then also have root access to your entire computer, which is obviously not good.

It is therefore recommended to use a normal user account for day to day computer usage and only use the root account for the tasks that require it.

3.2.1 su and sudo

If you need root access, you can always just log in at the console as root and you're good to go. However this isn't always practical. After all, if you're using a normal account, it would be inconvenient to log out just to install a piece of software. Unix has one primary and one secondary way of getting root access without having to use the console.

The first is the **su** command. su without any arguments allows you to enter the root password and become root for as long as you need. You can also pass su a username so that you can switch to a different user account temporarily. Traditionally, when you become root, your shell prompt becomes a # symbol instead of the usual $.

```
$ su
Password:
#
```

Note that the password is not echoed back to the screen in any way, not even asterisks. Many first time users mistakenly believe that the keyboard isn't working or the OS isn't responding. Once you've finished the task that needs root access, you can and should exit the root shell by using **exit, logout** or **Ctrl+d**.

Another way of obtaining root access is through a slightly more secure system called **sudo**. sudo allows you to enter a single command with root access and without entering or even knowing the root password. You must, however, enter your own password. For example, to install the 'screen' program on an Ubuntu or Debian Linux system, you may do the following:

```
$ sudo apt-get install screen
[sudo] password for jdoolin:
```

The user's password, rather than the root password, is then entered, and if successful, the program will execute with root access and immediately return to the normal user account. While this is a nice system, it is not default on all Unix or even Linux systems. On some systems it can be installed separately, usually by the package manager. It is the default on Ubuntu systems and Mac OSX, for example, but must be installed separately on Arch Linux systems or FreeBSD.

3.3 Adding and Removing a User

The method for adding and removing users varies between different Unix systems. Some have graphical user and group management tools while others may have their own custom command line or console applications for user management.

The traditional method of adding a user on Unix systems is with the **useradd** command. useradd modifies and creates a number of files and offers the opportunity to customize the account. useradd may also behave slightly differently on various Unix systems and provide a variety of options.

Among the options available to the useradd command are those to specify the user's home directory, whether or not to create the home directory (it may exist already), their numeric user id, initial login group, expiration date (for temporary accounts), the password or the user's shell (a user may wish, for example, to use zsh instead of bash).

To create a new normal user account you can use the following command:

```
$ sudo useradd -m dstoffel
[sudo] password for jdoolin:
$
```

Note that the '-m' option is required for the home directory to be created. Without any other options specified, the command will use default settings for the account. The home directory will default to /home/dstoffel but it could be set to anything else. There are many other options for the useradd command such as an expiration date for the account, the shell program, numeric user id, group membership and any additional comments, such as the user's full name.

On some Unix systems the command may be different, such as *adduser* on FreeBSD. FreeBSD's adduser command is also interactive and prompts for the various pieces of information required to create the user account.

Aside from creating a home directory, what else is modified on a Unix system that effectively brings the user account into existence? Let's move on to find out.

3.3.1 /etc/passwd

As mentioned in Chapter 2, most Unix systems by far use plain text files for the bulk of the configuration of the system. This is also true for user accounts. The primary file that stores user account information is **/etc/passwd**. This file contains the account information for one user per line. Each line consists of seven fields separated by colons (similar to how the PATH environment variable contains paths to executables separated by colons). For example:

```
jdoolin:x:1000:1000:Jeremy Doolin, CIT Instructor:/home/jdoolin:/bin/bas
```

The fields are as follows:

1. Username - a unique username field
2. Password - this previously held the user's encrypted password, however on modern systems this is usually an 'x' or '*', indicating that the encrypted password is actually in a separate file, such as /etc/shadow.
3. User ID - the numeric identifier used by the underlying operating system
4. Group ID - the numeric identifier of the user's primary group
5. GECOS - this is a field for any additional information, such as full name, address, phone, title, email, etc.
6. Home Directory - a full path describing the location of this user's home directory
7. Shell - The full path to the program that is run any time the user logs in. This is usually a shell, such as /bin/bash, but could also be a menu driven application that restricts the user to particular tasks and does not give a full command prompt.

The /etc/passwd file is readable by all users on the system but can only be changed by root. While it is possible to edit the file manually, it is preferable to use commands to add, delete or change information in the passwd file.

3.3.2 /etc/shadow and Account Aging

As previously mentioned, account passwords were previously stored in /etc/passwd along with all the other account information, however this became a security risk, as normal user accounts could read the passwd file and obtain the encrypted password. It was then only a matter of decrypting it and using it. The solution was to save password information in a separate file that is restricted only to the root user. On most Linux and Unix systems, this file is **/etc/shadow**. On BSD systems it is /etc/master.passwd.

/etc/shadow has a very similar format as /etc/passwd, with one account's information per line, with fields separated by colons. The fields in /etc/shadow are usually as follows:

1. Username
2. Algorithm ID, salt and hashed password - combination of information required to authenticate the user at login
3. Days since last password change
4. Days until password change is allowed
5. Days before a change is required
6. Number of expiration warning days
7. Days before an account becomes inactive
8. Days until account expires
9. Reserved

As you may have noticed, /etc/shadow also contains some very useful information about account aging and expiration. User accounts may be set to expire on a certain date and policies may be put into place to enforce changing passwords on a regular basis. Some of this information is located in /etc/shadow and yet more is located in /etc/login.defs on many systems.

To enforce restrictions on password complexity you must edit the file **/etc/pam.d/system-auth**. More on this as I get time to write more of the book. But you can set things like minimum length, number of lowercase, uppercase, digits and other characters required and number of characters that must be different from the previous password.

3.4 Groups

All Unix user accounts must belong to at least one group. This group is the user's "primary group", and any files the user creates will be owned by this group (more on file ownership next chapter). However a user may also belong to multiple other groups as well. Group membership not only allows permission to read, write or execute certain files, but also often grants privileges not available to those who are not members of the group.

For example, on BSD systems, a normal user cannot use the 'su' command to become root unless they are members of the **wheel** group[1]. The wheel group also exists by default in some Linux distributions such as Arch, but wheel membership is not required to use the 'su' command. However 'sudo' can be configured so that you must belong to wheel to use the 'sudo' command.

Other groups may grant permission to use USB storage devices, optical drives, printers or view the system log.

3.4.1 /etc/group

Not surprisingly, group information is also stored in a text file, **/etc/group**. It also has the same style and format as /etc/passwd, but the fields are different. For example, on an Arch Linux system:

```
optical:x:93:jdoolin
```

The fields are as follows:

1. Group name
2. Password - privileged groups can be created that require passwords
3. Group ID - analogous to the numerical User ID
4. Group List - a list of usernames that belong to the group, separated by commas

[1]The term 'wheel' is used in reference to the slang "big wheel" to refer to someone with power or influence

3.4.2 Adding a New Group

Adding a new group requires root access and the command is rather simple: **groupadd**. You can supply arguments that specify the numeric group ID and an optional password. Group passwords allow users who are not members of a group to be granted temporary privileges supplied by the group. Think of it like 'sudo', but for group membership. This isn't used very often due to security reasons.

When the new group is created it will be empty.

3.4.3 Adding Users to a Group

There are a few utilities that can be used to add a user to a group. The most commonly available is the command **usermod**. The primary purpose of usermod is to modify user account information, however part of that information is group membership. To add a user to a new group (called a "supplementary group"), use a command like the following example:

```
# usermod -G wheel -a jdoolin
```

This will add the user 'jdoolin' to the 'wheel' group. The other command frequently found on Unix systems is **gpasswd**. Think of this program as one that modifies the *group* information rather than the user information. To add a user to a group using gpasswd:

```
# gpasswd -a jdoolin wheel
```

Notice that the command syntax is opposite for usermod. The -a argument is followed by the username to be added to the group, then the group name that is being modified.

3.5 Modifying or Removing Users and Groups

As previously mentioned there are multiple commands for adding users to a group. These same commands, usermod and gpasswd, may also be used to modify users and groups in other ways.

usermod may be used to change the home directory (as well as moving the contents of their current home directory to the new location), the shell program, numeric user ID, primary group ID, password, expiration date and even lock or unlock the account. See the usermod man page for all of the options, but here is an example:

```
# usermod -d /Users/jdoolin -m -s /bin/zsh -u 5000 jdoolin
```

This command will change the home directory location, move the contents of the current home directory, change the shell and user ID of the account 'jdoolin'.

gpasswd can be used to add or remove a user from a group, add or remove a password for the group and a few other tasks.

Removing users is accomplished with the **userdel** command. The command is simple and has only a few optional arguments. An example usage would be as follows:

```
# userdel -rf jdoolin
```

This command will remove the jdoolin account from the system as well as their home directory and all of its contents, which is the reason for the '-rf' options.

The command to remove a group is even simpler:

```
# groupdel lenders
```

This removes the 'lenders' group from the system.

3.5.1 Viewing user and group information

To view information about a user account as well as group membership, the **id** command may be used. For example, on my Arch linux laptop, id shows the following information:

```
$ id
uid=1000(jdoolin) gid=1000(jdoolin) groups=1000(jdoolin),10(wheel),
14(uucp),93(optical),108(vboxusers),1001(lpadmin)
```

3.6 Back to sudo

Now that we've reviewed user accounts and groups, let's return to the sudo command. sudo must be configured with a (surprise) text file, **/etc/sudoers**. On BSD systems that install from their ports software system, it may be located in /usr/local/etc/sudoers.

Chapter 4

UNIX Permissions

As mentioned in the first chapter, one of the reasons for developing Unix in the first place was to establish a timesharing operating system that multiple people could use simultaneously. Each user was given their own directory on the system separate from others. A natural consequence of this was the necessity for a permissions system that prevented users from accessing unauthorized files. It certainly wouldn't be ideal if one user could overwrite the resume of another, or for an unprivileged user to access sensitive operating system files that should only be accessible by root.

4.1 File Ownership

Files (including directories) on Unix systems are "owned" by a particular user and group. Files created by a particular user are usually owned by that user and their primary group by default. Processes, that is running programs and applications, even those such as ls, cat, sort and export, are also owned by a particular user. There is a command that can be used to see which user account you are currently using called **whoami** that can be used to demonstrate this. whoami reflects the user account that owns the process. It will be different if you use sudo to run whoami:

```
$ whoami
jdoolin
$ sudo whoami
[sudo] password for jdoolin:
root
$
```

The first command reports back jdoolin, while running whoami under sudo reports that I am root. This is because sudo causes the subsequent process to be run under the ownership of root. Another way to see who owns the processes running on a system is to use the **ps -aux** command.

To be more specific, files are owned by the same user and group as the *process* that created them. So if you use nano to create a new text file, it will be owned by your user account. If you were to use sudo nano, it would be owned by root because the nano process would be running as root.

To view file ownership and other file permissions you can use the **ls -l** command. This will show the user and group that owns the file as well as the other permissions settings that we'll be discussing shortly.

```
$ ls -l
-rw-r--r--  1 jdoolin  cit       4625 Mar 12 17:16 midterm.zip
drwx------  4 jdoolin  jdoolin    512 Feb 26 14:40 old_midterm
-rw-r--r--  1 jdoolin  jdoolin   4265 Feb 26 14:41 old_midterm.zip
drwxrwxr-x  2 jdoolin  cit        512 Feb 26 18:39 old_results
drwxrwxr-x  2 jdoolin  cit        512 Mar 12 18:40 results
```

The above entries are all owned by the jdoolin user, while a few of them are owned by the cit group (of which jdoolin is a member).

4.2 Read, Write and Execute

Files and directories have three permissions: **read, write** and **execute**.

The read permission allows the contents of a file to be accessed. For directories it allows only the names of files in the directory to be known, but not any other information about the files therein, such as file size, ownership or contents of the files.

The write permission allows the contents of a file to be modified. For directories it allows one to create, delete or rename files, but it also requires execute permissions.

The execute permission allows a user to run the file as a program. This includes binary executable files, such as /usr/bin/firefox or even script files. For directories it allows a user to be able to change to that directory and access the file information.

These three permissions can be seen with the "ls -l" command:

```
$ ls -l
-rwxr-xr-x 1 jdoolin jdoolin 264 Mar 19 17:15 install.sh
```

In the above ls output you can see the file permissions to the far left. In this case they are -rwxr-xr-x. 'r' means the read permission is set, 'w' means the write permission is set and 'x' means the executable permission is set. A dash ('-') means that a permission is not set. If you notice, however, there appear to be three sets of permissions.

4.3 User, Group and Everybody Else

The permissions of a Unix file are separated into three classes of users: the user that owns the file, the group that owns the file and anybody who isn't that user or in that group (aka, "everybody else"). These permissions are listed *in that order*. So let's take the permissions from that last command and examine them more closely.

For now, let's ignore the very first dash in the permissions and focus on the remaining nine. We see:

```
rwxr-xr-x
```

Separated into the permissions for each class we have:

User	Group	Everyone Else
rwx	r-x	r-x

The user permissions show that the user that owns the file (jdoolin in this case) has permissions to read the file, write to it and execute it as a program. Anyone who is a member of the group that owns the file (cit in this case) has permissions to read the file and execute it as a program, but *not* to write to the file. Everybody else on the system (anyone who is *not* jdoolin or a member of the 'cit' group, can also read and execute the file, but not write to it.

4.4 Changing Ownership

It is occasionally necessary to change the ownership of a file, directory or a directory *and* its files. Changing the user that owns a file can typically only be done by root. Changing the group ownership can be performed by the user who owns the file, but only to a group to which the user belongs. For example, if jdoolin is *not* a member of the wheel group, jdoolin may not change group ownership of a file to wheel. If, however, he is a member of the cit group, he may change the group ownership to cit. The command for changing ownership is **chown**. It has the following syntax:

```
chown  username:groupname  filename
```

where *username* is the user to which you would like to change ownership and *groupname* is the group to which you would like to change ownership. Take the following example:

```
$ ls -l unix_book.tex
-rw-rw-r-- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
$ chown jdoolin:admin unix_book.tex
chown: changing ownership of "unix_book.tex": Operation not permitted
$ chown jdoolin:cit unix_book.tex
$ ls -l unix_book.tex
-rw-rw-r-- 1 jdoolin cit 2268 Mar  4 22:20 unix_book.tex
```

ls -l shows that the file is owned by the user jdoolin and the group jdoolin. The first attempt is to change the group ownership to the admin group, to which jdoolin does not belong. Thus, the chown command fails. The second attempt changes the group ownership to cit, to which jdoolin does belong and therefore succeeds. The second example will change user ownership:

```
$ ls -l unix_book.tex
-rw-rw-r-- 1 jdoolin cit 2268 Mar  4 22:20 unix_book.tex
$ chown dstoffel:cit unix_book.tex
chown: changing ownership of "unix_book.tex": Operation not permitted
$ sudo chown dstoffel:cit unix_book.tex
[sudo] password for jdoolin:
$ ls -l unix_book.tex
-rw-rw-r-- 1 dstoffel cit 2268 Mar  4 22:20 unix_book.tex
```

The first chown attempts to change user ownership to dstoffel. jdoolin does not have permission to do this. The second attempt uses sudo to gain root privileges to change ownership, which succeeds.

Note that it is also possible to use a different syntax with chown if you merely need to change user ownership or group ownership. For example:

```
$ chown dstoffel unix_book.tex
$ chown :cit unix_book.tex
```

The first command will change user ownership while the second will change group ownership without specifying the other unnecessary information.

4.5 Changing Permissions

There are two ways of changing permissions on a file. The command for both is the same: **chmod**. The original way uses octal permissions modes, while the newer way uses a completely different syntax.

4.5.1 The Old Way

As mentioned previously, setting permissions the old way (but not necessarily worse way) involves translating the desired permissions to an octal number. You will need three of these numbers to set the full permissions: one for user permissions, one for group permissions and one for everybody else. Let's return to the previous example:

User	Group	Everyone Else
rwx	r-x	r-x

Again note that this set of permissions is divided into the three user categories. Each one of these categories will require a digit from 0 to 7 to properly set the three permissions (read, write and execute). Examine the following chart to see how to calculate the digit:

Read	Write	Execute
4	2	1

If you wish to set the file to be readable, add 4. If you wish it to be writeable, add 2. If you wish the file to be executable, add 1. Observe the following examples:

Read	Write	Execute	Formula	Mode
4	2	1		
yes	yes	no	= 4 + 2 + 0	= 6
yes	no	no	= 4 + 0 + 0	= 4
yes	no	yes	= 4 + 0 + 1	= 5
no	no	no	= 0 + 0 + 0	= 0

Repeat this formula for each of the three user classes: user, group and everyone else. Say for example you would like the user who owns the file to be able to read and write to the file, you would like the members of the group to only have read access, but everyone else to have no access at all. If you look at the above chart, you can see this corresponds to a 6 for the user, 4 for the group and 0 for everyone else. Thus, our chmod command would be as follows:

```
$ ls -l unix_book.tex
-rw-rw-r-- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
$ chmod 640 unix_book.tex
$ ls -l unix_book.tex
-rw-r----- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
```

The first ls -l shows that the permissions are almost what we want, with the exception that the jdoolin group has write access. We would like to remove this write access, which is accomplished with the 4 mode for group permissions in the chmod command.

4.5.2 The Recent Way

A more recent way of changing permissions is by using abbreviations. The syntax is as follows:

```
chmod [userclass][+/-][permission]
```

The *userclass* is one of the following:

- **u** - the user who owns the file
- **g** - the group that owns the file
- **o** - others/everyone else
- **a** - all user classes (same as ugo together)

The permission is one of the following:

- **r** - read permission
- **w** - write permission
- **x** - execute permission

The +/- is for whether you want to add or remove the permissions for the user class. Let's look at the previous example using this method:

```
$ ls -l unix_book.tex
-rw-rw-r-- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
$ chmod g-w unix_book.tex
$ ls -l unix_book.tex
-rw-r--r-- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
```

The chmod command uses "g-w" to remove write partitions from the group owner. These can also be combined. Let's say we would like to give write permissions back to the group and to everyone else as well:

```
$ ls -l unix_book.tex
-rw-r--r-- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
$ chmod go+w unix_book.tex
$ ls -l unix_book.tex
-rw-rw-rw- 1 jdoolin jdoolin 2268 Mar  4 22:20 unix_book.tex
```

Using "go+w" translates to "add write permissions for group owner and others". This works well for simple changes, but for more complex changes it requires multiple statements. For example, if you would like to remove write permissions for everyone and add executable permissions for the user, the command would be as follows:

```
$ ls -l backup.sh
-rw-rw-rw- 1 jdoolin jdoolin 238 Mar  1 12:20 backup.sh
$ chmod o-w,u+x backup.sh
$ ls -l backup.sh
-rwxrw-r-- 1 jdoolin jdoolin 238 Mar  1 12:20 backup.sh
```

Which method you use is entirely up to you, though it is best to know the numerical method as it is most likely to be available on all Unix platforms.

4.6 setuid and setgid

There is one more aspect to chmod that hasn't been mentioned thus far, and that is the ability to set a program to always run as a particular user or group, even if the account running the program is not that user or in that group. For example, you can set a program to run as user root and group root even if you are not root yourself, or even have the root password or sudo access. This is called the **setuid** and **setgid** permissions. Like other permissions they are set with chmod. Instead of using three digits to set the permissions, you must use four. The values for this digit are 4 for setuid and 2 for setgid, and the digit used is the first one, followed by the usual three. Here is an example:

```
 $ ls -l backup.sh
-rwxrw-r-- 1 jdoolin jdoolin 238 Mar  1 12:20 backup.sh
 $ chmod 6764 backup.sh
 $ ls -l backup.sh
-rwsrwSr-- 1 jdoolin jdoolin 238 Mar  1 12:20 backup.sh
```

Notice that the permissions now change to display an 's' in the user permissions and a 'S' in the group permissions, indicating that this program is now setuid and setgid for its user and group owner. This program will now *always* run as user jdoolin and group jdoolin, even if a different user executes the script.

Chapter 5

Installing Software

While many Unix systems and Linux distributions come with a lot of very useful software and tools, you will at some point be installing additional software. There isn't a standard method for installing software in Unix/Linux and it varies by the individual OS or even Linux distribution.

A lot of software is available as binary packages built for specific Linux distributions or other operating systems. As much of the software that's available is also open source, it can also be installed by compiling this source to a binary format. What this means is that the original source code written for the application is translated into the executable format for the operating system.

Many modern systems also offer a package management system that connects to online software repositories, tracks software updates and patches, installs necessary dependencies and more.

5.1 Binary Packages

Users familiar with installing Windows applications are most familiar with binary packages. For Windows these files are often in the form of installer files, usually with a .exe or .msi file extension. These files contain all of the application executables and binaries, resources and documentation, and also take care of things such as updating menus, creating desktop icons and registering the application.

Many Linux distributions and Unix systems offer similar installer files. There are many package formats found in the various distributions, but I'll discuss the most common ones.

5.1.1 RPM

RPM is the Redhat Package Manager, obviously developed by the maintainers of the Red Hat Linux distribution. It has also been ported to other Linux distributions and even other Unix systems, such as IBM's AIX. Distributions that use the RPM installer format are often said to be "RPM based".

The files end in a .rpm file extension. RPM files contain a compressed archive of all necessary files and documentation, and when the package is installed, information about the software is saved in a database. The .rpm file also contains references to any additional software that is required to be installed first. This is called a **dependency**, because the package is dependent on another package that must also be installed. The files can also be digitally signed, which is a way of verifying that the package you are installing is not corrupt or tampered with.

Because RPM files are binary, they must be built for a particular CPU architecture, such as x86 (32 bit Intel/AMD), amd64 (64 bit Intel or AMD), ppc, sparc, arm or other architecture. If you cannot download an RPM for your architecture, you will have to build it from source.

RPM allows packages to be removed and updated and the RPM database can even be queried to see which versions of certain packages are installed or which files are installed by certain packages.

In days of yore, one may have to download RPM files individually from a source such as http://www.rpmfind.net. It would also be necessary to download any dependencies not already installed. This was a tedious and messy process. Later, package managers would solve this problem.

5.1.2 Debian

Debian GNU/Linux has a different binary package format of its own. The files end in the .deb extension and can be managed with the **dpkg** utility. These installer files offer the same advantages as the RPM format, however for many years, Debian packages have been able to be managed with the **apt** package manager, to be discussed shortly.

5.1.3 OSX dmg and pkg files

Mac OSX usually distributes installers either as .pkg binary installer files, or as .dmg disk images. The .pkg file is the format recognized by the OSX Installer program, which originated in the NeXT Operating System, a Unix variant that was the precursor to OSX. .pkg files can be customized to prompt the user for information, change welcome messages, display README files or documentation or require license agreements.

.dmg files are Apple Disk Images. When the file is opened, it is mounted as if it were a hard disk. At this point it can be browsed like a filesystem. From here you may be required to drag an application icon to the Applications directory or open a .pkg file.

5.1.4 .tar.gz binary distributions

Some software is distributed as binaries in a .tar.gz file. This is a compressed archive similar to a Windows .zip file. This file, when extracted, contains the complete directory structure necessary to run the application, very similar to a Windows application distributed in a .zip file. These binaries aren't made for any specific Linux distribution, but are still compiled for particular architectures, so make sure you have the right one.

These files may be extracted anywhere, as long as you know the path to the executable, but some common places to extract them are /usr/local and /opt.

One example is the Android ADT development IDE based on Eclipse. This is distributed as a single compressed archive rather than as RPM, .deb or other installer package.

5.1.5 .tar.gz source distributions

Some software is only distributed as source code in the form of .tar.gz compressed archives. It is also often provided as an alternative to binary packages such as .rpm or .deb files. Some prefer to compile from source as it provides more of a guarantee that it will function with currently installed libraries and allows the user to customize the installation.

Because the files are source code (most often C or C++), a compiler for the language in which the application is written is required. For C and C++ applications, this most often means gcc or clang/llvm must be installed. It is also often necessary to install other build utilities such as autotools, m4 and make. Some distributions provide these in a single package, such as *build-essentials* in Debian/Ubuntu.

The most common commands required for installing from source are as follows, assuming the package is called "application" and is version 1.0:

```
$ cd application-1.0/
$ ./configure
[lots of output here]
$ make
[lots more output]
$ make install
[even more output]
```

The first step simply changes to the source code directory that's created after extracting the .tar.gz file.

The second step allows the user to configure the installation, such as destination directories, options to enable or disable or support for various features. "configure" is a script that resides in the present working directory. This is why the ./ is required before the name of the file. Without any arguments, configure will use default settings.

The third step, "make", reads what is called a Makefile and performs the actual compiling from code to binary formats. Finally, "make install" runs the make program again, but with the argument to install the software.

However, if the software is being installed on a RPM, Debian or other system with a package manager, the software installed from source in this manner will *not* be registered in the official package database. To remove it, you will either have to manually remove files or hope that the maintainers provide an uninstall option in the Makefile.

Some distributions provide tools to install source packages under the package management system, such as Debian's "checkinstall".

5.2 Package Managers

Binary distributed packages and installers are convenient, but resolving dependencies and finding and downloading all the packages is tedious. It's also possible that not all RPM or .deb files provided by software maintainers will work well with the distribution they are designed for. If, however, the packages are maintained by those who also maintain the distribution, they should theoretically work much better.

Package managers were developed to resolve these issues. The distribution maintainers (the Red Hat people, Ubuntu people, etc) build an official online **repository** of packages that have been built specifically for their distribution, tested and are updated regularly. The package manager software can then automatically download packages, resolve dependencies by downloading any other package that is required (any any required by those, and so-on), the packages are digitally signed so they can be verified and they are tracked in a database that can be queried.

Some software may not make it into official repositories for a distribution, but it is usually possible to add additional repositories to the package manager.

There are several *major* benefits to using software repositories.

1. **automatic downloading** - You don't have to browse the web to find the software. You need only know the package name.
2. **dependency resolving** - Automatically fetches all software needed by the package
3. **updating** - Software repositories update frequently, allowing you to have current software versions
4. **querying** - You can search the repository, find which package a file comes from, see what is installed, etc.
5. **digital signatures** - software is verified and digitally signed so you can ensure integrity and prevent malware

Digital signatures alone make the spread of malware nearly impossible on Unix systems that use software repositories. It doesn't mean there won't be unintentional security vulnerabilities in software that gets installed, but intentionally malicious software is practically non-existent in the Linux/Unix world for this reason.

5.2.1 yum

yum is the package manager most often used on RPM based systems such as Red Hat, Fedora and CentOS. Installing a package via yum is done with a command such as:

```
$ sudo yum install fpc
```

This will automatically download the Free Pascal Compiler from the Fedora/RedHat repositories along with any dependencies (software that fpc needs to have installed first).

5.2.2 apt

apt is the package manager most often used on Debian based systems such as Debian itself and Ubuntu and its variants, such as Raspbian on the Raspberry Pi. The example installation is as follows:

```
$ sudo apt-get install fpc
```

This will perform the same tasks as the yum command above, but for Debian based systems.

5.2.3 BSD ports

The BSD systems have a package management system that is also entirely based on compiling from source. In FreeBSD and OpenBSD it is called **ports** and in NetBSD it is called **pkgsrc**. Even Mac OSX (which has FreeBSD code as part of its core) has a ports tree called **MacPorts**. This system is installed to the hard drive as a directory tree, usually called the ports tree. To install a package, you cd to the necessary directory on the system and type ″make install″. So for example:

```
$ cd /usr/ports/lang/fpc
$ sudo make install
```

This will download the source code package for the Free Pascal Compiler (fpc), as well as any other packages that it may depend on, then automatically compile the source code into the executables and then install them.

The advantage of the ports tree is that all of your software is custom compiled for your particular computer, its architecture and installed software. It also has the advantage of being maintained by the operating system maintainers, it resolves dependencies, allows customization and the other advantages of binary package management systems.

The BSDs do still have a binary package system that can be used with what is called the **pkg tools**. On FreeBSD, for example, you can issue the following command:

```
$ sudo pkg install gprolog
```

This will download a *binary* package for the GNU Prolog programming language that does not need to be compiled, and install it automatically.

5.2.4 pacman

Arch Linux uses its own package manager called **pacman**. To install a program using pacman, use the following command:

```
$ sudo pacman -S gprolog
```

5.2.5 OSX App Store

Many applications for OSX are offically distributed on Apple's App store, such as the popular Garage Band software. You do not necessarily have to use the App Store, however. Many applications can simple be installed with installer packages downloaded from the official websites.

Chapter 6

Linux On The Desktop

Thus far we have explored Linux and Unix as purely command driven operating systems. While it is immensely capable and preferable for servers and other headless systems, command line only is not ideal for the desktop environment.

It is no secret that Microsoft's Windows dominates the desktop market both at home and in the office, but Linux (and other Unix systems) is capable of providing a full desktop environment with all of the programs and applications one would expect, such as web browsing, office software, chat, audio, video, software development and an increasing number of games.

Many operating systems have their main graphical interface integrated with the operating system. The case is very different with Linux. It implements the GUI in separate software packages and in multiple layers.

6.1 X Windows

The first layer is X Windows. X Windows is a windowing system that began development in the mid 80's at the time Microsoft and Apple were also working on their own graphical systems. There are many implementations of the X Window system but the most common are X.Org (the predominant implementation), XFree86 (predominant before X.Org) and X11.app on Mac OSX. Others include implementations for Android and Windows.

X Windows is very different from Microsoft's Windows GUI. X is run as a client/server application that can actually be run over the network. What this means is that a graphical application can run on one system (the X client), but it is displayed on the X server of another system. This means many different machines could connect to and use the same system with a graphical interface. This is still possible today through an SSH session and graphical display managers.

X provides the underlying graphical interface command primitives that allows an application to draw to the screen. At the very minimum, one could run an instance of X Windows from a command line, then from the same command line run multiple applications. You would have to specify things like the location and size of the window and there would not be any window decorations or ways to close, minimize, maximize or start applications from within the GUI. This is why it is very rare that X is ever run on its own. At the very least, X requires a window manager.

6.2 Window Managers

Window managers perform exactly the task that their name describes: they manage the windows that are displayed on an X Windows display. This includes adding titlebars, borders, buttons that allow one to close, minimize (or iconify), maximize and roll-up windows, as well as many other tasks. Windows can be selected, moved and repositioned with a variety of techniques.

Window managers also allow the X display to be split into **virtual pages and/or desktops.** This is in contrast to Microsoft Windows where there is only a single desktop and all applications display and run within it. Multiple pages/desktops allows the user to dedicate specific desktops to certain tasks. For example, a user may have 4 desktops, one for web browsing and chat, one for image editing, one for games and one for software development.

Window managers may provide taskbars, start menus, application menus, configuration utilities, desktop switchers/pagers, system trays, docks and more.

There is a very large number of window managers available, some more popular than others. Some are very minimal and lightweight while others offer a larger number of features but at the expense of requiring more system resources.

6.2.1 twm

twm is a very early window manager for X Windows. It is packaged with X windows so that there is always an option for a window manager, albeit very basic. It doesn't feature much more than the ability to place, move, resize, close and iconify (minimize) windows.

6.2.2 FVWM

FVWM[1] is originally based on twm. FVWM is extraordinarily flexible and customizable to the point that it can be said that FVWM does not have a look and feel of its own. It all depends on how you configure it. It essentially gives you the tools and ability to make a window manager that suits you.

FVWM does, however, require that the user spend a fair amount of time learning how it works and how to configure the system via text files. There is a lot to learn, but a lot of power in FVWM's system. It is probably best thought of as the hacker's window manager. Some FVWM users[2] spend years tweaking and refining their desktop, discovering new tricks and easier ways to improve workflow.

6.2.3 Fluxbox

Fluxbox is another lightweight window manager based on another called Blackbox. Fluxbox forked from Blackbox and added features, primarily the ability to tab windows similar to the way you can have multiple tabs open in web browsers. It supports wallpaper, themes, multiple desktops and an application menu accessible by right clicking on the desktop.

6.2.4 Enlightenment

Enlightenment (or simply 'E') is a slightly larger window manager that requires a bit more resources than the likes of Fluxbox or FVWM, but supports some interesting features out of the box. It offers multiple desktops, window grouping (so multiple related windows can be minimized, maximized or

[1]There is no official meaning for the 'F' in the acronym, but it is often said to mean "flexible", "feeble", "feline" and other odd words, but the VWM stands for Virtual Window Manager

[2]Such as yours truly

closed at the same time), more advanced theming, customizable keyboard shortcuts, a Mac OSX like dock, a built-in file manager and visual effects.

6.2.5 Windowmaker

Windowmaker is a lightweight window manager based on the NeXTStep operating system. The unique feature of Windowmaker is the use of something called "dockapps", which are small utilities that fit in a single small window. Examples are a weather app, disk usage, performance monitor, email notification and audio control. This idea would later be extended to similar features such as Windows Vista Gadgets, Gnome Widgets and OSX Dashboard Widgets.

6.2.6 Others

The window managers decribed here comprise a very tiny tip of a very large iceberg. There are literally hundreds more, from very lightweight, minimalist designs to those that can be programmed with programming languages, to window managers that emulate Microsoft Windows or the Silicon Graphics GUI.

Feel free to experiment and find the one that suits you best.

6.3 Desktop Environments

Beyond the role of the window manager is the Desktop Environment. A desktop environment includes a window manager (sometimes it can be selected by the user), but more importantly provides a set of integrated programs, tools and utilities, as well as features like desktop icons and recycle bins.

The programs provided by a desktop environment are designed to work with each other and integrate with other elements of the system like the taskbar, menus, desktop and wallpaper, system sounds and system trays. Desktop environments also provide configuration utilities not only for the desktop itself, but also for the system. It may provide a program for installing software, enabling or disabling services, setting up a network or wireless connection, managing users, adding disks, mounting external media or mounting network shares.

Because desktop environments provide so much they require substantially more hard drive space, processing power and time and memory. For some, it is worth using the extra system resources, while others prefer to optimize the usage of their resources and use lightweight window managers.

6.3.1 CDE

In the early and mid-90's there were many proprietary variants of Unix available and they all had different interfaces. As part of the Single Unix Specification, an effort was made to provide a desktop environment that would be available and common to all major Unix systems from vendors willing to participage. IBM, Sun Microsystems, HP, DEC and Novell all contributed to the effort and developed the Common Desktop Environment.

CDE was the standard Unix desktop for several years until open source alternatives matured.

CDE provided many features expected of a desktop system such as a calendar, file manager, configuration tools, a mail client and customizable colors and themes.

CDE was released as an open source project in 2012, so it is now a free option for Linux systems.

6.3.2 KDE

The K Desktop Environment[3] began development in 1996 as an open source alternative to CDE. The aim was a cross-platform, easy to use desktop that provided a suite of applications that all looked and behaved similarly. The KDE developers opted to use the proprietary licensed QT graphical libraries to build its applications.

KDE is known for its distinctly Windows-like default appearance, with a start menu, taskbar and desktop icons. Among KDE's applications are Konqueror (a web browser and file manager), k3b (cd and dvd burning), kopete (instant messaging), KMail email client and Amarok, an advanced audio manager and player.

[3]The K doesn't really stand for anything. It was chosen as wordplay on CDE

6.3.3 Gnome

There were a number of developers that liked the mission of the KDE project but were opposed to using the proprietary QT graphics library. This group opted to start their own project using their own open source graphics library, GTK+. This would become the GNOME project.

GNOME version 1 would have a very similar appearance as KDE, with a Windows-like taskbar, start menu, desktop icons, system tray and a set of integrated utilities like mail clients, a web browser, file manager, configuration tools and the ability to customize the appearance. GNOME 2 would see a slight difference in default setup, with menus such as Applications and System being a part of a taskbar at the upper part of the screen. GNOME 3 changed things rather drastically, moving from the traditional desktop model to what it calls the "Gnome Shell".

Because the GNOME Shell was not universally well received, two other projects forked from GNOME 2 and GNOME 3 codebases called MATE and Cinnamon, respectively. They both seek to provide an experience more like GNOME 2 but also have fragmented the GNOME community.

6.3.4 XFCE

XFCE started as a project to provide a free desktop environment for Linux that is similar to CDE. Early versions of XFCE were quite obviously inspired by CDE but as the software matured it began to take on a style of its own. It's new focus would be less about being inspired by CDE and more about providing a desktop that's lightweight and fast but still have visual appeal. It's a bit more lightweight than GNOME or KDE, yet still has a file manager, compositing window management (meaning it can have drop shadows and other effects for windows) and a set of built-in applications. The Xubuntu distribution is basically Ubuntu that defaults to the XFCE desktop environment.

6.3.5 LXDE

LXDE goes beyond XFCE in terms of resource requirements. It aims to be usable on computers with much less resources (particularly RAM), such as netbooks, small devices or older computers. It is faster and more lightweight than even XFCE, but not necessarily some of the more minimal window managers like Fluxbox or FVWM. LXDE still provides a desktop environment that is easy to use and has a suite of applications built-in.

LXDE is the default environment on the Lubuntu distribution as well as the Raspbian variation of Debian that's built for running on the Raspberry Pi.

6.4 Starting X Windows and Display Managers

The old way of starting X Windows was to create a hidden file in your home directory called **.xinitrc**. In this file you would set certain variables, paths and execute the window manager, desktop or applications of your choice. There are a number of other X configuration files that can be used to personalize your X Windows session, but they are beyond the scope of this book at the time of this writing. The user would log in via the command line then type the **startx** command which processes the .xinitrc file and begins the X Session.

This method can still be used but it has its limitations. First, if you decide to change window managers you have to edit the .xinitrc file, quit X Windows and restart it. Rebooting is not necessary. Second, it requires the user to login at the command prompt first.

A more popular method is to use an X Display Manager. These are typically graphical programs that present a login box, user selection, window manager/desktop environment selection and options for rebooting, shutting down or hibernating and sleep. Display managers may also allow starting an X session on a remote host.

There are a number of display managers available. **xdm** is the oldest and most basic display manager. Its only benefit is that it is very likely to be found on most machines that have X Windows installed. Otherwise there are more sophisticated display managers available. **GDM** and **KDM** are the display managers provided by the GNOME and KDE projects, respectively. They allow logins to other window managers or desktops as well, however you must install a large portion of the GNOME and KDE desktop systems, which require a fair amount of disk space. **LightDM** is a more lightweight manager that is installed independently of any other desktop environment. **SLiM** is the Simple Log-in Manager. It's small, fast, lightweight and themeable but requires a bit more work to configure.

6.5 Desktop Applications

While Linux may be more often used for servers, embedded systems, supercomputers and other non-desktop applications, Linux does provide many desktop applications that are often as good or better than proprietary versions. For most open source projects the same applications will also exist for FreeBSD, Solaris, OSX and other Unix variants.

6.5.1 Office

Both Libre Office and OpenOffice.org are available for Linux and contain all of the office software you might come to expect: word processing, spreadsheets, presentation, database, drawing and others. The interface and options may not be exactly the same as Microsoft Office but most of the usual tasks can still be performed. Calligra Office is another suite that is part of the KDE project and will integrate very well with the KDE desktop.

There are also individual office applications such as Abiword and Gnumeric.

6.5.2 Browsers

Linux offers all the same web browsers you see on other operating systems with the exception of Safari and Internet Explorer[4]. Firefox, Google Chrome, Chromium (the open source version of Chrome) and Opera all work perfectly in Linux. KDE has its own browser called Konqueror as well.

There are other options for those looking for leaner browsers, such as Dillo, Arora and Midori and even the text based web browser, elinks. These browsers come at a price, such as lack of plugins, themes and extensions. But they serve their purpose of being small and fast.

6.5.3 Audio and Video

Linux is loaded with audio software for both playing, recording and mixing. Rhythmbox, Amarok, Audacious, Banshee, Beatbox, MPD and XMMS (a clone of the popular Winamp) are all full featured audio players. They allow management of your digital audio collection, playlists, ratings and

[4]There are even hacks for running older versions of Internet Explorer through the wine emulator, but there probably isn't much purpose in that any more

favorites, album artwork, etc. Audacity is available for Linux as is Ardour and Qtractor among others for recording and mixing. There are, of course, many command line options as well for playing and recording audio.

Linux allows CD ripping and burning with programs like K3b, xcdroast, the above audio players and some very powerful command line programs like cdparanoia, lame, oggenc, flac, faac and many encoders.

mplayer, xine, VLC and Miro are four obvious video players that support a multitude of codecs and formats as well as online streams and channels.

Beyond simply playing audio and video, there is the excellent XBMC for a complete HTPC solution supported fully in Linux, including a variation of Linux for the Raspberry Pi specifically designed for XBMC.

The powerful Handbrake is available for transcoding digital video from and to many formats such as MP4, MKV, Theora, H.246, MPEG4 and MPEG2.

6.5.4 Graphics

Right away: you will not get Adope Photoshop for Linux. It's simply not going to happen any time soon and if you require the specific advanced features of Photoshop you will need to use Windows or OSX.

However, for those simply looking to edit images from the basics to slightly advanced tasks, the GIMP (GNU Image Manipulation Program) is definitely up to the task. It supports layers, filters, scripting, brushes, patterns, masking, alpha blending and many other features.

Inkscape is an excellent vector graphics drawing program, while Dia and xfig allow more traditional diagrams such as those produced by Visio.

6.5.5 Chat and VOIP

Chat is very well covered in Linux. There are a number of multi-protocol chat clients that support popular systems such as AIM, ICQ, Yahoo, MSN and Jabber, but foremost among them is Pidgin and Kopete. They support far more than those protocols mentioned.

There are also many IRC clients available, including the XChat graphical client and irssi console client. There is also a secure IRC client for the SILC protocol.

Skype is available for Linux and OSX and provides text and video chat.

6.6 Software Development

Perhaps the area in which Linux and Unix shines the most is software development. With only a few exceptions (usually Microsoft related) Linux provides a development environment for any major programming language, past or present, and includes many powerful programming editors and development environments.

6.6.1 Languages

If you can think of a language, Linux and other Unix variants will probably have a compiler for it. Here is a nowhere-near-complete list:

- C - gcc, clang
- C++ - g++, clang
- C# - Mono
- Java - OpenJDK, Oracle Java JDK
- PHP
- Perl
- Python
- Ruby
- COBOL - GNU Cobol
- Fortran - GNU Fortran (gfortran)
- Common LISP - sbcl, clisp, cmucl, Clozure CL, Allegro
- Pascal - Free Pascal, GNU Pascal
- Assembly - nasm, dasm and many others for nearly any architecture (x86, MIPS, ARM, MOS 6502, Zilog Z80, SPARC, etc)

6.6.2 IDEs

Most enthusiastic programmers in the Unix world use one of the many powerful programmer's editors available. Of those there are two that are the most used and are part of what could be considered a "holy war" between two factions: vim and emacs. vim is "VI Improved". We talked about the vi editor earlier in the book. vim operates the same as vi but has many more features including syntax highlighting and coloring, auto indentation, folding, completion, building and debugging and many others. emacs offers most of the same only perhaps more. Many jokingly say that emacs would be a great operating system if only it had a good text editor.

There are graphical editors as well, such as jedit, gedit and Sublime[5].

Beyond the editor is the full blown Integrated Development Environment, possessing graphical interface builders, project trees, menus, log viewing, build setups debuggers and more.

Eclipse is an IDE familiar to many Java and Android programmers. It is free and cross-platform, which includes Linux. While it is meant for Java it can also be used for other programming languages. It is also capable of building Android applications from start to finish, including a GUI builder. Oracle Studio and Netbeans are two IDEs from Sun Microsystems and Oracle.

KDevelop is a full IDE for developing KDE applications. Anjuta is capable of developing GTK applications and is part of the GNOME project. MonoDevelop is meant for developing C# and other .NET applications.

[5]Sublime is powerful but is also proprietary and costs money

Chapter 7

Unix Filesystems and Hard Disks

Dealing with hard disks, optical drives and other storage media are one of the most common tasks with computer and systems administration. This chapter covers many of the common utilities used to add, configure, manage and get information from hard disks and storage media on Unix systems.

7.1 UNIX Directory Structure

While not universal, most Unix systems, such as Linux, have a very similar directory structure. Some of the directories found in typical Unix systems are analogous to some found on Windows systems, but not all of them.

One thing to keep in mind is that Unix treats nearly everything as files, even access to hardware devices as we shall see.

7.1.1 /

/ is the ”root” directory of a Unix filesystem. All disk drives and partitions will be accessible within this directory. This is different from typical Windows setups where each partition or drive is designated its own drive letter. All Unix systems, at the very least, *must* have one partition on which / is mounted.

If you list the contents of /, you will see the top level directories of the system, which I will now discuss in more detail.

7.1.2 /home and /root

/home is the traditional directory that contains all user account directories or "home" directories. This is analogous to C:\Users on Windows systems. Each user account with a valid login should have a home directory within /home, such as /home/jdoolin. On MacOSX systems it is /Users.

/root is root's home directory. But why wouldn't it simply be /home/root?

It is not uncommon in Unix network settings for /home to be mounted on a network drive, such as one provided by NFS or Samba. This allows one to login on any Unix machine on a network, but still have access to the exact same content in their home directory. It is also common for /home to be mounted on a different disk or partition.

If the /home mountpoint is unavailable, perhaps due to a network problem or a failing hard disk, then the users that have data stored within /home will then be unable to retrieve it. The system is now potentially in a state that requires repair that will necessitate root access. If root cannot access his own home directory, this could cause significant problems. It is therefore a best practice to keep root's home directory on the **same partition as the root partition**, thus /root is rarely on its own partition or disk.

7.1.3 /etc

The bulk of Unix configuration files are located in /etc. Think of it as analogous to the Windows registry, but usually consists of text files. Key files are /etc/passwd, /etc/group, /etc/shadow, /etc/init.d and many, many others.

7.1.4 /bin and /sbin

Traditionally, /bin contains the primary system binaries required for basic system operation. Such programs could be ls, mkdir, rm, rmdir, cp, mv, ps, kill, bash, chmod, cat and pwd.

/sbin typically contains system binaries that require root access. These may include fdisk, fsck (filesystem check), mount and umount, ifconfig, route, or mkfs programs.

Some modern Linux systems do not use /bin or /sbin, and instead place all binaries in /usr/bin and /usr/sbin or even just /usr/bin.

7.1.5 /usr

The /usr directory is typically the largest aside from home directories. It contains the majority of the rest of a Unix system's applications and all resources they require. On Unix networks, it is common for /usr partitions to be mounted as a network share so that all users have access to the same applications that can now be managed on one server.

/usr/bin, /usr/sbin and /usr/local/bin

/usr/bin contains the majority of application programs, such as web browsers, compilers, text editors, audio and video applications and others that are needed by typical computer users. On Arch Linux, however, /usr/bin contains nearly all binaries for the whole system.

/usr/sbin usually contains system services such as sshd, web servers, mail servers, directory servers, ntpd, nfs, samba, dns and dhcp servers and others. It also contains some utilities often needed by users, such as tcpdump or traceroute.

/usr/local/bin often contains binaries local to the system or installed by means other than official package managers. /usr/local will be discussed in more detail shortly.

/usr/share

Within /usr/share you may find platform independent resources required by other programs. These aren't binary executables, but are files needed by other binaries. This could be icons and images, audio files, documentation and html, time zone information, man pages and a dictionary of common words. These files are in formats common to all operating systems and distributions.

/usr/local

Because some Unix networks mount /usr on a network share, it may be necessary for a local program to be installed that the network administrators do not wish to have on the file server. /usr/local is a place for applications to be installed on the local machine. It contains a set of directories that looks very similar to what is found in the / directory. /usr/local has its own bin, share, sbin, include, etc and other directories.

7.1.6 /var

/var contains primarily variable data, or any data or files that change frequently, such as log files, print jobs, database files, caches and lockfiles.

/var/log

/var/log is a nearly ubiquitous directory for storing operating system log files, such as the main system log, login failures and successes, server logs (mail, web, ftp), firewall logs, boot logs or errors.

7.1.7 /dev

/dev is a unique and interesting directory. It doesn't contain conventional files, such as text, image, audio, video or program data. /dev contains files that refer to devices. Some of these devices are physical, such as hard drives, the audio interface, keyboards, mice, joysticks, hard drive partitions, thumb drives, optical drives or terminals. Other /dev entries are virtual, such as a file containing all zero data, random data and the mysterious /dev/null.

/dev entries can be read from and sometimes written to. For example, the following command will read the contents of a CD or DVD ROM and send the output, bit for bit, to an ISO file:

```
$ dd if=/dev/sr0  of=/home/jdoolin/arch_linux.iso
```

This command will read a floppy disk image and write it to a disk in the floppy drive:

```
$ dd if=boot_floppy.img of=/dev/fd0
```

7.1.8 /tmp

As the name may imply, /tmp is a temporary storage location. It is used by the operating system and is often also the only other directory on a system that is writable by normal users. /tmp is usually emptied after every reboot, so do not use it as permanent storage.

7.1.9 /boot

On many Unix systems, /boot contains bootloader and bootloader configuration data, as well as the operating system kernel. It is vital to the boot process.

7.1.10 /lib, /usr/lib, /lib64

Those familiar with Windows systems may be aware of .DLL files. These are Dynamic Linked Libraries that provide functionality for a variety of other programs. For example, a programmer doesn't need to write their own code for accessing the network because there will already be a library (.DLL) file for it.

In Unix, /lib and /usr/lib contain most of the libraries used by the operating system and applications. On Linux, these files are typically .so files.

7.1.11 Other common directories

/opt

/opt is another directory that is self describing. It is an optional directory that can be used when no other directory is quite appropriate. This is usually in the form of locally installed, self contained application directories, such as those needed by google chrome, eclipse, android SDKs or other programs that extract to their own directory. /usr/local can also be used for this purpose.

/proc

Like the /dev directory, /proc doesn't contain actual data files. It is a virtual filesystem containing virtual files that have information about running processes and the kernel.

/mnt and /media

/mnt and /media are normally used for accessing removeable media or disks that are separate from the main operating system. This inludes thumb drives, optical drives or hard drives with other operating systems installed. Mountpoints may be permanently created, such as in the case of other hard drives or partitions within the system, but can also be created and deleted automatically, such as with USB, SD or other flash media. The new directory will appear in /mnt when the drive is mounted, but then will disappear when the drive is removed.

7.2 Hard Disk Related Commands

This section specifically covers some useful utilities related to disks on Unix systems.

7.2.1 df

df (for "disk free") is a command that will show the current disk usage of *all mounted filesystems*. Note that it will not report usage for any partition or disk that is not mounted. Example output:

```
$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
/dev/sda1       76765216 18338260  54504372  26% /
/dev/sda3       40958972 28670548  12288424  70% /home
$
```

The default output uses 1 kilobyte block counts, which may not be immediately readable by many users. The "-h" command option enables "human readable" output as such:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        74G   18G   52G  26% /
/dev/sda3        40G   28G   12G  70% /home
$
```

The command lists the filesystem block device as listed in /dev, the total size of the partition, how much is used, how much is available, percentage used and the mountpoint on which the partition can be accessed.

On many Linux systems you may also see entries for any dev, run and tmpfs filesystems also currently in use. These do not refer to physical disks, but virtual filesystems.

7.2.2 du

To find how much disk space is used by a file or directory, the **du** command ("disk usage") command may be used. Like df, du also has the "-h" option available. Another common option is "-c" to show the total of all files and directories listed in the command. Here is an example:

```
$ du -h unix_book
4.0K    unix_book/exroot/one/another
8.0K    unix_book/exroot/one
4.0K    unix_book/exroot/programs
77M     unix_book/exroot
78M     unix_book
$ du -hc unix_book
4.0K    unix_book/exroot/one/another
8.0K    unix_book/exroot/one
4.0K    unix_book/exroot/programs
77M     unix_book/exroot
78M     unix_book
78M     total
```

This command shows the disk space used by the unix book directory. This is a useful command for finding which directories and files are using the most space on a filesystem. For very long file and directory listings, consider redirecting the output to a file. Also consider using grep if you are searching for specific file or directory names within the output.

7.2.3 fdisk

While there are several disk partitioning options (such as the graphical utility **gparted**), fdisk is the ubiquitous program found on most Unix systems. Note that fdisk usage requires root privileges. fdisk may be used to list available disks and partitions with the "-l" option, but more importantly, may be used to edit the partition table of a disk, including adding, removing or changing partition information.

fdisk may be launched by simply supplying it a hard disk node in /dev, such as /dev/sda. Once fdisk is launched it uses a menu driven system that prompts the user for information. Example:

```
$ sudo fdisk /dev/sdb
[sudo] password for jdoolin:

Welcome to fdisk (util-linux 2.24.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help):

Help:

  DOS (MBR)
     a   toggle a bootable flag
     b   edit nested BSD disklabel
     c   toggle the dos compatibility flag

  Generic
     d   delete a partition
     l   list known partition types
     n   add a new partition
     p   print the partition table
     t   change a partition type
     v   verify the partition table

  Misc
     m   print this menu
     u   change display/entry units
     x   extra functionality (experts only)

  Save & Exit
     w   write table to disk and exit
     q   quit without saving changes

  Create a new label
     g   create a new empty GPT partition table
     G   create a new empty SGI (IRIX) partition table
```

```
   o   create a new empty DOS partition table
   s   create a new empty Sun partition table
```

From here the user may choose various options and will be prompted for further information depending on the option chosen. The most common tasks will be deleting and adding partitions, printing the current partition table, viewing partition types and either quiting without saving changes or writing the table to disk and exit. There will be a more detailed example later.

7.2.4 mkfs

When a new partition has been created it must then be formatted with a filesystem. In the Windows world, the common filesystems are NTFS and FAT32. A filesystem is the way the operating system organizes the data on a hard drive. It also enables other features such as permissions, encryption and compression, journals that enable better recovery and other features.

In Linux systems, the command for creating a new filesystem typically begins with **mkfs** followed by a dot (.) and a filesystem name. Examples would be mkfs.ext4, mkfs.ext2, mkfs.reiserfs, mkfs.xfs, mkfs.ntfs and mkfs.vfat. Other systems, such as FreeBSD may have their own unique command such as 'newfs'. Always read the documentation specific to your operating system or distribution.

The mkfs command requires at least one argument, and that is the partition itself, as it is listed in /dev. For example:

```
$ sudo mkfs.ext2 /dev/sdb1
```

This will create a new ext2 partition on the /dev/sdb1 partition.

7.2.5 mount and umount

Even if a partition has been formatted, it still cannot be accessed by applications until it is **mounted**. In other words, the partition becomes attached to a directory in the Unix / file hierarchy. That directory could be / itself, /home, /usr, /var, /tmp, /mnt/windows, /media/THUMB-DRIVE or even something like /home/jdoolin/media/music.

The mount command requires a partition as it is listed in /dev, a mountpoint directory and often a partition type specified by the "-t" option.

There may also be a number of options that can be passed for a filesystem, such as read-only support. Here are a few examples:

```
$ sudo mount /dev/sda1 /
$ sudo mount /dev/sda2 /home
$ sudo mount -t ntfs /dev/sdb1 /mnt/windows -o remount,ro,noatime
```

An exception for the mount arguments is found when a mountpoint is listed in /etc/fstab. For example, if there is an entry in /etc/fstab to mount /home on /dev/sda2 and another to mount the cdrom drive (/dev/sr0) on /media/cdrom, then they could be mounted without specifying the partition. Only the mountpoint is required:

```
$ sudo mount /home
$ sudo mount /media/cdrom
```

It may also be necessary to unmount partitions. When doing so, the command is **umount** (that is NOT a typo, it is umount and not UNmount) and the only option necessary is the mountpoint for the partition you wish to unmount.

```
$ sudo umount /media/cdrom
```

7.2.6 fsck

fsck (for FileSystem ChecK) is analogous to the chkdsk command in Windows. It is used to check for and repair errors in the filesystem. fsck is often forced after a certain number of days or reboots. It is also a good idea to use fsck when any filesystem or hard disk errors become apparent. To check a filesystem simply type a command such as this:

```
$ sudo fsck /dev/sda2
$ sudo fsck /home
```

Note that you can pass either a device name or a mountpoint to fsck.

7.3 Device Nodes

7.3.1 /dev Devices and Partitions

On most Unix systems, hard drives and partitions will appear in the /dev directory as "block devices", which basically means any storage device that accesses its data in chunks of bytes or bits called "blocks". The naming conventions for block devices varies between operating systems. In Solaris, for example, a hard disk may be listed with a name similar to /dev/dsk/c0t2d0s3 (meaning controller 0, target 2, logical unit 0 and slice/partition 3). FreeBSD disks may be named similarly to /dev/ad0s1 (disk 0, slice/partition 1). In Linux, hard disks typically have the format /dev/sda, where the 'a' is the disk id. If you have two disks, you may see /dev/sda and /dev/sdb. Thumb drive inserted would add /dev/sdc. Each partition is then listed as a number appended to the disk id. So the first partition on /dev/sda would be /dev/sda1. The second partition on the second disk would be /dev/sdb2 and so-on.

SD cards may appear as /dev/mmc0blk1.

7.4 Various Filesystems

While in modern Windows systems you are limited to using NTFS and FAT32 for hard disks, in the Linux and Unix world your choices are much more broad. For the average user, the choice of filesystem doesn't mean much or warrant much thought. But for those managing high performance systems or have specific needs, there may be filesystems available that are more appropriate for the task at hand. The following are some of the filesystems currently available for use in Linux.

7.4.1 ext2, ext3 and ext4

ext2, ext3 and ext4 have been the closest to a default filesystem in Linux. ext2 was the first filesystem native to Linux and it was used as the default in many Linux distributions. ext3 added many features to ext2, but of most importance is journaling, which is the ability of the filesystem to track its reads and writes, thereby minimizing damage during improper shutdowns or other possible filesystem problems. ext4 is basically ext3 with a few more improvements.

All three are good general purpose filesystems though ext2 is still preferred for flash media such as solid state drives or USB storage. Journaling is sacrificed for speed and fewer writes. For most users, ext4 will be just fine.

The ext2,3,4 systems experience very low levels of fragmentation, so the need for a defragmentation utility is next to nil.

7.4.2 btrfs

btrfs is widely considered to be the next standard filesystem for Linux, particularly for high performance systems. It is well suited for server and supercomputing environments and supports more advanced features such as resource pooling, snapshots of files and compression.

7.4.3 XFS

XFS was developed by Silicon Graphics (SGI) for use in their Irix Unix system. It was developed with one thing in mind: very large files. SGI workstations were used for developing and rendering 3D graphics, games and movies and thus required very large file support. Being able to read and write large files as quickly as possible was of vital importance, so XFS supports this very well.

If you find yourself in a similar situation, where large media files will be a big part of your system and work, then XFS would be a good choice. Note that you will sacrifice some hard drive space for the filesystem structures themselves, but you will gain performance.

7.4.4 Reiser4

Reiser4 is more adequate for filesystems containing numerous small files such as logs, man pages, docs, icons, etc. This makes it more suitable for directories such as /usr or /var.

7.4.5 JFS

JFS is a project by IBM tarteting their AIX Unix system and eventually the OS/2 operating system, but was ported to Linux. It is a journaling filesystem, meaning it survives unintentional shutdowns very well. Many claim that it uses less CPU resources and that it is an adequate system for numerous small files. This may also be an adequate system for interoperability between Linux, AIX and/or OS/2.

7.4.6 ZFS

ZFS was created by Sun Microsystems (which was purchased by Oracle) as a highly advanced, scalable and reliable filesystem. Its license is not compatible with the GNU Public License that the Linux kernel falls under, so it cannot be distributed with the Linux kernel. It can, however, be installed as a third party kernel module (driver) and be used in Linux. However, FreeBSD natively supports a free implementation of ZFS.

ZFS protects against data corruption, very high storage capacity, is scalable (meaning ZFS filesystems can be easily adjusted dynamically without having to create and re-create ZFS partitions), supports Access Control Lists for security, filesystem level compression and disk volumes without an additional layer, such as LVM.

7.4.7 NTFS and FAT32

NTFS and FAT32 are familiar to those in the Windows world. It is possible read and write to these filesystems in Linux, which means you can use live CDs and bootable Linux USBs as data rescue systems. You can also use Linux to create these filesystems for use on Windows systems.

7.5 How To Add a Hard Drive

7.5.1 Install the Hard Disk

First add the physical disk to the system and verify through the BIOS that the drive is working and recognized by the system.

7.5.2 Find the Device Node

When Linux boots it should find the new disk and create a new entry or entries in /dev. If you have only one disk, /dev/sda, the next will likely be /dev/sdb, then /dev/sdc and so on. If this disk has partitions already, they will also be present in /dev. You can use the **dmesg** command or simply ”ls /dev/sd*” to see the new entry in /dev.

7.5.3 Partition the Disk

Now repartition the disk if necessary. Use fdisk on the disk entry in /dev:

```
$ sudo fdisk /dev/sdb
```

Delete any existing partitions and create the new ones you wish to use using fdisk’s menu system. When finished, write the partition table to disk and exit.

7.5.4 Format the Partition(s)

You must now use an mkfs command to format any of the partitions you created on the new disk.

```
$ sudo mkfs.ext2 /dev/sdb1
```

7.5.5 Mount the Partitions

Finally, you can now mount the formatted filesystem on the mountpoint of your choice:

```
$ sudo mount /dev/sdb1 /home/jdoolin/media
```

7.5.6 /etc/fstab

If you wish this mountpoint and filesystem access to be persistent across reboots, you need to add an entry to fstab. The format is mostly the same between Unix variants. However, while some Unices still use device nodes such as /dev/ad0s3, Linux requires the UUID of the partition. First find the UUID of the filesystem using the **blkid** command as root:

```
$ sudo blkid /dev/sda1
[sudo] password for jdoolin:
/dev/sda1: UUID="51dfcc2e-b71c-4357-b3e7-2f8878f5f19c" TYPE="ext2" PARTUU
```

You can then use this UUID to add a line to /etc/fstab using your favorite text editor. Here is an example from my Linux workstation:

```
UUID=216370fe-d3fd-425d-9e73-82aed041dce0  /home ext4  defaults
0 2
```

The information is separated by whitespace (tabs and spaces). The first field is the UUID of the partition. The second is the mountpoint where the partition will be accessed. The third is the filesystem type (ext4, vfat, btrfs, xfs, etc). The fourth is any arguments being passed to the mount program. The fifth field is whether or not the filesystem information should be dumped and the final field is the order in which the disk should be checked. The root system, /, should be 1 and all others 2.

Chapter 8

System Administration

8.1 Archives and Compressed Files

In any operating system it is a common task to work with compressed and archive files. In the Windows world these are usually in the .zip, .rar and .cab formats. While these files can be created, extracted and worked with in Unix, they are not as common as some others, particularly for software distribution. It is therefore important to know about these formats and how to work with them. Using .zip, .rar and .7z files in Unix also usually requires the installation of third party software, but this is typically as simple as installing from a repository (see Installing Software).

8.1.1 tar

tar is a very common archive file format. In and of itself, it does not perform any compression. tar was originally and can still be used for creating backup tape archives (hence, the name: "t"ape "ar"chive). But more commonly today tar is used to create simple archive files. This allows one to create one file that contains multiple others (including other tar files).

To create a tar file, use a command with the following structure:

```
$ tar -cvf destination.tar  sourcefile1 sourcefile2 sourcefile3 ...
```

This will create a new file called destination.tar that contains all of the source files listed after it. The source files may also be directories. For example:

```
$ tar -cvf music.tar Music/
```

This will create a tar file called music.tar that contains the Music directory and all of its contents. For those curious, the arguments mean the following: c - create, v - verbose (show information) and f - destination file.

As always, see the tar man page for complete documentation.

8.1.2 gzip, bzip2

Creating an archive is not always enough. While Windows users are accustomed to normally using the ZIP file format, which compresses and archives, tar files are not compressed. This allows the Unix user to choose the compression utility. While there are many different compression algorithms and tools, by far the most common are gzip and bzip2.

gzip is probably a bit more common than bzip2. While bzip2's compression is usually (but not always) better, it also takes more CPU power and time to compress and decompress, so it is typically used when space and bandwidth are a bigger constraint than time and CPU power.

To gzip or bzip a file, simply execute the respective command on the target file. Note, however, that it will not create a new file. It will compress the target file and replace it, as well as adding a new file extension, either .gz or .bz2.

```
$ gzip destination.tar
$ bzip2 music.tar
$ ls
destination.tar.gz    music.tar.bz2
```

8.1.3 Others

While a combination of tar and either gzip and bzip2 is by far the most common archival and compression method used in the Unix world, there is still support for many other archive utilities which are usually available through software repositories or BSD ports. Among them are zip, xz, 7zip,rar, cpio, compress and many other lesser known utilities.

8.2 The Mighty grep

grep, as mentioned a few times earlier in the book, is a tremendously useful system administration tool. It provides two primary functions: filtering text streams and locating files containing certain text patterns. grep is so common and prevalent among Unix admins, that the program has become a verb, such as in the sentence, ”I had to grep through a few thousand log files.”

grep is used in one of two ways: in a Unix command pipeline or as aprogram in and of itself.

```
[jdoolin@thompson  ]$cat/var/log/messages | grep sda
[jdoolin@thompson  ]$ grep "randint" *.py
```

The previous two commands do not display what the normal output would be. The first command would send the contents of the system logto the grep command, which would filter out any lines that contain thestring ”sda”. This could be a command used to search the system log forany information or errors on the hard disks in the system. The second command will search all files ending in .py for the string ”randint”. This is an effective way to search the contents of files for certain text patterns.grep has a very useful flag, -v, that allows one to search for all lines that do not contain a particular string or pattern.

```
[jdoolin@thompson  ]$ df -h | grep -v tmpfs
System                                Size   Used Avail Use% Mounted on
/dev/dm-0                              64G    46G   15G  76% /
udev                                   10M      0   10M   0%
/dev/dev/sda1                         236M    33M  191M  15% /boot
/dev/mapper/thompson--home-home       266G    32G  221G  13% /home
```

The previous command filtered out all lines that contain the string’tmpfs’, so that one does not see the many virtual filesystems in the ’df’output.

8.3 dd - The Ultimate Data Copying Tool

dd is among the most versatile and useful Unix commands at your disposal. Think of it like ’cp’ on steroids. Where ’cp’ can copy only whole files and directories, dd can copy only certain sections of files if you wish, or copy data into a certain area of a file. Better yet, dd can even access and use /dev/ devices, meaning you can directly access the data of devices such as hard disks, flash media and optical disks. This makes dd a useful tool for many tasks. Here are a few examples.

8.3.1 dd: as an image or clone tool

dd can be used to create images of CDROMs, DVDs and BluRays, floppy-disks, USB drives, SD cards and even entire hard drives.

8.4 head and tail

8.5 Logs

8.6 Cron

8.7 Backups

8.8 Single User Mode

8.9 SSH

8.9.1 screen

Chapter 9

L.A.M.P

Linux.

Apache.

MySQL.

PHP.

This set of four open source projects is often referred to as the **L.A.M.P. Stack**. It consists of a Linux distribution, the Apache web server, the MySQL relational database server and the PHP web programming language. With these four pieces of software, one can set up a fully functioning web server capable of serving professional, data driven websites. Many web hosting services rely on the LAMP stack for its services.

However, it doesn't necessarily need to be this exact combination of software. It may be a FreeBSD system with the NGINX web server, PostGRES database server and Ruby on Rails web programming language. Even with the different software, the concept is still the same: four open source projects with the goal of serving professional web sites.

If you are a web designer or are interested in web design, you owe it to yourself to become comfortable with this software.

9.1 Web Server Software

The web server is software that runs in the background, listens on a network interface and receives data from the operating system as it comes in through the network. The web server receives requests for documents, HTML, CSS, images, flash players and any other data hosted on the server required to display the page. There are many web server options, both proprietary and open source, but by far the most common are the open source web servers. In particular, Apache.

9.1.1 Apache

Apache is practically the foundation of the majority of the World Wide Web. It has had the greatest market share of web servers for many years. It's an open source project, meaning its code is seen by millions, reviewed, patched quickly and enjoys excellent security[1]. It is known for being very stable and capable of handling very large installations. It supports virtual hosts, which means the ability to host many web sites on one system. It is also extensible with its module system, allowing you to include only features that are needed.

9.1.2 nginx

nginx is a relative newcomer to the web server party, but it is a big one. It was designed to be very high performance with low memory requirements, as well as being able to serve as a proxy for several network protocols and as a load balancer. It has a simpler setup than Apache, but has a shorter history.

9.1.3 lighttpd

lighttpd is worth mentioning because it has a totally different philosophy: to be a strong, capable but very lightweight web server. While it is convenient for smaller systems, it is also capable of running very large installations, such as Wikimedia.

[1]Keep in mind, however, that web servers may be vulnerable through the applications that it hosts. Even a simple login page may be an entry into the whole system.

9.2 Database Server

The database server is the process that allows a connection to a relational database system. The relational database is how most website data is stored. From small database systems such as SQLite and Access, to large scale databases that run on Oracle or Bigtable, the world runs on databases. User accounts, product listings, social media posts, order history, recipes, student information, course offerings, billing systems and countless other systems have a database storing the information.

As with web servers, there are many options for relational database systems (RDBMS). There are proprietary systems such as Microsoft SQL Server and Oracle and open source systems such as MySQL/MariaDB, PostGRESQL and SQlite.

9.2.1 MySQL/MariaDB

9.2.2 PostGRES

9.2.3 SQLite

9.3 Web Programming Languages

Once you have a web server for providing pages and documents and a database for storing all of the valuable data, you need some way of tying them together. A way to retrieve data from the database and present it to the user. A way for the user to change, add or remove data. A way to present this data differently depending on what it is. This is where the web programming language comes into play.

The web programming language allows the developer to write code that builds HTML documents on the fly and populate them with data retrieved from the relational database. They are often built with elaborate frameworks that enable very quick development and connecting code with databases.

Nearly any language can be used as a programming language for a web site, but there are some that are far more common than others. Among the common languages are ASP, Perl, PHP, Python, Java and Ruby (on Rails), but perhaps the most popular is PHP.

9.3.1 PHP

9.3.2 Ruby on Rails

9.3.3 Python

9.4 A Basic Web Page

Chapter 10

UNIX Processes

Similarly to Windows, any running program or task is considered to be a **process**. Some processes are graphical programs that we see and interact with or even command line programs like ls, grep, cat or nano. Others are background tasks that aren't "seen", such as web servers, print services, wireless services or DHCP clients. Understanding processes allows us to get information about them, how they are affecting the system and to troubleshoot any that are causing problems.

10.1 What is a Process?

A Unix process consists of the following:

- **Instructions** - these are the actual binary machine code instructions as contained in the binary file (such as /bin/bash).
- **Data** - These are variables and intermediate data required for the process to run or that it is producing.
- **Resources** - this refers to open files, sockets or devices
- **Environment Variables** - passed from the shell that executed the process
- **Permissions** - each process is owned by a user, typically whichever initiated the process.

All processes are given a process id, or **pid**. This id can be used for many tasks, such as changing the priority, tracking whether a process is currently still running or terminating the process.

10.2 Listing Processes

In Windows, one may need to list currently running processes to see if some particular application is running, not responding using a lot of resources and memory, or has a particular file open. The graphical program most often used is Task Manager, however there is also a command prompt utility called "tasklist" that will show running processes.

The Unix command usually used for listing all process is **ps**. ps has many command options, so a thorough understanding of the command will require reading the man page. But a few common examples are:

```
## List All processes
$ ps -A
## List all processes along with full path to the binary and user ownersl
$ ps -aux
```

There are many other great examples in the man page. ps output can be combined with grep to retrieve information for only particular processes, users or even binaries or their arguments. One example that we will discuss later is using ps to retrieve the process id for a process you wish to terminate.

10.2.1 Process Resource Usage

Most Unix systems have a Task Manager type program called **top** that shows running processes sorted by certain criteria, such as those using the most CPU time or memory. This is quite useful for seeing which programs are causing performance problems or that may have memory leaks. There are other, more modern versions of top that you can also install from software repositories, such as **htop**. They provide the same function but are easier to read, colorful and more intuitive to use.

10.3 Background and Foreground

Conventionally, when you run a program or command from the command prompt, you must wait for it to complete before issuing the next command. This is because the process interrupts the execution of the bash shell until the process has completed. For simple, quick programs like ls, mkdir or touch, this may be too brief to even notice. But occasionally these programs can run for much longer. For example, you may be creating a .tar.gz file of the entire /home partition. This may take several minutes to hours to complete. You don't have time to wait for it to finish before moving on to other administrative duties.

Another example is if you are using a graphical interface and you launch a graphical program from a command prompt, such as firefox. Until firefox is finished running, you can no longer use that command prompt window to enter further commands.

In either case, one solution provided by the shell itself is to put the process into the background by adding an ampersand (&) at the end of the command, like so:

```
$ tar -czvf home.tar.gz /home &
$
```

The tar process will now run in the background and you will get your command prompt back. The command will continue to run until you either exit the program (such as with firefox) or until it finishes on its own (tar).

Occasionally you may initiate a process that takes much longer than you expect and that would be better off running in the background. This can be done, but first we need to examine pausing a process.

If you enter a command that takes an appreciable amount of time, you may stop/pause the process by pressing **Ctrl-z** (hold the control key and press 'z'). This now pauses the process. It is no longer running, yet it is not terminated either. At this point you have a choice. You can now put this program in the background by typing the **bg** command, or bring it back into the foreground with **fg**.

There is a possibility that you may have several active processes that are either stopped or running in the background. You can get a list of these processes by using the **jobs** command, which will list these background and stopped processes by number. You can then specify the number of a process to either put in the background or to resume in the foreground, like so:

```
$ jobs
[1]+ Stopped firefox
$ fg 1
```

These commands list current background or stopped processes, which shows only one, firefox. The fg command then brings firefox back to the foreground.

10.4 Terminating Processes

Killing a process can be accomplished in one of three ways. For a command line process that is currently running in the foreground, such as a large file download with wget, for example, you can terminate this process with **Ctrl-c**. You can try this by running the 'cat' program by itself, which will wait for input. Press Ctrl-c to terminate cat.

But occasionally you want to terminate a process that is not or has not ever run in your terminal. In this case you need its PID (process id), which can be obtained from the 'ps' command. First issue ps, then you can use the **kill** command to terminate the process:

```
$ ps -A | grep ntpd
2122 ?        00:01:46 ntpd
$ kill 2122
```

If you know the exact name of the process, such as ntpd or firefox, you can kill it by name with the **killall** command:

```
$ killall firefox
```

The caveat here is that its default behavior is to kill all occurences of the processes running under that name.

The kill command actually sends a signal to a running process that tells it to terminate. Some programs are written to accept particular signals that allow it to perform certain tasks to allow it to terminate gracefully, such as closing files, network connections or writing configuration data. The default signal used by kill is SIGTERM, which stands for TERMINATE. There are other signals as well, but the most common other signal is SIGKILL. This signal terminates a process immediately and it cannot be ignored. It's the sledgehammer of process termination. The typical, shorthand manner of issuing SIGKILL is **kill -9**, like so:

```
$ kill -9 2122
```

Two other signals that are useful are SIGSTP, which the equivalent of pressing "Ctrl-Z" to pause a process and SIGCONT, which resumes stopped processes. These signals are how shells like bash implement job control.

10.5 Daemons

10.5.1 Init and Runlevels

10.5.2 Init Scripts and Services

Chapter 11

UNIX Security

11.1 Linux Anti-Virus and Malware Removal

Malware has a long and interesting history. It was first simply annoyances, written by rather intelligent programmers seeking to show the world how clever they were. It grew to be expensive, network hogging worms and system damaging viruses. Eventually it became apparent that malware could actually be profitable, so rather than seeing a decrease in malware, there has actually been an *increase*.

Malware comes in many forms, such as viruses, worms, trojans, spyware, adware, rootkits, browser hijackers, ransomware and many others. It has become a concern for the safety of our private information, sensitive data and even our money. With this in mind, we should be very mindful of preventing malware .

With this in mind, we need to take care in protecting our Linux and UNIX systems from the perils of malware.

11.1.1 Just Kidding

Hahahahahahahahaha.... malware on UNIX! Oh, that was a good one. Hope you got as much of a laugh out of that one as I did.

One of the greatest advantages that Linux and UNIX users have is that malware is simply not a problem we have to deal with. Oh, I already know what some of you are saying. You're saying, "Yeah, but Linux has such a tiny market share on the desktop, so why would malware writers even consider it as a target? If Linux were more popular, it would have just as much malware as Windows".

That seems like a legitimate argument at first, but when you consider the huge number of servers, routers, appliances and other network devices running Linux or some other UNIX variant, why would you be happy taking out a few thousand home PCs when you could take out a whole continent?

Or perhaps you've read about some Linux or macOS malware in the news and have concluded it's just as much of a problem as it is in Windows. Yes, it's true that every so often, some piece of malware is discovered that targets macOS or Linux. However, it often requires that the system have a particular Linux distribution with particular services running or configurations. It also usually requires the interaction of less informed users. But these are few and far between.

There are a number of reasons why malware is more or less non-existant for Linux and UNIX. Among them are:

- **Not running as root/administrator**. Most UNIX users do not use the root account for day-to-day activities. As such, malware is severely limited to what it could do on a UNIX system. In order to infect binaries, the malware has to have root access, which is more difficult to achieve with accounts that do not themselves have root access.
- **Software repositories**. In Windows, when you install software, you either go buy it or you download it and install it from any number of sites. A third option is to pirate the proprietary software. Windows software can come from unreliable sources, especially if it's pirated. When you install pirated software, there could be anything hiding in those executables. Even when you install legitimate software, you often have to use the "Custom Settings" for "Advanced Users" to disable the installation of various toolbars and other malware. On most UNIX systems, software is installed either from repositories or from packages provided on the original developers' sites. This prevents malicious code from sneaking in. Packages are also digitally signed to verify their integrity. It doesn't mean it's impossible, just far more difficult to do on a large scale.
- **Open source software**. A lot of Windows software is proprietary or closed source, meaning that only the developers ever get to see the code. Who knows what could be hiding in that program you just installed? Open source software is an open book. There are a lot of eyes on the source code, so malware has no chance to get in.
- **Linux Distribution and UNIX variety**. It would be difficult to write malware that works on all Linux systems, because not all distribu-

tions are built the same, use the same software or have the same configuration systems. This is further complicated by the variety in UNIX systems themselves, as malware for Linux is unlikely to work on FreeBSD, Illumos, or OpenBSD.

- **User knowledge and experience**. - The typical Linux or UNIX user has a technology education and knows well enough to not click on suspicious email attachments, visit suspicious websites or otherwise fall for the social engineering tactics used to facilitate the spread of malware. That's not to say it's true for all UNIX users, but it's a safe bet.

If you're running Linux, you can be pretty confident without running any anti-malware or anti-virus software. I myself have used Linux, BSD and macOS systems for 18 years as of this writing and have never once had anything you could call a malware infection.

11.2 But Seriously

Having said all that, this doesn't mean you can just do whatever you want with a Linux system and you'll be fine for all time. Just because malware isn't a problem doesn't mean security isn't a problem at all. It *is* a problem. A very big problem.

Linux and UNIX systems still have security vulnerabilities outside of malware and thus they are still major targets for security attacks. Webservers can be compromised, UNIX hosted web sites can be hacked, remote exploits can be conducted. So you must remain vigilant with software updates and make sure you aren't running unneccessary services.

You should approach the security of Linux and UNIX system as seriously as you would any Windows system, even if malware isn't a concern.

11.3 UNIX Security Practices

Specific security measures may depend on your particular UNIX system, as Linux, FreeBSD, OpenBSD, macOS, Illumos or other UNIX systems may all have different available security measures. We'll touch on a few of these later. However, there are some general policies that one can follow to mitigate security risk on UNIX systems.

11.3.1 Keep Software Updated

The majority of UNIX security breaches occur through vulnerable server software or web applications. Still others occur due to root access being gained by unprivileged user accounts through vulnerable programs. These can be mitigated by simply keeping your software up to date and applying patches, just as it does on a Windows machine.

For the truly security conscious sysadmin, it is advisable to keep aware of security vulnerabilities that are discovered for the systems you are managing and the software they use. FreeBSD, for example, will publish known discovered vulnerabilities on a mailing list, and even individual projects like Apache and MySQL will do the same. Knowing what you are running can help you know what is serious enough to update.

11.3.2 Use Repositories and Package Managers

By installing software from repositories using package managers, such as apt, yum, and pkg, you have a much lower risk of installing programs that contain any malware or malicious code. These packages have been patched, tested and reviewed. While it is still not impossible for there to be vulnerabilities, the likelihood of intentionally malicious code is very small.

11.3.3 Disable Unnecessary Services

By turning off all unnecessary services you decrease the number of **attack surfaces** on the system. Many UNIX systems enable certain services by default during installation. If they aren't needed they should be disabled. This can include mail servers, print servers, Samba file sharing or name-service and others. The Irix system by Silicon Graphics was notorious for being insecure, but this was largely due to the large number of insecure services enabled by default. When these were disabled and more secure alternatives installed, Irix could be secured.

Most BSD systems enable only the bare minimum of services and require the admin to enable those they wish to run. The same is not true for Linux distributions, so always be mindful of what services are running after installing and disable those that are not needed.

11.3.4 Use Secure Remote Access

Rather than use telnet or FTP protocols for remote access, use the OpenSSH alternatives. While this isn't as much of an issue any more, it is still important for sysadmins to know that FTP and telnet are insecure, unencrypted clear text protocols. Usernames, passwords, files, commands, command output and all sorts of dangerous information can be read very easily by anyone sniffing packets on a replicated port or a Man in the Middle attack. OpenSSH mitigates this possibility by using asymmetric (public/private key) encryption.

However even OpenSSH must be protected. SSH servers have a variety of configuration options that can help reduce the chances of being compromised. For example, it is a good idea to disable root SSH access. This means that even someone who stole or guessed the root password would not be able to use SSH to login with the root account. SSH can also be used to allow only certain users or groups to login remotely, or even to disallow password authentication entirely, requiring all users to use public/private SSH key authentication.

Taking it a step further would include only allowing SSH connections from inside the network, or even from particular IP addresses.

11.3.5 Proper User and Group Management

When creating users and granting permissions to various resources, use the Principle of Least Privilege. In other words, when creating users, only grant them access to the minimal resources required to perform their duties. For example, you may not want to give them sudo access or put them in the BSD wheel group. Perhaps you don't want them access to certain folders, drives or devices.

It is also a good practice to enable user account expiration. This allows accounts to be automatically disabled after a certain amount of time or lack of activity. Orphaned or abandoned accounts are potential security problems.

11.3.6 Use a Firewall

Most UNIX systems have firewalls either built-in or as optional software. Linux has **iptables**, which is being replaced by **nftables**. The BSD systems have **pf**, **ipfw** and **ipfilter**. Enable and configure the firewall to block all network connection attempts other than those from legitimate sources and to legitimate services.

The typical firewall strategy is to block everything incoming by default, then "poke holes" in the firewall to allow the traffic you need through. Firewalls can be very specific as well, such as allowing Port 22 traffic (SSH) but only for certain IP addresses.

Firewalls should be configured at the network level and host level. Network firewalls are systems through which all traffic must pass and the firewall filters it as needed, blocking all traffic that is not allowed. There are numerous network firewalls available from many vendors, but it is also possible to build one with a UNIX or Linux system. **pfSense** is a very popular FreeBSD based firewall distribution, while IPFire and ClearOS are Linux based firewall distros. One need not even use a particular distribution, and can build one from a base OS install.

11.3.7 Make Regular Backups

The three key aspects of Information Security are Confidentiality, Integrity and Availability. There is a huge focus on Confidentiality, but many often overlook the importance of Integrity and Availability. Integrity means the data is accurate and unmodified from its intended state. Availability means data is not lost or inaccessible.

Keeping good backups helps to ensure Availability and Integrity. It prevents loss of data and gives administrators the freedom of reinstalling compromised operating systems and restoring any data that may have been modified. Even using good storage systems such as FreeBSD's ZFS promotes data integrity even by protecting against hardware faults. Using RAID, ZFS or Linux software raid ensures that data is not lost due to failed disks.

11.3.8 Encryption

For sensitive data, it is a good idea to use encryption. This can take many forms, from single file encryption to archive encryption, encrypted containers and even entire disk encryption.

PGP/GPG can be used to encrypt and decrypt files, as well as verify their senders. This is often done in email clients, but can also be performed on the command line.

Encrypted containers can be created by VeraCrypt (also available on Windows and macOS). These containers are something like folders that can even be moved around and transferred to other systems.

Whole disk encryption can also be done with VeraCrypt but also by the Linux utility dm-crypt and in FreeBSD with GELI, which can be enabled at install. However, be mindful that while whole disk encryption is beneficial, it can also result in misery when one must attempt to recover data from an encrypted disk that is not in its original PC. This depends on the encryption method used, but always read documentation and keep good backups.

11.3.9 Intrusion Detection and Rootkit Detection

There is an open source IDS (Intrusion Detection System) available called tripwire. This is a signature based IDS that is used to scan for and detect unauthorized changes to key operating system files. Rootkits often require replacing low level operating system files with compromised versions that hide themselves. Tripwire will detect when any of these files are changed or an attempt is made to change them.

It is also advisable to regularly run rootkit scans using tools such as rkhunter.

11.3.10 Use Virtualization

Particularly useful for servers, virtualization allows one to run many instances of Linux virtual machines, each running its own service. For example, it is possible on a LAMP machine to run each service on its own VM. This means that if only one VM is compromised, such as Apache, the MySQL server will remain protected.

Depending on the CPU features, Linux supports **Kernel Virtual Machines** (or KVM), meaning that the Linux operating system provides low-level access to hardware, increasing the speed and efficiency of virtual machines. FreeBSD does the same through it's **bhyve** hypervisor.

FreeBSD has the unique **jails** feature, similar to Linux **Docker** containers, which are very similar to virtual machines. These are lightweight virtual machines that do not require full machine emulation. It is sometimes referred to as **os-level virtualization**. Linux and FreeBSD can run each service in its own isolated 'contaienr' or 'jail'. Each of these is a stand-alone copy of the operating system (minus the kernel) in separate folders. Each of these runs in its own memory space and cannot access any data outside of itself, meaning that if one jail were to be compromised it would leave the others safe. Jails and containers can be easily cloned, backed up or deleted, making them very easy to manage.

11.4 Summary

While malware may not be a major concern for UNIX systems, the usual security risks still apply. UNIX systems can still be hacked and compromised and are popular targets. Systems must still be secured through whatever means necessary, including software updates, good user management, firewalls, secure remote connection protocols and configurations, disabling services and many other techniques.

Good sysadmins need to be aware of the systems they manage and be aware of the vulnerabilities that may affect them, and take the steps to mitigate the risks.

Chapter 12

Linux/Unix Networking

TCP/IP networking has been an important part of Unix systems since 1984,when 4.2BSD first introduced the protocol to the operating system. 4.3BSD improved on the prior version and this became a near defacto standard forTCP/IP networking implementations. To this day, many modern operating systems owe their own TCP/IP network drivers to 4.3BSD source code, including Microsoft's.

Therefore, it is no surprise that the fundamentals of Unix networking are the same as in other major operating systems, and the same terminology, acronyms and even tools will be quite similar.

12.1 Network Interfaces

Just as in Windows, on any Unix machine there will be a network interface created by the operating system for every operational Network Interface Card (NIC) that is supported. It is possible, though not very common, forsome NICs to not be supported by particular flavors of Unix.

The name given to network interfaces varies between Unix variants and even Linux distributions. On most popular Linux distributions, network interfaces begin with **eth0** and continue with **eth1**, **eth2** and so on. For everynetwork adapter, it will have an 'eth' abbreviation followed bit a numerical identifier. macOS names ethernet interfaces similarly, with en0, en1 and soon.

FreeBSD, NetBSD and OpenBSD's interface names depend on the NIC manufacturer or driver, and may have names such as em0, xl0 or bge0. However, FreeBSD allows these to be renamed in/etc/rc.conf, so one can theoretically establish a more Linux style naming scheme, though most don't bother. Solaris does something similar, as does Arch Linux.

Wireless interfaces may differ from wired interfaces. In popular Linux distros, for example, it is common for them to be named **wlan0**. However,because WiFi adapters are still ethernet, macOS still applies the enX nameeven to wireless adapters.

All systems will also have what is called a loopback interface, which is a virtual interface that allows the operating system to communicate withitself using TCP/IP. On most systems this interface is called **lo0**.

Finding the names of the interfaces on a Unix machine depends on the Unix variant or Linux distribution. On many machines, simply running the **dmesg** command (usually as root) will reveal the name of network interfaces as they are setup at boot by the OS. However, this does require scanning through a lot of output, even if piped to less. For decades, the most common means would be to run the command **ifconfig -a**, which would list all network interfaces. This is still a relatively safe assumption, but in recent years on Linux, ifconfig has been replaced with the **ip** command, and ip awill perform the same task. More on these two tools later in the next section.

12.2 Network Configuration

Unfortunately, there is no one set of instructions for configuring a network interface on a Unix systems. It is all entirely dependent upon both the Unix variant or the Linux distribution. Many Unix systems configure the primary network interface during installation, which may mean you will not haveto change it later. However some systems require multiple interfaces and others may need to be modified, so it is still worth knowing.

Many Unix systems have graphical utilities or menu driven commandline utilities for configuring network interfaces or information. OpenSUSE, for example, has the **yast** configuration tool and AIX uses its **smitty** command line, menu driven application. Some Linux distributions use applications such as **Network Manager** or for wireless, **wicd**, which while easy to use, may be difficult to work with when attempting to make changes to configuration files.

The two most common configuration tools on the Unix command line are, as mentioned, **ifconfig** and **ip**. ip is the newer of the two and is now quite common in newer Linux distributions. These tools can be used to list interfaces, query their status, set IP addresses and netmasks, take them down or bring them back up. ip can also modify the routing table, while ifconfig cannot and usually requires the use of the **route** command to do so. On modern systems without ifconfig, it is usually an apt-get away, by instaling the **net-tools** package. The following table lists operations and their equivalent 'ip' and 'ifconfig'commands, or other command if ifconfig cannot perform the task.

As you can see, it is possible to configure an interface through only thesecommands. However, this will not be enough to make the settings per-manent. These must be saved in configuration files. This, however, is where Unix variants and distributions differ the most. The best advice for permanent configuration changes is to read the documentation for the Unix variant or Linux distribution you are using. However, there are some common configurations that appear often enough to be worth mentioning.

Debian systems

On Debian systems, most of the configuration can be entered in /etc/network/interfaces. In this file you can give your interface a static IP addressor use DHCP and set up routing information. A DHCP entry may look as follows:

```
auto eth0
allow-hotplug eth0
iface eth0 inet dhcp
```

This will allow the interface to use a DHCP client (more on that in a later section) to automatically configure the network interface. A manual configuration will look similar to the following:

```
auto eth0
iface eth0 inet static
    address 192.168.1.143
    netmask 255.255.255.0
    gateway 192.168.1.254
```

RedHat Systems

On Redhat systems, such as Fedora, CentOS and RHEL, each network interface has its own configuration file in /etc/sysconfig/network-scripts/ifcfg-eth0 (replace 'eth0' with the name of the interface to be configured). The following would be the contents of a file set for DHCP:

```
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
```

... and for static IP:

```
DEVICE=eth0
BOOTPROTO=none
ONBOOT=yes
NETMASK=255.255.255.0
IPADDR=192.168.1.143
GATEWAY=192.168.1.254
USERCTL=no
```

***BSD Systems**

All major BSD systems, such as FreeBSD, use config files in /etc to configure network adapters. On FreeBSD and NetBSD the lines appear thusly in /etc/rc.conf:

```
    # nfe0 will be replaced with your particular network adapter name
    ifconfig_nfe0="inet 192.168.1.10 netmask 255.255.255.0"
    defaultroute="192.168.1.1"
```

OpenBSD in /etc/hostname.smsc0:

```
# inet <ip address> <netmask> <broadcast ip>
inet 192.168.1.10 255.255.255.0 192.168.1.255
```

12.2.1 DNS

Along with the IP address, netmask and default gateway, one more important piece of information is required for nearly all operational network connections. The operating system must know the IP addresses of one or more DNS servers, so that connections can be made by name (such as www.wvncc.edu) instead of by IP address only.

On nearly all Unix machines, this file is found at /etc/resolv.conf and it has a simple format. The following is an example resolv.conf:

```
    domain cit.wvncc.edu
    search cit.wvncc.edu wvncc.edu
    nameserver 192.168.1.200
    nameserver 192.168.1.201
    nameserver 192.168.1.202
```

The 'domain' directive specifies the domain this machine should beconsidered a part of. The 'search' directive specifies which domains should be searched when attempting to access a computer or host by its name only. For example, if I try to "ping thompson", the OS will first try to ping 'thompson.cit.wvncc.edu', followed by 'thompson.wvncc.edu'. The nameserver directive is the most important here. This is a list of IP addresses that correspond to valid DNS servers. Preferrably, these servers should be located on the local network, but in the event that none are available, it is possible to use a public DNS server such as Google's '8.8.8.8'. This file may be modified directly, but be aware of services that may modify it automatically, such as certian DHCP clients and the resolvconf program on FreeBSD and other Unix systems.

12.3 Troubleshooting

Troubleshooting Unix network connections involves tools that are similar or identical to the tools used on other operating systems, such as ping, traceroute and nslookup. This section will cover a number of useful utilities for finding network information and troubleshooting connections.

12.3.1 ping

ping is the quintessential network connection testing tool. The concept is quite simple: ping sends an "echo request" packet to a remote host. If your connection is working, the remote host is available and responds to pings, an "echo response" will be received from the remote host. Obviously, this can be used to check the availability of remote hosts but it also tells a bit more about the connection. Here are some examples.

- If the machine you're testing cannot ping a remote host but a neighboring machine can, then it is likely a problem with the test machine.
- If you can ping by IP address but not by name, you likely have a DNS problem
- If you can ping your default gateway/router, but you cannot ping anything outside your network, then your router's connection to the outside world is likely at fault
- If you can ping other hosts on your network, but not your router, then your router may be down
- If you cannot ping anything it is likely a problem with your local machine.

ping will also report any lost packets and round trip times. High packet loss may indicate a problem with connection quality or hardware failure. High round trip times indicates a bottleneck somewhere in the route from the local host to the remote host.

12.3.2 netstat

The **netstat** command, and on BSD systems, **sockstat**, can be used to list currently open network ports as well as currently established network connections. The valid command arguments for netstat and its ouput format varies beteen Unix variants. On Linux, the following command will list currently open TCP ports:

```
    $ netstat -at
```

On BSD systems you must use a command flag to limit the list to IPv4 connections, and the -l lists only listening ports. You can leave that out if you want to see active connections:

```
    $ sockstat -4l
```

What Program is Listening?

Sometimes it is necessary to find out which process is listening on a particular port. In some cases it may be needed to disable unneeded services, although it is also useful when you suspect something may be listening that shouldn't in the event of an intrusion. Both the netstat and sockstat programs are useful here as well.

On Linux you must be root to list the process information:

```
$ sudo netstat -tlup
```

And on FreeBSD (root not required):

```
$ sockstat -4
USER     COMMAND    PID   FD PROTO  LOCAL ADDRESS         FOREIGN ADDRES
root     sendmail   1307  4  tcp4   127.0.0.1:25          *:*
root     sshd       1301  4  tcp4   *:22                  *:*
root     sendmail   1262  3  tcp4   192.168.1.22:25       *:*
root     sshd       1257  3  tcp4   192.168.1.22:22       *:*
root     syslogd    1209  5  udp4   192.168.1.22:514      *:*
```

Chapter 13

Shell Scripting

In the Windows world, you may have encountered something called a "batch file" ending in the .bat extension. These are treated as executable files or programs that you can double click to run. These files are essentially the equivalent of the Unix shell script.

Shell scripts are extraordinarily useful programs that can be used for an enormous amount of tasks involving automation, program launchers, backups, startup scripts, software building and installation. The only limits to what a shell script can do are constrained by your own imagination.

13.1 Shells are Programming Languages

Unix shell scripts are literally computer programs, but they happen to often include the use of day-to-day programs and commands that you routinely use on the command line, such as echo, cd, mkdir, mv, cp and many others. Just as with other programming languages, shell scripts are capable of conditionals (if/else and case statements), loops (for and while), variable assignment, math and even subroutines. There is a caveat, however.

Back in Chapter 2 we covered the topic of shells and how users in the Unix/Linux world often have their choice of many different shell programs, such as csh, tcsh, bash, zsh, ksh and others, where bash is the most popular and most common on modern Unix systems. While it is nice that we have this choice, it does present a problem with shell scripts. Not all shells support the same scripting features nor do they have the same syntax. For example, the next two lines perform the same task in Bash and CSH:

```
-BASH-
$ name=Jeremy
-CSH-
$ set name = Jeremy
```

That is just one of many little differences between shell syntax. What this means is that what you write for one shell may not work in another, so be aware of this. Most of the time and for this book, however, you will be using Bash.

13.2 List of Commands

So far you have likely typed thousands of commands directly into the shell (probably bash). The very simplest of shell scripts consist of a list of commands to be run, or even one command that has a list of complex arguments. For example, you could write a file named ″mkprojects.sh″ containing the following lines:

```
mkdir projects
cd projects
mkdir python java lisp c++
cd ../..
```

Then if you run this file using the following command...

```
$ bash mkprojects.sh
```

... this would proceed to create a directory calld ′projects′, change to that directory and create four more directories within it, then cd back to the original directory. This sequence of commands is precisely the same sequence you would use to perform the same task at the command prompt, but they are written in a file. You could now keep this file and run it any time you would like to perform the same task.

Another simple example is one I′m using personally to start a Minecraft server. It contains only one line, but simplifies the process of starting the server. In a file simply called ′mcserver′, I have the following line:

```
java -jar -Xms128M -Xmx960M spigot-1.9.jar
```

This starts the Minecraft server with the necessary arguments to the Java virtual machine to optimize performance on a Raspberry Pi. Since this more complex command is in a file, I always have those command line arguments available and do not have to memorize them.

But shell scripts can go way beyond this. Unix shell is a full fledged programming language. Let's look at how.

13.3 SheBANG!

Thus far I have shown you only one way to execute a shell script: by passing the filename as an argument to the shell command:

```
$ bash mkprojects.sh
```

But there are more convenient ways of running shell scripts. One of the ways we can simplify it is by specifying which shell program to use in the very first line of the file. This line begins with a **hash symbol** followed by an **exclamation point**, often referred to as "hash bang" or simply, "shebang". These two characters are immediately followed by the full path to the shell binary along with any other necessary arguments. For example:

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

Using this as the first line in your shell script file would be the first step in allowing it to be run as a self-contained launcher, provided that your bash shell is installed in /bin. It could be that it is installed in /usr/bin or perhaps even /usr/local/bin. If your script is designed to run with ksh, your first line would look like:

```
#!/usr/bin/ksh
```

ksh would have to be installed in /usr/bin, of course. However, this alone isn't enough to make your script a self-contained launcher. You must now make the file executable using chmod, like so:

```
$ chmod a+x mkprojects.sh
```

This uses the newer, non-numeric permissions mode to set executable permissions for all users on the system. Of course, use whichever permissions make the most sense in your particular situation. You should now be able to run this shell script with the following command:

```
$ ./mkprojects.sh
```

This assumes mkprojects.sh is in your present working directory, which is also why we precede the filename with './'. You could also use the full path to execute the program, like so:

```
$ /home/jdoolin/mkprojects.sh
```

Of course, assuming you have root access or sudo, you could simply copy this file to one of the directories in your PATH environment variable, such as /usr/local/bin. You would then be able to execute the program just by typing the filename.

```
$ sudo cp mkprojects.sh /usr/local/bin
$ mkprojects.sh
```

One final step could be to eliminate the .sh extension, which would make it appear like most of the other Unix commands we type.

```
$ sudo mv /usr/local/bin/mkprojects.sh  /usr/local/bin/mkprojects
$ mkprojects
```

13.4 Variables

Bash and other shells are able to save variables. This is an extremely common task in programming languages and usually involves the assignment operator, typicall the '=' sign. To create a variable in Bash, type the following:

```
$ fname=Jeremy
```

This will create a variable named 'fname' that contains the data, "Jeremy". It is very important to note that **there are no spaces between the '=' sign and the variable name or data**. This is because bash considers spaces the necessary separator between arguments. So if you typed

```
$ lname = Doolin
```

Bash would look for a program called 'lname' and try to execute it with the arguments '=' and 'Doolin'. It would likely respond with a "command not found" error in this case. In order to see the contents of a variable, you can use the 'echo' command, like so:

```
$ echo $fname
Jeremy
$
```

You may remember from previous chapters or exercises that **echo** is like a print statemnt. It just prints something to the terminal and exits. However, echo can be used in this way to print the contents of variables. Also note the dollar sign before the variable name. This is required in order to refer to the contents of the variable. Otherwise, echo would have just printed the word ″fname″ to the screen.

If your data requires the use of spaces, you can use double quotes to begin and end the data. For example:

```
$ fullname="Jeremy Doolin"
```

You can also combine variables with literal data. Consider the following script:

```
$ fname=Jeremy
$ lname=Doolin
$ fullname="$fname $lname"
$ echo $fullname
Jeremy Doolin
```

This time I created three variables, but the third one used the contents of the first two to construct my full name.

13.5 Prompting for Data

You may occasionally want to acquire input from a user. This is common in setup scripts that may need to repeatedly use usernames, numbers and other options. The command for reading data from the user is, appropriately, **read**. For example:

```
$ read classnum
CIT220 <--- this would be typed in
$ echo $classnum
CIT220
```

This provides an empty prompt for someone to type something as input. This input will be saved in a variable called ′classnum′. Pay careful attention to my note that the first ′CIT220′ is what I typed into the prompt, not what is printed to the screen. The second one is printed to the screen after the echo. But this prompt isn′t very helpful. You can make it better by using the ″-p″ argument and supplying a prompt, as follows:

```
$ read -p "Enter your course number: " classnum
Enter your course number: CIT220
$ echo $classnum
CIT220
```

You can also read multiple variables in the same command by providing a list of variable names and separating the data by spaces when you enter the data. For example:

```
$ echo "Enter the month, day and year of your birthday"
$ read -p "separated by spaces: " month day year
Enter the month, day and year of your birthday
separated by spaces: 12 8 1978
$ echo $year
1978
```

13.6 for Loops

One of the most useful parts of the Bash shell is the ability to write 'for' loops. These will be second nature to any programmers in the class, but may be new to those who have no programming experience. A for loop allows you to perform the same task repeatedly, usually with different data each time. For example, you may wish to perform the same task on every .png file in a directory or for every letter of the alphabet or many other sequences that can even be generated by other commands. One example of these commands is the **seq** command, which will print a sequence of numbers. 'seq 1 10' will print the numbers 1 to 10. 'seq 20 5' will print the numbers backward from 20 to 5. This can be used to create a 'for' loop as follows:

```
$ for num in ‘seq 1 10‘; do touch Logfile${num}.txt; done;
```

In this loop, 'seq 1 10' is placed inside backticks (we'll get to that shortly), which means it will run the seq program, capture its output and use it for the items in the for loop. This loop will then create ten files named Logfile1.txt through Logfile10.txt. If this loop were written in a file, it could also be written in a more readable form, like so:

```
for num in ‘seq 1 10‘
do
    touch Logfile${num}.txt
done
```

13.7 if/else conditionals

In many programs, you may need your code to branch such that it does one thing if a condition is met, but do something else if the condition is not met. For example, if you may need to create a new file if it does not already exist, but if it does, you will do create a different one or simply add to the previous one. This requires what is called a **conditional** statement. The most basic of these is the **if** statement, something programmers will be all to familiar with. Here is a simple 'if' statement example:

```
if [ -f config.txt ]
then
    echo "SUCCESS: Configuration detected, setup will commence"
else
    echo "ERROR: Config file not found!"
fi
```

This simple statement will check to see if "config.txt" exists as a file in the present working directory. If so, it will print a success message. Otherwise, it will print an error. Strangely, the Bash if statement is ended with **fi**, a backward spelling of "if". This will also be used for the "case" statement as we shall soon see.

13.8 Backquote/Backtick

13.9 The Rabbit Hole

Appendix A

Common Commands

Appendix B

Linux Distro and BSD Comparison

Appendix C

Linux Equivalents of Windows Software

About the Author

Jeremy Doolin is a graduate of Ohio University in Athens, Ohio, with Bachelor's Degrees in Computer Science and Mathematics. He has worked as a system administrator (primarily UNIX/Linux) for a variety of businesses, including an Internet Service Provider, a casino, a bank and sports media. He has developed software for internet service, sports media, military sensor equipment, malware detection and mobile applications. He is currently a Computer and Information Technology Instructor at West Virginia Northern Community College. He enjoys software development in his spare time and is a UNIX enthusiast. He also enjoys computer history, playing guitar, cooking, teaching, building electronics and many other constructive activities.