Unix and Linux System Administration and Shell Programming

version 56 of August 12, 2014

This book includes material from the http://www.osdata.com/ website and the text book on computer programming.

Distributed on the honor system. Print and read free for personal, non-profit, and/or educational purposes. If you like the book, you are encouraged to send a donation (U.S dollars) to Milo, PO Box 5237, Balboa Island, California, USA 92662.

This is a work in progress. For the most up to date version, visit the website http://www.osdata.com/ and http://www.osdata.com/programming/shell/unixbook.pdf — Please add links from your website or Facebook page.

Professors and Teachers: Feel free to take a copy of this PDF and make it available to your class (possibly through your academic website). This way everyone in your class will have the same copy (with the same page numbers) despite my continual updates. Please try to avoid posting it to the public internet (to avoid old copies confusing things) and take it down when the class ends. You can post the same or a newer version for each succeeding class. Please remove old copies after the class ends to prevent confusing the search engines. You can contact me with a specific version number and class end date and I will put it on my website.

Unix and Linux Administration and Shell Programming

chapter 0

This book looks at Unix (and Linux) shell programming and system administration.

This book covers the basic materials needed for you to understand how to administer your own Linux or Unix server, as well as how to run your own personal desktop version of Linux or Mac OS X.

This book goes beyond the typical material in a shell scripting class and presents material related to either downloading and compiling existing software (including ports to new hardware and/or operating systems) or for preparing your own software for release via the internet.

requirements

You need a willingness to learn.

You need a working computer or server or access to one.

The computer needs a working version of Unix, Linux, Mac OS X, AIX, HP/UX, Solaris, etc. (it can be a dual boot computer).

The new version of Mac OS X 10.9 (Mavericks) is now available on the Mac App Store at www.apple.com as of October 22, 2013. The new version of Mac OS X 10.8.1 (Mountain Lion) is now available on the Mac App Store at www.apple.com as of August 23, 2012. The new version of Mac OS X 10.8 (Mountain Lion) is now available on the Mac App Store at www.apple.com as of July 25th, 2012. Tell them you heard about it from www.osdata.com when you register your new copy.

A working connection to the internet is recommended, but not required, preferably a high speed connection.

You may use LAMP on Linux, MAMP on Mac OS X, or WAMP on WIndows to set up and experiment with a local web server.

You may want to have a domain name of your own and web hosting to try out controlling a server. You can use GoDaddy or HostGator or most any other major hosting provider to obtain these servcies for low cost with great telephone tech support. The OSdata.com website where this book is offered to the public is hosted by Host Gator. You may use any other hosting service you want.

options

Almost anyone can learn this material. It isn't complicated. It really helps if you enjoy doing this kind of thing. You will learn faster and you will enjoy the work. If you don't enjoy this kind of computer activity, please think carefully about whether or not you want to spend decades at a job you hate.

Almost anyone can slog through and learn at least some of this material, but an aptitude for this material greatly helps learning. If you are strong at grammar, then you will probably be able to master. This material. mathematical ability is useful, but not necessary.

Many portions of this book require root or administrator access. While you learn better if you can actually try out each command for yourself, you can just read about root material if you don't have root or administrator access.

Some portions of this book require special software. Most of the software can be downloaded for free. Those with Mac OS X should have the Developer Tools installed. These are available for free on either the install DVD/CD or from Apple at http://connect.apple.com/

A static IP address is in general useful and is required for some portions of this book.

proletarian

This book is intentionally proletarian and is specifically intended to allow anyone, regardless of socio-economic position, to have access to the basic information and the power against the established authorities that such knowledge provides.

This subversive intent is consistent with Linux, which was created by a college student who couldn't afford the dominant tools of the time.

This contrasts strongly with Apple's Macintosh, Mac OS X, and iOS, all of which are specifically targeted for sales to the trendiest members of the wealthy class. Apple does offer a discount to school children, but that discount is smaller than the discount Apple offers to large corporations.

This also contrasts with Microsoft's Windows, which is specifically targeted for sales to large corporations and their employees.

And this contrasts with Google's Android, which is specifically targeted for businesses. The users of Android products are part of Google's product, not the customer. The customers are businesses who want detailed information on the masses for purposes of advertising.

This book is intended to make knowledge available to everyone, especially those who are increasingly being shut out of the mainstream education system.

goals

Please note that at the time this paragraph was written, these goals are not yet implemented in the text. Now is a good time to make suggestions on modifications of the goals.

The reader will understand the Unix computing environment and history.

The reader will be able to access the Unix system and perform basic operations, including using help features.

The reader will be able to access and manipulate files and directories, including basic and advanced directory and file management. The reader will be able to use file system utilities.

The reader will be able to design and implement file system security.

The reader will be able to print documents.

The reader will be able to perform system backups and restores.

The reader will be able to troubleshoot system processes.

The reader will be able to perform environment customization.

The reader will become acquainted with networks, web servers, web clients, transaction security, and other basic network concepts.

The reader will learn how to create a web page that accepts input, program the server response, and interface the input with a database.

The reader will understand web server functionailty and be able to install and configure Apache.

The reader will be able to work with, understand, and configure DNS functionality.

The reader will be able to set up and maintain his or her own general purpose server, including hosting, server administration, security, user interactivity, and database integration.

The reader will be familiar with SQL Server architecture and able to efficiently administer and maintain SQL Server instances and databases.

The reader will be able to secure a web server and its websites.

personal reference book

I strongly recommend that each reader print out this PDF and the manual pages from your computer and place the resulting pages into a looseleaf binder. That will allow you to organize the chapters in the order that makes the most sense for you. It will also allow you to make notes in the margins. It will allow you to easily add materials from other sources. The resulting binder will be personalized to meet your individual needs.

If you have a limited printing budget (ink is expensive), print out portions of this PDF as needed.

chapter contents

27. pipes
28. scripts
29. sysadmin and root/superuser
30. sudo
31. su
32. who
33. advanced file systems
34. major directories
35. network file system (NFS)
36. tail
37. wc
38. chmod
39. chown
40. shell levels and subshells
41. substitutions
42. command substitutions
43. arithmetic substitutions
44. flow control
45. management tools
46. df
47. du
48. processes
49. ps
50. kill
51. nice
52. w
53. date
54. uname
55. uptime
56. top
57. lsof
58. free
59. vmstat
60. polkit
61. defaults (screencapture; Mac Flashback Trojan)
62. init (init; Linux run levels)
63. sendmail
64. ifconfig (view configuration; static IP address)
65. arp
66. netstat (view connections; main info; routing address)
67. route (view connections; routing commands)
68. ping (test packets; measuring)
69. nslookup
70. traceroute (entire route; etiquette)
71. ftp and sftp
72. curl
73. sysstat
74. at (example; removing a job; timing)
75. back ups
76. tar
77. touch (multiple files; specific time)
78. find
79. text processing
80. basename
81. sed (fixing end of line; adding line numbers)
82. awk (remove duplicate lines)

83. screencapture (from graphic user interface; changing defaults; command line screenshots)
84. signals
85. kernel modules
86. LAMP
87. mysql
88. PHP
89. Perl
90. Tcl
91. installing software from source code
92. computer programming
93. size of programs
94. kinds of programs
95. programming languages
96. standards and variants
97. test bed

Appendix:

A. sommand summaries
B. computer history
C. critical reasoning
D. Forth-like routines

cool shell tricks for Unix, Mac OS X, and Linux

chapter 1

summary

This chapter looks at cool shell tricks for Unix, Linux, and Mac OS X to give you an idea of the power of the shell.

A quick summary of how to get to the shell is included in this chapter (more detailed explanations, including what to do when things go wrong, are in following chapters.

If you need a primer on computer terminology, please look at the next chapter on basics of computers.

WARNING: Never trust any Terminal/Shell commands you find on the internet. Only run shell commands you understand. In particular, *never* run anything that includes `sudo` anywhere in the command line unless you are absolutely certain what the command does. When you run a command line with `sudo` you are giving permission to have complete (possibly destructive) control of your computer at the root level. And, yes, this advice applies even to this book. Don't run any commands with `sudo` unless you know for sure what you are doing.

cool shell tricks

This chapter has a handful of cool shell tricks. These are intended to show a beginner that a command line shell can be as fun as any graphic user interface and get across the idea that there is a lot of power in the shell that simply doesn't exist in a standard graphic user interface.

definitions

Unix is one of the ground-breaking operating systems from the early days of computing. Mac OS X is built on top of Unix. Linux is a variation of Unix.

The shell is the command line interface for running Unix (and Mac OS X and Linux) with just typing (no mouse).

operating system The software that provides a computer's basic tasks, such as scheduling tasks, recognizing input from a keyboard, sending output to a display screen or printer, keeping track of files and folders (directories), running applications (programs), and controlling peripherals. Operating systems are explained in more detail for beginners just below.

Unix Unix (or UNIX) is an interactive multi-user multitasking timesharing operating system found on many types of computers. It was invented in 1969 at AT&T's Bell Labs by a team led by Ken Thompson and Dennis Ritchie. Some versions of Unix include: AIX, A/UX, BSD, Debian, FreeBSD, GNU, HP-UX, IRIX, Linux, Mac OS X, MINIX, Mint, NetBSD, NEXTSTEP, OpenBSD, OPENSTEP, OSF, POSIX, Red Hat Enterprise, SCO, Solaris, SunOS, System V, Ubuntu, Ultrix, Version 7, and Xenix.

Linux An open-source version of the Unix operating system.

graphical user interface A graphical user interface (GUI) is a windowing system, with windws, icons, and menus, operated by a mouse, trackball, touch screen, or other pointing device, used for controlling an operating system and application programs (apps). The Macintosh, Windows, Gnome, and KDE are famous examples of graphical user interfaces.

command line interface A command line interface (CLI orcommand line user interface CLUI) is a text only interface, operated by a keyboard, used for controlling an operating system and programs.

shell The shell is the command line interface for Unix, Linux, and Mac OS X. In addition to intrepetting commands, it is also a programming language.

shell uses

The Unix shell is a very powerful programming language, with access to hundreds of useful tools that are designed to work with each other, and access to the very heart of the operating system (although this is only available to the root or superuser account for security reasons).

Unix (and therefore also Mac OS X and Linux) has more than 200 basic commands (also called tools or utilities) that are distributed with the standard operating system. This collection and the ease with which they work together is the major source of the power of Unix. The vast majority of these standard tools are designed to be used from a command line (the shell).

The shell is most commonly used to control servers. Servers are the computers used to host websites. The most common operating system for the world's web servers is Linux. If you learn shell scripting and system administration, you can run your own server and possibly get a job.

The shell can be used to control a desktop or portable computer. Some tablets and smart phones have a shell. The iPhone actually has a shell, but it can't be accessed witout jailbreaking the iPhone.

The shell will often run even when a computer is partly broken. Both Mac OS X and Linux (as well as almost all versions of Unix) can be run in a special single user mode. This starts up the computer or server with just the command line shell running. This can be used to service a computer or server, including both diagnosis and repair.

The shell is extremely useful for programming. Even when a programmer uses a graphical integrated development environment (IDE), the programmer is likely to still heavily use the shell for programming support. Some IDEs even have shell access built-in.

Shell scripts are widely used by system administrators for performing common tasks.

command line interface

Before the widespread introduction of graphic user interfaces (GUI), computers were controlled either by punched cards, paper tape, or magnetic tape (a batch system) or a command line interface (CLI) using an interactive terminal (originally, some variation of a teletype machine from the telegraph technology). The earliest computers were controlled by front panel lights and switches or even by directly changing the wiring.

The command line interface on interactive terminals was a major advance. Because of limitations of the early hardware (at a time when a computer's entire memory might be measured in hundreds of bytes), the first CLIs compressed the number of characters, using two or three letter abbreviations for commands and single character switches for options to commands.

The modern Unix/Linux shells carry over this early limitation because there is the need to remain backward compatible and still run shell scripts that are decades old, but essential to continued operation of legacy systems.

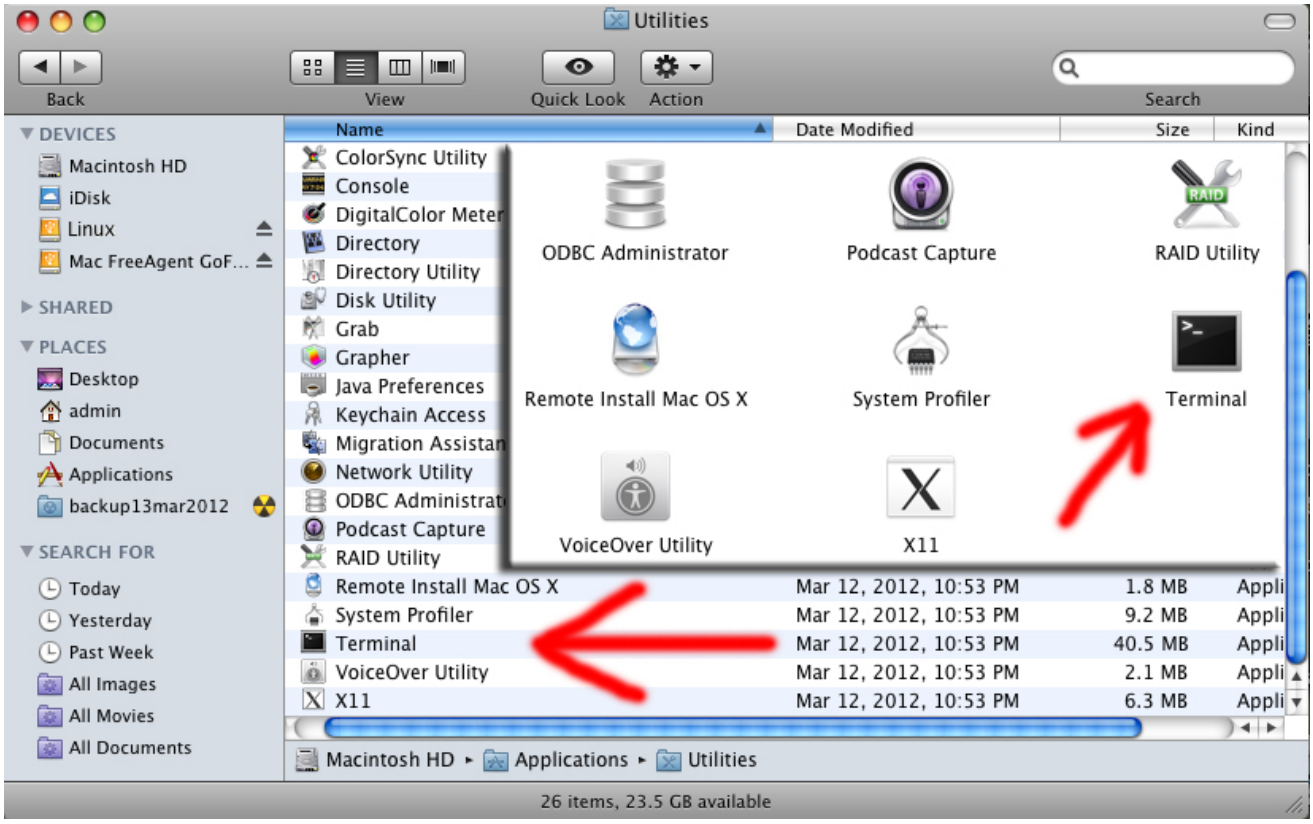
This historical limitation makes it difficult for newcomers to figure out how to use a Unix/Linux shell.

how to find Terminal

You can run these commands by copying and pasting into Terminal, a program that is available for free and preinstalled on Mac OS X and most versions of Linux. Some of these commands will only work on a particular operating system (this will be indicated), but most can be run from Mac OS X, any distribution Linux, and any kind of Unix.

On Mac OS X, you will find Terminal by opening the Applications folder (on your main hard drive), then opening the Utilities folder, then scrolling down the list until you find Terminal or Terminal.app. Drag it to your Dock, because you will

be using it a lot.



On Ubuntu Linux, look in Applications menu > Accessories > Terminal. Single click and hold, then drag to your panel, because you will be using it a lot.

On some versions of Linux, you can press the CONTROL and the ALT and the F1 keys all at once to bring up Terminal.

In Gnome, use the Applications menu > Accessories > Terminal or the keyboard shortcut of CONTROL and ALT and T (all at the same time).

In KDE, use the KMenu > System > Terminal Program (Konsole).

In Linux Mint you can use the keyboard shortcut of holding down the CONTROL and ALT and T keys all at once.

As a last resort, you can use the file search feature in Linux or Mac OS X and search for “Terminal”.

warnings

Be careful about any hints you find on the internet. There are people who suggest very destructive commands in the guise of a useful or fun hint just to trick beginners into destroying their computers.

Be very careful if you use any command that includes `sudo`, because it runs at the root level, allowing complete access to the entire computer with all safeguards turned off. Very powerful. Potentially very distructive in a hurry. There are legitimate hints and cool tricks that use `sudo`, but be careful to type them *exactly* as you you see them in the hint (or copy and paste) and only use `sudo` hints from trusted soruces.

Watch out for anything that includes the command `rm` or `rm *`. That is the remove command versions of it can literally wipe out all of your hard drives in seconds in such a way that only a very expensive data recovery specialist (thousands of dollars) can get your data back.

Also watch out for anything that includes the command `shred`. That is the secure delete and even the most expensive data recovery specialist in the world can’t get your data back.

cool ASCII art

If your computer is connected to the internet, you can use the shell to watch the entire *Star Wars Episode IV* in old fashioned ASCII. Type `telnet towel.blinkenlights.nl` followed by ENTER or RETURN. If you have IPv6, you get extra scenes and color support.

```
$ telnet towel.blinkenlights.nl
```

play a CD

On Linux you can play a CD from the command line. The following example plays the first track of the CD. Change the number to play a different track.

```
$ cdplay play 1
```

On Linux you can get a free coffee cup holder (eject the CD-ROM tray).

```
$ eject
```

basics of computers

chapter 2

summary

This chapter will cover some very basic information on how computers work. I'm not trying to insult anyone's intelligence. This material addresses some of the questions asked by novice test readers.

major desktop systems

There are only three major desktop computing systems left in common existence: Windows, Macintosh, and Linux.

Apple **Macintosh** was the first commercially successful graphical user interface. That is, a computer where one uses a mouse (or other pointing device) and icons, windows, and menus to control the computer. Since the turn of the century, **Mac OS X** has actually been built on top of Unix.

Unix is the last major survivor of the mainframe era (back when computers were so large that they took up an entire room, or even an entire building). Its big advantage of its compeitors was that it is available in free open source versions and it runs on an extremely wide variety of computer hardware (including many computers that are no longer used).

Microsoft **Windows** is loosely based on the Macintosh and for decades dominated the personal computer market.

Linux is a very popular open source variation of Unix. It is the most common operating system for servers and the third most popular desktop computing operating system. In its early days it was very geeky and very difficult to use, but it now sports two major graphical user interfaces (Gnome and KDE) and is reasonably easy to use.

FreeBSD is the next most popular open source version of Unix and is commonly used for servers. **Solaris** is the next most popular commercial version of Unix. IBM and HP both have their own commercial versions of Unix.

definitions

Unix is one of the ground-breaking operating systems from the early days of computing. Mac OS X is built on top of Unix. Linux is a variation of Unix.

The shell is the command line interface for running Unix (and Mac OS X and Linux) with just typing (no mouse).

operating system The software that provides a computer's basic tasks, such as scheduling tasks, recognizing input from a keyboard, sending output to a display screen or printer, keeping track of files and folders (directories), running applications (programs), and controlling peripherals. Operating systems are explained in more detail for beginners just below.

Unix Unix (or UNIX) is an interactive multi-user multitasking timesharing operating system found on many types of computers. It was invented in 1969 at AT&T's Bell Labs by a team led by Ken Thompson and Dennis Ritchie. Some versions of Unix include: AIX, A/UX, BSD, Debian, FreeBSD, GNU, HP-UX, IRIX, Linux, Mac OS X, MINIX, Mint, NetBSD, NEXTSTEP, OpenBSD, OPENSTEP, OSF, POSIX, Red Hat Enterprise, SCO, Solaris, SunOS, System V, Ubuntu, Ultrix, Version 7, and Xenix.

Linux An open-source version of the Unix operating system.

graphical user interface A graphical user interface (GUI) is a windowing system, with windws, icons, and menus, operated by a mouse, trackball, touch screen, or other pointing device, used for controlling an operating system and application programs (apps). The Macintosh, Windows, Gnome, and KDE are famous examples of graphical user interfaces.

command line interface A command line interface (CLI orcommand line user interface CLUI) is a text only interface, operated by a keyboard, used for controlling an operating system and programs.

shell The shell is the command line interface for Unix, Linux, and Mac OS X.

the shell

Unix (and therefore also Mac OS X and Linux) has more than 200 basic commands (also called tools or utilities) that are distributed with the standard operating system. This collection and the ease with which they work together is the major source of the power of Unix. The vast majority of these standard tools are designed to be used from a command line (the shell).

A shell is primarily a command interpreter.

In a graphical user interface such as Macintosh or Windows, the user controls the computer and programs primarily through pointing and clicking, supplemented by some typing.

In a shell, all of the commands are typed. The shell figures out the meaning of what you typed and then has the computer do as instructed.

But the shell is much more than just a command interpreter. It is also a complete programming language.

Because the shell is a complete programming language, with sequences, decisions, loops, and functions, it can do things well beyond pointing and clicking. It can take control of your computer and react to changing circumstances.

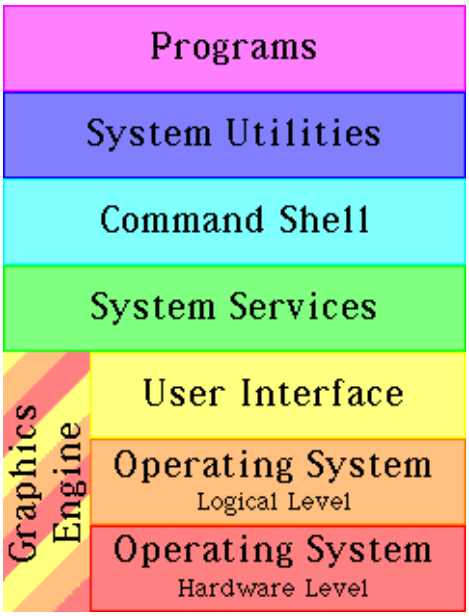
Programming languages are used to make all of the computer programs and smart phone apps you've ever seen or used. Your imagination is the only limit on the power of the shell. Anything that can be done with a computer can be done with the shell.

operating systems

The seven layers of software are (top to bottom): Programs; System Utilities; Command Shell; System Services; User Interface; Logical Level; and Hardware Level. A Graphics Engine stradles the bottom three layers. Strictly speaking, only the bottom two levels are the operating system, although even technical persons will often refer to any level other than programs as part of the operating system (and Microsoft tried to convince the Justice Department that their web browser application is actually a part of their operating system).

The following are examples of each category:

- Programs: Examples of Programs include your word processor, spreadsheet, graphics programs, music software, games, etc.
- System Utilities: Examples of System Utilities include file copy, hard drive repair, and similar items. On the Macintosh, all the Desk Accessories (calculator, key caps, etc.) and all of the Control Panels are examples of System Utilities.
- Command Shell: The Command Shell on the Macintosh is the Finder and was the first commercially available graphic command shell. On Windows, the Command Shell is a poorly integrated comination of the File Manager and the Program Manager. The command line (C:\ prompt) of MS-DOS or Bourne Shell of Unix are examples of the older style text-based command shells.
- System Services: Examples of System Services are built-in data base query languages on mainframes or the QuickTime media layer of the Macintosh.
- User Interface: Until the Macintosh introduced Alan Kay's (inventer of the personal computer, graphic user interfaces, object oriented programming, and software agents) ground breaking ideas on human-computer interfaces,



operating systems didn't include support for user interfaces (other than simple text-based shells). The Macintosh user interface is called the Macintosh ToolBox and provides the windows, menus, alert boxes, dialog boxes, scroll bars, buttons, controls, and other user interface elements shared by almost all programs.

- Logical Level of Operating System: The Logical Level of the operating system provides high level functions, such as file management, internet and networking facilities, etc.
- Hardware Level of Operating System: The Hardware Level of the operating system controls the use of physical system resources, such as the memory manager, process manager, disk drivers, etc.
- Graphics Engine: The Graphics Engine includes elements at all three of the lowest levels, from physically displaying things on the monitor to providing high level graphics routines such as fonts and animated sprites.

Human users normally interact with the operating system indirectly, through various programs (application and system) and command shells (text, graphic, etc.), The operating system provides programs with services thrrough system programs and Application Program Interfaces (APIs).

basics of computer hardware

A **computer** is a programmable machine (or more precisely, a programmable sequential state machine). There are two basic kinds of computers: analog and digital.

Analog computers are analog devices. That is, they have continuous states rather than discrete numbered states. An analog computer can represent fractional or irrational values exactly, with no round-off. Analog computers are almost never used outside of experimental settings.

A **digital computer** is a programmable clocked sequential state machine. A digital computer uses discrete states. A binary digital computer uses two discrete states, such as positive/negative, high/low, on/off, used to represent the binary digits zero and one.

The French word **ordinateur**, meaning that which puts things in order, is a good description of the most common functionality of computers.

what are computers used for?

Computers are used for a wide variety of purposes.

Data processing is commercial and financial work. This includes such things as billing, shipping and receiving, inventory control, and similar business related functions, as well as the "electronic office".

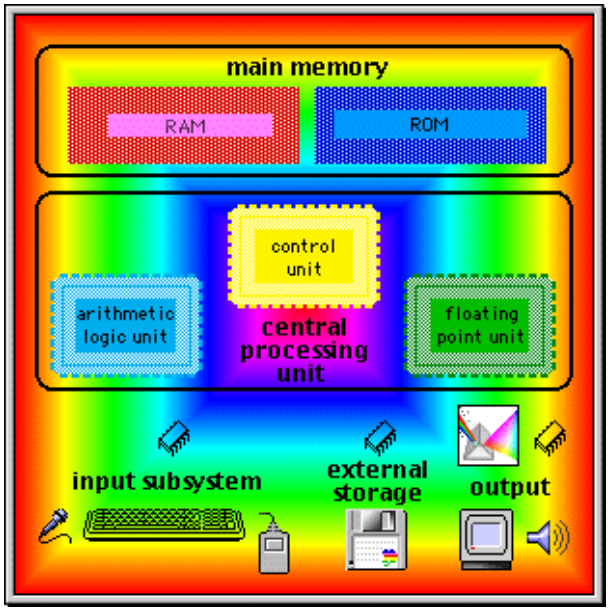
Scientific processing is using a computer to support science. This can be as simple as gathering and analyzing raw data and as complex as modelling natural phenomenon (weather and climate models, thermodynamics, nuclear engineering,

etc.).

Multimedia includes **content creation** (composing music, performing music, recording music, editing film and video, special effects, animation, illustration, laying out print materials, etc.) and multimedia playback (games, DVDs, instructional materials, etc.).

Servers includes web servers. Every website is hosted on a computer called a server. When you connect to a website in your web browser, your computer connects to a web server that provides your web browser with all of the parts (text, pictures, Flash, style sheets, JavaScripts, etc.) needed for your web browser to display any particular web page.

parts of a computer



The classic crude oversimplication of a computer is that it contains three elements: processor unit, memory, and I/O (input/output). The borders between those three terms are highly ambigious, non-contiguous, and erratically shifting.

A slightly less crude oversimplification divides a computer into five elements: arithmetic and logic subsystem, control subsystem, main storage, input subsystem, and output subsystem.

- processor
- arithmetic and logic
- control
- main storage
- external storage
- input/output overview
- input
- output

processor

The **processor** is the part of the computer that actually does the computations. This is sometimes called an **MPU** (for main processor unit) or **CPU** (for central processing unit or central processor unit).

A processor typically contains an arithmetic/logic unit (**ALU**), control unit (including processor flags, flag register, or status register), internal buses, and sometimes special function units (the most common special function unit being a floating point unit for floating point arithmetic).

Some computers have more than one processor. This is called **multi-processing**.

The major kinds of digital processors are: CISC, RISC, DSP, and hybrid.

CISC stands for Complex Instruction Set Computer. Mainframe computers and minicomputers were CISC processors, with manufacturers competing to offer the most useful instruction sets. Many of the first two generations of microprocessors were also CISC.

RISC stands for Reduced Instruction Set Computer. RISC came about as a result of academic research that showed that a small well designed instruction set running compiled programs at high speed could perform more computing work than a CISC running the same programs (although very expensive hand optimized assembly language favored CISC).

DSP stands for Digital Signal Processing. DSP is used primarily in dedicated devices, such as MODEMs, digital cameras, graphics cards, and other specialty devices.

Hybrid processors combine elements of two or three of the major classes of processors.

arithmetic and logic

An arithmetic/logic unit (**ALU**) performs integer arithmetic and logic operations. It also performs shift and rotate operations and other specialized operations. Usually floating point arithmetic is performed by a dedicated floating point unit (**FPU**), which may be implemented as a co-processor.

An arithmetic/logic unit (**ALU**) performs integer arithmetic and logic operations. It also performs shift and rotate operations and other specialized operations. Usually floating point arithmetic is performed by a dedicated floating point unit (**FPU**), which may be implemented as a co-processor.

control

Control units are in charge of the computer. Control units fetch and decode machine instructions. Control units may also control some external devices.

A **bus** is a set (group) of parallel lines that information (data, addresses, instructions, and other information) travels on inside a computer. Information travels on buses as a series of electrical pulses, each pulse representing a one bit or a zero bit (there are trinary, or three-state, buses, but they are rare). An **internal bus** is a bus inside the processor, moving data, addresses, instructions, and other information between registers and other internal components or units. An **external bus** is a bus outside of the processor (but inside the computer), moving data, addresses, and other information between major components (including cards) inside the computer. Some common kinds of buses are the system bus, a data bus, an address bus, a cache bus, a memory bus, and an I/O bus.

main storage

Main storage is also called **memory** or internal memory (to distinguish from external memory, such as hard drives).

RAM is Random Access Memory, and is the basic kind of internal memory. RAM is called “random access” because the processor or computer can access *any* location in memory (as contrasted with sequential access devices, which must be accessed in order). RAM has been made from reed relays, transistors, integrated circuits, magnetic core, or anything that can hold and store binary values (one/zero, plus/minus, open/close, positive/negative, high/low, etc.). Most modern RAM is made from integrated circuits. At one time the most common kind of memory in mainframes was magnetic core, so many older programmers will refer to main memory as **core memory** even when the RAM is made from more modern technology. **Static RAM** is called static because it will continue to hold and store information even when power is removed. Magnetic core and reed relays are examples of static memory. **Dynamic RAM** is called dynamic because it loses all data when power is removed. Transistors and integrated circuits are examples of dynamic memory. It is possible to have battery back up for devices that are normally dynamic to turn them into static memory.

ROM is Read Only Memory (it is also random access, but only for reads). ROM is typically used to store thigns that will never change for the life of the computer, such as low level portions of an operating system. Some processors (or variations within processor families) might have RAM and/or ROM built into the same chip as the processor (normally used for processors used in standalone devices, such as arcade video games, ATMs, microwave ovens, car ignition systems, etc.). **EPROM** is Erasable Programmable Read Only Memory, a special kind of ROM that can be erased and reprogrammed with specialized equipment (but not by the processor it is connected to). EPROMs allow makers of industrial devices (and other similar equipment) to have the benefits of ROM, yet also allow for updating or upgrading the software without having to buy new ROM and throw out the old (the EPROMs are collected, erased and rewritten centrally, then placed back into the machines).

Registers and **flags** are a special kind of memory that exists inside a processor. Typically a processor will have several internal registers that are much faster than main memory. These registers usually have specialized capabilities for arithmetic, logic, and other operations. Registers are usually fairly small (8, 16, 32, or 64 bits for integer data, address, and control registers; 32, 64, 96, or 128 bits for floating point registers). Some processors separate integer data and address registers, while other processors have general purpose registers that can be used for both data and address purposes. A processor will typically have one to 32 data or general purpose registers (processors with separate data and address registers typically split the register set in half). Many processors have special floating point registers (and some processors have

general purpose registers that can be used for either integer or floating point arithmetic). Flags are single bit memory used for testing, comparison, and conditional operations (especially conditional branching).

external storage

External storage (also called **auxillary storage**) is any storage other than main memory. In modern times this is mostly hard drives and removeable media (such as floppy disks, Zip disks, DVDs, CDs, other optical media, etc.). With the advent of USB and FireWire hard drives, the line between permanent hard drives and removeable media is blurred. Other kinds of external storage include tape drives, drum drives, paper tape, and punched cards. Random access or indexed access devices (such as hard drives, removeable media, and drum drives) provide an extension of memory (although usually accessed through logical file systems). Sequential access devices (such as tape drives, paper tape punch/readers, or dumb terminals) provide for off-line storage of large amounts of information (or back ups of data) and are often called I/O devices (for input/output).

input/output overview

Most external devices are capable of both input and output (I/O). Some devices are inherently input-only (also called read-only) or inherently output-only (also called write-only). Regardless of whether a device is I/O, read-only, or write-only, external devices can be classified as block or character devices.

A **character** device is one that inputs or outputs data in a stream of characters, bytes, or bits. Character devices can further be classified as serial or parallel. Examples of character devices include printers, keyboards, and mice.

A **serial** device streams data as a series of bits, moving data one bit at a time. Examples of serial devices include printers and MODEMs.

A **parallel** device streams data in a small group of bits simultaneously. Usually the group is a single eight-bit byte (or possibly seven or nine bits, with the possibility of various control or parity bits included in the data stream). Each group usually corresponds to a single character of data. Rarely there will be a larger group of bits (word, longword, doubleword, etc.). The most common parallel device is a printer (although most modern printers have both a serial and a parallel connection, allowing greater connection flexibility).

A **block** device moves large blocks of data at once. This may be physically implemented as a serial or parallel stream of data, but the entire block gets transferred as single packet of data. Most block devices are random access (that is, information can be read or written from blocks anywhere on the device). Examples of random access block devices include hard disks, floppy disks, and drum drives. Examples of sequential access block devcies include magnetic tape drives and high speed paper tape readers.

input

Input devices are devices that bring information into a computer.

Pure input devices include such things as punched card readers, paper tape readers, keyboards, mice, drawing tablets, touchpads, trackballs, and game controllers.

Devices that have an input component include magnetic tape drives, touchscreens, and dumb terminals.

output

Output devices are devices that bring information out of a computer.

Pure output devices include such things as card punches, paper tape punches, LED displays (for light emitting diodes), monitors, printers, and pen plotters.

Devices that have an output component include magnetic tape drives, combination paper tape reader/punches, teletypes,

and dumb terminals.

Unix and Linux history

chapter 3

summary

This chapter looks at the history of Unix and Linux.

The history can help you understand why things are they way they are in Unix and Linux.

Of particular concern are:

- the small storage space of early computers which resulted in short command names and one character options
- the switchover from teletype I/O to video terminal I/O
- the use of eight bit bytes, ASCII characters, and plain text for interprocess communication
- treating files as collections of bytes and devices, directories, and certain kinds of inter-process communications as files
- the use of small, single-purpose programs that can be easily combined together to perform complex tasks

Unix history

When Unix came into existence digital computers had been in commercial use for more than a decade.

Mainframe computers were just starting to be replaced by minicomputers. The IBM 360 mainframe computer and clones by other manufacturers was still the dominant computer. The VAX, with its virtual memory addressing, was still in development. Microcomputers were just an experimental idea. The CDC mainframe, a forerunner of the more famous Cray supercomputers, was still the world's fastest computer.

COBOL was the most common programming language for business purposes. FORTRAN was the most popular programming language for scientific and systems programming. PL/I was a common third language. ALGOL was dominate in the academic community.

Most data entry was still performed using punched cards. Directly connected terminals were typically teletypes (TTY), a technology originally developed for telegraph systems. New terminals combining a keyboard and a monitor were still rare. These were also called CRTs (for Cathode Ray Tube) or dumb terminals or smart terminals (depending on the capabilities). Unlike modern black letters on white background terminals (pioneered with the Apple Lisa and Apple Macintosh) and full color monitors (popularized with the Atari and Commodre Amiga personal computers), these monitors were a single color phosphor (usually green) on a dark gray background. Even though disk drives existed, they were expensive and had small capacities. Magnetic tape was still the most commonly used form of storage of large data sets. Minicomputers were making use of punched paper tape.

Interactive computing was still rare.

Bell Telephone Laboratories was still using the BESYS operating system from 1957. In 1964, Bell Labs decided to update their operating system, leading to a 1965 partnership between the Massachusetts Institute of Technology (MIT), AT&T Bell Labs, and General Electric to create an experimental operating system called Multics (MULTIplexed operating and Computing System) for the GE-645 mainframe computer. AT&T intended to offer subscription-based computing services over the phone lines, an idea similar to the modern cloud approach to computing.

While the Multics project had many innovations that went on to become standard approaches for operating systems, the project was too complex for the time.

Bell Labs pulled out of the Multics project in 1969.

Ken Thompson and Dennis Ritchie offered to design a new operating system using a DEC PDP-7 computer that was available at the time.

UNICS (UNIplexed operating and Computing System, a pun on MULTICS) was created at AT&T's Bell Labs in 1969 by a group that included Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, Michael Lesk and Joe Ossanna.

This group of researchers, the last of the AT&T employees involved in Multics, decided to attempt the goals of Multics on a much more simple scale.

"What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication," according to Dennis Ritchie.

Unix was originally intended as a programmer's workbench. The original version was a single-user system. As Unix spread through the academic community, more general purpose tools were added, turning Unix into a general purpose operating system.

While Ken Thompson still had access to the Multics environment, he wrote simulations on Multics for Unix's file and paging system.

Ken Thompson ported the Space Travel computer game from Multics to a Digital Equipment Corporation (DEC) PDP-7 he found at Bell Labs.

Ken Thompson and Dennis Ritchie led a team of Bell Labs researchers (the team included Rudd Canaday) working on the PDP-7 to develop a hierarchial file system, computer processes, device files, a command-line interpreter, and a few small utility programs.

Early work was done in a BCPL (Basic Combined Programming Language) a language intended for writing compilers and system software. BCPL was simplified and revised into the B programming language.

By the end of 1969, UNICS had a primitive kernel, an editor, an assembler, a simple shell command interpreter, and a few basic command utilities, including `rm`, `cat`, and `cp`.

The first Unix command shell, called the Thompson shell and abbreviated `sh`, was written by Ken Thompson at AT&T's Bell Labs, was much more simple than the famous Unix shells that came along later. The Thompson shell was distributed with Versions 1 through 6 of Unix, from 1971 to 1975.

In 1970 Dennis Ritchie and Ken Thompson traded the promise to add text processing capabilities to Unix for the use of a Digital Equipment Corporation (DEC) PDP-11/20. The initial version of Unix, a text editor, and a text formatting program called `roff` were all written in PDP-11/20 assembly language.

The PDP-11/20 computer had 24K of RAM. The operating system used 12K. The remaining 12K was split between application programs and a RAM disk. The file size limit was 64K and the disk size limit was 512K.

Soon afterwards `roff` evolved into `troff` with full typesetting capability. The first Unix book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, was published on November 3, 1971. The book described more than 60 commands, including `b` (compile a B program), `chdir` (change working directory), `chmod`, `chown`, `ls`, and `who`.

By 1971, many fundamentally important aspects of Unix were already in place, including file ownership and file access permissions.

The first commercial Unix system was installed in early 1972 at the New York Telephone Co. Systems Development Center under the direction of Dan Gielan. Neil Groundwater build an Operational Support System in PDP-11/20 assembly language.

With funding from the Patent Department, the Unix team upgraded to a DEC PDP-11/45.

In 1972, Dennis Ritchie and Ken Thompson rewrote the B programming language into C. Work started on converting Unix to C. Unix was originally written in assembly language and B, but by 1973 it had been almost completely converted to the C language. At the time it was common belief that operating systems must be written in assembly in order to perform at reasonable speeds. Writing Unix in C allowed for easy portability to new hardware, which in turn led to the Unix operating system being used on a wide variety of computers.

Ken Thompson invented the pipe in 1972. Pipes allow the output of one process to be used as the input for another. Pipes allow the Unix philosophy of having many small programs that each do one function very well and then write scripts that combine these small utility programs to accomplish bigger tasks.

By 1973, Unix was running at sixteen (16) sites, all within AT&T and Western Electric. Unix was presented to the public in an October 1973 conference and within six months the number of sites running Unix had tripled.

A version of Unix was published in the July 1974 issue of the *Coomunications of the ACM*, leading to widespread requests for copies. Under the antitrust consent decree with the U.S. government, Bell Labs was required to give a free copy to anyone who requested it. Ken Thompson personally signed a note for many of these early copies.

Because C was a complete, platform-independent programming language and Unix's more than 11,000 lines of code were written in C, it was possible to easily port Unix to any computer platform. And Unix did quickly spread to almost every available computer platform, becoming the first true universal operating system.

In fall of 1974, Ken Thompson went to the University of California at berkeley to teach for a year. Ken Thompson, Bill Joy, and Chuck Haley worked together to create the Berkeley version of Unix, called the Berkeley Software Distribution, or more commonly BSD.

The souce code for BSD Unix was widely distributed to students and other schools. Students at UC-Berkeley and other schools around the world worked on improvements and the BSD Unix replaced the Bell Laboratories Unix as the primary Unix.

BSD Unix added the `vi` editor, the C shell, the `Sendmail` email system, virtual memory, and support for TCP/IP (Transmission Control Protocol/Internet Protocol). Because email was built into the operating system, email was the method Unix used for sending notifications to the system administrator of system status, reports, problems, etc.

The Fifth Edition of Unix was released in 1975 and licensed to universities for free.

The PWB shell or Mashey shell, abbreviated `sh`, was a variation of the Thompson shell that had been augmented by John Mashey and others at Bell Labs. The Mashey shell was distributed with the Programmer's Workbench Unix in 1976.

The Bourne shell, created by Stephen Bourne at AT&T's Bell Labs (in New Jersey) as a scripting language, was released in 1977 as the default shell in the Version 7 Unix release distributed to colleges and universities. It gained popularity with the publication of *The UNIX Programming Environment* by Brian W. Kernighan and Rob Pike. The book was the first commercially published tutorial on shell programming.

The Bourne shell provided process control, variables, regular expressions, flow control, input/output control, and functions.

In 1977, the University of California, Berkeley, released the Berkeley Software Distribution (BSD) version of Unix, based on the 6th edition of AT&T Unix.

The C shell, abbreviated `csh`, was created by Bill Joy, a graduate student at the University of California, Berkeley. With additional work by Michael Ubell, Eric Allman, Mike O'Brien, and Jim Kulp, it was released in the 2BSD release of BSD Unix in 1978.

The C shell offered command history, aliases, file name completion, and job control.

It turned out that while the C shell was easier to use than the Bourne shell, it failed miserably as a scripting language.It

became common to use the C shell for everyday interactive computing and to use the Bourne shell for script programming.

The improved C shell, abbreviated `tcsh`, was created by Ken Greer by September 1975 and merged into the C shell in December 1983. Ken Greer based the shell on the TENEX operating system (hence, the "t" in `tsch`). Mike Ellis, Paul Placeway, and Christos Zoulas made major contributions to `tcsh`.

The release of the Seventh Edition of Unix in 1978 led to the divergence of Unix on the System V (Bell Labs) and BSD (Berkeley) paths. System V was often called sys-five. System V was the for pay version and BSD was the free open source version.

The Korn shell, abbreviated `ksh`, was created by David Korn at AT&T's Bell Labs and announced at USENIX on July 14, 1983. Mike Veach and Pat Sullivan were also early contributors. The Korn shell added C shell features to the Bourne shell (command history and history substitution, command aliases, and file name completion). The Korn shell also added arrays and built-in integer arithmetic.

When the AT&T broke up in 1984 into "Baby Bells" (the regional companies operating local phone service) and the central company (which had the long distance business and Bell Labs), the U.S. government allowed them to start selling computers and computer software.

AT&T gave academia a specific deadline to stop using "encumbered code" (that is, any of AT&T's source code anywhere in their versions of Unix). This led to the development of free open source projects such as FreeBSD, NetBSD, and OpenBSD, as well as commercial operating systems based on the BSD code.

Meanwhile, AT&T developed its own version of Unix, called System V. Although AT&T eventually sold off Unix, this also spawned a group of commercial operating systems known as Sys V Unixes.

Unix quickly swept through the commercial world, pushing aside almost all proprietary mainframe operating systems. Only IBM's MVS and DEC's OpenVMS survived the Unix onslaught.

Some of the famous official Unix versions include Solaris, HP-UX, Sequent, AIX, and Darwin. Darwin is the Unix kernel for Apple's OS X, AppleTV, and iOS (used in the iPhone, iPad, and iPod).

The BSD variant started at the University of California, Berkeley, and includes FreeBSD, NetBSD, OpenBSD, and DragonFly BSD.

The original Sun OS was based on BSD, but Solaris V5 was based on System V, release V.

Most modern versions of Unix combine ideas and elements from both Sys-V and BSD. HP-UX and Solaris are primarily System V, while AIX and Mac OS X are hybrids of both flavors of Unix.

Other Unix-like operating systems include MINIX and Linux.

In 1983, Richard Stallman founded the GNU project, which eventually provided the tools to go with Linux for a complete operating system.

In 1986, Maurice J. Bach of AT&T Bell Labs published *The Design of the UNIX Operating System*, which described the System V Release 2 kernel, as well as some new features from release 3 and BSD.

In 1987, Andrew S. Tanenbaum released MINIX, a simplified version of Unix intended for academic instruction.

`bash`, which stands for Bourne Again SHell, was created by Brian Fox for the Free Software Foundation and first released on June 7, 1989. `bash` combined features from the Bourne shell, the C shell, and the Korn shell. Bash also introduced name completion for variable names, usernames, host names, commands, and file names, as well as spelling correction for pathnames in the `cd` command, arrays of unlimited size, and integer arithmetic in any base between 2 and 64. `bash` is now the primary shell in both Linux and Mac OS X.

The Z shell, abbreviated `zsh`, was written by Paul Flastad in 1990 when he was a student at Princton University.

The Linux operating system was first released on September 17, 1991, by Finnish student Linus Torvalds. With the permission of Andrew S. Tanenbaum, Linus Torvalds started work with MINIX. There is no MINIX source code left in Linux. Linux Torvalds wrote Linux using the GNU C compiler running on MINIX on an Intel 80386 processor.

Linus Torvalds started work on open source Linux as a college student. After Mac OS X, Linux is the most widely used variation of Unix.

Linux is technically just the kernel (innermost part) of the operating system. The outer layers consist of GNU tools. GNU was started to guarantee a free and open version of Unix and all of the major tools required to run Unix.

In 1992 Unix System Laboratories sued Berkeley Software Design, Inc and the Regents of the University of California to try to stop the distribution of BSD Unix. The case was settled out of court in 1993 after the judge expressed doubt over the validity of USL's intellectual property.

See the appendix for a more detailed summary of the history of computers.

choice of shells

chapter 4

summary

This chapter looks at the shells available for Linux and Unix.

This includes an explanation of what a shell is and a list of the popular shells.

what is a shell?

Unix (and therefore also Mac OS X and Linux) has more than 200 basic commands (also called tools or utilities) that are distributed with the standard operating system. This collection and the ease with which they work together is the major source of the power of Unix. The vast majority of these standard tools are designed to be used from a command line (the shell).

A shell is primarily a command interpreter.

In a graphical user interface such as Macintosh or Windows, the user controls the computer and programs primarily through pointing and clicking, supplemented by some typing.

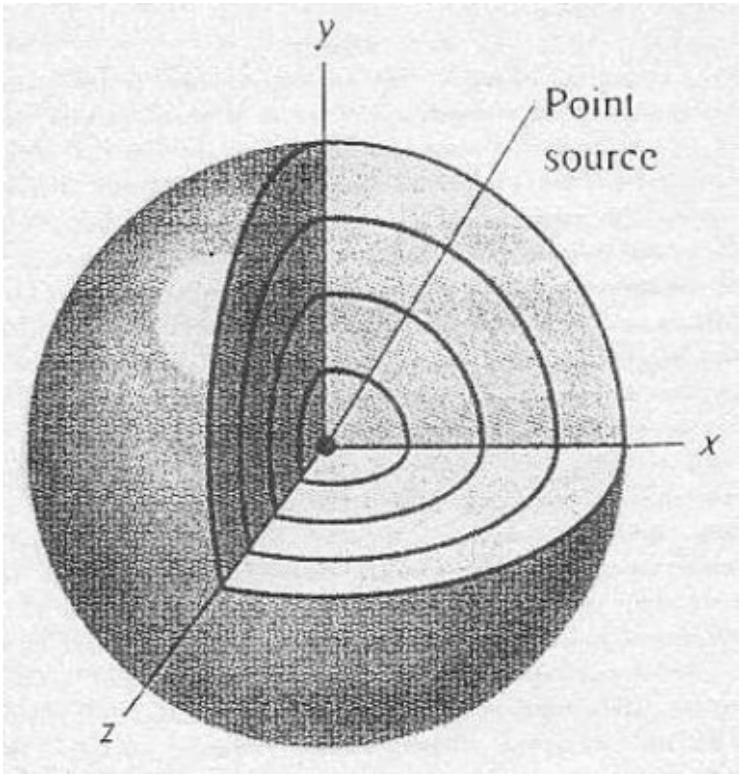
In a shell, all of the commands are typed. The shell figures out the meaning of what you typed and then has the computer do as instructed.

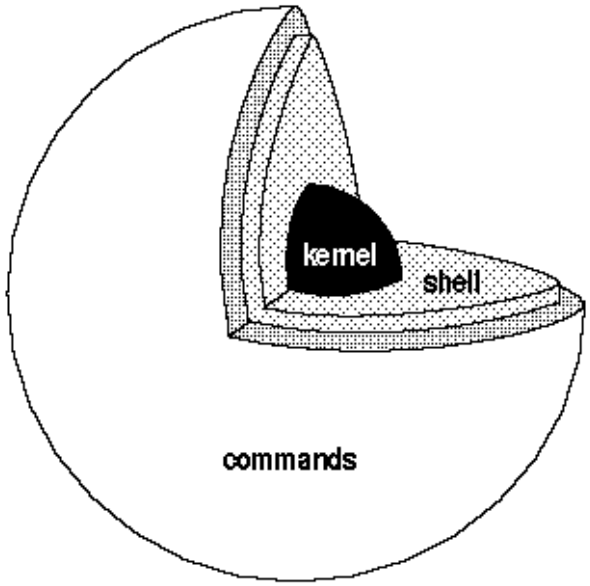
But the shell is much more than just a command interpreter. It is also a complete programming language.

Because the shell is a complete programming language, with sequences, decisions, loops, and functions, it can do things well beyond pointing and clicking. It can take control of your computer and react to changing circumstances.

Programming languages are used to make all of the computer programs and smart phone apps you've ever seen or used. Your imagination is the only limit on the power of the shell. Anything that can be done with a computer can be done with the shell.

The term *shell* is derived from the idea of concentric layers that occur in many seashells. One view is that programs can be run inside of the shell, including other copies of the shell. These layers can be arbitrarily deep. An inverse view is that the kernel is the inner most layer, with the shell wrapped around the kernel and application programs wrapped around the shell.





choice of shells

Most versions of Unix and Linux offer a choice of shells.

The term shell comes from a concept of Unix being a series of layers of software that work together rather than one single monolithic operating system program. The command line shell was part of the outer shell of a series of programs that eventually reach down to the kernel or innermost portion of the operating system.

Steve Bourne created the original Bourne shell called sh.

Some other famous shells were C Shell (csh), Korn Shell (ksh), Turbo C Shell (tsch), and the Z Shell (zsh). A shell that combined the major advntages of each of these early shells was the Bourne Again SHell (BASH).

The default shell in Linux and MacOS X is normally BASH (Bourne Again SHell). The following instruction assumes the use of BASH.

It is possible to change the shell you are currently using (especially useful if you need to run a line or script from an alternative shell).

original shell

The original shell, called the Thompson shell and abbreviated `sh` was written by Ken Thompson at AT&T's Bell Labs, was very rudimentary. The Thompson shell was distributed with Versions 1 through 6 of Unix, from 1971 to 1975.

Note that the same `sh` is also used for other shells, including the Bourne shell.

The PWB shell or Mashey shell, abbreviated `sh`, was a variation of the Thompson shell that had been augmented by John Mashey and others at Bell Labs. The Mashey shell was distributed with the Prorgammer's Workbench Unix in 1976.

Bourne shell

Stephen Bourne created the Bourne shell, abbreviated `sh`, at AT&T's Bell Labs in New Jersey as a scripting language.

The Bourne shell was released in 1977 as the default shell in the Version 7 Unix release distributed to colleges and universities. It gained popularity with the publication of *The UNIX Programming Environment* by Brian W. Kernighan and Rob Pike. The book was the first commercially published tutorial on shell programming.

The Bourne shell (or a shell that is compatible and will run Bourne shell scripts) is usually located at `/bin/sh`.

You will also see references to the Bourne shell as `bsh`.

The Bourne shell has a command language based on the ALGOL programming language, including process control, variables, regular expressions, flow control (decisions and loops), powerful input and output control, and functions.

Despite its power, the original Bourne shell was difficult to use and lacked file name completion, command history, aliases, and had difficulty running multiple background jobs.

C shell

The C shell, abbreviated `csh`, was created by Bill Joy, a graduate student at the University of California, Berkeley. With additional work by Michael Ubell, Eric Allman, Mike O'Brien, and Jim Kulp, it was released in the 2BSD release of BSD Unix in 1978.

The C shell was based on the style of the C programming language.

The C shell was much easier to use than the Bourne shell, but was unable to perform anything other than the most simple scripts.

The C shell lacked functions, had weak input and output control, and had limited programming ability due to a poorly written command interpreter (Bill Joy explained that at the time he didn't know about many standard compiler techniques).

The power of the C shell for everyday interactive use was due to its introduction of command history, aliases, file name completion, and job control.

It became common to use the C shell for everyday interactive computing and to use the Bourne shell for script programming.

The improved C shell or TENEX C shell, abbreviated `tcsh`, was created by Ken Greer by September 1975 and merged into the C shell in December 1983. Ken Greer based the shell on the TENEX operating system (hence, the "t" in `tsch`). Mike Ellis, Paul Placeway, and Christos Zoulas made major contributions to `tcsh`.

The Hamilton C Shell was written in 1988 by Nicole Hamilton of Hamilton Laboratories for the OS/2 operating system. In 1992, the Hamilton C Shell was released for Windows NY. The OS/2 version was discontinued in 2003. The Windows version, with adaptations for Windows peculiarities, is still in use.

Korn shell

The Korn shell, abbreviated `ksh`, was created by David Korn at AT&T's Bell Labs and announced at USENIX on July 14, 1983. Mike Veach and Pat Sullivan were also early contributors.

After years of Unix users having to learn two different shells, one for scripting and oen for interactive control, David Korn built the Korn shell by adding C shell features to the Bourne shell.

In addition to adding command history, history substitution, aliases, and file name completion from the C shell, the Korn

shell also included arrays and built-in integer arithmetic.

The three major versions of the Korn shell are the official version `ksh`, the public domain version `pdksh`, and the desktop version `dtksh`

CDE shipped with the Desktop version of the Korn shell.

The POSIX shell is a variant of the Korn Shell. HP shipped the POSIX shell with HP-UX 11.0

BASH shell

`bash`, which stands for Bourne Again SHell, was created by Brian Fox for the Free Software Foundation and first released on June 7, 1989.

`bash` combined features from the Bourne shell, the C shell (both the original C shell and tcsh), and the Korn shell, while using the syntax of the original Bourne shell, allowing it to run almost all historical Bourne shell scripts. `bash` is now the primary shell in both Linux and Mac OS X.

tt>bash introduced name completion for variable names, usernames, host names, and commands (in addition to file names), spelling correction for pathnames in the `cd` command, arrays of unlimited size, and integer arithmetic in any base from 2 to 64.

Z shell

The Z shell, abbreviated `zsh`, was written by Paul Flastad in 1990 when he was a student at Princton University.

The Z shell combines features of the Bourne Shell, the C shell, and `bash`.

other shells

The POSIX shell, abbrevaited `sh`, (POSIX = Portable System Interface) is based on the Korn shell and is the official IEEE P1003.2 shell. It is the `/bin/sh` shell on some systems.

The Almquist shell, abbreviated `ash`, was written by Keneth Almquist as a replacement for the Bourne shell written for BSD systems. The `sh` in FreeBSD and NetBSD were based on the Almquist shell.

The Debian Almquist shell, abbreviated `dash`, is a modern replacement for the Almquist shell in Debian and Ubuntu Linux. While the default shell in both Debian and Ubuntu Linux are `bash`, `/bin/sh` is a symlink to `ash`.

`ash` and `dash` are minimalist shells that interpret scripts and are not intended for interactive use. Their small size and fast execution time make them good for embedded systems. Because many scripts call for the use of `/bin/sh`, Debian and Ubuntu Unix gain speed of execution for scripts, while still allowing `bash` as the primary interactive shell.

The Public domain Korn shell, abbreviated `pdksh`, is based on the Korn shell.

The MirBSD Korm shell, abbreviated `mksh`, is based on the OpenBSD version of the Korn shell and the `pdksh` and was released as part of MirOS BSD.

Busybox is a set of tiny utilities, including a version of `ash`, used in embedded systems.

`rc`, written by Tom Duff, was the default shell for AT&T's Bell Labs' Plan 9 and Version 10 Unix. Plan 9 was an improved operating system based on Unix, but was not a significant enough improvement to overtake its predecessor. `rc` is available on some modern versions of Unix and Unix-like operating systems.

`es` is an `rc`-compatible shell written in the mid-1990s for functional programming.

Perl shell, abbreviated `psh`, is a shell that combines `bash` and the Perl scripting language. it is available for Unix-like and Windows operating systems.

Friendly interactive shell, abbreviated `fish`, was released in 2005.

`pysh` is a profile in the IPython project that combines a Unix-style shell with a Python scripting language shell.

`wish` is a windowing shell for Tcl.Tk.

Which Shell to Use

from Google Shell Style Guide

`Bash` is the only shell scripting language permitted for executables.

Executables must start with `#!/bin/bash` and a minimum number of flags. Use `set` to set shell options so that calling your script as `bash <script_name>` does not break its functionality.

Restricting all executable shell scripts to **bash** gives us a consistent shell language that's installed on all our machines.

The only exception to this is where you're forced to by whatever you're coding for. One example of this is Solaris SVR4 packages which require plain Bourne shell for any scripts.

connecting to a shell

chapter 5

This chapter looks at how to connect to a shell.

On your local desktop or laptop computer, many of these steps won't apply, but you will probably need most or all of the steps when logging in remotely to control a server.

If you are connecting to a remote computer (such as web server), then you can use `telnet` or `SSH`.

If you connecting to the shell from a modern graphic user interface, then you need to start up a terminal emulator.

If you connecting to a mainframe computer, minicomputer, or the equivalent, you use any terminal connected to the system.

NOTE: On a single-user Linux or Mac OS X desktop/laptop computer, you can start the Terminal program and be automatically signed in (no need for account login or password). On a remote server, you will always need to login in with at least account name and password.

physical terminal

For old style systems where Unix runs on a mainframe or minicomputer, there will be actual physical terminals that are connected by wires or MODEM to the mainframe or minicomputer. These systems are increasingly rare.

The terminal may already be at the `login` prompt. If not, try pressing the RETURN key several times until the `login` prompt shows up.

Telnet and SSH

`telnet` or `SSH` are the two basic methods for connecting to a remote server.

`telnet` was the original method. There are `telnet` client programs available for most computer systems. Unfortunately, `telnet` performs all communications in plain text, including passwords, which means that anyone malicious along the path between you and your server can easily read the information needed to hack into your system using your account. For this reason, many servers no longer allow `telnet` access.

`SSH` is Secure Shell. It has the same basic capabilities as the older `telnet`, but includes security, including encrypted communications. There are `SSH` client programs for most modern computer systems.

You can also use `SSH` through a terminal emulator, which is how many system administrators now access their servers from their personal or portable computer.

terminal emulator

As mentioned in the history chapter, the Unix shells were accessed through interactive terminals, originally teletype machines and later special combinations of keyboards and cathode ray tubes (CRTs).

There are programs for graphic user interfaces that allow you to interact with a shell as if you were using one of those ancient terminal devices. These are called terminal emulators (they emulate the signals of a terminal).

Some common terminal emulators include eterm, gnome-terminal, konsole, kvt, nxterm, rxvt, terminal, and xterm.

Note that if you are logging in to a web server or other remote computer, you should use an SSH (Secure SHell) client

program instead of a terminal emulator. A terminal emulator program is for gaining shell access to a personal computer or workstation. With a little bit more knowledge, you can use the shell on a personal computer or workstation as your SSH client.

special settings

If you are starting up Terminal from a modern operating system, you will almost always be able to start using the shell without any login name or password.

If you are logging in to a remote server you may need a few pieces of additional information beyond just your login name and password.

Terminfo: You may need to know the Terminfo name for your terminal. The Terminfo name is used to set the characteristics of your terminal. A common default is the old DEC VT-100 terminal. On old (early) versions of Unix you may have to use a Termcap name instead of a Terminfo name.

End of line: On most modern systems, the RETURN key will indicate the end of line. On some older systems you may find a RET, NEWLINE, ENTER< or other key that serves this purpose. On some systems, that have both a RETURN and an ENTER key, both will serve the same purpose.

Erase key: On most modern systems, the DELETE key will erase one character at a time. Typically the CONTROL-H combination (hold down both the H key and the CONTROL key, sometimes marked CNTRL or CTRL) will also work as an erase key. On some older systems, the # key will also serve as the erase key.

Line kill key: On some older systems the @ key deletes the entire line being entered.

Interrupt key: There is typically a key that will interrupt the execution of running programs.

You can find out these items from the tech support for your web hosting, from the system administrator of a system, or from the manual for your system.

shell basics

chapter 6

summary

This chapter looks at Unix (and Linux) shell basics.

This includes an explanation of what a shell is, how to get to your shell, and a simple example of how to use your shell.

The short version of this chapter is to start up your terminal emulator (or SSH client), type in your account name at the `login` prompt, followed by your password at the `password` prompt (using the RETURN or ENTER button at the end of each line). Then type a valid command (such as `who`) and an invalid command (such as `asdf`) and see the results. The rest of the commentary helps you through common questions and problems.

book conventions

The following conventions are used in this book.

Examples are displayed in a gray box. In the PDF version of this book the gray box does *not* appear, but the box is indented from the regular text. In the on-line version at www.osdata.com the gray box displays correctly. I need to find someone with a legal copy of Adobe Acrobat to help out. If you are in the Costa Mesa, California, area and can help out with a legal copy of Adobe Acrobat, please contact me.

Additionally, anything that will appear on the screen (whether you type it or the shell produces it as output) will appear `in a monospace type`. This will help identify the examples in the PDF version. This also applies to material contained in ordinary descriptive paragraphs that you will type.

```
$ date
Wed Nov 10 18:08:33 PST 2010
$
```

Bold face type is used to indicate items that you type.

There will be no indication of the correct locations to type RETURN or ENTER or other similar special characters. These normally happen at the end of every input line. There will be reminders in the descriptive text near the beginning of the book, as well as in other descriptive text where there is a need for a special reminder or an unusual convention or an unusual keystroke.

```
$ date
```

Bold italics type is used to indicate items that you must customize, such as file names or user account name. In a descriptive paragraph, these items will be in *ordinary italics*.

```
$ AccountName
```

Material that the shell will output will be in ordinary `monospace type`. In some cases where the material will be variable, the shell output will be in *`italics`*.

```
Wed Nov 10 18:08:33 PST 2010
VariableInformation
$
```

root or superuser

Do **not** run your terminal emulator or shell in root or superuser.

Root or superuser has complete control over the entire computer. A regular user account has limitations.

Experienced programmers run BASH as a regular user to protect themselves from destructive mistakes — and they are already skilled at using BASH.

A beginner running BASH as superuser or root can turn a simple typing mistake into a disaster.

If you already have a regular user account created, make sure you use it. If you only have the superuser or root account running, immediately create a regular user account and switch to it for all of your learning exercises.

Both the Bourne style shells (including BASH) and the C style shells display a # prompt when you are running the shell as root. Be careful.

```
#
```

starting your shell

NOTE: On a single-user Linux or Mac OS X desktop/laptop computer, you can start the Terminal program and be automatically signed in (no need for account login or password). On a remote server, you will always need to login in with at least account name and password.

Some information you may need includes the name or URL of your server or computer (sometimes called the host), your account name, and your assigned password. if you are working on a remote server (such as a web server), you need to get this information from your hosting company or system administrator. If you are using a large multi-user system, you need to get this information from your system administrator. If you are using a personal computer or workstation, you probably set these for yourself.

If you have just purchased a new personal computer or workstation, ask your salesperson for the defaults or look them up in the manual. Your install CD or DVD is likely to have a utility that can be used to create accounts and set passwords. The Mac OS X install DVD has utilities for both purposes.

If you are the first person to work with a large system or computer, refer to the manuals that came with your computer. You will find information on the root/superuser account. That is a powerful account and should be reserved for special occasions when that kind of power is needed. Look in the manual section with a title similar to “Getting Started” and then look for an account named *user*, *guest*, *tutor*, or similar name that suggests an ordinary user account.

You login remotely into a web server through SHH (Secure SHell). There are many SSH programs available. You do not have to have a matching operating system. You can use an SSH client program on Windows, Macintosh, Linux, or almost any other operating system to sign into a Linux or UNIX server.

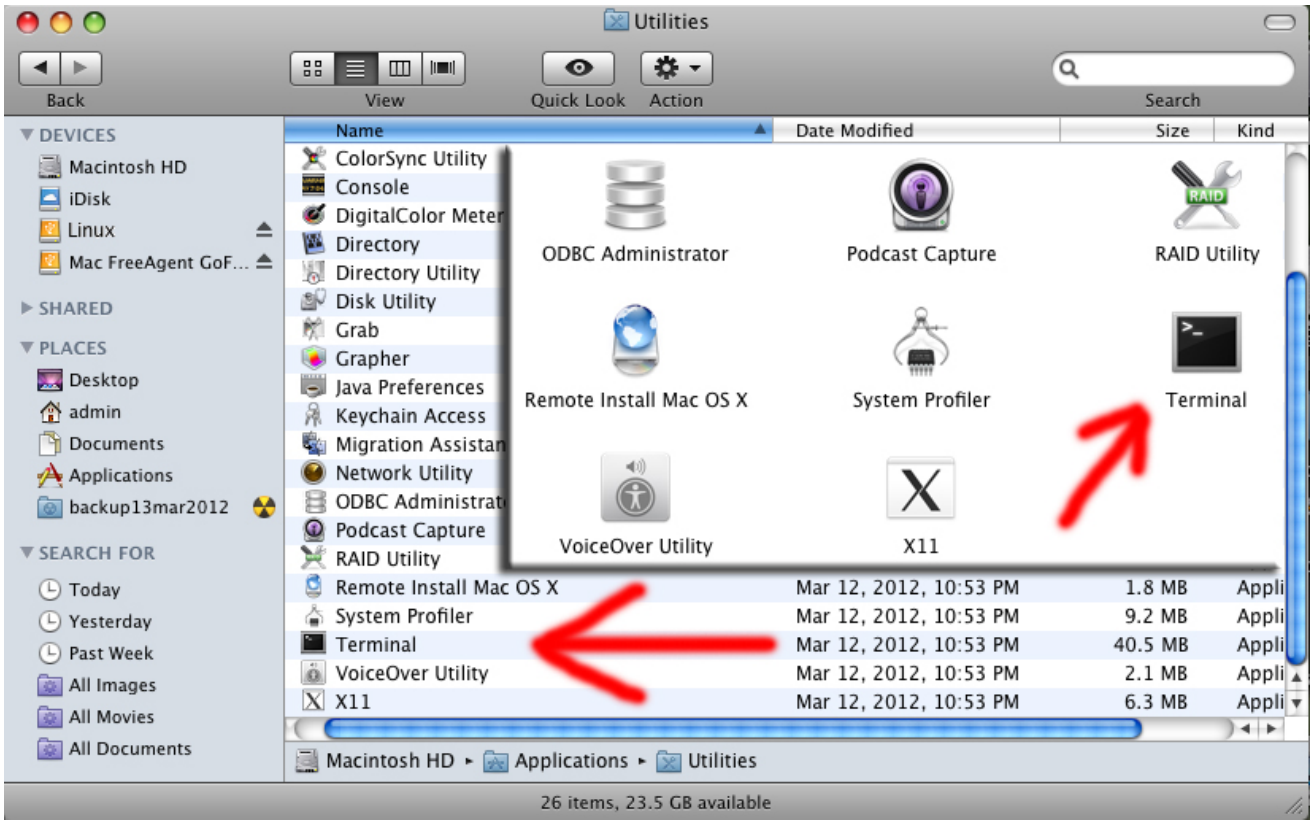
You login into a large multi-user traditional command line UNIX system by typing your user name, ENTER, your password, and ENTER (in that order). This will enter you into the default shell set up for your account.

On a computer that runs a graphic interface (such as Gnome, KDE, and Mac OS X) you will want to use a terminal emulator program. Typically a terminal emulator program starts with you already logged-in to the same account that you

have running in the graphic user interface.

On Mac OS X and most versions of Linux, the icon for the Terminal program will look like this:

On a Mac OS X computer, Terminal will be located in the Utilities directory, which is located in the Applications directory. When you find it, you can drag it onto the Dock to make it easily accessible. Apple purposely hid the UNIX command line to avoid confusing the typical consumer.



On Linux/KDE, look for Konsole or Terminal in the Utilities menu.

On Linux/Gnome, look for color xterm, regular xterm, or gnome-terminal in the Utilities menu.

On Ubuntu Linux, look in Applications menu > Accessories > Terminal. Single click and hold, then drag to your panel, because you will be using it a lot.

On some versions of Linux, you can press the CONTROL and the ALT and the F1 keys all at once to bring up Terminal.

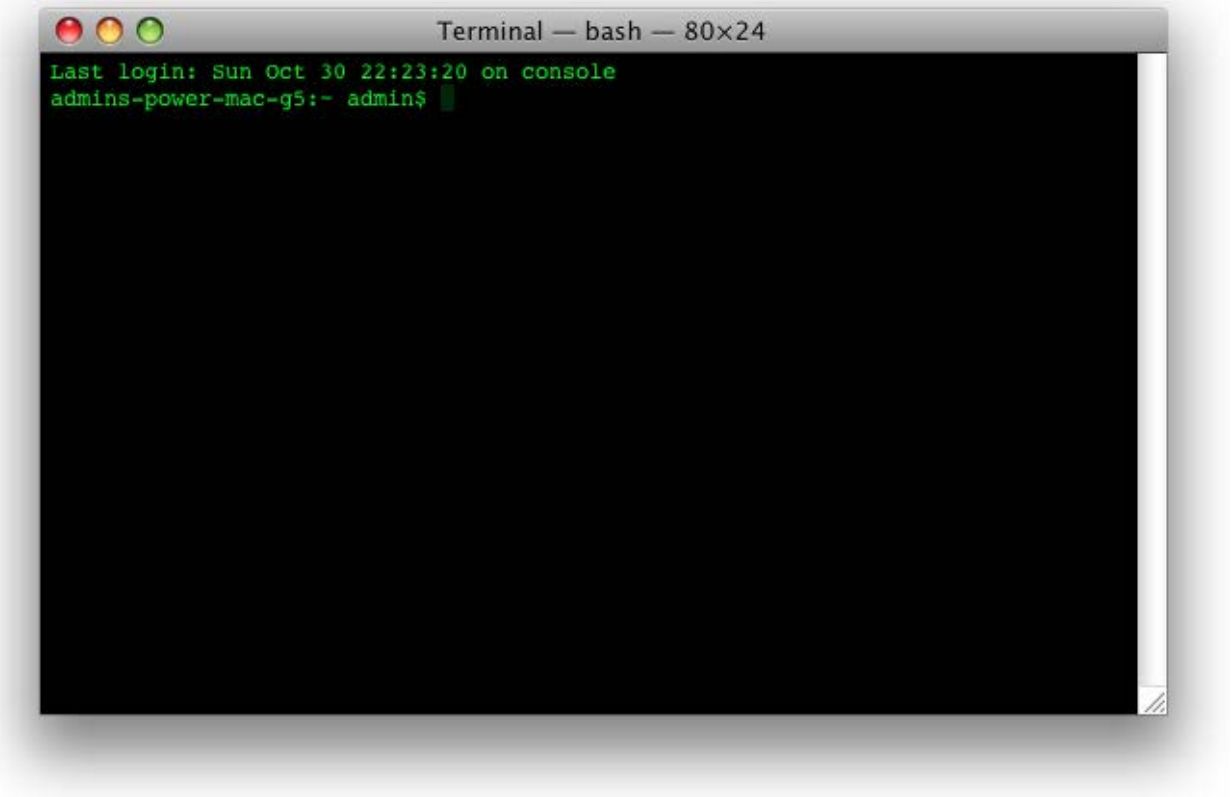
In Gnome, use the Applications menu > Accessories > Terminal or the keyboard shortcut of CONTROL and ALT and T (all at the same time).

In KDE, use the KMenu > System > Terminal Program (Konsole).

In Linux Mint you can use the keyboard shortcut of holding down the CONTROL and ALT and T keys all at once.

As a last resort, you can use the file search feature in Linux or Mac OS X and search for “Terminal”.

You may want to try out the various terminal emulators and decide which works best for you (each offer slightly different features).



login and password

NOTE: On a single-user Linux or Mac OS X desktop/laptop computer, you can start the Terminal program and be automatically signed in (no need for account login or password). On a remote server, you will always need to login in with at least account name and password.

At the `login` prompt type your account name (in lower case letters) followed by ENTER or RETURN.

```
login: accountname
Password:
```

At the `password` prompt type your password followed by ENTER or RETURN.

```
Password: password
$
```

If you encounter problems, see the next chapter for help.

prompt

Once the shell is running it will present a shell prompt. This shell prompt is normally the U.S. dollar sign ($). Other common prompts are the percent sign (%) and the pound sign (#). The shell prompt lets you know that the shell is ready and waiting for your input.

```
$
```

The most common prompt in the Bourne shell (`sh` or `bsh`) and Bourne Again shell (`bash`) and Korn shell (`ksh`) is the U.S. dollar sign ($).

```
$
```

When the the Bourne Again shell (`bash`) or Korn shell (`ksh`) is running as root or superuser, the prompt is changed to the U.S. number or pound sign (#) as a clear reminder that you are using a much more powerful account.

```
#
```

The Bourne Again shell (`bash`) shell typically reports the current working directory and user name before the `$` prompt, often inside of square braces. An example:

```
[admin:~ admin]$
```

The most common prompt in the C shell (`csh`) and TENEX C shell (`tsch`) is the percent sign (%).

```
%
```

When running as root or superuser, the C shell (`csh`) shows the U.S. number or pound sign sign (#).

```
#
```

The Z shell (`zsh`) shows the short machine name followed by the percent sign (%).

```
mac%
```

When running as root or superuser, the Z shell (`zsh`) shows the short machine name followed by the U.S. number or pound sign sign (#).

```
mac#
```

Almost all of the shell examples in this book will use **bold face** to show what you type and plain fixed space characters to show shell output. Items that you need to modify according to your local needs are normally going to be presented in *italics.*

You are likely to see information before the actual command prompt. It may the current working directory. It may be the current user name. There are ways to change what is presented. For now, don’t worry about it, just be aware that you are likely to see some text before the prompt.

example command

You can run a UNIX (or Linux or Mac OS X) command (also called a tool) by typing its name and then the ENTER or

RETURN key.

The following example uses the **date** command or tool.

```
$ date
Wed Nov 10 18:08:33 PST 2010
$
```

The format for the output is: day of the week, month, day of the month, 24 hour time, time zone, year.

failed command

BASH will let you know when you have typed something it doesn't understand.

Type "asdf" and then the ENTER or RETURN key. You should see a message similar to the following:

```
$ asdf
-bash: asdf: command not found
$
```

no news is good news

BASH follows the rule that no news is good news. If a command runs successfully, it won't generate any output message.

As an example, when you use the `cp` command to coy a file, there will be no special notice on success. You will just be returned to the command prompt (`$`), ready for the next command.

login/logout

chapter 7

summary

This chapter looks at `login` and `logout`, a pair of Unix (and Linux) commands.

On a personal computer or workstation you probably don't need to `login`. You can just start your terminal emulator (see previous chapter).

On a multi-user UNIX or Linux system, such as a web server, you will need to `login`. This chapter furthers the discussion `login` in more detail than the previous chapter's introduction, including special cases and solutions to common problems.

In particular, this chapter discusses choosing the system and terminal type, rare options that you might encounter.

This chapter also discusses the `logout` command and the importance of always logging out from a running system.

NOTE: On a single-user Linux or Mac OS X desktop/laptop computer, you can start the Terminal program and be automatically signed in (no need for account login or password). On a remote server, you will always need to login in with at least account name and password.

login

`login` is a UNIX command for logging into a UNIX system.

`login` is a builtin command in `csh`. There is also an external utility with the same name and functionality.

In many modern systems, the functionality of login is hidden from the user. The login command runs automatically upon connection. In some old computers you may need to type a special character or even type `login` to bring up the login program.

If you login remotely to a server you will be prompted for your user/account name and password. If you start up a terminal emulator and shell from a graphic user interface you will probably already be authenticated and immediately go to the shell (although some of the startup activity of login will still be done for you).

The activities that login performs at startup of a shell will be covered in a later chapter, because you need more familiarity with the shell before you can understand that material.

Starting a shell is covered in the previous chapter about shell basics.

select system or host

If you are signing into a system that combines multiple computers, you will first need to indicate which computer you are logging into.

You may already be logged in if you are using a terminal emulator on a personal computer or workstation. If so, you don't need this step.

You will see a prompt indicating a choice of computer, such as (only one of these will appear):

```
host:
pad:
request:
system:
```

You will see only one of these (or similar) choices.

If you see `login:` instead, then you don't have to worry about choosing the system.

Enter the identifying name of the computer system you were assigned to use, followed by the ENTER or RETURN key. Your system administrator can provide you with this information.

On some networked systems you can abbreviate with the first two or three characters of the computer name (as long as it is enough to uniquely identify which computer is intended).

If you are successful, you will be presented with the `login:` prompt.

account name

You may already be logged in if you are using a terminal emulator on a personal computer or workstation. If so, you don't need this step.

Ask your system administrator for your account name. If this is your own personal system, then use the account name set up during installation. Many modern install disks include a utility for creating a new account.

At the `login:` prompt, enter your account name, followed by the ENTER or RETURN key.

Type your account name in lower case letters. UNIX assumes that an account name in ALL CAPITAL LETTERS indicates an old input/out device that doesn't support lower case letters (such as an old style teletype, or TTY). Also, if you use upper case letters for the login, then you can not use lower case letters in the password.

If you enter a valid account name, you will be asked for your password. On many computers, you will be asked for a password even if you enter an incorrect account name. This is a security measure to help prevent guessing account names.

```
login: accountname
Password:
```

password

You may already be logged in if you are using a terminal emulator on a personal computer or workstation. If so, you don't need this step.

Ask your system administrator for your password. If this is your own personal system, then use the password set up during installation. Many modern install disks include a utility for changing the password for an existing account or for creating a new account with a new password.

At the `Password:` prompt, enter your password, followed by the ENTER or RETURN key.

```
Password: password
$
```

In some cases, a new account doesn't have a password. Also, it is common for Mac OS X to have no password for the user account.

If your account doesn't have a password, just press the ENTER or RETURN key.

Change your password to a secure password at your earliest opportunity. If you don't know how to do this yet, the upcoming chapter on passwd (yes, that is the correct spelling) will tell you how. That same chapter gives advice on how to select a secure password.

Failure will be indicated by an error message. The exact error message varies by system, but will probably be something similar to:

```
Login incorrect
```

The most likely reasons are:

1. Incorrect account name.
2. Incorrect password.
3. You typed the account name in upper case letters (check the CAPS LOCK key).

You should have a new login prompt and can easily start over.

If you logged in in upper case, the shell will display in upper case letters and expect you to type only in upper case letters.

If you accidentally typed the account name in upper case letters, but haven't yet typed in the password, many UNIX systems will display aa backslash character in front of the password prompt.

If you realize you made a mistake on the account name and want to start over, hold down the CONTROL key (often marked CTRL) and the D key at the same time. This will stop the login program and start over again from the beginning.

Note that at one time there was a security hole because of the Control-D. A hacker would start the login, type Control-D, and then try to slip in a command to the shell before a new login program could be started. This security hole no longer exists on shell login, but this same type of timing attack is still used for other security holes.

terminal type

On some older UNIX systems you may be asked to provide the terminal type you are using. Some modern terminal emulator programs offer the choice of emulating various different terminals from the early UNIX era. These different terminal types had different specialized capabilities.

Because the early traditional UNIX was created primarily on Digital Equipment Corporation (DEC) computers (first the PDP, followed by the VAX), the default terminal device was the DEC VT100. Most modern terminal emulator programs act like the VT100.

On older UNIX systems you may see the terminal type prompt:

```
Term:
```

In some cases, the prompt may give the default terminal setting in parenthesis.

```
Term = (VT100)
```

If the default terminal setting is correct, simply press the RETURN or ENTER key.

Otherwise, type in the correct terminal type, followed by the ENTER or RETURN key.

If you are uncertain, try the `vt100` or `UNKNOWN`, followed by ENTER or RETURN.

Some common terminal types:

- h19
- tvi925
- vt100
- wyse50

After you are logged in, you can set the terminal type (capitalization matters).

Shell	Command
csh or tcsh	`setenv TERM vt100`
sh	`TERM=vt100; export TERM`
ksh, bash, or zsh	`export TERM=vt100`
VMS	`set term/device=vt100`

If you do not know your shell type, use the following command (followed by ENTER or RETURN):

```
$ echo $SHELL
```

You will need to enter the appropriate command each time you login.

Alternatively, you can add the appropriate command line to the initialization file in your home directory.

Shell	Login file
csh	`.cshrc` or `.login`
tcsh	`.cshrc`
ksh	`.profile` `.kshrc`
zsh	`.zshrc`
bash	`.bash_profile`
remote shell	`.rhosts`

VMS users need to modify the login.com file, adding the line `$set term/device=vt100 .`

logout

When you finish using the shell, type the `exit` or `logout` command. This will either display a new login prompt or close the terminal emulator window.

When you finish using the shell, type the `exit` or `logout` command. This will either display a new login prompt or close the terminal emulator window. if you are logged in remotely to a server, this will break your connection. If you are logged in physically on a large UNIX system, this will prevent someone else from abusing your account.

Some of the possible variations of this command include: `bye`, `exit`, `lo`, `logout`, and `quit`.

Typing CtrlL-D (holding down the CONTROL key and the D key at the same time) will also log you out of most shells. The Ctrl-D tells the shell it has reached the end-of-file (EOF). Because the shell is a filter and filters terminate when EOF is reached, the shell terminates and you log off the system or return to the parent process.

While it might not be a big deal to skip this step on a single user computer that you never leave unattended, forgetting to logout from a remote server is a serious security hole. If you leave your computer or terminal unattended, someone else can sit down and gain access to all of your files, including the ability to read, modify, or delete them. The imposter can send spam emails from your user ID. The imposter can even use your computer or account to attempt to hack or damage or break into any system in the world from your user ID. If this occurs from a school or business account, you will be responsible for any damage done.

Most modern systems accept either `logout` or `exit`.

On Mac OS X, typing `logout` produces the following result:

```
$ logout

[Process completed]
```

On Mac OS X, typing `exit` produces the following result (notice that the shell automatically runs `logout` for you):

```
$ exit
logout

[Process completed]
```

A successful `logout` will stop all running foreground or background processes. There is a way to keep processes running, which is useful for things like starting up your Apache web server and MySQL data base, but the method will be discussed later.

```
$ logout
There are suspended jobs.
```

If you see the message “There are stopped jobs.” it means that you have one or more suspended jobs. The shell is letting you know in case you forgot about them. If you type `logout` again without running the `jobs` in between, the system will go ahead and log you out and terminate all suspended jobs.

`logout` is a builtin command in `csh`.

exit

`exit` is a built-in shell command (for the shells that support it), while `logout` is a program. In the csh (C Shell), `logout` is an internal built-in command. sh (the original Bourne shell) and ksh (the Korn Shell) do not officially have the `logout` command, but many modern versions support it.

`exit` can be used by itself as a substitute for `logout` (as mentioned above).

`exit` can also be used to send a return value (often an error code) to the parent process. Simply add an integer following `exit`. This is particularly useful in scripting.

```
$ exit n
```

`exit` will be covered later in its own chapter.

passwd

chapter 8

summary

This chapter looks at `passwd`, a Unix (and Linux) command.

`passwd` is used to change your password.

This chapter also includes the list of the 100 worst (most commonly used) passwords.

As mentioned in the previous chapter, you should change your password from the original default or assigned password to a secure password that nobody else knows. And please don't leave the new password on a sticky note attached to your computer.

syntax:

```
passwd [option…] [user]
```

shells:	ash	bash	bsh	csh	ksh	sh	tcsh	zsh
File Name:	`passwd`	**Directory:**	/usr/bin/			**Type:**	External	

setting your password

Type `passwd` followed by the ENTER or RETURN key.

```
$ passwd
```

You will be prompoted to give your current (old) password (to make sure it is really you) and then prompted to enter your new password twice. For security purposes, the password is typically replaced with asterisks or some other character so that nobody can read your password over your shoulder. To make sure that you have typed what you thought you typed you are asked to type the new password twice. The two copies must match before your new password replaces your old password.

local password

The password set by `passwd` is your *local* password. On a single user system, this is probably your only password.

On Mac OS X the use of the `passwd` may or may not be sufficient to change your password for the entire system. This depends on which version of Mac OS X you are using. It is best to change your password using the install disc. If you do not have a copy of the install disc, there are instructions on the internet on how to manually change the password.

On a large system, there may be multiple passwords spread across multiple computers. The `passwd` command will only change the password on the one server that you are currently logged into (normally through SSH). You may need to use `yppasswd` or a web interface to change your password for the entire system.

You can check for your account or username in `/etc/passwd`. If it's not listed there, then don't use the `passwd`. Check with your system administrator.

periodic changes

Whenever you first login into a new system, the first thing you should do is change your password. In particular, immediately change the initial root password for a new system. Leaving the initial default password is a huge security hole and hackers do try all of the standard default passwords to see if they can find an easy way into a computer. Even with a user account, it is common for initial passwords to be generated poorly and be easy for hackers to guess.

Additionally, you want to change your password on a regular basis. It only takes a few months to figure out a password through brute force attacks. Some systems require that you change your password on a regular basis. Once a month is a good time period. More often if you suspect that someone saw you typing or there is any other possibility that your password might have been compromised.

You can set up your account to remind you to change your password on a regular basis. If you are the system administrator, you can set up these reminders for everyone (and should do so). As system administrator you can even require that users change their passwords on a regular basis (or they become locked out and have to come to you to beg for re-entry). As system administrator you can also set up a system that requires (or even suggests) secure passwords.

100 most common passwords

Always avoid the common passwords. These are the most common passwords as of June 2012:

1. password
2. 123456
3. 12345678
4. 1234
5. qwerty
6. 12345
7. dragon
8. pussy
9. baseball
10. football
11. letmein
12. monkey
13. 696969
14. abc123
15. mustang
16. michael
17. shadow
18. master
19. jennifer
20. 111111
21. 2000
22. jordan
23. superman
24. harley
25. 1234567
26. trustno1
27. iloveyou
28. sunshine
29. ashley
30. bailey
31. passw0rd
32. 123123
33. 654321

34. qazwsx
35. Football
36. seinfeld
37. princess
38. peanut
39. ginger
40. tigger
41. fuckme
42. hunter
43. fuckyou
44. ranger
45. buster
46. thomas
47. robert
48. soccer
49. fuck
50. batman
51. test
52. pass
53. killer
54. hockey
55. babygirl
56. george
57. charlie
58. andrew
59. michelle
60. love
61. jessica
62. asshole
63. 6969
64. pepper
65. lovely
66. daniel
67. access
68. 123456789
69. joshua
70. maggie
71. starwars
72. silver
73. william
74. dallas
75. yankees
76. 666666
77. hello
78. amanda
79. orange
80. biteme
81. freedom
82. computer
83. sexy
84. nicole
85. thunder
86. heather
87. hammer
88. summer
89. corvette

90. taylor
91. fucker
92. austin
93. 1111
94. merlin
95. matthew
96. 121212
97. golfer
98. cheese
99. martin
100. chelsea

Approximately 4.7% of all users have a password of *password*. 8.5% have one of the top two passwords. 9.8% (nearly one tenth) have one of the three top passwords. 14% have one of the top 10 passwords. 40% have one of the top 100 passwords. 79% have one of the top 500 passwords. 91% have one of the top 1,000 passwords.

secure passwords

It is important to have secure passwords.

The more characters, the more secure. A minimum of six or eight characters is barely adequate.

A strong mixture of characters for a password includes at least one capital letter, at least one lower case letter, at least on digit, and at least one punctuation character. You should avoid repeating any character more than once in the same password. The special character (such as !@#$%^&*,;) should not be th efirst or last character in the password.

Avoid using any word that occurs in your own or any other natural langauge. Hackers use a **dictionary attack** that tries words from the dictionary. Also avoid spelling words backwards, using common misspellings, or using abbreviations. Avoid using dates that are important to you (someone can easily look up your birthday or anniversary on the world wide web). Avoid using names of family, friends, or even pets.

secure technique

A technique that generates decent passwords is to use a key phrase and then use the first letter of each word in the keyword. Sprinkle in digits and special characters (punctuation) and make some of the letters upper case and some lower case.

Never use the same password for more than one purpose. People have the tendency to reuse the same password over and over. If a hacker gets your password from one system, the hacker will see if it also works on your bank account and other systems.

superuser

The super user (root) can use the `passwd` command to reset any other user's password. There is no prompt for the current (old) password.

```
$ passwd username
```

The super user (root) can also remove a password for a specific user with the `-d` option. The disable option then allows the specified user to login without a password. This applies to Linux and Solaris only.

```
$ passwd -d username
```

command structure

chapter 9

summary

This chapter looks at simple Unix/Linux commands to get you started with using the shell.

You will learn the basic format or structure of a shell command.

You can run a UNIX (or Linux or Mac OS X) command (also called a tool) by typing its name and then the ENTER or RETURN key.

All commands (and file names) are **case senitive**. `ls` is a valid command, while `LS` is unrecognized.

commands and utilities

A quick note on the difference between commands and utilities.

A **utility** is a program. `who`, `date`, and `ls` are examples of utility programs.

A **command** is all of the text typed at the command line, the utility name and all of the flags, switches, file names, and other text.

This subtle distinction between utilities and commands is generally ignored and the two terms are often used interchangeably. As I type this, I am pretty sure that I have ignored this distinction elsewhere in this book and dread double-checking every use. You can probably use the terms interchangeably except in cases like tests, papers, or books.

The kernel is loaded from hard drive (or other storage) when the computer is started and remains running until the computer is turned-off or crashes.

Utilities are also stored on hard drives, but are only loaded into main memory when used.

single command

The most simple form of a UNIX command is just a single command by itself. Many Unix/Linux commands will do useful work with just the command by itself. The next examples show two commands (`who` and `date`) by themselves.

The output generated by a single command by itself is called the default behavior of that command.

who

The `who` command will tell you all of the users who are currently logged into a computer. This is not particularly informative on a personal computer where you are the only person using the computer, but it can be useful on a server or a large computing system.

Type `who` followed by the ENTER or RETURN key.

```
$ who
admin   console Aug 24 18:47
admin   ttys000 Aug 24 20:09
$
```

The format is the login name of the user, followed by the user's terminal port, followed by the month, day, and time of login.

failed command

The shell will let you know when you have typed something it doesn't understand.

Type "asdf" and then the ENTER or RETURN key. You should see a message similar to the following:

```
$ asdf
-bash: asdf: command not found
$
```

date

The following example uses the **date** command or tool.

```
$ date
Wed Nov 10 18:08:33 PST 2010
$
```

The format for the output is: day of the week, month, day of the month, 24 hour time, time zone, year.

options, switches, or flags

A command may optionally be followed by one or more options. The options are also commonly called flags or switches.

Options are usually a single character. Usually the option is preceded by a minus sign or hyphen character.

See the following example.

universal time

Adding the `-u` flag to the command `date` will cause it to report the time and date in UTC (Coordinated Universal) time (also known as Zulu Time and formerly known as Greenwich Mean Time or GMT). The seconds are slightly higher in the second example because of the passage of time.

```
$ date
Sat Aug 25 19:09:19 PDT 2012
$
$ date -u
Sun Aug 26 02:09:27 UTC 2012
$
```

option grouping

You can combine multiple options for a single command.

The following example shows the `ls` command by itself and with two options, first individually, and then combined.

```
$ ls
Desktop     Downloads   Movies   Pictures   scripts
Documents   Library     Music    Public

$ ls -F
Desktop/     Downloads/   Movies/   Pictures/   scripts/
Documents/   Library/     Music/    Public/
$ ls -a
.               .cups       Library    scripts
..              .vminfo     Movies
.DS_Store       Desktop     Music
.Trash          Documents   Pictures
-bash_history   Downloads   Public

$ ls -a -F
./              .cups/       Library/    scripts
../             .vminfo      Movies/
.DS_Store       Desktop/     Music/
.Trash/         Documents/   Pictures/
-bash_history   Downloads/   Public/
$
```

You can group option by listing them in any order (with a few rare exceptions) and either with their own minus sign or after a single minus sign. All of the following versions of the `ls` command work the same way:

```
$ ls -a -F
$ ls -F -a
$ ls -aF
$ ls -Fa
$
```

arguments

Commands may also optionally have arguments. The most common arguments are the names of files.

Technically, the options just mentioned are also arguments, but in common practice options are separated from other arguments in discussions. Also, technically, the operators mentiond below are also arguments, but again it is useful to separate operators from other arguments in discussions.

The following shows the difference between the default behavior (no arguments) of `who` and a version with arguments `who am i`.

The default behavior of `who` lists all users on the computer.

```
$ who
admin   console Aug 24 18:47
admin   ttys000 Aug 24 20:09
$
```

The use of the arguments with `who am i` causes the command to only lists the one user who typed the command.

The command is `who` and the arguments are `am i`.

```
$ who am i
admin    ttys000  Aug 25 17:30
```

```
$
```

one argument

In an upcoming chapter on the `cat` you will use the command `cat` with the file name `file01.txt` to confirm that you correctly entered your sample file.

just observe this example
do *not* type this into your shell

```
$ cat file01.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
abc
xyz
$
```

You may download this sample file from http://www.osdata.com/programming/shell/file01.txt.

two arguments

In the upcoming chapter on the `cat` you will use the command `cat` with two file names (`file01.txt` and `numberfile.txt`) to confirm that you correctly entered two of your sample files.

just observe this example
do *not* type this into your shell

```
$ cat file01.txt numberfile.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
abc
xyz
1
2
3
4
5
6
7
8
9
10
11
12
```

```
13
14
15
16
17
18
19
20
$
```

You may download the second sample file from

http://www.osdata.com/programming/shell/numberfile.txt.

options and arguments

You can combine options and arguments on the same command line.

The following example uses the command `cat` with the `-b` option and the file name `file01.txt`.

just observe this example
do *not* type this into your shell

```
$ cat -b file01.txt
    1  This is a line of text in my first file.
    2  This is another line.

    3  To be, or not to be: that is the question:

    4  1234567890
    5  ABC
    6  XYZ
    7  abc
    8  xyz
$
```

operators and special characters

Command lines may include optional operators or other special characters.

In an upcoming chapter on the `cat` you will use the command `cat` with the operator `>` and the file name `file01.txt` to enter your sample file.

just observe this example
do *not* type this into your shell

```
$ cat > file01
```

If you accidentally typed in the above command, hold down the CONTROL key and the D key at the same time to return to your shell.

quick tour of shell commands

chapter 10

summary

This chapter gives you a quick tour of shell commands.

You should do each of the exercises and observe what happens.

Do not worry about trying to learn how each of the commands works.

The goal here is to give you a basic idea of how a shell works and the kinds of things that the most common shell commands and operations do.

This quick tour will give you a background to understand the upcoming chapters and their lessons. If you have the need to use anything you observe in this quick tour, you can always jump ahead to the appropriate chapter and learn the details.

failure

Type `asdf` (a non-existent command) into the terminal. Remember to press the ENTER or RETURN key after this command.

```
$ asdf
-bash: asdf: command not found
$
```

This shows you what it look like when the shell reports mistakes. Note that you will not always see any reports of a particular error. Commands that have no output will just return silently to the prompt.

echo text

Use the `echo` command to send text to the terminal. Remember to press the ENTER or RETURN key after each command. Note that this will be particularly useful in your future Unix/Linux scripts to report information back from your scripts.

```
$ echo hello world
hello world
$
```

echo text with variable information

Use the `echo` command with an environment variable. it should use your account name.

```
$ echo hello $USER
hello admin
$
```

list directory contents

Use the `ls` command to list the contents of the current directory.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures
Library          Public
$
```

create a text file

Use the `cat` command to create a small text file.

Type the command line, followed by RETURN or ENTER, then type each of the six suggested lines, each followed by the RETURN KEY. After having enetered each line, make sure you are at the beginning of a new (blank or empty) line and type Ctrl-D (hold downt he CONTROL key and the D key at the same time).

```
$ cat > names
James
Mary
John
Patricia
Robert
Linda
CONTROL-D
$
```

The choice of names is based on the most popular names in the United States in the year before this chapter was written. Bribe me if you want your name put into the example.

check the file was created

Use `ls` to make sure the file was created properly. It should be added to your directory listing.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library          Public
$
```

display file contents

Use the `cat` command to show the contents of your new file.

```
$ cat names
James
Mary
John
Patricia
```

```
Robert
Linda
$
```

count the number of words

Use the `wc` command to count the number of words in your new file.

```
$ wc names
       6       6      38 names
$
```

The format is the number of lines (`6`), followed by the number of words (`6`), followed by the number of characters (`38`), followed by the name of the file (`names`).

copy a file

Use the `cp` command to make a copy of a file.

```
$ cp names saved_names
$
```

Notice that there is no confirmation of the file copy being made.

This silent behavior is typical of any UNIX shell. The shell will typically report errors, but remain silent on success. While disconcerting to those new to UNIX or Linux, you become accustomed to it. The original purpose was to save paper. When UNIX was first created, the terminals were mostly teletype machines and all output was printed to a roll of paper. It made sense to conserve on paper use to keep costs down.

You can use the `ls` command to confirm that the copy really was made. You won't be using up any paper.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library          Public        saved_names
$
```

rename a file

Use the `mv` command to rename a file.

```
$ mv saved_names old_names
$
```

Notice that the shell is once again silent with success. You can use the `ls` command to confirm that the rename really was made.

```
$ ls
```

```
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library          Public        old_names
$
```

delete a file

Use the `rm` (remove) command to delete a file.

```
$ rm old_names
$
```

You can use the `ls` command to confirm that the file was really deleted.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library
$
```

current directory

Use the `pwd` command to determine the current directory. Your version should give the name of your home directory, which normally matches the name of your user account. The example include the directory as part of the prompt (your system may not include this) and the tilde (~) character indicates the home directory.

```
$ pwd
/Users/admin
admins-power-mac-g5:~ admin$
```

make a directory

Use the `mkdir` command to make a new directory (folder).

```
$ mkdir testdir
admins-power-mac-g5:~ admin$
```

change directory

Use the `cd` command to change to your new directory (folder). if your prompt includes the current directory, then you will see your prompt change to show the new location.

```
$ cd testdir
admins-power-mac-g5:testdir admin$
```

confirm directory change

Use the `pwd` command to confirm that you are now in your new directory.

```
$ pwd
/Users/admin/testdir
admins-power-mac-g5:testdir admin$
```

return to home directory

Use the `cd` command without any additional arguments to return to your home directory from anywhere.

```
$ cd
admins-power-mac-g5:~ admin$
```

confirm directory change

Use the `pwd` command to confirm that you are now back in your home directory.

```
$ pwd
/Users/admin
admins-power-mac-g5:~ admin$
```

Now you are ready to dive in and start becoming proficient at using the UNIX or Linux shell.

man

chapter 11

summary

This chapter looks at `man`, a Unix (and Linux) command.

`man` is the UNIX equivalent of a help function.

`man` is the command used to view manual pages.

This chapter show you how to use the `man` command to get help. Because of huge variation in the various flavors of UNIX and Linux, the `man` installed on your computer or server is the best source of detailed information. If you have trouble with any of the lessons in this book or in general use of your UNIX or Linux system, always refer to the local `man` for help.

UNIX was the first operating system distributed with online documentation. This included `man` (manual pages for each command, library component, system call, header file, etc.) and doc (longer documents detailing major subsystems, such as the C programming language and `troff`.

example of man command with options

Most BASH commands accept options. These options follow a space character and are typed before you press RETURN or ENTER.

The format is `man` is followed by the name of the command or tool that you want to view. You will get the manual pages for the named command or tool.

The following example uses the manual page for the **date** command or tool.

```
    $ man date
DATE(1)                   BSD General Commands Manual                  DATE(1)

NAME
      date -- display or set date and time

SYNOPSIS
      date [-ju] [-r seconds] [-v [+|-]val[ymwdHMS]] ... [+output_fmt]
      date [-jnu] [[[mm]dd]HH]MM[[cc]yy][.ss]
      date [-jnu] -f input_fmt new_date [+output_fmt]
      date [-d dst] [-t minutes_west]
DESCRIPTION
      When invoked without arguments, the date utility displays the current
      date and time. Otherwise, depending on the options specified, date will
      set the date and time or print it in a user-defined way.

      The date utility displays the date and time read from the kernel clock.
      When used to set the date and time, both the kernel clock and the hard-
      ware clock are updated.

      Only the superuser may set the date, and if the system securelevel (see
      securelevel(8)) is greater than 1, the time may not be changed by more
      than 1 second.
:
```

Typing the RETURN or ENTER key will bring up the next line of the manual page.

Typing the SPACE-BAR key will bring up the next page of text.

Typing the UP-ARROW or DOWN-ARROW key will move up or down one line.

Typing **q** will end viewing the manual page and return to the BASH prompt (the entire manual page will disappear from view).

The **man** command or tool will be a very useful reference.

A `man` page is typically organized:

`NAME:` Command name and brief description.

`SYNOPSIS:` The syntax for using the command, along with the flags (options) the command takes.

`DESCRIPTION:` Details about the command.

`ENVIRONMENT:` The environment variables used by the command.

`EXIT STATUS:` Information about how the command reports errors or success.

`FILES:` File related to the command.

`SEE ALSO:` related commands.

`STANDARDS:` The international standards, if any.

Some systems might have an `AUTHOR` or other sections.

Type `man man` for information about the local version of `man`.

man sections

The first seven distributions of UNIX (called V1 UNIX through V7 UNIX and collectively called traditional UNIX) included a two volume printed manual, which was divided into eight sections.

Some manual pages are located in sections. There are generally eight sections:

1. General commands
2. System [kernel] calls
3. C library functions
4. Special files (such as devices) and drivers
5. File formats, conventions, and miscellaneous information
6. games and screensavers
7. Macro packages
8. System administration commands and daemons

Note that on some systems, the system administration commands are in section 1m. This section is also sometimes called the Maintenance commands.

Note that on some systems, section 7 is Miscellaneous.

You can look at a particular section by adding the section number to the command line.

```
$ man SECTION-NUMBER commandname
```

You can use the `whatis` command to find the sections and then look at the particular section of your choice.

```
$ whatis crontab
$ whatis crontab
crontab(1)               - maintain crontab files for individual users (V3)
crontab(5)               - tables for driving cron

$ man 5 crontab
```

-f option

The `-f` option is used to show all `man` page titles for entries that begin with a particular word. `man -f file` will show all the manual pages that have the word “file” in the title.

```
$ man file
```

-k option

The `-k` option runs a search on all manual pages for any manual page with a particular word anywhere on the page. The `man -k file` will show all the manual pages that have the word “file” anywhere in the manual page. The `-k` option can take a while to run.

```
$ man file
```

cat

chapter 12

summary

This chapter looks at `cat`, a Unix (and Linux) command.

The `cat` (as in con**cat**enate) utility can be used to concatenate several files into a single file.

`cat` is most often used to obtain the contents of a file for input into a Linux or UNIX shell script.

`cat` is used concatenate file. The name is an abbreviation of catenate, a synonym of concatenate.

`cat` was part of the original 1969 version of UNICS (the original name for UNIX).

create names file

If you did not create the names file suggested in the quick tour chapter, please do so now, because you will use this file in future exercises. If you already created the names file, then you can safely skip this step.

Type the command line, followed by RETURN or ENTER, then type each of the six suggested lines, each followed by the RETURN KEY. After having enetered each line, make sure you are at the beginning of a new (blank or empty) line and type Ctrl-D (hold downt he CONTROL key and the D key at the same time).

```
$ cat > names
James
Mary
John
Patricia
Robert
Linda
CONTROL-D
$
```

The choice of names is based on the most popular names in the United States in the year before this chapter was written. Bribe me if you want your name put into the example.

check the file was created

Use `ls` to make sure the file was created properly. It should be added to your directory listing.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library          Public
$
```

display file contents

Use the `cat` command to show the contents of your new file.

```
$ cat names
James
Mary
John
Patricia
Robert
Linda
$
```

creating files

One simple use of `cat` is to create simple files.

Type `cat > file01.txt`, followed by ENTER or RETURN.

```
$ cat > file01.txt
```

The cursor will be at the beginning of a line that has no prompt. You are no longer in the shell, but instead in the `cat` utility. There is no `cat` prompt.

Type the following lines (by convention, all of the input should be in bold, but to make it easier on the eye, it is in italics here):

```
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
abc
xyz
```

Once you have typed in these lines, press RETURN to make sure you are at the beginning of a new line.

Press Ctrl-D (hold down the CONTROL key and the D key at the same time). This indicates end-of-file (EOF), which informs `cat` that you have no more input. This will return you to the shell.

Note that unlike some operating systems, the ".txt" file extension is *not* required by UNIX or Linux. It is optional, but the file extensions are often used to make it easier for a human to distinguish file types.

Now repeat the process with the following file (by convention, all of the input should be in bold, but to make it easier on the eye, it is in italics here):

There is a lot of typing in this example, but it is important for the upcoming exercises. Type the following lines (by convention, all of the input should be in bold, but to make it easier on the eye, it is in italics here):

Because of the length of this file, you may want to download a copy from the internet. There is a copy at http://www.osdata.com/programming/shell/file02.txt — if you know how to download a copy and place it in your home

directory, save yourself some typing time.

```
$ cat > file02.txt

This is a line of text in my second file.
This is another line.

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to, 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
To sleep: perchance to dream: ay, there's the rub;
For in that sleep of death what dreams may come
When we have shuffled off this mortal coil,
Must give us pause: there's the respect
That makes calamity of so long life;
For who would bear the whips and scorns of time,
The oppressor's wrong, the proud man's contumely,
The pangs of despised love, the law's delay,
The insolence of office and the spurns
That patient merit of the unworthy takes,
When he himself might his quietus make
With a bare bodkin? who would fardels bear,
To grunt and sweat under a weary life,
But that the dread of something after death,
The undiscover'd country from whose bourn
No traveller returns, puzzles the will
And makes us rather bear those ills we have
Than fly to others that we know not of?
Thus conscience does make cowards of us all;
And thus the native hue of resolution
Is sicklied o'er with the pale cast of thought,
And enterprises of great pith and moment
With this regard their currents turn awry,
And lose the name of action. - Soft you now!
The fair Ophelia! Nymph, in thy orisons
Be all my sins remember'd.

1234567890
ABC
XYZ
abc
xyz
```

Now repeat the process with the following file (by convention, all of the input should be in bold, but to make it easier on the eye, it is in italics here — note the cheat method below):

```
$ cat > numberfile.txt
1
2
3
4
5
```

```
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Once again, use the RETURN key to enter a new line and then use Ctrl-D to exit `cat`.

If you want to cheat on making this file, you can use the following command on Mac OS X or BSD systems:

```
$ jot 20 1 > numberfile.txt
$
```

You will use these files in many of the exercises in this book.

PC-DOS equivalent

`cat` is the UNIX equivalent of the MS-DOS or PC-DOS command `TYPE`. You can add the PC-DOS equivalent to your shell session with the `alias` command. To make the change permanent, add the following line to the `.bashrc` file in your home directory. Note that if you add this PC-DOS/MS-DOS equivalent, only add the all upper case version, because the lower case `type` is an important UNIX command that you will also need.

```
$ alias TYPE="cat"
```

view a file with cat

Type `cat file 01`, followed by the RETURN or ENTER key. You should see the contents of your first file. The beginning of the file may scroll off the top of your screen (this is normal).

```
$ cat file01.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
abc
xyz
$
```

combine files with cat

Type `cat file 02 numberfile.txt`, followed by the RETURN or ENTER key. This will show you the contents of both files, one immediately after the other. This is the use for which `cat` was named.

```
$ cat file02.txt numberfile.txt
This is a line of text in my second file.
This is another line.

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to, 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
  …many text lines…
The fair Ophelia! Nymph, in thy orisons
Be all my sins remember'd.

1234567890
ABC
XYZ
abc
xyz
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
$
```

This should list your new file that combines combines two of the files you created. The beginning of the listing will scroll off the top of your screen. Don't worry about it. We'll see how to view long files in the chapter about `less`.

display with line numbers

The following example uses the command `cat` with the `-n` option to add line numbers to the listing of the file.

Type `cat -n file01.txt`.

```
$ cat -n file01.txt
     1  This is a line of text in my first file.
     2  This is another line.

     3  To be, or not to be: that is the question:
     4
     5  1234567890
     6  ABC
     7  XYZ
     8  abc
     9  xyz
$
```

Notice that the blank line is numbered. You can use the `-b` option to get line numbers, skipping all blank lines.

Type `cat -b file01.txt`.

```
$ cat -b file01.txt
     1  This is a line of text in my first file.
     2  This is another line.

     3  To be, or not to be: that is the question:

     4  1234567890
     5  ABC
     6  XYZ
     7  abc
     8  xyz
$
```

empty a file

You can use `cat` to empty a file by sending `/dev/null` to the file to be emptied. The null device is bit nothingness.

Start by making a copy of one of the files you just created so that you don't wipe out your hard work.

```
$ cp file01.txt emptyfile
$
```

Use `cat` to verify that the new file really does exist and has a complete copy of the original file.

```
$ cat emptyfile.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
abc
xyz
$
```

Now type `cat /dev/null > emptyfile.txt`. This will leave the file in existence, but empty it of all characters.

```
$ cat /dev/null > emptyfile.txt
$
```

Confirm this by using both `cat` and `wc` (word count). The `cat` will simply return to the prompt because there is nothing to print out. The `wc` will show zero lines, zero words, and zero characters.

```
$ cat file01.txt > emptyfile.txt
$ wc names
       0       0       0 emptyfile.txt
$
```

replacing files

The previous example brings up the point that if you use `cat` to output to an existing file, the previous file will be completely replaced by the new one. There is no warning at all.

appending files

You can append to the end of an existing file by using two greater than signs (no spaces in between) >>.

Start by making a copy of an existing file and then confirming the copy is correct.

```
$ cp file01.txt file03.txt
$ cat file03.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
abc
xyz
$
```

Now try the append.

```
$ cat numberfile.txt >>.file03.txt
$
```

And confirm that your append worked.

```
$ cat file03.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:
```

```
1234567890
ABC
XYZ
abc
xyz
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
$
```

hidden characters and spaces

There are several switches that can be used to view files with non-printable characters, trailing spaces, and other hidden characters that can drive you crazy.

On Linux (*not* on BSD or SYS V or Mac OS X), you can use the `-E` command to place a `$` dollar sign at the end of each line. This will reveal trailing spces that would not otherwise be visible.

```
$ cat -E names
James$
Mary$
John$
Patricia$
Robert$
Linda$
$
```

The `-v` switch will display non-printing characters. Control characters are preceded by the `^` caret character, so control-X would be listed as `^X`. The delete character (octal 0177) is displayed as `^?`. The non-ASCII characters (high bit set) are displayed with `M-` (for meta) followed by the character for the low seven bits.

On Linux only, you can combine the `-vE` with the `-e` character (show nonprinting characters and the end of line marker).

On Linux only, you can use the `-T` switch to show tab characters as `^I`.

You can use the `-t` switch to show tab characters as `^I` and non-printing characters in the manner described for the `-v` switch. This is the equivalent of the Linux only `-vT`.

On Linux only, you can use the `-A` switch as a shorter version of the `^vET` combination.

squeeze lines

You can use the `-s` switch to suppress multiple consecutive empty lines. Only one empty line will be displayed.

cat in scripts

While `cat` is a great tool for qork from the command line, if you find yourself using `cat` in a script, you should probably rethink your algorithm. It is rare that you are on the right path when you find yourself using `cat` inside a shell script. The use of `cat` in a shell script is a common sign of amateur hacking and the script probably has serious beginner mistakes and problems as well.

command separator

chapter 13

This chapter looks at the command separator in UNIX or Linux.

The semicolon (;) is used as a command separator.

You can run more than one command on a single line by using the command separator, placing the semicolon between each command.

```
$ date; who am i
Mon Aug 27 19:15:41 PDT 2012
admin    ttys000  Aug 27 17:50
$
```

It does not matter if you have a space before the semicolon or not. Use whichever method is more readable and natural for you.

```
$ date ; who am i
Mon Aug 27 19:15:41 PDT 2012
admin    ttys000  Aug 27 17:50
$
```

Each command is processed in order, as if you had typed each individually, with the exception that the line with the prompt is only displayed once, at the end of the series of commands.

If you forget the semicolon, you will get an error message. The following two examples are from different systems.

```
$ date who am i
date: illegal time format
$
```

```
$ date who am i
date: bad conversion
$
```

The semicolon is a common terminator or separator in many common programming languages, including Ada, C, Java, Pascal, Perl, PHP, and PL/I.

Many programmers automatically place the semicolon at the end of any shell command. The shell is fine with this.

```
$ date;
Mon Aug 27 19:15:41 PDT 2012
$
```


data

less, more, pg

chapter 14

summary

This chapter looks at `less`, `more`, and `pg`, a related family of Unix (and Linux) commands.

`less`, `more`, and `pg` are utilities for reading a very large text file in small sections at a time.

`pg` is the name of the historical utility on BSD UNIX systems.

`more` is the name of the historical utility on System V UNIX systems.

`pg` and `more` are very similar and have almost the same functionality and are used in almost the exact same manner, except for whether you type `pg` or `more`.

`less` is a more modern version that has the capaibilites of `more` along with additional new capabilities.

Mac OS X has all three versions installed. It is common to find both `more` and `less` on Linux systems. The older versions are typically included so that old scripts written before the invention of `less` will still work.

The name `less` is a pun, from the expression “less is more.”

Try using `less` with your test file `file01.txt`.

You are going to try `less` first.

```
$ less file01.txt
```

If that doesn’t work, you are going to try `more` next.

```
$ more file01.txt
```

If that doesn’t work, you are going to try `pg` last.

```
$ pg file01.txt
```

One of the three will work. You can try the others to see if they are available on your system, but for the exercise, use the first one that works.

```
$ less file01.txt
This is a line of text in my first file.
This is another line.

To be, or not to be: that is the question:

1234567890
ABC
XYZ
```

```
abc
xyz
$
```

Now we are going to see the real power of less with a long file.

```
$ less file02.txt
This is a line of text in my second file.
This is another line.

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to, 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
To sleep: perchance to dream: ay, there's the rub;
For in that sleep of death what dreams may come
When we have shuffled off this mortal coil,
Must give us pause: there's the respect
That makes calamity of so long life;
For who would bear the whips and scorns of time,
The oppressor's wrong, the proud man's contumely,
The pangs of despised love, the law's delay,
The insolence of office and the spurns
That patient merit of the unworthy takes,
When he himself might his quietus make
File02.txt
```

Notice that you are not yet back to the shell prompt. You are still in the less command. You are only one page into the file. less allows you to view long files one page at a time, instead of scrolling them off the top of your screen.

Type the SPACE key to see another page of text.

```
That patient merit of the unworthy takes,
When he himself might his quietus make
With a bare bodkin? who would fardels bear,
To grunt and sweat under a weary life,
But that the dread of something after death,
The undiscover'd country from whose bourn
No traveller returns, puzzles the will
And makes us rather bear those ills we have
Than fly to others that we know not of?
Thus conscience does make cowards of us all;
And thus the native hue of resolution
Is sicklied o'er with the pale cast of thought,
And enterprises of great pith and moment
With this regard their currents turn awry,
And lose the name of action. - Soft you now!
The fair Ophelia! Nymph, in thy orisons
Be all my sins remember'd.

1234567890
ABC
XYZ
abc
xyz
(END)
```

Notice that you are not yet back to the shell prompt. Even though the entire file contents have been listed, you are still in the `less` command.

Typing lower case `q` will quit the `less` (or `more` or `pg`) program.

`less` (and the other two utility commands) displays one page at a time. This is in contrast with `cat`, which displays the entire file at once.

file system basics

chapter 15

summary

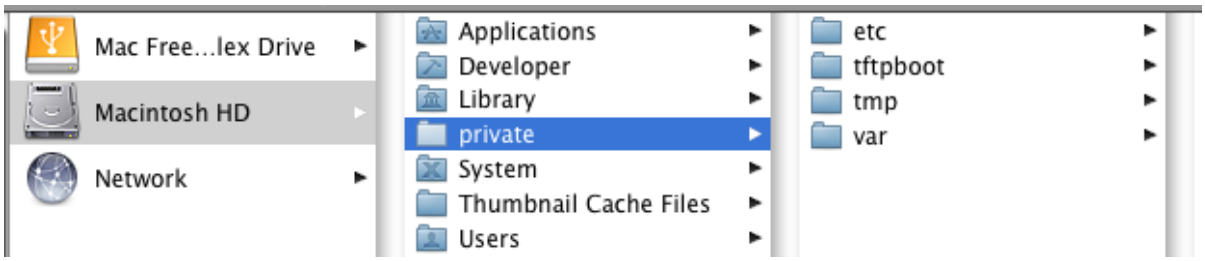
This chapter looks at the basics of the file system on UNIX or Linux.

By 1971, many fundamentally important aspects of UNIX were already in place, including file ownership and file access permissions.

graphics example

In a graphic user interface, you can see the file system directory structure in some kind of picture version. This varies by system.

The following example is from Mac OS X. If you use Mac OS X, the UNIX directories are kept hidden in Finder. You can see them by using Finder's **Go** menu, Go To Folder…, and type "private".



The following example is from Linux.

Do **not** mess with your system files until you know what you are doing. You can cause your computer to stop working.

Some of the typical Unix/Linux directories are etc, tmp, var, bin, sbin, dev, boot, or home.

some basics

Both UNIX and Linux organize the entire file system into a single collection (called the directory tree). This is in sharp contrast to Windows dividing the file system into multiple parts, each based on a particular drive or device.

File names are case sensitive. The file names "File" and "file" are different files. This is in sharp contrast with Windows and Macintosh (where they are alternate names for the same file). Note that this is a Mac OS X variation from the standard UNIX practice.

A file name that starts with a period is a hidden file. In Mac OS X, you can toggle between showing and hiding hidden files in Finder by typing Command-Shift-Period.

Neither UNIX nor Linux enforce or recognize file extensions (such as ".txt"). In some operating systems (such as Windows and traditional Macintosh), the file extensions are recognized by the operating system and are used for such

purposes as deciding which program is used to open the file. You can add file extensions UNIX or Linux file names (and this is commonly done to help humans organize their files), but the file extension is ignored by the operating system and has no special meaning.

Most modern Linux and UNIX operating systems support long file names with a wide variety of characters, including the space character. Shells are set up for old school UNIX, which had a limited number of characters in file names and a very limited choice of characters in file names. it is best to use these old school limits for files you manipulate through the UNIX or Linux shell. Mac OS X allows you to put file names inside quotation marks (to allow such things as spaces in file names). You can also access files through their inode numbers (which will let the shell work on any file name). Still, it is wise to use only letters, numbers, period, hyphen, and underscore in UNIX or Linux file names 9and particularly to avoid the space character).

directory tree

A directory in UNIX or Linux is the same as a folder in Macintosh or Windows.

Directories may contain any combination of files and subordinate directories.

This structure is called a tree. It branches out from a root.

The **root** is the beginning of the filesystem.

A **branch** is a directory and all of its subdirectories.

When a directory is inside another directory, the one inside is called a **child** and the outer one is the **parent**.

A **leaf** is anything that has no children, such as a file.

Unix and Linux are organized into a single file system tree, under a master directory called root and designated by the forward leaning slash (/). This is in sharp contrast with Windows, where there is a seperate file system tree for each drive letter.

In UNIX, everything is is a file. Even a directory is just a special file that stores information about other files.

some important directories

The following is a brief description of some of the major directories. Not all systems will have all of these directories.

/ The root directory of the entire file system.

/bin A collection of binary files, better known as programs.

/boot or **/kernel** The files needed to boot or start your computer.

/dev The devices connected to your computer (remember, everything, even hardware devices, is treated as a file in UNIX or Linux).

/etc System configuration files, startup procedures, and shutdown procedures.

/home or **/users** The home directories for each user.

/lib or **/Library** Library files.

/net Other networked systems (again, treated as files).

/opt Third party software. In older systems this might be **/usr/local**

/private On Mac OS X, this is the location of the UNIX files. It is normally kept hidden from all users in Finder, but is visible in your terminal emulator and shell. Mac OS X has links to the major UNIX directories at the root level so that UNIX and Linux tools can find their files in the standard locations.

/proc Information about processes and devices.

/sbin System binaries (that is, system programs).

/tmp Space for temporary files. Required by all forms of UNIX.

/usr User binaries (programs), libraries, manuals, and docs.

/var Variable files.

Swap Virtual memory on a hard drive. Allows the memory manager to swap some data and program segments to hard drive to expand the amount of memory available beyond the limits of physical memory.

For a more advanced and complete listing, see major directories.

home and working directory

The home directory is the directory that your system places you in when you first log into your system (start the shell). The home directory is defined in the `/etc/passwd` file. This is your personal space on a UNIX or Linux machine.

The tilde character can be used to name your home directory. *`~user-name`* (where user-name is your actual user name) indicates your home directory.

The working directory is whatever directory you happen to be working in. The working directory is always the same as the home directory when you start a shell session, but you can change your working directory at any time (and typically do change it). Commands normally are applied to the current working directory.

The working directory may also be called the current directory, current working directory, or present working directory.

parent and child directory

Any directories underneath a particular directory are called child directories. Any directory can have zero, one, or many child directories.

The directory above any particular directory is called the parent directory. Any directory has only one parent (with the exception of root, which has no parent).

absolute paths

Every directory or file can be defined by a complete absolute path. The complete absolute path gives every level of directory, starting from the root. An absolute path name is sometimes called the full path name.

An example might be the `/usr/java/bin`. This would be a directory containing Java binaries. In this example, under the `/` root directory there is a directory called `usr` and inside that directory there is a directory called `java` and inside that directory there is a directory called `bin` and inside that directory there are programs for running Java.

Absolute paths normally start at the root of the file system.

relative paths

Relative paths describe where to find a file or directory from the current working directory.

A single period (or dot) character (`./`) indicates that you are describing the path to a directory or file from the current working directory.

If your current working directory is `www` and that directory is under `/usr`, then the file `index.html` inside the current working directory can be described by the full or absolute path of `/usr/www/index.html` or a relative path name of `./index.html`.

The single period is called “dot”.

Two periods (`../` called “dot dot”) take you up one directory to the parent directory.

You can use a series of two dots (`../../`) to go up more than one directory level.

dots and tildes and slashes

`/` Slash is used as a separator between directory names. When slash is the first character in a path name, it indicates the root directory.

~ Tilde is used to get to your home directory.

`./` Single dot is used to indicate the current working directory.

`../` Two dots are used to indicate the parent of the current working directory.

. A single dot in front of a file name (such as `.Trash`) is used to make a hidden file. Hidden files are only listed when specifically requested.

hidden files

A file name that starts with a period is a hidden file. In Mac OS X, you can toggle between showing and hiding hidden files in Finder by typing Command-Shift-Period. In MS-DOS, a hidden file can’t be written to. In UNIX and Linux a hidden file can be read and written.

Some important hidden files are the files used to configure the working environment, including `.cshrc`, `.login`, and `.profile`.

Some programs keep application-specific resources in hidden files or hidden directories.

moving around

This will all make a bit more sense in the next chapter as you learn how to actually move around your file system.

pwd

chapter 16

summary

This chapter looks at `pwd`, a Unix (and Linux) command.

`pwd` is used to Print the Working Directory. It provides the full path for the current working directory. and is normally run without any options.

`pwd` is a builtin command in `bash`. There is also an external utility with the same name and functionality.

working directory

The working directory is whatever directory you happen to be working in. The working directory is always the same as the home directory when you start a shell session, but you can change your working directory at any time (and typically do change it). Commands normally are applied to the current working directory.

The working directory may also be called the current directory, current working directory, or present working directory.

using pwd

Type pwd from the command line, followed by ENTER or RETURN, and you will see the current working directory.

```
$ pwd
/Users/admin
$
```

There's not much else to tell you about this command, but this is an important one because you will continually have the need to know where you are.

moving around

We will go into more detail about these commands later, but for now you need to know how to move around in your file system.

The `ls` command is used to list the contents of the current directory.

Go ahead and type `ls` followed by ENTER or RETURN and you will see a list of the files and directories in your current working directory (which should still also be your home directory).

The `cd` command is used to change directories.

Go ahead and type `cd` *`directory-name`* followed by ENTER or RETURN and you will move to the new directory. You can confirm this by then typing `pwd` followed by ENTER or RETURN. You will see that you have successfully changed working directories.

```
$ pwd
/Users/admin
$ ls
```

```
Desktop     Movies      Send Registration
Documents   Music       Sites
Downloads   Pictures
Library     Public
$ cd Desktop
$ pwd
/Users/admin/Desktop
$
```

As you move around the directory tree, it is common to forget exactly where you are, especially if you do some other tasks (such as file editing). `pwd` provides an easy way to determine where you are and insure that your next actions are happening in the correct directory.

Use `cd` command all by itself to return to your home directory.

```
$ cd
$ pwd
/Users/admin
$
```

advanced topic

The remainder of this chapter covers an advanced topic. I am placing this here so that you will be easily able to find them when you are ready for this information.

Eventually you will make use of symlinks and symlinked directories. There will be times when you need toknow the actual physical path to the current working directory, not just the logical symlinked path.

options

`pwd` is almost always used without options and never uses any arguments (such as file names). Anything other than legal options typed after `pwd` are ignored. Some shells ignore anything other than the legal options typed after `pwd` without reporting any errors, while most shells report errors for illegal options. All shells seem to ignore all arguments without reporting any errors.

The two main options for `pwd` are `-L` and `-P`.

The `-L` option displays the logical current working directory. This is the default if no options are listed.

```
$ cd /symlink
$ pwd -L
/symlink
$
```

The `-P` option displays the physical current working directory, with all symbolic links resolved.

```
$ pwd -P
/Users/admin
$
```

You can use the results of the `pwd -P` switch to `cd` to the physical directory.

```
$ cd -P /Users/admin
$
```

You can also use the `-P` switch directly on the `cd` command to change to the physical directory.

```
$ cd -P /symlink
$ pwd
/Users/admin
$
```

You can use the `help` command to get information about `pwd`.

```
$ help pwd
```

Some early Linux shells allow the --*help* command to get information about `pwd`.

```
$ pwd --help
```

Some early Linux shells allow the --*version* option gives the version number for your copy of `pwd`.

```
$ pwd --version
```

command history

chapter 17

summary

This chapter looks at command history in a Unix (and Linux) BASH shell.

Command history is used to recall recent commands and repeat them with modifications.

You will find that you are often repeating a few common commands over and over with variations in the options and arguments (such as file names). Having the ability to recall a previous example and make a few changes will greatly save on typing.

arrow keys

Pushing the Up Arrow key will bring back previously typed commands one at a time (from most recently typed to the first typed).

Pushing the Down Arrow key will move from the oldest to the newest commands one at a time.

The commands brought back by the Up Arrow and Down Arrow keys will appear after the current prompt.

You can press the ENTER or RETURN key to immediately run any selected command.

After using the Up Arrow or Down Arrow keys, your cursor is at the end of the line. You can use the DELETE key to remove old arguments and then start typing in new ones.

Note that holding down the CONTROL key and pressing the p key at the same time will take you to the previous command just like the Up Arrow key. This is written ^p.

Holding down the CONTROL key and pressing the n key at the same time will take you to the next command just like the Down Arrow key. This is written ^n.

Pressing the META key (or on some systems pressing the ESCAPE key), then releasing it, then typing the less than < key will take you to the beginning of the command history. Pressing the META key (or on many modern systems pressing the ESCAPE key or ESC key), then releasing it, then typing the greater than > key will also move to the end (most recent) command history. These are written M-< and M->.

editing a line

As mentioned above, when you use command history, the cursor is at the end of the recalled command.

Either the Left Arrow or ^b (CONTROL-B) will go one character backward (towards the beginning of line).

Either the Right Arrow or ^f (CONTROL-F) will go one character forward (towards the end of line).

Typing inside a line will place characters in front of the cursor. The character inside the cursor remains unchanged.

Using the DELETE key will delete the character in front of the cursor. The character inside the cursor remains unchanged.

^k (CONTROL-K) will delete everything from the cursor to the end of the line.

^d (CONTROL-D) will delete the character inside (under) the cursor.

M-b will move you one word back (towards beginning of line). In this case, a word is a collection of characters, usually separated by a space character.

M-f will move you one word forward (towards end of line).

^e (CONTROL-E) will move to the end of the line and ^a (CONTROL-A) will move to the beginning of the line.

Once you have edited your command, use the RETURN or ENTER key to run the new command. You can use the ENTER or RETURN key from anywhere in the command line. You do not have to be at the end of the command line.

built-in commands

chapter 18

summary

This chapter looks at built-in commands in a Unix (and Linux) BASH shell.

tools

Most of the commands in the UNIX or Linux shell are actually programs. If you look at the `usr/bin` directory, you will see the actual programs. Most of these programs were written in the C programming language.

There is a common core of tools/commands that will be available on almost every UNIX or Linux machine, but exactly how many and which commands/tools are available varies widely.

The good news is that if a command or tool is missing from your system, you can go out and get the source code and recompile it for your local computer.

built-in

Many of the shells have special built-in commands. These are *not* separate programs, but are part of the code for the shell itself.

One example would be the shell command `cd` that you just saw in the previous quick tour chapter.

There are some built-in commands are only available in selected shells and these can make your scripts shell-dependent.

Some examples of built-in commands include the `history` command in the C shell, and the `export` command in the Bourne shell. The `cd` command is built-in in both `bash` and `csh`.

`echo` is an example of a command that is built into both `bash` and `csh`, but also exists externally as a utility.

overriding built-in commands

You can override any built-in commands by giving the full path name to an external command or utility. If `bash` finds a slash character ( ) anywhere in a command, the shell will not run the built-in command, even if the last component of the specified command matches the name of a builtin command.

As an example, using the command `echo` will run the version of the command that is built into `bash`, while specifying `/bin/echo` or `./echo` will ignore the built-in comamnd and instead run the designated utility.

Overriding can be used to run alternative versions of commands or to extend the built-in command to add additional features.

determining builtin or external

You can use the `type` command to determine if a particular command is a built-in command or an external utility. If the command is an external utility, you will also be told the path to the external command.

```
$ type echo
echo is a shell builtin
```

```
$ type mkdir
mkdir is /bin/mkdir
$
```

You can use the `which` command to locate a program in your path.

```
$ which echo
/bin/echo
$
```

You can use the `whereis` command to locate a program in your path.

```
$ whereis echo
/bin/echo
$
```

In `csh` and `tcsh` you can use the `where` command to locate a program in your path.

```
% where echo
/bin/echo
%
```

problems

You can use the `type` command to determine if a particular command is a built-in command or an external utility. If the command is an external utility, you will also be told the path to the external command.

If something bad happens to your computer, if the shell is still loaded in memory and running, any of the built-in commands will still work correctly, even if the entire file system (including all hard drives) disappears or becomes unavailable for any reason.

built in command chart

The following chart shows the built-in commands and external utilities for `bash` and `csh` for Mac OS X. This wil be similar for Linux and other UNIXes.

External commands marked No** under the External column do exist externally, but are implemented using the built-in command.

Command	External	csh	bash
`!`	No	No	Yes
`%`	No	Yes	No
`.`	No	No	Yes
`:`	No	Yes	Yes
`{`	No	No	Yes
`}`	No	No	Yes
`alias`	No**	Yes	Yes
`alloc`	No	Yes	No
`bg`	No**	Yes	Yes

bind	No	No	Yes
bindkey	No	Yes	No
break	No	Yes	Yes
breaksw	No	Yes	No
builtin	No	No	Yes
builtins	No	Yes	No
case	No	Yes	Yes
cd	No**	Yes	Yes
chdir	No	Yes	Yes
command	No**	No	Yes
complete	No	Yes	No
continue	No	Yes	Yes
default	No	Yes	No
dirs	No	Yes	No
do	No	No	Yes
done	No	No	Yes
echo	Yes	Yes	Yes
echotc	No	Yes	No
elif	No	No	Yes
else	No	Yes	Yes
end	No	Yes	No
endif	No	Yes	No
endsw	No	Yes	No
esac	No	No	Yes
eval	No	Yes	Yes
exec	No	Yes	Yes
exit	No	Yes	Yes
export	No	No	Yes
false	Yes	No	Yes
fc	No**	No	Yes
fg	No**	Yes	Yes
filetest	No	Yes	No
fi	No	No	Yes
for	No	No	Yes
foreach	No	Yes	No
getopts	No**	No	Yes
glob	No	Yes	No
goto	No	Yes	No
hash	No	No	Yes
hashstat	No	Yes	No
history	No	Yes	No
hup	No	Yes	No
if	No	Yes	Yes
jobid	No	No	Yes
jobs	No**	Yes	Yes
kill	Yes	Yes	No

limit	No	Yes	No
local	No	No	Yes
log	No	Yes	No
login	Yes	Yes	No
logout	No	Yes	No
ls-F	No	Yes	No
nice	Yes	Yes	No
nohup	Yes	Yes	No
notify	No	Yes	No
onintr	No	Yes	No
popd	No	Yes	No
printenv	Yes	Yes	No
pushd	No	Yes	No
pwd	Yes	No	Yes
read	No**	No	Yes
readonly	No	No	Yes
rehash	No	Yes	No
repeat	No	Yes	No
return	No	No	Yes
sched	No	Yes	No
set	No	Yes	Yes
setenv	No	Yes	No
settc	No	Yes	No
setty	No	Yes	No
setvar	No	No	Yes
shift	No	Yes	Yes
source	No	Yes	No
stop	No	Yes	No
suspend	No	Yes	No
switch	No	Yes	No
telltc	No	Yes	No
test	Yes	No	Yes
then	No	No	Yes
time	Yes	Yes	No
times	No	No	Yes
trap	No	No	Yes
true	Yes	No	Yes
type	No	No	Yes
ulimit	No	No	Yes
umask	No**	Yes	Yes
unalias	No**	Yes	Yes
uncomplete	No	Yes	No
unhash	No	Yes	No
unlimit	No	Yes	No
unset	No	Yes	Yes
unsetenv	No	Yes	No

`until`	No	No	Yes
`wait`	No**	Yes	Yes
`where`	No	Yes	No
`which`	Yes	Yes	No
`while`	No	Yes	Yes

ls

chapter 19

summary

This chapter looks at `ls`, a Unix (and Linux) command.

`ls` is the Unix (and Linux) list command.

The `ls` command was described in the first UNIX book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, published November 3, 1971.

list directory contents

Use the `ls` command to list the contents of the current working directory (files and directories).

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures
Library          Public
$
```

PC-DOS equivalent

`ls -la` is the UNIX equivalent of the MS-DOS or PC-DOS command `DIR`. You can add the PC-DOS equivalent to your shell session with the `alias` command. To make the change permanent, add the following line to the `.bashrc` file in your home directory. Note that if you add this PC-DOS/MS-DOS equivalent, only add the all upper case version, because the lower case `type` is an important UNIX command that you will also need.

```
$ alias DIR="ls -la"
```

`ls -la | less` is the UNIX equivalent of the MS-DOS or PC-DOS command `DIR /P`.

list all

Type `ls` with the `-A` option to get a list of all entries other than `.` (single dot) and `..` (double dot).

```
$ ls -A
```

list with indicators

Type `ls` with the `-F` option to have special indicators for special files.

```
$ ls -F
Desktop/               Music/                 file01.txt
```

```
Documents/              Pictures/               file02.txt
Downloads/              Public/                 names
Library/                Send registration@      numberfile.txt
Movies/                 Sites/                  testdir/
$
```

A slash / will be added after every pathname that is a directory. An asterisk * will be added after every file that is executable (including scripts). An at sign @ will be displayed after each symbolic link. An equals sign = will be displayed after each socket. A percent sign % will be displayed after each whiteout. A vertical bar | will be displayed after each item than is a FIFO.

`ls -F` is a builtin command in `csh`.

hidden files

The `-a` option will give a list of all files, including hidden files.

```
$ ls
Desktop     Downloads   Movies   Pictures   scripts
Documents   Library     Music    Public
$ ls -a
.               .cups       Library    scripts
..              .vminfo     Movies
.DS_Store       Desktop     Music
.Trash          Documents   Pictures
-bash_history   Downloads   Public
$
```

The hidden or invisible files are hidden to make it less likely that they are accidentally corrupted or deleted.

list specific directory

So far all of the examples have been on the current working directory. If you name a specific directory, you will get a listing for that directory/ You may use any of the flags you have alrady learned.

```
$ ls Music
GarageBand      iTunes
$
```

multiple directories

You can list multiple directories, each separated by a space character. This will give the listing for each directory, one at a time.

```
$ ls Movies Music
Movies:
iMovie Projects.localized

Music:
GarageBand      iTunes
$
```

separate lines

The `-1` option will cause each item on its own line. This is useful for scripts that will do further processing.

```
$ ls -1
Desktop
Documents
Downloads
Library
Movies
Music
Pictures
Public
scripts
$
```

option grouping

You can combine multiple options for a single command. For example, if you wanted to show visible and invisble files *and* show the file indicators, you can use both the `-F` and `-a` options together.

```
$ ls -a -F
./              .cups/       Library/    scripts
../             .vminfo      Movies/
.DS_Store       Desktop/     Music/
.Trash/         Documents/   Pictures/
-bash_history   Downloads/   Public/
$
```

You can group option by listing them in any order (with a few rare exceptions) and either with their own minus sign or after a single minus sign. All of the following versions of the `ls` command work the same way:

```
$ ls -a -F
$ ls -F -a
$ ls -aF
$ ls -Fa
$
```

list specific file

You can `ls` a specific file or files. This is primarily useful with other options that report additional information.

```
$ ls filename
```

list specific directory

You can `ls` a specific directory. It will list all of the files in the designated directory.

```
$ ls directory
```

You can the `-d` switch to have `ls` list just the named directory. This is pretty much useless by itself, but useful when combined with switches for specific kinds of information.

```
$ ls -d directory
```

list files and directories

You can `ls` a a combined list of specific files and directories.

```
$ ls filename directory
```

common error

The most common error is attempting to `ls` a directory or file that doesn't exist.

colorized

Use the `-G` option to colorize the output of `ls`.

```
$ ls -G
Desktop         Downloads       Movies          Pictures        file01.txt      names
scripts
Documents       Library         Music           Public          forthfunctions
numberfile.txt  testdir
$
```

You can edit your `.profile` file to set specific colors for directories, sockets, pipes, symlinks, etc. You can also set colors for executable files. You can even set colors for specific file extensions.

The default for Mac OS X (shown above) is royal blue for directories.

Daniel Sandman wrote:

> I have this as an alias in my `~/.zshrc`.
>
> ```
> alias ls="ls --color=auto -FshX"
> ```

list all subdirectory contents

Use the `-R` switch to show all of the contents of all of the subdirectories. This can be a huge list. The `-R` stands for recursive.

```
$ ls -R
```

long listing

Use the `-l` switch for a long listing. Each file or directory is on its own line. One sample line shown:

```
$ ls -l
drwxr-xr-x  2 admin  staff  68 Aug 20 16:43 testdir
$
```

The order of information

Mode file type (see chart below), file permissions (owner, group, other in the order of read, write, execute), possible special information.

b Block special file
c Character special file
d Directory
l Symbolic link
s Socket link
p FIFO (pipe)
- Regular file

If the file or directory has extended attributes the permissions field is followed by the `@` character.

If the file or directory has extended security information (such as an Access Control List), the permissions field is followed by the `+` character.

links The number of links.

user The user name. If the user name is unknown, the numeric ID is shown. Adding the `-n` switch will force the numeric ID instead.

group The group name. If the group name is unknown, the numeric ID is shown. Adding the `-n` switch will force the numeric ID instead.

size The number of bytes. If the file is a character special or block special file, the major and minor device numbers replace the size field.

date and time The abbreviated month, day-of-month, hour and minute the file was last modified. If the modification time is more than six (6) months past or future, then the year replaces the hour and minute fields.

pathname The name of the file or directory. If the file is a symbolic link, the pathname of the linked-to file is preceded by `->`.

The first line (before the individual file entries) is the number of blocks used by the files in the directory.

list in reverse order

Use the `-r` switch to show the results in reverse order.

```
$ ls -r
```

list by time of modification

Use the `-t` switch to show the results by time of modification (most recent first).

```
$ ls -t
```

list by time of last access

Use the `-u` switch to show the results by time of last access (most recent first).

```
$ ls -u
```

list by time of file creation

Use the `-U` switch to show the results by time of file creation (most recent first).

```
$ ls -U
```

show time information

Use the `-T` switch to show full time information: month, day, hour, minute, second, year. This switch needs to be comibined with the `-l` switch. Sample line below:

```
$ ls -Tl
drwxr-xr-x   2 admin   staff   68 Aug 20 16:43:15 2013 testdir
$
```

show extended information

Use the `-@` switch to show extended attribute keys and sizes. This switch needs to be comibined with the `-l` switch.

```
$ ls -@l
```

show Access Control List

Use the `-e` switch to show the Access Control List (ACL) associated with a file when there is an ACL. This switch needs to be comibined with the `-l` switch.

```
$ ls -el
```

show number of blocks

Use the `-s` switch to show the size in blocks for each file. The UNIX default block size is 512 bytes, but thsi can very different on modern versions of UNIX or Linux.

```
$ ls -s
```

advanced information

scripting gotchas

This is an advanced item, but it is placed in this chapter so that you can easily find where to look it up when you need it.

Do not use the output of `ls` in a BASH `for` loop.

```
$ for i in $(ls *.mp3); do # Wrong!
$    command $i # Wrong!
$ done # Wrong!

$ for i in $(ls *.mp3) # Wrong!
$ for i in `ls *.mp3` # Wrong!
```

If any file name has spaces in it, then it will be word split. So, the MP3 for Alex Clair's "Too Close" (number four song on his "The Lateness of the Hour" album) might be `04 Too Close`. In that case, your command would run on the files "04", "Too", and "Close", rather than "04 Too Close". Additional errors likely to follow. Listen to the song "Too Close" for free at http://www.thissideofsanity.com/musicbox/musicbox.php?mastersong=485 .

Double quote won't work either because that will cause the entire output of the `ls` command to be treated as a single item. Your loop will run once on all of the file names conactenated together.

```
$ for i in "$(ls *.mp3)"; # Wrong!
```

The solution is to run your loop without even using `ls`, relying on BASH's filename expansion. Don't forget to check for the possibility that there are no qualifying files in the current directory.

```
$ for i in *.mp3; do
$    [[ -f "$i" ]] || continue
$    command "$i"
$ done
```

Thanks to Daniel Massimillano for a correction.

cd

chapter 20

summary

This chapter looks at `cd`, a Unix (and Linux) command.

`cd` is used to Change Directory.

`cd` is a builtin command in `bash` and `csh`. There is also an external utility with the same name and functionality.

`cd` is used to change the directory you are working in. You type the command `cd` followed by a space and then the directory (folder) that you want to change to.

The `chdir` (change directory) command, the predecessor of `cd`, was described in the first UNIX book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, published November 3, 1971.

change directory example

For the purposes of an example, we need to first find a directory to change to. Type the command `ls -F`.

```
$ ls -F
Desktop/                Music/                  file01.txt
Documents/              Pictures/               file02.txt
Downloads/              Public/                 names
Library/                Send registration@       numberfile.txt
Movies/                 Sites/                  testdir/
$
```

Your listing will be different than this one. Look for any name that has a slash (`/`) after it and use it as the *directory_name* in the following example. If you created the testdir in the quick tour chapter, then use the second example listed here.

```
$ cd directory_name
admins-power-mac-g5:directory_name admin$
```

If you created the testdir in the quick tour chapter, then use the following example:

```
$ cd testdir
admins-power-mac-g5:testdir admin$
```

Use the `pwd` command to confirm that you are now in your new directory.

```
$ pwd
/Users/admin/testdir
admins-power-mac-g5:testdir admin$
```

Use the `cd` command without any additional arguments to return to your home directory from anywhere.

```
$ cd
admins-power-mac-g5:~ admin$
```

Use the `pwd` command to confirm that you are now back in your home directory.

```
$ pwd
/Users/admin
admins-power-mac-g5:~ admin$
```

return to home directory

You can always return to your home directory from anywhere by typing `cd` command all by itself.

```
$ cd
admins-power-mac-g5:~ admin$
```

If the `HOME` variable is set (which is normal), then you can use it to return to your home directory.

```
$ cd $HOME
admins-power-mac-g5:~ admin$
```

return to previous directory

To return to the previous directory, type `cd ~-`

```
    admins-power-mac-g5:~ admin$ cd testdir
changing to the testdir directory
    admins-power-mac-g5:testdir admin$ cd ../Library
changing to the Library directory
    admins-power-mac-g5:Library admin$ cd ~-
changing back to the testdir directory, the immediately preceding directory
    admins-power-mac-g5:testdir admin$
```

go one level up

Type `cd ../` to go up one directory level. The following exampel starts in the testdir directory and goes up one level to the home directory.

```
    admins-power-mac-g5:testdir admin$ cd ../
changing to the home directory, one level up from the testdir directory
    admins-power-mac-g5:~ admin$
```

You can type a series of `../` to go up multiple directories. `cd ../../` will go up two levels. `cd ../../../` will go up three levels. And so on.

common cd errors

The two most common `cd` errors:

- Attempting to `cd` to a file. You can only `cd` to a directory.
- Attempting to `cd` to a directory that doesn't exist. The directory must already exist.

PC-DOS equivalent

`cd ..` is the UNIX equivalent of the MS-DOS or PC-DOS command `CD..` You can add the PC-DOS equivalent to your shell session with the `alias` command. To make the change permanent, add the following line to the `.bashrc` file in your home directory.

```
$ alias CD..="cd .."
```

advanced topic

The following is an advanced topic. I am placing it in this chapter so that you can easily find it when you need it in the future.

If you use `cd` to change to a symlinked directory, `pwd` will show the logical symlinked path rather than the actual physical path. You can use the `-P` option to force the change of directory to be resolved to the actual physical path.

```
$ cd -P /symlink
$ pwd
$ /Users/admin
$
```

cp

chapter 21

summary

This chapter looks at `cp`, a Unix (and Linux) command.

`cp` is used to copy a file.

copy a file

Use the `cp` command to make a copy of a file. This example assumes you created the names file in the quick tour chapter.

```
$ cp names saved_names
$
```

Notice that there is no confirmation of the file copy being made.

This silent behavior is typical of any UNIX shell. The shell will typically report errors, but remain silent on success. While disconcerting to those new to UNIX or Linux, you become accustomed to it. The original purpose was to save paper. When UNIX was first created, the terminals were mostly teletype machines and all output was printed to a roll of paper. It made sense to conserve on paper use to keep costs down.

You can use the `ls` command to confirm that the copy really was made. You won't be using up any paper.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library          Public        saved_names
$
```

`cp` makes an exact copy of a file.

common errors

Common errors:

- the source is a directory
- the source does not exist
- attempt to copy a file to itself

overwriting files

The plain `cp` command will simply replace any existing file with a matching name. No warning.

```
$ cp names saved_names
$
```

In the above example, if the file `save_names` already existed, it will be replaced with a copy of `names`.

interactive confirmation

You can type `cp -i` to get an interactive confirmation before replacing a file. Enter `n` or just the RETURN key to stop the copy and prevent an overwrite. Enter `y` to go ahead with the copy and replace the old file.

```
$ cp -i names saved_names
overwrite saved_names? (y/n [n]) n
not overwritten
$
```

force overwrites

Use the `-f` option to force the copy to overwrite any file with a matching name. This is the same as the default action of the `cp` command.

```
$ cp names -f saved_names
```

copy to a directory

To copy a file to a different directory, name the directory with a leading slash (`/`) as the destination:

```
$ cp sourcefilename directory/
```

Try this example:

```
$ cp names testdir/
$ ls testdir
names
$
```

A copy of the file will be stored in the specified directory with the original name. You can add a new file name after the directory if you want to change the name of the copy at the same time you create it in the specified directory.

```
$ cp sourcefilename /directory/newname
```

Try this example:

```
$ cp names testdir/newnames
$ ls testdir
names           newnames
$
```

copying an entire directory

To copy an entire source directory to a different directory, name each directory:

```
$ cp sourcedirectory destinationdirectory/
```

This does not work on all systems without using the `-r` or `-R` switch (see next item).

copying an entire directory and subdirectories

If you want to copy a directory and all of its subdirectories (the directory subtree), then use the recursive switch. The recursive switch will be either `-r` or `-R`. Check the `man` page for your system.

```
$ cp -r sourcedirectory destinationdirectory/
```

Mac OS X supports both the `-r` and the `-R` switches for compatibility with legacy scripts, but on Mac OS X the `-r` option will not correctly copy special files, symbolic links, or fifos, while the `-R` option will copy them correctly.

Try this example:

```
$ cp -r testdir newdir/
$ ls newdir
names           newnames
$
```

copying multiple items to a directory

You can create a list of sources (both files and directories) that are all put into a single destination directory. The last item listed must be a directory and is the destination. You can optionally use the `-r` or `-R` flag (which ever is appropriate for your system) to recursively copy source directories with all of their subdirectories. SOme systems may require the `-r` or `-R` switch if you include directories as well as files int he source list.

```
$ cp sourcedirectory1/ sourcedirectory2/sourcefile destinationdirectory/
```

common error

A common error is that the destination folder already exists.

PC-DOS equivalent

`cp -i` is the UNIX equivalent of the MS-DOS or PC-DOS command `COPY`. You can add the PC-DOS equivalent to your shell session with the `alias` command. To make the change permanent, add the following line to the `.bashrc` file in your home directory. Note that this version adds interactive questioning before replacing a file, in a manner similar to PC-DOS. Note also that if you add this PC-DOS/MS-DOS equivalent, only add the all upper case version, because the lower case `type` is an important UNIX command that you will also need.

```
$ alias COPY="cp -i"
```


data

mv

chapter 22

summary

This chapter looks at `mv`, a Unix (and Linux) command.

`mv` is used to move or rename a file.

rename a file

Use the `mv` command to rename a file. This example assumes you created the names and saved-name files in the quick tour chapter.

```
$ mv saved_named old_names
$
```

Notice that the shell is once again silent with success. You can use the `ls` command to confirm that the rename really was made.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library          Public        old_names
$
```

common error

A common error is the source file not existing (often because of a typing or speling error).

interactive confirmation

Use the `-i` option to have an interactive confirmation before overwriting an existing file. Note that this does not seem to work on Mac OS X Mountain Lion.

```
$ mv -i names saved_names
overwrite saved_names? (y/n [n]) n
not overwritten
$
```

force overwrites

Use the `-f` option to force rename (`mv`) to overwrite any file with a matching name. This is the same as the default action of the `mv` command.

```
$ mv -f names saved_names
```

prevent overwrites

Use the `-n` option to prevent rename (`mv`) from overwriting any file with a matching name.

```
$ mv -n names saved_names
```

PC-DOS equivalent

`mv` is the UNIX equivalent of the MS-DOS or PC-DOS command `REN`. You can add the PC-DOS equivalent to your shell session with the `alias` command. To make the change permanent, add the following line to the `.bashrc` file in your home directory.

```
$ alias REN="mv"
```

rm

chapter 23

summary

This chapter looks at `rm`, a Unix (and Linux) command.

`rm` removes or deletes files or directories.

`rm` is used to remove or erase a file.

`rm` was part of the original 1969 version of UNICS (the original name for UNIX).

delete a file

Use the `rm` (remove) command to delete a file. This example assumes you created the names saved_names and old_names files in the quick tour chapter.

```
$ rm old_names
$
```

You can use the `ls` command to confirm that the file was really deleted.

```
$ ls
Desktop          Movies        Send registration
Documents        Music         Sites
Downloads        Pictures      names
Library
$
```

safety

Before using `rm`, especially `rm` with globbing (such as using wild cards), try the same glob with `ls` first to make sure you will be removing the files you intend to remove.

For example, before running `rm foo*.txt`, run `ls foo*.txt`. Read the list and be sure that you are going to delete the files you really intend to delete.

Some UNIX and Linux users modify the `rm` command by aliasing it with the interactive form, `rm -i`. This changes the behavior of `rm` to always ask before deleting files.

```
$ alias rm="rm -i"
```

Unfortunately, this alias can cause some scripts to fail (possibly including system scripts that were provided with your distribution).

An alternative is to alias to a different, non-conflicting name. In this case, `del` might be appropriate.

```
$ alias del="rm -i"
```

PC-DOS equivalent

`rm -i` is the UNIX equivalent of the MS-DOS or PC-DOS command `DEL`. You can add the PC-DOS equivalent to your shell session with the `alias` command. To make the change permanent, add the following line to the `.bashrc` file in your home directory. Note that this version adds interactive questioning before deleting a file, in a manner similar to PC-DOS.

```
$ alias DEL="rm -i"
```

deleting a file name with spaces

Sometimes a file name has space or tab characters in it. This is strongyl discouraged on a server, but is fairly common on a desktop Linux system. Enclose the file name in quotation marks.

```
$ rm "sample file"
$
```

recursive

Type `rm` followed by the option `-r` followed by *directory name* to delete recursively. This means that the directory and all of its subdirectories will be removed, even for non-empty direcories. This is a fast way to remove directories and files in a single command. This is also a very dangerous command. Always include the directory name rather than `*` to prevent accidentally destroying the wrong directories.

```
$ rm -r directory-name
$
```

This is also a very dangerous command. You may want to include the `-i` interactive flag to confirm that you are really deleting the correct files.

```
$ rm -ri directory-name
$
```

dangerous command

WARNING! Do not test these examples. They will delete all of the files on your ocmputer.

These commands often appear in cruel internet jokes.

```
$ rm -rf /
$ rm -rf *
```

advanced information

file won’t delete

The following information is rather advanced and requires knowledge of that hasn’t been introduced in this book yet. This advanced material is placed here so that you can easily find it when using this book as a reference rather a tutorial. For now, just note that this material is located here.

Sometimes you will attempt to delete a file and the shell will tell you the file can’t be deleted.

```
$ rm stubborn_file
rm: cannot remove `stubborn_file': Permission denied
$
```

You can try to `chmod 777` the file or `chown` the file or directory.

The next line of defense is to use `sudo` and delete the file as root or superuser.

```
$ sudo rm stubborn_file
$
```

If you can’t remove the file as root, then there are a number of different possibilities to consider. The following is a checklist of some of the possibilities.

It might be a matter of directory permissions. Directory permissions, not file permissions, control who can delete and create files. If you don’t have write-permissions for the directory, you can’t delete the file.

Interestingly, the converse is also true. A user can delete a file set to 000 and owned by root in a directory that he or she has access to, even though they can’t even view the file contents. Of course, the sticky bit on a directory sets files so that they can only be deleted by their owners. It is common to set the sticky bit on certain key directories, such as `/tmp/`.

The root or superuser should be able to override any regular permissions and delete any file anyway, so the you will want to check Access Control Lists (ACLs), extended attributes, and other special features that may be in play. You can try `lsattr` to list the attributes on a Linux second extended file system (ext2) and `chattr` to change the attributes. The `a` attribute is append mode and can be removed with `chattr -a stubborn_file`. The `i` attribute is immutable and can be removed with `chattr -i stubborn_file`. On Mac OS X, you can use the similar `chflags` command (for UFS, HFS+, SMB, AFP, and FAT). See the man pages for details on these commands.

Another possibility is that the file may be on Network File System (NFS). Your local root won’t matter. You need root access to the machine running NFS.

```
$ rm stubborn_file
$
```

There may be unusual characters in the file name.

You may be able to use auto-completion to correctly escape the file name. Type the first character of file name, then press the Tab key. The shell should auto-complete the file name with correct escapes for unusual characters.

You can check to see if there are invisible or non-printing characters or spaces or tabs at the beginning or end of the file name by comparing the results of `ls stubborn_file` with the output of `ls *stubborn_file*`. If you get no result from the first command and a result from the second command, then there are extra characters. Try using the inum solution mentioned below.

Another method for checking for the real file name:

```
$ ls -la stubborn_file | sed -n l
-rw-r--r-- 1 Mac staff 17 Jan 11 02:56 stubborn_file$
$
```

Another method for checking for the real file name is to use the `-B` option to force the printing of non-printable characters. The non-printable characters will be presented as `\xxx` where `xxx` is the numeric character code in octal. The `-b` option is similar, but will use C programming language escape codes whenever possible.

I have read that Linux supports the `-Q` option to place the file names within quotation marks, but I have not tested this on a Linux system.

Once you identify the real file name, you can use quotation marks to give the complete file name.

```
$ rm "stubborn file"
$
```

In some cases you may need to use single quotes. An example would be a file name with the character `$` in it.

```
$ rm 'stubborn $file'
$
```

You can try to remove the file using its inode rather than name. Use the correct inum output from your real session. Note there may be a long delay before success, so you may want to run this command in the background.

```
$ ls - i stubborn_file
3982096 stubborn_file
$ find . -inum 3982096 -exec rm -f {} \;
$
```

shred

chapter 24

summary

This chapter looks at `shred`, a Linux command.

`shred` is a secure method for eliminating data, overwriting a file.

```
$ shred filename
```

Files are normally deleted by removing their file table entries. The data remains out on the hard drive (which is how file recovery programs can work).

Writing over the file actually eliminates the data so that nobody can ever read it.

This can take a while for large files or for many files.

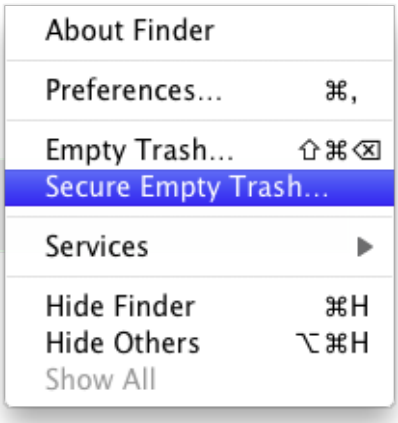
Files deleted with `shred` can *not* be recovered (unless you previously stored them in a backup somewhere else).

Be especially careful when using `shred` with any wildcard characters.

You should probably never run `shred` from root or superuser.

Mac OS X

Mac OS X offers secure empty trash from the graphic interface (Finder menu, Secure Empty Trash menu item, which is directly under regular Empty Trash).



Mac OS X also has the `srm` command, which is the secure equivalent of regular `rm`.

```
$ srm -rfv -s filename
```

The `-z` option will overwrite with zeros. The `-s` option will overwrite in a single pass (one pass) of random data. The `-m` option will perform a Department of Defense seven-pass deletion. No options will result in a 35-pass file erase, which will take a long time.

mkdir

chapter 25

summary

This chapter looks at `mkdir`, a Unix (and Linux) command.

`mkdir` is used to make a directory.

make a directory

Use the `mkdir` command to make a new directory (folder).

```
$ mkdir testdir
admins-power-mac-g5:~ admin$
```

The example includes an extended prompt that shows the current working directory. Using the `ls` command will confirm that the directory now exists in the current working directory.

common errors

`mkdir` will fail if the directory already exists.

`mkdir` will fail if a file with the same name already exists.

multiple directories

You can name multiple directories with the `mkdir` command. Simply list each of the new diretcories after the `mkdir` with a space character between each directory name.

```
$ mkdir directory1 directory2 directory3
admins-power-mac-g5:~ admin$
```

create with permissions

Use the `-m` option to create a directory with specific permissions. The old school version of the `-m` switch used octal permissions.

The permission chocies are the same as with the `chmod` command. Replace *mode* (`a=rwx` in the following example) with an appropriate mode argument. In the following example, the*mode* is `a=rwx`, which would give all users read, write, and execute permissions.

```
$ mkdir -m a=rwx directory1 directory2 directory3
admins-power-mac-g5:~ admin$
```

This option is often used in shell scripts to lock down temporary directories.

use a path

All of the previous examples involved the current working directory. You can use `cd` to move to a new directory and create a directory there. Or you can give an entire path name to a specific directory, using either a relative or absolute path name to create the directory anywhere in the file system.

```
$ mkdir directory1/directory2/directory3
admins-power-mac-g5:~ admin$
```

common error

The most common error in using `mkdir` with a path is one or more of the intervening directories haven't been made yet. `mkdir` will fail.

creating parent directories

There is a solution to this problem. You can use the `-p` option to create directories along the path. You can therefore, create a series of parent and child directories all at once.

If you use the `-p` option and the directory already exists, the `mkdir` command will continue on through your path and create new directories as needed down to the last one you list.

```
$ mkdir -p directory1/directory2/directory3
admins-power-mac-g5:~ admin$
```

Intermediate directories are created with the permission bits of `rwxrwxrwx` (0777) as modified by the current umask, plus the write and search permission for the owner.

view directories made

You can use the `-v` to get a list of the directories created. This is most often used with the `-p` option, but you can use it by itself.

```
$ mkdir -v directory1/directory2/directory3
$
```

```
$ mkdir -pv directory1/directory2/directory3
$
```

The `-v` option is considered nonstandard and should not be used in shell scripts.

spaces in names

To create a directory with space in the directory name, you need to use quoting for the entire name so that the shell treats it as a single name rather than a series of individual directory names separeted by spaces.

```
$ mkdir "directory name"
$
```

While it is common to use spaces in file and directory names in graphic user interfaces (such as Macintosh, OS/2, and Windows, as well as Gnome or KDE), this may cause failure with older scripts or programs. Some programs may convert the spaces in a directory name into `%20`, such as converting "Name with spaces" into "Name%20with%20spaces". This might cause confusion or failure.

long options

The long options are:

```
-m    --mode=MODE
-p    --parents
-v    -verbose
-Z    --context=CONTEXT
      --help
      --version
```

advanced information

The following items are advanced topics. They are included here so that you can easily find them when needed.

scripting mkdir

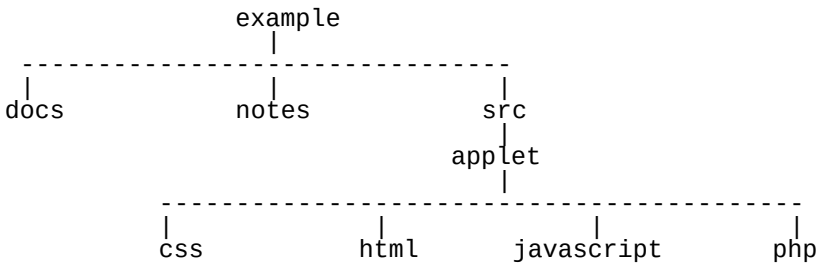
If you create a directory in a script, you will want to check first to see if the script actually exists. The following line of code will create a directory called `tmp`

```
$ test ! -d $HOME/tmp && mkdir $HOME/tmp
```

This line relies on the short circuit evaluation by the `test` statement. The first part of the `test` checks to see if the specified directory does not exist. If the specified directory already exists, the `test` fails and further evaluation stops. If the `test` passes, then the `mkdir` command is run.

create an entire directory tree

You can create an entire directory tree all at once. The example will create the following diretcory tree (which will be confirmed with `ls` commands):




```
$ mkdir -p example/{src/applet/{css,html,php,javascript},docs,notes}
$ ls example
docs    notes   src
$ ls example/src
applet
$ ls example/src/applet
css      html     javascript     php
```

```
$
```

The `{}` curly braces are used to create directories at the same level and the directories at the same level are separated by the `,` comma.

create a directory and change to it

You can use the following function to both `mkdir` a new directory and `cd` to the newly created directory, all in one command.

```
$ function mkdircd () { mkdir -p "$@" && eval cd "\"\$$#\""; }
$
```

This function will work even if the directory name has spaces.

various operating systems

The `mkdir` command appears in the Linix, Mac OS X, OS/2, PC-DOS (also called MS-DOS), UNIX, and Windows operating systems. It also appears in the PHP programming language. In OS/2, PC-DOS, and Windows, the command can be abbreviated `md`.

set security context

When running SELinux, you can use the `-Z` or `--context` switch to set the security context. You must give the full context as `user:role:type`.

An example from the CentOS forum:

```
$ mkdir --context=system_u:object_r:httpd_sys_content_t:s0 /var/www/example.org
$
```

history

The `mkdir` command appeared in Version 1 of AT&T UNIX. In early versions of UNIX, the `mkdir` command had to be `setuid root` because the kernel did not yet have a `mkdir` syscall. The `mkdir` command made a directory with the `mknod` syscall, followed by linking in the `.` and `..` directories.

The `mkdir` syscall originated in 4.2 BSD and was added to System V in Release 3.0.

alias

chapter 26

This chapter looks at `alias`, a Unix (and Linux) command.

`alias` is used to create short names for long strings of commonly performed commands, scripts, and functions, and their combinations.

`alias` is a builtin command in `bash` and `csh`. There is also an external utility with the same name and functionality.

example

You can use the alias command to modify the `rm` so that it always displays an interactive query before deleting a file.

```
$ alias rm="rm -i"
```

Unfortunately, this alias can cause some scripts to fail (possibly including system scripts that were provided with your distribution). Any scripts that rely on the unmodified version of an aliased `rm` are likely to fail or produce strange results because of your alias.

A better alternative is to use an alias to a different, non-conflicting name. In this case, `del` might be appropriate.

```
$ alias del="rm -i"
```

overriding aliases

If you have set an alias on a command or utility, you can always run the raw unmodified command or utility by typing the absolute directory path to the command. For example, if you set an alias to have `ls` always show hidden files, you can still run the regular `ls` by typing its full path name.

```
$ alias ls="ls -a" #modify the ls command
$ /bin/ls #run the raw ls command
```

Some system administrators always type out the complete path name for a command when using the root or superuser account to help prevent mistakes.

pipes

chapter 27

summary

This chapter looks at pipes.

Pipes are most commonly used as a method to chain different utilities together, with the output of one piped tot he input of the next.

history

Ken Thompson invented the pipe in 1972.

The pipe is a standard mechanism for allowing the output of one program or process to be used as the input for another program or process.

Pipes allow the UNIX philosophy of having many small programs that each do one function very well and then write scripts that combine these small utility programs to accomplish bigger tasks.

how to pipe

To pipe the output of one UNIX command or utility to the next, simply place the | (vertical bar character) between the two processes:

```
$ cat file01.txt | sort | lp
```

The example obtains the contents of the designated file, sorts the contents, and then sends the result to the printer.

scripts

chapter 28

summary

This chapter looks at scripts.

Scripts are the power tools for system administrators. Scripts allow you to automate tasks.

shell initialization files

When you start Terminal (or, in some cases, run `login`) the shell will run two initialization scripts:

```
/etc/profile
.profile
```

Note that these are the file names for BASH. See the login chapter for the default initialization file names for other shells.

The `/etc/profile` initialization script is maintained by the system administrator and is the first initialization script run for all users.

Each user has their own `.profile` script in their home directory.

The `.profile` includes three basic operations (set terminal type, set path to commands, and set path to `man` pages) and any customization created by the individual user.

Before the initialization scripts are run, the shell is considered to be an *uninitialized shell*.

.profile

The `.profile` initialization script sets the terminal type:

For example: `TERM=vt100`

The `.profile` initialization script sets the path to the directory or directories that contains the commands and utilities.

For example: `PATH=/bin:/usr/bin`

Notice the colon character (`:`) is used as a separator between directories, allowing commands and utilities to be organized in many directories.

The `.profile` initialization script sets the path to the directory or directories containing the manual (`man`) pages.

For example: `MANPATH=/usr/man:/usr/share/man`

interactive and noninteractive

When you directly enter a command or commands and the shell performs the commands for you, that is the *interactive* mode.

You can also have the shell run scripts or commands in a *noninteractive* or *batch* mode. This is great for a system administrator automating a server or other large scale Unix/Linux/Mac OS X system.

When an non-interactive shell (or subshell) finishes running a script, it exits.

making a script executable

Scripts are just text files with lists of instructions.

To make a text file into an executable script that you can run, simply type `chmod a+x filename`.

```
$ chmod a+x filename
$
```

Also, add the *magic line* `#!/bin/sh` as the first line in the text file. This special magic line tells the shell which program to use to run the script (in this case, the basic shell program).

Shell Files and Interpreter Invocation
File Extensions
from Google Shell Style Guide

Executables should have no extension (strongly preferred) or a `.sh` extension. Libraries must have a `.sh` extension and should not be executable.

It is not necessary to know what language a program is written in when executing it and shell doesn't require an extension so we prefer not to use one for executables.

However, for libraries it's important to know what language it is and sometimes there's a need to have similar libraries in different languages. This allows library files with identical purposes but different languages to be identically named except for the language-specific suffix.

scripting languages

Some of the scripting languages available on Mac OS X are: sh, bash, perl, PHP, python, and ruby.

Examples of running scripts in these languages:

```
$ sh scriptfilename
$ bash scriptfilename
$ perl scriptfilename
$ php scriptfilename
$ python scriptfilename
$ ruby scriptfilename
$
```

shell script example

Create a directory (folder) for your personal scripts called `scripts`:

```
$ mkdir scripts
$
```

Create a shell script called `script.sh` and save it in the new scripts directory (folder):

```
#!/bin/sh
echo "Hello World!"
```

Use the `chmod` command to make the new file an executable script:

```
$ chmod u+x /scripts/script.sh
$
```

Add the scripts directory to your command path:

```
$ export PATH="$PATH:~/scripts"
$
```

Run your new script:

```
$ script.sh
Hello World!
$
```

You can run your script directly in a specific shell:

```
$ rbash script.sh

$ sh script.sh

$ bash -x script.sh
```

php script example

This example assumes that you have created the `scripts` directory.

Create a php script called `script.php` and save it in the new scripts directory (folder):

```
<?php
echo "Hello World!"
?>
```

Notice that we skip the `chmod` step.

Run your new script by running the `php` program with your script as the file to execute:

```
$ php ~/scripts/script.php
Hello World!
$
```

Note that the shell does *not* render HTML, so if you run a web scirpt, you will see raw HTML, CSS, and JavaScript as plain text in your terminal window.

You can run perl, ruby, and python scripts in the same manner.

There is more complete information in the chapter on php and shell.

When to use Shell

from Google Shell Style Guide

Shell should only be used for small utilities or simple wrapper scripts.

While shell scripting isn't a development language, it is used for writing various utility scripts throughout Google. This style guide is more a recognition of its use rather than a suggestion that it be used for widespread deployment.

Some guidelines:

- If you're mostly calling other utilities and are doing relatively little data manipulation, shell is an acceptable choice for the task.
- If performance matters, use something other than shell.
- If you find you need to use arrays for anything more than assignment of `${PIPESTATUS}`, you should use Python.
- If you are writing a script that is more than 100 lines long, you should probably be writing it in Python instead. Bear in mind that scripts grow. Rewrite your script in another language early to avoid a time-consuming rewrite at a later date.

sysadmin and root/superuser

chapter 29

summary

This chapter looks at system administration and the root or superuser account.

As we look at more advanced features and commands, we need to become aware of the root or superuser account and capabilities and how to safely access that power.

sysadmin

In the UNIX world, the system administrator is often called the sysadmin.

root/superuser

The root or superuser account has total authority to do anything on the system. This power is great for fixing problems, but bad because one accidenttally mistyped character could be very destructive. Some systems also have admin accounts of similar power.

Because of the potential for destructiveness, system administrators typically login with either a normal user account or a limited admin account for every day work and then switch to superuser or root only when absolutely necessary (and then immediately switch back).

The UNIX command for temporarily switching to root or superuser power is the `sudo` command, discussed in the next chapter.

special powers

The root or superuser account has powers that “mere mortal” accounts don’t have.

The root account has access to commands that effect the entire computer or system, such as the ability to halt or shutdown the system.

The root account is not affected by read and write file access permissions. The root or superuser account can create, remove, read, or write any file anywhere on the system.

Some commands have built-in restrictions that the root or superuser can ignore. For example, the system administrator can change any user’s password without knowing the old password.

sudo

chapter 30

summary

This chapter looks at `sudo`, a Unix (and Linux) command.

WARNING: Never trust any Terminal/Shell commands you find on the internet. Only run shell commands you understand. In particular, *never* run anything that includes `sudo` anywhere in the command line unless you are absolutely certain what the command does. When you run a command line with `sudo` you are giving permission to have complete (possibly destructive) control of your computer at the root level. And, yes, this advice applies even to this book. Don't run any commands with `sudo` unless you know for sure what you are doing.

sudo

The `sudo` command allows you to run a single command as another user, including at superuser or root level from a normal account. You will be asked for the password before the command will actually run.

This keeps you firmly in a normal account (with less danger of catastrophic failures), while still giving easy access to root or superuser power when really needed.

The `sudo` program was originally written by Bob Coggeshall and Cliff Spencer in 1980 at the Department of Computer Science at SUNY/Buffalo.

`sudo` is a concatenation of `su` (substitute user) and `do` (perform an action).

To run a single command as superuser or root, type `sudo` followed by a command.

```
$ sudo command
```

You will normally be asked for your password (exceptions listed below).

`sudo` can be configured to not require a password (very bad idea other than single user personal systems). `sudo` can also be configured to require the root password (rather than the current user's password).

On Mac OS X the `sudo` command will fail if your account has no password.

On Mac OS X the `sudo` commands `password` prompt will not display anything (not even bullets or asterisks) while you type your password.

You will not be asked for a password if you use `sudo` from the root or superuser account. You will not be asked for a password if you use `sudo` and the target user is the same as the invoking user.

Some systems have a timer set (usually five minutes). You can run additional `sudo` commands without a password during the time period.

run in root shell

To change to in the root shell, type `sudo` followed by the option `-s`. The following warning is from Mac OS X (entered a root shell and then immediately returned to the normal shell). Note the change to the pound sign (#) prompt.

```
$ sudo -s

WARNING: Improper use of the sudo command could lead to data loss
typing when using sudo. Type "man sudo" for more information.

To proceed, enter your password, or type Ctrl-C to abort.

Password:
bash-3.2# exit
$
```

other users

To run a command as another user, type `sudo` followed by the option `-u` followed by the user account name followed by a command.

```
$ sudo -u username command
```

To view the home directory of a particular user:

```
$ sudo -u username ls ~username
```

edit files as www

To edit a file (this example is for index.html) as user www:

```
$ sudo -u www vim ~www/htdocs/index.html
```

which password

On most systems, you will authenticate with your own password rather than with the root or superuser password. The list of users authorized to run `sudo` are in the file /usr/local/etc/sudoers or /etc/sudoers (on Mac OS X, /private/etc/sudoers). These authorized users are identified in the sudoers file as `admin`.

The sudoers configuration file offers a wide variety of configuration options, including enabling root commands only from the invoking terminal; not requiring a password for certain commands; requiring a password per user or per group; requiring re-entry of a password every time for particular command lines; never requiring re-entry of a password for a particular command line. The sudoers configuration file can also be set support the passing of arguments or multiple commands and also supports commands with regular expressions.

timeout

`sudo` can set timeout limits. This is done with the `timeout` option. This can be configured globally, per user, or per application. The timeout can be retained only per tty or globally per user. The user or application only has root authentication until the timeout occurs.

forgot to use sudo

Sometimes you type a command and forget that you needed to use `sudo` until you see the error message. You can type

`sudo !!` to run the previous command with root privileges.

```
$ head /etc/passwd
head: /etc/passwd: Permission denied
$ sudo !!
```

unreadable directories

To view unreadable directories:

```
$ sudo ls /usr/local/protected
```

shutdown

To shutdown a server:

```
$ sudo -r +15 "quick reboot"
```

saving system file in vim

The ideal method for editing and saving a system file that can only be saved by the root user is to prepend the `vim` command with `sudo`. Then the `vim` command `:w` will work because the `vim` program was launched with root privileges.

```
$ sudo vim /etc/passwd
$ some editing commands
$ :w
```

So, what do you do if you start editing the file and then remember that you need root permission to save it? Add `!sudo tee %` to the `vim` save command.

```
$ vim /etc/passwd
$ some editing commands
$ :w !sudo tee %
```

usage listing

To make a usage listing of the directories in the /home partition (note that this runs the commands in a sub-shell to make the `cd` and file redirection work):

```
$ sudo sh -c "cd /home ; du -s * | sort -rn > USAGE"
```

view sudoers configuration

To view the `sudoers` current configuration settings, type:

```
$ sudo -ll
```

editing sudoers file

Run the `visudo` command line tool to safely edit the sudoers configuration file. You will be presented with the `vi` editing interface (this can be changed by setting the shell EDITOR environment variable to a different text editor, such as `emacs`.

Any syntax error in the `sudoers` configuration file will make `sudo` stop working globally. Therefore, always use `visudo` to edit the `sudoers` file. `visudo` also provides security locks to prevent multiple simultaneous edits and other possible security problems.

graphic equivalents

The utilities `kdesudo` (KDE) and `gksudo` (Gnome) provide a graphic user interface version of `sudo` (both are based on `sudo`). Mac OS X Authorization Services provides a graphic user interface with adminstrative privileges (but is not based on the UNIX `sudo`).

start a shell as root

If you need to do extended work as root, you can start up a root shell from your user account:

```
$ sudo bash
```

running commands as root

`sudo` only works for programs, not for any built-in commands. If you attempt it, you will get an error message stating `command not found`. The solution is to start a root shell:

```
$ sudo bash
```

security

The system can be set up to send a mail to the root informing of unauthorized attempts at using `sudo`.

The system can be set up to log both successful and unsuccessful attempts to `sudo`.

Some programs (such as editors) allow a user to run commands via shell escapes, avoiding `sudo` checks. You can use `sudo`'s `noexec` functionality to prevent shell escapes.

`sudo` never does any validation of the ARGUMENTS passed to a program.

`sudo` defaults to extensive logging, using the `syslogd` system log daemon to log all commands issued with `sudo` into a cnetral host and local host file. This allows a complete audit trail of system access.

A system can be setup so that all machines in a system use the same `sudoers` file, allowing better central administration of a network.

trace

You can't `sudo strace …` (fill in the rest of the command any way you want) because `sudo` can't gain its privileges

while being traced.

polkit alternative

`polkit` (formerly PolicyKit) is an alternative control component for system-wide privileges.

SUID/SGID

from Google Shell Style Guide

SUID and SGID are *forbidden* on shell scripts.

There are too many security issues with shell that make it nearly impossible to secure sufficiently to allow SUID/SGID. While bash does make it difficult to run SUID, it's still possible on some platforms which is why we're being explicit about banning it.

Use `sudo` to provide elevated access if you need it.

su

chapter 31

summary

This chapter looks at su, a Unix (and Linux) command.

su is used to switch user accounts.

switch to another user account

You can use the su command to switch another user account. This allows a system administrator to fix things signed in to a particular user's account rather than from the root account, including using the user's settings and privileges and accesses rather than those of root.

who

chapter 32

summary

This chapter looks at `who`, a Unix (and Linux) command.

The `who` command was described in the first UNIX book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, published November 3, 1971.

list of who is using a system

The `who` command will tell you all of the users who are currently logged into a computer. This is not particularly informative on a personal computer where you are the only person using the computer, but it can be useful on a server or a large computing system.

Type `who` followed by the ENTER or RETURN key.

```
$ who
admin   console Aug 24 18:47
admin   ttys000 Aug 24 20:09
$
```

The format is the login name of the user, followed by the user's terminal port, followed by the month, day, and time of login.

which account is being used

The command is `who` with the arguments `am i` will tell you which account you are currently using. This can be useful for a system administrator if the system administrator is using the `su` command and wants to be sure about which the system administrator is currently using.

```
$ who am i
admin    ttys000  Aug 25 17:30
$
```

A related command without any spaces, `whoami`, lets you know which account you are currently logged in with.

```
$ whoami
admin    ttys000  Aug 25 17:30
$
```

advanced file systems

chapter 33

summary

This chapter is an advanced look at Unix (and Linux) file systems.

Some of the information in this chapter is a refresher carried over from previous chapters. Also see the chapter on major directories.

Important reminder: Always use the root or superuser account sparingly. Do most of your work with an ordinary user account and switch to the root account only when needed. And then switch back as soon as possible. The `sudo` command can be used for temporarily switching to the root account to run a particular command.

absolute paths

It is common for a system administrator logged in to the root or superuser accunt to type out the absolute path to a command or utility. This will skip any aliases. This will also skip over any commands in the user's defined search path. This guarantees that the correct command or utility will run.

everything is files

In UNIX, everything is a file, even hardware devices such as printers, hard drives, and other devices. Even UNIX abstractions such as directories, pipes, and file redirection are treated as files.

Becuase everything is a file, the operating system can provide a common processing and communication interface. This is essential to the UNIX approach of using lots of small programs or utilities that do one job well and connecting these programs with shell scripts to do a wide variaety of activities.

man pages

Many forms of UNIX have a `man` page on filesystems.

```
$ man filesystems
```

If your system has this man page, you will find a general overview of UNIX file systems of all types, including CDROMs, floppy disks, and tape drives.

diskless

Almost every UNIX system has a root directory and file system, which is stored on one or more hard drives connected to the computer system.

There are diskless nodes. These are computers that don't have their own local hard drive. Diskless nodes run off a network server. Mac OS X makes it easy to set up a computer to run in this manner. This approach is often used when an organization wants central control of all of the computers in the organization, such as a computer lab at a school.

root directory

You can examine the contents of the root directory with the `ls -F /` command.

```
$ ls -F /
Applications/    Volumes/        mach_kernel.ctfsys
Desktop DB       bin/            net/
Desktop DF       cores/          private/
Developer/       dev/            sbin/
Library/         etc@            tmp@
Network/         home/           usr/
System/          mach-Kernal     var@
Users/
```

The above example is for Mac OS X 10.5.8 Leopard.

On many systems you will find a file called `unix` This is the main UNIX operating system kernel.

On some systems you will find `root.home`. This is a temporary file created when running Netscape as root. This would be an example of an application-specific file.

some important directories

The following is a brief description of some of the major directories. Not all systems will have all of these directories. This list has additional information beyond the similar list in chapter on file system basics.

/ The root directory of the entire file system.

/bin A collection of binary files, better known as programs. Many of the standard system commands are stored here.

/boot or **/kernel** The files needed to boot or start your computer.

/dev The devices connected to your computer (remember, everything, even hardware devices, is treated as a file in UNIX or Linux). Examples are the keyboard, disks, and printers.

/etc System configuration files, startup procedures, and shutdown procedures.

/home or **/users** The home directories for each user.

/lib or **/Library** Library files used by executable programs.

/net Other networked systems (again, treated as files).

/opt Third party software. In older systems this might be **/usr/local**

/private On Mac OS X, this is the location of the UNIX files. It is normally kept hidden from all users in Finder, but is visible in your terminal emulator and shell. Mac OS X has links to the major UNIX directories at the root level so that UNIX and Linux tools can find their files in the standard locations.

/proc Information about processes and devices.

/sbin System binaries (that is, system programs).

/tmp Space for temporary files. Required by all forms of UNIX. This directory is normally erased on bootup.

/usr User binaries (programs), libraries, manuals, and docs. Common subdirectories include online help (/usr/share), header files for application development (/usr/include), and system configuartion files related to low-level hardware (such as /usr/cpu and /usr/gfx).

It is rare that a system administrator would even look at the low level hardware directories, such as /usr/cpu. The system automatically manages these directories and their contents. It is unwise for a system administrator to modify these directories and their contents. Messing with these will almost certainly bring the system down and may require reinstalling the entire system.

/var Variable files. Some common subdirectories include X Windows files (/var/X11), system services files, applications related files (/var/netscape), system administration files and data (/var/adm or /var/spool), a second temporary directory (/var/tmp) that is not normally erased on bootup.

Swap Virtual memory on a hard drive. Allows the memory manager to swap some data and program segments to hard drive to expand the amount of memory available beyond the limits of physical memory.

For a more advanced and complete listing, see the chapter on major directories.

FAQs

You can find a FAQ (Frequently Asked Questions) file for all UNIX variants. Find the appropriate FAQ for your version of UNIX or Linux and look up the details for your system.

/etc

A key directory for new system administrators to learn is the `/etc` directory. The `/etc` directory contains key system administration files.

Because the `/etc` files change system behavior and properties, a system administrator spends a lot of time modifying and updating these files.

A new system administrator can learn a lot about UNIX by studying the files in the `/etc` directory. Do not change these files until you have figured out what you are doing. This is not the place for random experimentation.

A few key `/etc` files a system administrator should know.

`/etc/sys_id` is the name of the system. It may contain the full domain. Not used on Mac OS X.

`/etc/hosts` is the summary of the full host names. A system administrator will add to this standard file.

`/etc/fstab` is a list of file systems to mount on bootup. Mac OS X has `/etc/fstab.hd`.

`/etc/passwd` is the password file. It includes a number of items of basic information on user accounts.

`/etc/group` is the group file. It contains details of all user groups.

/var

Another important directory for system administrators to know is the `/var` directory, which contains configuration files for many system services.

For example, the `/var/named` file configures Domain Name Service and the `/var/yp` file configures Network Information Service.

where to find answers

Surprisingly, there won't be a grand tour of the UNIX file system here.

A UNIX system typically has thousands of files. Maybe tens of thousands of files. A system administrator doesn't normally need to know what these files are or where they are located.

A system administrator should know where to find relevant information when a problem occurs.

Many system administrators keep a handy notebook with information on where to find answers to various problems. It is far more useful to know where to find correct answers than to know try to memorize all answers.

And don't forget popular search engines as a source for answers. There is very little chance that you are the first system administrator in the world to encounter a particular problem. A quick internet search should reveal many others asking for help on the same problem, along with answers.

hidden files and directories

Any file or directory that starts with the period or dot (`.`) character is considered hidden.

In MS-DOS (and Windows) and the original Macintosh, files and directories are hidden by setting a special attribute.

Mac OS X has a hybrid. When using the command line shell, Mac OS X acts exactly like any other version of UNIX. But when using the Finder, Mac OS X also recognizes the invisible bit. This is how the entire UNIX operating system is kept out of the hands of the typical Mac OS X user.

Access permissions are separate from invisibility. In MS-DOS, a hidden file can not be written to. In UNIX, you can freely read and write and otherwise modify a hidden file.

In UNIX, file access is controlled by access permissions.

As a reminder from the `ls` chapter, the `-a` option will give a list of all files, including hidden files.

```
$ ls -a
```

Each user has a few important hidden files that configure their basic working environment when they login. These include:

- `.cshrc`
- `.login`
- `.profile`

major directories

chapter 34

summary

This chapter looks at some of the major directories (folders) on Unix/Linux/Mac OS X.

directory listing

The following is a brief description of some of the major directories. Not all systems will have all of these directories.

/ The root directory of the entire file system.

/Applications On Mac OS X, this is the location of the Macintosh programs.

/Applications/Utilities On Mac OS X, this is the location of the graphic user interface based utility programs.

/bin A collection of binary files, better known as programs. These are the programs needed by the operating system that normal users might use. Many of the standard system commands are stored here. Contrast with /sbin below.

/boot or **/kernel** The files needed to boot or start your computer. Includes the bootstrap loader, the very first thing loaded when your computer starts. On Linux, it usually includes the Linux kernel in the compressed file `vmlinuz`.

/dev The devices connected to your computer (remember, everything, even hardware devices, is treated as a file in UNIX or Linux). Examples are the keyboard, disks, and printers.

/etc System-wide configuration files, startup procedures, and shutdown procedures.

/home or **/users** The home directories for each user. Contains such things as user settings, customization files, documents, data, mail, and caches. The contents of this directory should be preserved during operating system updates or upgrades.

Every new user account starts with a few basic files such as `.login`, `.cshrc`, and `.profile`. A system administrator may want to add additional customization such as an introductory `README` file.

/lib or **/Library** Shared library files and kernel modules.

/lost+found Files recovered during filesystem repair.

/mnt Mount points for removable media (floppy disks, CD-ROM, Zip drives, etc.), partitions for other operating systems, network shares, and anything else temporarily mounted to the file system. Linux and UNIX don’t use Windows-style drive letters.

/net Other networked systems (again, treated as files).

/opt Optional large applications and third party software. In older systems this might be **/usr/local**.

/private On Mac OS X, this is the location of the UNIX files. It is normally kept hidden from all users in Finder, but is visible in your terminal emulator and shell. Mac OS X has links to the major UNIX directories at the root level so that UNIX and Linux tools can find their files in the standard locations.

/proc A Linux-only directory. Information about processes and devices. The files are illusionary. The files don’t exist on the disk, but are actually stored in memory. Many Linux utilities derive their information from these files.

/sbin System binaries (that is, system programs). These are programs that general users don't normally use (although they can). This directory is not in the PATH for general users. Contrast with /bin listed above.

/root The home directory for the system administrator (superuser or root).

/tmp Space for temporary files. Required by all forms of UNIX. This directory may automatically be cleaned on a regular basis. This directory is normally erased on bootup.

/usr User binaries (programs), libraries, manuals, and documentation (docs). Common subdirectories include online help (/usr/share), header files for application development (/usr/include), and system configuartion files related to low-level hardware (such as /usr/cpu and /usr/gfx).

/var Files that change, such as spool directories, log files, lock files, temporary files, and formatted (on use) manual pages. Some common subdirectories include X Windows files (/var/X11), system services files, applications related files (/var/netscape), system administration files and data (/var/adm or /var/spool), a second temporary directory (/var/tmp) that is not normally erased on bootup.

Swap Virtual memory on a hard drive. Allows the memory manager to swap some data and program segments to hard drive to expand the amount of memory available beyond the limits of physical memory.

Network File System (NFS)

chapter 35

summary

This chapter looks at Network File System (NFS) on Unix/Linux/Mac OS X.

The Network File System (NFS) allows a directory on one machine to be accessed from another machine. This is an important feature of UNIX.

The procedure for setting up the Network File System is called mounting.

NFS provides a seemless method to merge different file systems from different physical machines into a large shared structure. NFS works across different kinds of UNIX or Linux and even different kinds of processors.

With NFS, a user doesn't need to know or keep track of the actual physical locations of directories and files. The NFS files and directories appear to be a normal part of the file system and are accessed with the exact same methods as any local file or directory.

mount/export

The `mount` command will replace a local directory with a directory from another machine. Anyone viewing the mounted directory will see the remote directory as if it were local.

The `export` command is used to make a directory available for other computers to mount it.

The `/etc/exports` file contains a list of directories to be exported.

`mount` should be run from root.

availability

Network File System (NFS) was developed by Sun Microsystems in 1984 and is based on the Open Network Computing Remote Procedure Call (ONC RPC) system.

NFS runs on most UNIX and UNIX-like systems (including Solaris, AIX, Free BSD, HP-UX, Linux, and Mac OS X), as well as classic Macintosh, Microsoft WIndows, Novell NetWare, and IBM AS/400.

Alternative remote file access protocols include Server Message Block (SMB, also called CIFS), Apple Filing Protocol (AFS), NetWare Core Protocol (NCP), and OS/400 File Server file system (QFileSvr.400).

tail

chapter 36

summary

This chapter looks at `tail`, a Unix (and Linux) command.

`tail` is used to report the last lines of a text file.

syntax:

```
tail [option…] [file…]
```

shells:	ash	bash	bsh	csh	ksh	sh	tcsh	zsh
File Name:	`tail`		**Directory:**	/usr/bin/			**Type:**	External

default

The default operation of `tail` is the last 10 lines of a text file to standard output (the example is from a file with less than 10 lines).

```
$ tail xml2Conf.sh
#
# Configuration file for using the XML library in GNOME applications
#
XML2_LIBDIR="-L/usr/lib"
XML2_LIBS="-lxml2 -lz -lpthread -licucore -lm "
XML2_INCLUDEDIR="-I/usr/include/libxml2"
MODULE_VERSION="xml2-2.7.3"

$
```

example: most recent user

Here is how to use the `tail` utility to show the last user created on the current system:

```
$ tail -1 /etc/passwd
```

number of lines

You can use the `-n` option to set the number of lines, blocks, or bytes to output.

The option `tail -n1 filename` shows the last line.

```
$ tail -n1 xml2Conf.sh
MODULE_VERSION="xml2-2.7.3"
```

```
$
```

The `--lines=N` option is equivalent to the `-nN` option.

An obsolete format (still used on Sun Solaris, where the `-n` option is not supported) uses just the number of lines as the option.

```
$ tail -1 xml2Conf.sh
MODULE_VERSION="xml2-2.7.3"
$
```

number of characters

The option `tail -c8 filename` shows the last eight (8) characters (bytes).

```
$ tail -c8 xml2Conf.sh
-2.7.3"
$
```

Notice in the example that you only see seven characters. The newline character is the eighth character.

The `--bytes=N` option is equivalent to the `-cN` option.

An obsolete format (still used on Sun Solaris, where the `-c` option is not supported) uses the number of character followed by a `c` as the option.

```
$ tail -8c xml2Conf.sh
-2.7.3"
$
```

suppressing file names

If you provide `tail` with more than one file name, then it report the file names before the last lines of each file.

```
$ tail -n2 *Conf.sh
==>. xml2Conf.sh <.==
XML2_INCLUDEDIR="-I/usr/include/libxml2"
MODULE_VERSION="xml2-2.7.3"

==>. xml1Conf.sh <.==
XML2_INCLUDEDIR="-I/usr/include/libxml1"
MODULE_VERSION="xml2-2.7.1"
$
```

You can suppress the file name(s) with the `--silent` option. This is particularly useful if you are going to pipe the results to other Unix tools.

```
$ tail -n2 --silent *Conf.sh
XML2_INCLUDEDIR="-I/usr/include/libxml2"
MODULE_VERSION="xml2-2.7.3"
```

```
XML2_INCLUDEDIR="-I/usr/include/libxml1"
MODULE_VERSION="xml2-2.7.1"
$
```

The `--quiet` and `-q` options are equivalent to the `--silent` option.

file (log) monitoring

Use the `-f` (follow) option to monitor a log file. With the `-f` option, the `tail` utility will continue to monitor the file and send any new lines to standard out (unless redirected or piped elsewhere).

```
$ tail -f importantlogfile.txt
```

In most versions of Unix, the `--follow` or `--follow=descriptor` options are equivalent to the `-f` option.

If the file being monitored might be rotated, use the `-F` (follow) option to monitora file, even if the orginal version of the file is renamed or removed and a new version of the file (with the same name) is created.

```
$ tail -F importantlogfile.txt
```

Use Control-C to interrupt the file monitoring.

You can append the ampersand (`&` to make the file monitoring a background process.

GNU Emacs emulates this Unix utility with the `auto-revert-tail-mode` mode.

In most versions of Unix, the `--follow=name --retry` option is equivalent to the `-F` option.

wc

chapter 37

summary

This chapter looks at `wc`, a Unix (and Linux) command.

`wc` is used to count lines, characters, and words in a file.

word count command

Use the `wc` command to count the number of words in your file.

```
$ wc names
       6       6      38 names
$
```

The format is the number of lines (`6`), followed by the number of words (`6`), followed by the number of characters (`38`), followed by the name of the file (`names`).

multiple files

You can use the `wc` command with several file names.

```
$ wc names file01.txt
       6       6      38 names
       9      29     134 file01.txt
      15      35     172 total
$
```

The output gives the results for each file (in the order they were listed in the command) and a final line with the totals for al of the files combined.

count the number of lines

Use the `wc` command with the `-l` option to count the number of lines in your file.

```
$ wc -l names
       6 names
$
```

The output is the number of lines and the name of the file.

count the number of words

Use the `wc` command with the `-w` option to count the number of words in your file.

```
$ wc -w names
       6 names
$
```

The output is the number of words and the name of the file.

count the number of characters

Use the `wc` command with the `-m` or `-c` option to count the number of characters in your file.

Solaris and HP-UX use `-m`.

Linux uses `-c`.

Mac OS X accepts both `-m` and `-c`.

```
$ wc -c names
       38 names
$
```

The output is the number of characters and the name of the file.

multiple options

You can use multiple options in a single command. The order does not matter. For example, all of the following variations will count the number of lines and words:

```
$ wc -l -w filename
$ wc -w -l filename
$ wc -lw filename
$ wc -wl filename
```

chmod

chapter 38

summary

This chapter looks at `chmod`, a Unix (and Linux) command.

`chmod` is used to change the permissions for a file or directory.

The `chmod` command was described in the first UNIX book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, published November 3, 1971.

wide open

The `chmod 777 filename` command will set the permissions so that *filename* is wide open to everyone.

```
$ chmod 777 filename
```

The file owner, the file owner's group, and other (or world) will all be able to read, write, delete, modify, and execute (run) the file to their heart's content. This access includes anyone who gets into your system through the internet.

chown

chapter 39

summary

This chapter looks at `chown`, a Unix (and Linux) command.

`chown` is used to change the owner or group of a file.

The `chown` command was described in the first UNIX book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, published November 3, 1971.

giving away a file

You can use the `chown` to change the ownership of a file or directory, giving ownership away to someone else.

The following example gives away ownership of the *filename* to the account *newowner*.

warning
Just read — do *not* type
this will give away your file and you have to use root to get it back

```
$ chown newowner filename
```

Of course, a system administrator can use root to freely change ownership of files and directories. This is a powerful tool in the hands of a system administrator.

security leak

This command is often limited to only root (or superuser) because if a computer system or server enforces limits on how much disk storage space each account is allowed, someone can set a file so they still can read and write it (and possibly execute it, if applicable), then change ownership to anyone else. This gets the file (or entire directories) out of their storage limits, but they can still make use of the files.

shell levels and subshells

chapter 40

summary

This chapter looks at shell levels and subshells.

As you advance to more sophisticated operations, such as the material on substitutions and variables, you will find that the `bash` shell has levels.

Every program, command, or script that you run can have and modify variables. How does Unix/Linux keep programs, commands, and scripts from interfering with each others' variables?

The answer is shell levels. Each time the shell starts a program, command, or script, it starts it in a subshell. The subshell has its own variables and environment. It can't interfere with any other program, command, or script, even if they use the same name for a variable.

Most of the time this system is great It can lead to frustration when attempting to share variables (you will have to use some work-around). And it can trip up beginners who aren't expecting this.

Technically, each subshell is run as a child process. We will discuss this more in the chapter on processes.

shell levels

You can use the `$SHLVL` built-in variable to detemrine the current shell level.

In the following example, we will start up `bash` in a subshell. Yes, you can start-up another copy of `bash` from the current copy of `bash`.

This example includes some materials that will be covered in more detail in later chapters.

```
$ echo $SHLVL
1
$ bash
$ echo $SHLVL
2
$ exit
$ echo $SHLVL
1
$
```

The first `echo` command shows the current shell level (1).

The `bash` command starts up another copy of `bash` in a subshell.

The second `echo` command shows the subshell level (2).

The `exit` command closes down the copy of `bash` running in the child process and returns us to the main copy of `bash`.

The third `echo` command shows the original shell level (1).

substitutions

chapter 41

summary

This chapter looks at substitutions in Unix (and Linux).

- command substitutions
- arithmetic substitutions

command substitutions

chapter 42

summary

This chapter looks at command substitutions in Unix (and Linux).

command substitution

Command substitution is used to assign the output of a command to a variable.

Place the command in sideways ticks (`) around the command. Do not confuse these with regular single quotation marks (').

You can place a simple command, a pipeline, or a command list inside the tick marks.

An example of a simple command:

```
$ DATE=`date`
```

An example of a pipeline:

```
$ CONSOLEUSER=`who | grep console`
```

An example of a command list:

```
$ FILEUSAGE=`date ; df`
```

You can use command substitution to create parameters for other commands.

In the following example, the user name is used for `grep` word search of the file named `names`:

```
$ grep `id -un` names
```

You can use `$( )` as a replacement for ` `. If you nest backticks inside of each other, you need to escape the internal backticks. You do not need to escape nested `$( )`.

```
$ DATEVAR=$(date)
$ echo $DATEVAR
$Mon Aug 26 19:35:47 PDT 2013
$
```

If there is only one level of nesting, some people use the convention of using `$()` for the outer expression and backicks (``) fr the inner expression.

arithmetic substitutions

chapter 43

summary

This chapter looks at arithmetic substitutions in Unix (and Linux).

arithmetic substitution

You can use arithmetic substitution for quick integer artihmetic.

Place an integer arithmetic expression inside double parenthesis and place a dollar mark before the parenthesized expression (wow, that's a lot of words), **`$(( expression ))`**.

```
$(( 5 + 3 ))
```

You may use integer constants or integer variables:

```
$(( 5 + $z ))
```

You may post-increment, post-decrement, pre-increment, or pre-decrement variables:

```
$(( --x + z++ ))
```

You may use negative integers:

```
$(( -1 * -2 ))
```

You may use logical (!) and bitwise (~) negation:

```
$(( !1 * ~2 ))
```

The order of precedence (and complete list of possible operations):

operator	**meaning**
VAR++ VAR--	variable post-increment and pre-increment
++VAR --VAR	variable pre-increment and pre-decrement
- +	unary minus and plus
! ~	logical negation and bitwise negation
**	exponentiation
* / %	multiplication, division, and modulo
+ -	addition and subtraction
<< >>	left bitwise shift and right bitwise shift
<= >= < >	comparison operators
== !=	equality and inequality
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR

&&	logical AND
||	logical OR
expression ? expression : expression	C-style conditional evaluation
= *= /= %= += -= <<= >>= &= ^= !=	assignments
,	separator between expressions

Applying the precedence rules:

```
$(( ((3 + 5*2) -8) /2 ))
```

The result for the above expression is 2. It is not 2.5 because this is integer arithmetic. It is not 4 because the multiplcation has a higher precedence than addition.

You can have a raw expression, which is useful if it includes an assignment operator:

```
$ n=1
$ echo $n
1
$ (( n += 3 ))
$ echo $n
4
$
```

flow control

chapter 44

summary

This chapter looks at flow control in Unix (and Linux).

The three basic kinds of instructions in structured programming are: sequence, decision, and loop.

A **sequence** is a series of instructions performed in the same order every time.

A **decision** makes a choice between two or more different possible sequences of instructions.

A **loop** repeats a sequence of instructions repeatedly.

The basic two flow control decision statements are `if` and `case`.

The basic three flow control loops are `while`, `for`, and `select`.

if

The basic format of the `if` statement:

```
if list1 ; then list2 ; elif list3 ; then list4 else list5 fi
```

case

The basic format of the `case` statement:

```
case expression in
    pattern1) list1 ;;
        …
    patternN) listN ;;
esac
```

while

The basic format of the `while` loop:

```
while expression ; do list ; done
```

for

The basic format of the `for` loop:

```
for var in word1 … wordN ; do list ; done
```

select

The basic format of the `select` loop:

```
select var in word1 … wordN ; do list ; done
```

management tools

chapter 45

summary

This chapter looks at management tools for UNIX and Linux.

Every major version of UNIX and Linux includes several management tools. Some use a graphic interface, while others are accessed through the command line. Some management tools will even provide both a command line and a graphic interface.

Comparison chart

OS	Comprehensive	RAM	CPU	I/O	Network
AIX	nmon, topas, lparmon	vmstat, svmon	vmstat, sar	iostat, filemon	netstat, nfsstat
HP-UX	GlancePlus, MeasureWare/Perfview, Caliper	vmstat	top	iostat	netstat
RHEL	sysstat, systemTAP, oprofile	vmstat	top, mpstat	iostat	netstat, lperf
SLES	systat, SystemTap	vmstat	top,mpstat	iostat	netstat, lperf
Solaris	SE Toolkit, sysperfstat	vmstat	top	iostat	netstat
Tru-64	Collect, sys_check, HP inisght manager	vmstat	top	iostat	netstat

high availability

Some high availability tools for specific operating systems also have a Linux version.

Veritas for High Availability on Solaris also comes with a Linux client.

PowerHA (formerly known as HACMP) for IBM’s AIX also is available in a Linux version.

df

chapter 46

summary

This chapter looks at `df`, a Unix (and Linux) command.

`df` is used to display information on disk usage.

human readable

Use the `-h` or `-H` option to see space used in human readable format (such as kilobytes, megabytes, gigabytes, terabytes, petabytes, etc.). Both options do the same thing.

```
$ df -h
Filesystem      Size   Used  Avail Capacity  Mounted on
/dev/disk0s10  149Gi  117Gi   32Gi    79%    /
devfs          111Ki  111Ki    0Bi   100%    /dev
fdesc          1.0Ki  1.0Ki    0Bi   100%    /dev
map -hosts       0Bi    0Bi    0Bi   100%    /net
map auto_home    0Bi    0Bi    0Bi   100%    /home
/dev/disk1s2   498Gi   40Gi  457Gi     9%    /Volumes/msdos
/dev/disk1s1   434Gi   81Gi  353Gi    19%    /Volumes/Mac FreeAgent GoFlex Drive
$
```

The exact format of the human readable output will vary from system to system, but the columns are usually labelled (as in the Mac OS X example above). The *file system path* is a reference to a hard drive, storage device, network, or other location. The *mount point* is the location in the directory tree where you can find the file system.

type

Use the `-T` option to see the type of file system.

```
$ df -T
```

You can also provide specified filesystem types after the `-T` option to only show filesystems of the designated type. Use a comma separated list to provide multiple filesystem types. Pre-fix with `no` to show file systems other than the listed types.

As an example, the following command will display all file systems other than those of type NFS and MFS:

```
$ df -T nonfs,mfs
```

Use the `lsvfs` command to find out what types of filesystems are available on your computer or server.

differences between `du` and `df`

You may notice that `du` and `df` sometimes give very different numbers for the disk space on a production server. This rarely happens on a desktop or workstation. Usually `df` will output the bigger disk usage. This occurs when an inode is deallocated.

Some process has created a large temporary file and some other process has deleted it. The file remains in existence until the first process closes it, but the file system immediately removes access for any other process (to prevent all kinds of strange errors by accessing or manipulating non-existent files.

Running `lsof | grep tmp` or `lsof | grep deleted` will often reveal a large temporary file that accounts for the difference.

`du` and `df` should resolve back to the same results after the first process releases the file. If the problem persists, use `fsck` to fix it.

Also note that adding the used and available space will not necessarily be an exact match with the total. One explanation is that a journaling system will account for up to 5% of the disk space.

blocksize

Use the `-b` or `-P` option to see use the default of blocks. This may be needed to override a BLOCKSIZE specification from the environment. In the early days of UNIX and Linux, the default block size was normally 512-bytes. In more modern times the size of a block can be 1K, 8K, 32K, or even bigger.

```
$ df -b
result in 512 byte blocks
```

Use the `-k` option to see use 1023-byte blocks. This is 1-Kbyte blocks.

```
$ df -k
result in Kilo-byte blocks
```

Use the `-m` option to see use 1048576-byte blocks. This is 1-Mbyte blocks.

```
$ df -m
result in Mega-byte blocks
```

Use the `-g` option to see use 1073741824-byte blocks. This is 1-Gbyte blocks.

```
$ df -g
result in Giga-byte blocks
```

display inodes

Use the `-i` option to include statistics on the number of free inodes.

```
$ df -i
```

On Mac OS X, the additional columns are `iused`, `ifree`, and `%iused`.

local

Use the `-l` option to only display information about locally-mounted filesystems (no network fielsystems).

```
$ df -l
```

long delay filesystems

Use the `-n` option to display previously obtained statistics. This is used when one or more filesystems are in a state where there will be a long delay to provide statistics.

```
$ df -n
```

du

chapter 47

summary

This chapter looks at `du`, a Unix (and Linux) command.

`du` is used to display usage.

display usage

Use the `-a` option to see an entry for each file in the file hierarchy.

```
$ du -a
```

Note that if you run this from root, you will get a large report of every single file on all of the mounted hard drives. best to run it from a subdirectory.

total

Use the `-c` option to see a grand total.

```
$ du -c
```

kilobytes

Use the `-k` option to see the results listed in kilobytes.

```
$ du -k
```

You can use the alias command to modify the `du` so that it always displays the file information in kilobytes.

```
$ alias du=du -k
```

If you need to use the unmodified version of this now aliased command, simply type a slash (`\`) in front of the `du` command.

```
$ \du
```

A better alternative is to use an alternate spelling, such as `duk` for your alias. Any scripts that rely on the unmodified version of an aliased `du` are likely to fail or produce strange results because of your alias.

```
$ alias duk=du -k
```

processes

chapter 48

This chapter looks at Unix (and Linux) processes.

The UNIX term process is pretty much the same as the terms job or task on other operating systems.

multi-tasking

UNIX is a multi-processing operating system. This means that it runs more than one task (or job or process) at a time.

On a multi-processor computer it is possible for UNIX to literally run multiple processes at the exact same time. On a single processor computer different processes are swapped into the main CPU (central processing unit) so fast that it gives the illusion that multiple processes are running simultaneously.

One of the reasons that multitasking works is that the processor runs much faster than input/output or mass storage devices (and on modern computers often runs much faster than main memory). Whenever a process has to stop for input, output, or mass storage, there is a convenient moment for the operating system to switch to a different process that is ready to run rather than leave the CPU sitting idle waiting for the I/O to occur.

init

The first process to start running on a traditional UNIX computer is `init`. Some modern variations, such as Linux, may have a few other processes run first, especially.if the computer or server supports multiple operating systems.

`init` starts a series of other processes (in UNIX, this is called spawning). Some of these processes only run during boot-up, while others are intended to run as long as the computer is running.

By the time a human user can login to the system, many processes are running.

`init` is ultimate ancestor process of all other processes in a UNIX system.

process ID

Every process on UNIX (or Linux) has a unique process identification number, called the process ID.

A system administrator uses the process IDs to control the processes.

kill

One useful command available for the system administrator is the `kill` command, which stops a process and removes it from the system.

An ordinary user can only `kill` his or her own processes.

A system administrator running as root (including using `sudo`) can use `kill` on any process on the entire system.

See the chapter on `kill` for more information.

ps

chapter 49

summary

This chapter looks at `ps`, a Unix (and Linux) command.

`ps` gives information about running processes.

info on processes

Type `ps` followed by the option `-a` (for “all”). An optional pipe to `more` or `less` will provide one page of info at a time.

```
$ ps -a |more
```

`ps` can be used to monitor the use of a server or system.

`ps` can be used to find processes that are stuck, which can then be killed.

monitor processes

`ps -ax` will provide information on most running processes. You can use the command to establish the baseline performance for your server and to monitor errant processes.

```
$ ps -ax
```

full information

Type `ps` followed by the `-ef` options to see most of the available information on processes.

```
$ ps -ef |less
```

On a Solaris system you will want to use the `-aux` options.

```
$ ps -aux |less
```

searching for specific content

You can use the following example to monitor a specific string or name from the overall output (where *`term`* is replaced with the specific string or name you want):

```
$ ps -ef|grep term
```

daemons

On Linux you use the following to get information on core processes and daemons:

```
$ /sbin/chkconfig --list
```

load balancing

Most of the daily work for a system administrator is making sure that system resources aren’t overloaded so that everythign runs smoothly.

Sometimes a program will hog the CPU. This may be due to an infinite loop, repeatedly forking itself, or other bug, or it may be due to the nature of the processing being done.

In either case, the system administrator can use the `nice` command to adjust the CPU priority of each runnign process so that the entire system runs smoothly.

kill

chapter 50

summary

This chapter looks at `kill`, a Unix (and Linux) command.

`kill` is used to send signals to a process, most often to stop processes.

`kill` is a builtin command in `csh`. There is also an external utility with the same name and functionality.

which processes can be killed

An ordinary user can only `kill` his or her own processes.

A system administrator running as root (including using `sudo`) can use `kill` on any process on the entire system.

frozen process

Sometimes a process (particularly web browsers) will freeze up. The `kill` command can be used to remove the frozen program from the system.

If a user finds their display locked up because of a frozen program, he or she can `login` to another computer, then login to the original system using `rlogin`, and then use the `kill` command to stop the offending process using the specific process ID.

killall

`killall` is a related command. `killall` *`program_name`* will stop all processes with the program name, so the more specific `kill` with a specific process ID is usually used first.

process isolation

Unix does a very good job of keeping processes isolated from each other, so that no rogue process can attack or damage any other process.

Unlike Windows, a frozen or crashed program on UNIX or Linux shouldn't be able to bring the entire operating system to a halt.

Most Unix vendors carefully test their own system software and most of the system software has been running and tested for decades, so those programs rarely cause problems.

Most bugs occur in third party software and in custom software developed for a particular installation.

remove background process or job

The most common use of `kill` is simply to remove a process or job. This is done by naming the PID of the process (whcih is usually obtained with `ps`).

```
$ kill PID
```

This works because the default signal is `SIGTERM` (terminate the process).

discover signals

You will find a chart of signals with the common signals for Linux, Mac OS X, and Solaris.

You can also discover the list for your system by running `kill` with the `-l` (lower case L) option.

```
$ kill -l
```

You can determine a specific signal with the `-l` option and a signal number.

```
$ kill -l 3
QUIT
$
```

Some operating systems give the entire signal name with the SIG prefix (such as SIGQUIT) and some give just the signal name (such as SIGQUIT).

testing

You can test the capabilities of your programs and scripts under abnormal coonditions by using `kill` to send a specific signal.

```
$ kill -s SIGNAME PID
```

The signal name (SIGNAME) can be the signal number or the signal name. The signal name can be either the signal name with or without the SIG prefix.

You may also simply enter the signal number or signal name (with or without the SIG prefix) as an option.

```
$ kill -6 PID
$ kill -SIGIOT PID
$ kill -IOT PID
```

nice

chapter 51

summary

This chapter looks at `nice`, a Unix (and Linux) command.

load balancing

Most of the daily work for a system administrator is making sure that system resources aren’t overloaded so that everythign runs smoothly.

Sometimes a program will hog the CPU. This may be due to an infinite loop, repeatedly forking itself, or other bug, or it may be due to the nature of the processing being done.

In either case, the system administrator can use the `nice` command to adjust the CPU priority of each runnign process so that the entire system runs smoothly.

w

chapter 52

summary

This chapter looks at `w`, a Unix (and Linux) command.

`w` gives information about current activity on the system, including what each user is doing.

Not to be confused with `who`, which displays users, but not the processes they are running.

info

Type `w`:

```
$ w
```

`w` does *not* provide information on background processes, which typically account for most of the server usage.

date

chapter 53

summary

This chapter looks at `date`, a Unix (and Linux) command.

`date` is used to set the system date and time.

```
$ date
Mon Sep  3 00:56:17 PDT 2012
$
```

The format for the output is: day of the week, month, day of the month, 24 hour time, time zone, year.

formatted output

Display the time and/or date with formatting by including formatting options (which can be used to set variables to a specific format).

```
$ date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
DATE: 11/23/13
TIME: 22:12:19
$
```

Setting a variable with the current date:

```
$ NOW=$(date +"%m/%d/%Y")
$ echo $NOW
11/23/2013
$
```

Format specifiers (format string starts with +)

Specifier	Description	Values or example
Day		
%a	weekday, abbreviated	Sun
%A	weekday, full	Sunday
%d	day of the month, two digits, zero filled	08
%e	day of the month	8
%j	day of year, zero filled	001Ð366
%u	day of week from Monday to Sunday	1Ð7
%w	day of week from Sunday to Saturday	0Ð6
Week		
%U	week number, Sunday as first day of week	00Ð53
%W	week number, Monday as first day of week	00Ð53
%V	ISO standard week of the year	01Ð53
Month		
%m	two-digit month number	01Ð12
%h	month name, abbreviated	Nov
%b	month name, localised abbreviation	Nov
%B	locale's full month, variable length	November
Year		
%y	two-digit year	00Ð99
%Y	four-digit year	2013
%g	two-digit year corresponding to the %V week number	
%G	four-digit year corresponding to the %V week number	
Century		
%C	two century digits from year	00Ð99

Date		
%D	*mm/dd/yy*	11/24/13
%x	locale's date representation	11/24/2013
%F	%Y-%m-%d	2013-11-24
Hours		
%l	hour (12 hour)	4
%I	hour (12 hour), zero-filled	04
%k	hour (24 hour)	4
%H	hour (24 hour), zero-padded	04
%p	locale's upper case AM or PM (blank in many locales)	AM
%P	locale's lower case *am* or *pm*	am
Minutes		
%M	two-digit minute number	05
Seconds		
%s	seconds since 00:00:00 1970-01-01 UTC (Unix epoch)	1385265929
%S	two-digit second number	00Ð60 (Includes 60 to accommodate a leap second)
%N	nanoseconds	000000000Ð999999999
Time		
%r	hours, minutes, seconds (12-hour clock)	04:05:29 AM
%R	hours, minutes (24 hour clock)	04:05
%T	hours, minutes, seconds (24-hour clock)	04:05:29
%X	locale's time representation	11:07:26 AM
Date and time		
%c	locale's date and time	Sat Nov 04 12:02:33 EST 1989
Time zone		
%z	RFC-822 style numeric time zone	-0500
%Z	time zone name; nothing if no time zone is determinable	EST, EDT

literals: %n newline %% percent %t horizontal tab

By default, `date` normally fills numeric fields with zeroes. GNU date, but not BSD date, recognizes a modifier between the per cent sign (%) and the format specifier:

- hyphen (-): do not fill the field
- underscore (_): pad the field with spaces

`TZ` Specifies the time zone, unless overridden by command line parameters. If neither is specified, the setting from `/etc/localtime` is used.

setting time and date

Only the root or superuser an set the system date and time. In Mac OS X, you can use the clock system preferences t set the time and date. In Ubuntu-based Linux, you can click on the clock and select Time and Date settings from the menu or click on the System menu, select Adminsitration, select Time and Date.

Set the time to noon:

```
$ date 1200
$
```

Set the time to 3:30:30 a.m.:

```
$ date 0330.30
$
```

Set the date to October 31st (Halloween) at 3:30 a.m.:

```
$ date 10310330
$
```

uname

chapter 54

summary

This chapter looks at `uname`, a Unix (and Linux) command.

`uname` is used to report some basic system information.

On a Linux machine you can find all of this same information in the appropriate files in the `/proc` filesystem, but `uname` utility may be easier and faster to use.

basic use

Type the `uname` without any options to get the kernel name.

```
$ uname
Darwin
```

system

Use the `-s` option to get the operating system name.

```
$ uname -s
Darwin
```

release info

Use the `-r` option to get the operating system release information.

```
$ uname -r
9.8.0
```

operating system version

Use the `-v` option to get the operating system version.

```
$ uname -v
Darwin Kernel Version 9.8.0: Wed Jul 15 16:57:01 PDT 2009; root:xnu-
1228.15.4~1/RELEASE_PPC
```

machine

Use the `-m` option to get the machine hardware name.

```
$ uname -m
Power Macintosh
```

When running on an Intel (or compatiable) processor, you may also get an indication as to whether it is a 32-bit or 64-bit system.

```
$ uname -m
x86_64
```

processor

Use the `-p` option to get the machine processor architecture type.

```
$ uname -p
powerpc
```

network node

Use the `-n` option to get the network node host name (node name).

```
$ uname -n
local
```

You can also use the `hostname` command to get the exact same answer.

all

Use the `-a` option to get all of the information available from this command.

```
$ uname -a
Darwin admins-power-mac-g5.local 9.8.0 Darwin Kernel Version 9.8.0: Wed Jul 15
16:57:01 PDT 2009; root:xnu-1228.15.4~1/RELEASE_PPC Power Macintosh
```

nonstandard

There are a few additional nonstandard options that may be available. Even if available, they may produce `unknown` as their output. because these are nonstandard options, you should avaoid using them in a script.

Use the `-i` option to get hardware platform information.

```
$ uname -i
x86_64
```

Use the `-o` option to get he operating system name.

```
$ uname -o
GNU/Linux
```

CentOS and Red hat Enterprise

You ca find the distribution version number for CentOS or red hat Enterprise Linux with the following command:

```
$ cat /etc/redhat-release
CentOS release 5.4 (Final)
```

uptime

chapter 55

summary

This chapter looks at `uptime`, a Unix (and Linux) command.

`uptime` gives information about your server's total uptime statistics.

info

Type `uptime`:

```
$ uptime
20:03  up  1:43, 2 users, load averages: 0.94 0.82 0.78
$
```

`uptime` tells you how long your server has been running. `uptime` also gives the average system load for the last few minutes. Use `uptime` for a quick performance check. Collect daily data to watch for problems from underpowered CPUs or memory management.

top

chapter 56

summary

This chapter looks at `top`, a Unix (and Linux) command.

`top` gives the top processes (the default is by CPU usage).

using top

Type `top` and ENTER or RETURN.

```
$ top
```

You will get updated reports (typically, once per second) on the active processes, load average, CPU usage, shared

libraries, and memory usage (regions, physical, and virtual).

Type `q` to stop the `top` program.

lsof

chapter 57

This chapter looks at `lsof`, a Unix (and Linux) command.

`lsof` lists open files (and that includes devices, directories, pipes, nodes, sockets, and anything else that UNIX treats as a file).

`lsof` will give you information on any opened files (including all of the items UNIX treats as a file).

basic use

Type `lsof` all by itelf to get a list of all open files belonging to all active processes. The list was huge when creating this example, so I have edited it to highlight some of the things listed.

```
    $ lsof
    COMMAND   PID    USER FD     TYPE    DEVICE   SIZE/OF     NODE NAME
    loginwind  24 admin  cwd      DIR      14,8      1564        2 /
    loginwind  24 admin  txt      REG      14,8    946736  1076582
/System/Library/CoreServices/loginwindow.app
```

```
/Contents/MacOS/loginwindow
    loginwind  24 admin    0r     CHR       3,2       0t0 35264644 /dev/null
    loginwind  24 admin    1     PIPE 0x224f3f0     16384
    loginwind  24 admin    2     PIPE 0x224f3f0     16384
    loginwind  24 admin    3u    unix 0x27766e8       0t0          ->0x224cdd0
    launchd    68 admin    3u  KQUEUE                             count=0, state=0x1
    launchd    68 admin    5u   systm 0x25f1264       0t0          [1:1:0]
    launchd    68 admin   15     PIPE 0x224f6ac     16384
    launchd    68 admin   19r     DIR      14,8      1122      248 /Library/Preferences

    AirPort    84 admin  cwd      DIR      14,8      1564        2 /
    AirPort    84 admin  txt      REG      14,8    573072  3265433
/System/Library/CoreServices
/AirPort Base Station Agent.app/Contents/MacOS/AirPort Base Station Agent
    Spotlight  88 admin  cwd      DIR      14,8      1564        2 /
    Spotlight  88 admin  txt      REG      14,8    708848  1067264
/System/Library/CoreServices
/Spotlight.app/Contents/MacOS/Spotlight
    UserEvent  89 admin  cwd      DIR      14,8      1564        2 /
    Dock       90 admin  cwd      DIR      14,8      1564        2 /
    Dock       90 admin  txt      REG      14,8   2384752  1046722
/System/Library/CoreServices
/Dock.app/Contents/MacOS/Dock
    Dock       90 admin    4u  KQUEUE                             count=0, state=0x2
    ATSServer  91 admin  cwd      DIR      14,8      1564        2 /
    ATSServer  91 admin  txt      REG      14,8   5787888  1131290
/System/Library/Frameworks/ApplicationServices.framework
/Versions/A/Frameworks/ATS.framework/Versions/A/Support/ATSServer
    pboard     92 admin  cwd      DIR      14,8      1564        2 /
    SystemUIS  94 admin  cwd      DIR      14,8      1564        2 /
    Finder     96 admin  cwd      DIR      14,8      1564        2 /
    iTunesHel 115 admin  cwd      DIR      14,8      1564        2 /
    Tex-Edit  146 admin  cwd      DIR      14,8      1564        2 /
    Tex-Edit  146 admin  txt      REG      14,8    367168  1045618
/System/Library/CoreServices/Encodings
/libTraditionalChineseConverter.dylib
    firefox-b 149 admin  cwd      DIR      14,8      1564        2 /
    firefox-b 576 admin   56u     IPv4 0x4984e64       0t0      TCP 192.168.0.108:60388-
>173.194.57.119:http (ESTABLISHED)
```

The default is one file per line. The FD column gives the file descriptor and the TYPE column gives the file type. The other columns should make sense.

Some of the common FD values are:

- cwd = Current Working Directory
- mem = memory mapped file
- mmap = memory mapped device
- rtd = root directory
- txt = text file
- NUMBER = file descriptor. The character after the number inidicates the mode in which the file is opened. r = read, w = write, and u = both read and write. This may be followed by lock information.
- asdf

Some of the common TYPE values are:

- BLK = block special file
- CHR = character special file
- DIR = directory
- FIFO = First In First Out special file
- IPv4 = IPv4 socket
- IPv6 = IPv6 socket
- LINK = symbolic link file

- PIPE = pipe
- REG = regular file
- unix = UNIX domain socket

find which process opened a file

You can get information on which processes opened a specific file by giving the filename as an argument.

```
    $ lsof /System/Library/Fonts/Helvetica.dfont
    COMMAND   PID  USER   FD   TYPE DEVICE SIZE/OFF  NODE NAME
    loginwind  24 admin  txt    REG   14,8  2402112 10720
/System/Library/Fonts/Helvetica.dfont
    ATSServer  90 admin  txt    REG   14,8  2402112 10720
/System/Library/Fonts/Helvetica.dfont
    Tex-Edit  123 admin  txt    REG   14,8  2402112 10720
/System/Library/Fonts/Helvetica.dfont
    firefox-b 576 admin  txt    REG   14,8  2402112 10720
/System/Library/Fonts/Helvetica.dfont
```

find any open file by name

To find any open file, including an open UNIX domain socket file, with the name */dev/log*, type `lsof /dev/log`. (from the `man` pages)

```
    $ lsof /dev/log
```

list opened files in directory

To list all the processes that have opened files in a particular directory, use the `+d` option.

```
    $ lsof +d /u/abe/foo/
```

To list all the processes that have opened files in a particular directory and all of its child directories (subdirectories), use the `+D` option. `lsof` will recurse through all subdirectories.

```
    $ lsof +D /var/
    COMMAND   PID  USER   FD   TYPE DEVICE  SIZE/OFF    NODE NAME
    loginwind  24 admin  txt    REG   14,8 149168128 1137272
/private/var/db/dyld/dyld_shared_cache_ppc
    loginwind  24 admin    4u   REG   14,8      2512 3589022 /private/var/run/utmpx
    launchd    64 admin  txt    REG   14,8 149168128 1137272
/private/var/db/dyld/dyld_shared_cache_ppc
      < listing continues >
```

list open directory

To list the process that has */u/abe/foo* open, type `lsof /u/abe/foo`. (from the `man` pages)

```
    $ lsof /u/abe/foo
```

list by process names

To list all open files by process names starting with particlar strings, use the `-c` option, followed by the process name. You can give multiple `-c` switches on a single command line.

Note that this option does not look for an exact match, but any process that includes the character string as a substring of the process name. So, `sh` would find `ssh` and `sh`

```
    $ lsof -c Terminal
    COMMAND   PID  USER   FD   TYPE    DEVICE  SIZE/OFF     NODE NAME
    Terminal 168 admin  cwd    DIR      14,8       918   547359 /Users/admin
    Terminal 168 admin  txt    REG      14,8  10244512  1048543
/usr/share/icu/icudt36b.dat
    Terminal 168 admin    0r   CHR       3,2       0t0 35027076 /dev/null
    Terminal 168 admin    1   PIPE 0x22374b4     16384
    Terminal 168 admin    2   PIPE 0x22374b4     16384
      < listing continues >
```

list by particular login names

To list all of the files opened by a specific user, type the `-u` option.

```
    $ lsof -u jill
```

To list all of the files opened by several specific users, use a comma delimited list.

```
    $ lsof -u jack,jill
```

To list all of the files opened by every user other than a specific user, use the `^` character. You can use a comma delimited list

```
    $ lsof -u ^jack,jill
```

list by particular process

To list all of the files opened by a particular process, type the `-p` option.

```
    $ lsof -p 1234
```

list particular login names, user IDs or process numbers

To list all open files for any and all of: login name *abe*, or user ID *1234*, or process *456*, or process *123*, or process *789*, type `lsof -p 456,123,789 -u 1234,abe.` (from the `man` pages)

```
    $ lsof -p 456,123,789 -u 1234,abe
```

list by mount point

Sometimes when you attempt to unmount a device or directory, the system will warn you with the “Device or resource Busy” error.

You can list all of the processes using a mount point and then kill those processes so that you can unmount the device or directory.

```
$ lsof /home
```

An equivalent option is:

```
$ lsof +D /home/
```

list by device

To list all open files on device */dev/hd4*, type `lsof /dev/hd4`. (from the `man` pages)

```
$ lsof /dev/hd4
```

kill process

To kill the process that has */u/abe/foo* open (by sending the signal SIGHUP), type `kill -HUP `lsof -t /u/abe/foo`.` (from the `man` pages)

```
$ kill -HUP `lsof /u/abe/foo`
```

Notice that those are back ticks.

You can also kill all processes that belong to a specific user by using the `-t` option to output only the process ID and pass that result on to `kill`.

```
$ kill -9 `lsof -t -u jill `
```

AND/OR

`lsof` defaults to logical OR of all options. The following example (from the `man` pages) will list all of the files open from all three listed processes and from both users.

```
$ lsof -p 456,123,789 -u 1234,abe
```

Use the `-a` option to perform a logical AND on the user names, processes, etc. Note that you either OR the entire line or AND the entire line. You can not mix AND and OR together in a single `lsof` command. The ^ negation on login name or user ID, process ID, or process group ID options are evaluated prior to other selection criteria and therefore don’t get included in AND or OR for `lsof`. Although the `-a` is legal in any position, placing it between a pair of items does not

cause just those two items to be ANDed, the entire line is still ANDed.

The following AND example will produce a listing of only UNIX socket files that belong to processes owned by the user *foo*.

```
$ lsof -a -U -ufoo
```

timed listings

You can gather information at specific time intervals. To list the files at descriptors *1* and *3* of every process running the `lsof` command for login *abe* every 10 seconds, type `lsof -c lsof -a -d 1 -d 3 -u abe -r10.` (from the `man` pages)

```
$ lsof -c lsof -a -d 1 -d 3 -u abe -r10
```

Use the `+r` or `-r` options for timed repeats. The `+r` switch will stop when no open files that meet the selected criteria are open. The `-r` will continue until interrupted by a signal. The number after the `r` is the time in seconds for each delay.

Between each cycle, `lsof` will print a sequence of equal signs (`=======`).

```
=======
```

internet connections

Because UNIX and Linux (and Mac OS X) treat internet connections as files, you can use the `-i` switch to view all of your open internet connections.

```
$ lsof -i
COMMAND   PID  USER   FD   TYPE    DEVICE SIZE/OFF NODE NAME
SystemUIS  93 admin   10u  IPv4 0x2152f48      0t0  UDP *:*
firefox-b 127 admin   75u  IPv4 0x43c5270      0t0  TCP 192.168.0.108:49816-
>63.141.192.121:http (CLOSE_WAIT)
Fetch     294 admin   23u  IPv4 0x27ffe64      0t0  TCP 192.168.0.108:50539-
>reliant.websitewelcome.com:ftp (ESTABLISHED)
Fetch     294 admin   24u  IPv4 0x2d2be64      0t0  TCP 192.168.0.108:50541-
>reliant.websitewelcome.com:36975 (LAST_ACK)
Fetch     294 admin   25u  IPv4 0x444a66c      0t0  TCP 192.168.0.108:50542-
>reliant.websitewelcome.com:22271 (TIME_WAIT)
```

internet files

To list all open Internet, x.25 (HP-UX), and UNIX domain files, type `lsof -i -U.` (from the `man` pages)

```
$ lsof -i -U
```

IPv4 network files by PID

To list all open IPv4 network files in use by the process whose PID is *1234*, type `lsof -i 4 -a -p 1234.` (from the `man` pages)

```
$ lsof -i 4 -a -p 1234
```

IPv6 files

To list all open IPv6 network files (assuming your UNIX system supports IPv6), type `lsof -i 6`. (from the man pages)

```
$ lsof -i 6
```

list by port

List all of the processes that are listening to a particular port by using colon (:) followed by the port number(s).

```
$ lsof -i:21
```

To list all files using any protocol on ports *513*, *514*, or *515* of host *wonderland.cc.purdue.edu*, type `lsof -i @wonderland.cc.purdue.edu:513-515`. (from the man pages)

```
$ lsof -i @wonderland.cc.purdue.edu:513-515
```

list TCP or UDP connections

List all of the TCP connections:

```
$ lsof -i tcp
```

List all of the UDP connections:

```
$ lsof -i udp
```

list from default domain

Assuming a default domain of *cc.purdue.edu*, list all files using any protocol on any port of *mace.cc.purdue.edu*, type `lsof -i @mace`. (from the man pages)

```
$ lsof -i @mace
```

socket files

To find an IP version 4 socket file by its associated numeric dot-form address, type `lsof -i@128.210.15.17`. (from the man pages)

```
$ lsof -i@128.210.15.17
```

To find an IP version 6 socket file by its associated numeric colon-form address, type `lsof -i@[0:1:2:3:4:5:6:7]`. (from the man pages)

```
$ lsof -i@[0:1:2:3:4;5;6:7]
```

To find an IP version 6 socket file by its associated numeric colon-form address that has a run of zeros in it (such as the loop-back address), type `lsof -i@[::1]`. (from the man pages)

```
$ lsof -i@[::1]
```

Network File System (NFS)

List all of the Network File System (NFS) files by using the `-N` switch.

```
$ lsof -N
```

To find processes with open files on the NFS file system named */nfs/mount/point* whose server is inaccessible (assuming your mount table supplies the device number for */nfs/mount/point*), type `lsof -b /nfs/mount/point`. (from the man pages)

```
$ lsof -b /nfs/mount/point
```

To do the preceding search with warning messages suppressed, type `lsof -bw /nfs/mount/point`. (from the man pages)

```
$ lsof -bw /nfs/mount/point
```

ignore device cache file

To ignore the device cache file, type `lsof -Di`. (from the man pages)

```
$ lsof -Di
```

obtain specific multiple info on each file

You can combine flags to gather specific information. To obtain the PID and command name field for each process, file descriptor, file device number, and file inode number for each file of each process, type `lsof -FpcfDi`. (from the man pages)

```
$ lsof -FpcfDi
```

using regular expressions

To list the current working directory of prcoesses running a command that is exactly four characters long and has an upper or lower case “O” or “o” in character position three, type `lsof -c /^..o.$/i -a -d cwd`. (from the man pages)

```
$ lsof -c /^..o.$/i -a -d cwd
```

free

chapter 58

summary

This chapter looks at `free`, a Linux command.

`free` is used to show free memory.

`free` displays the total amount of free and used physical memory and swap space in the system, as well as the buffers and cache consumed by the kernel.

size

The default is to report the memory size in bytes. Use the `-g` option to display in Gigabytes. Use the `-m` option to display in Megabytes. Use the `-k` option to display in Kilobytes. Use the `-b` option to display in Bytes.

total

Use the `-t` option to view total memory.

low/high

Use the `-l` option to view low vs. high memory usage.

count

The default is to display the information once and then return to the normal shell. Use the `-s` *n* option (where *n* is the number of times to display) to keep displaying the memory usage (until interrupted). Use the `-c` *n* or `--count=`*n* options (where *n* is the number of times to display the memory statistics).

Mac OS X

Use the Activity Monitor for this purpose. From the command line, you can get similar results from `vm_stat`. `vm_stat` is a Mach-specific version of the UNIX command `vmstat` Also:

```
$ echo -e "\n$(top -l 1 | awk '/PhysMem/';)\n"
```

vmstat

chapter 59

summary

This chapter looks at `vmstat`, a Unix (and Linux) command.

`vmstat` is used to show virtual memory usage.

usage

Type `vmstat` to see a list of virtual memory statistics. Especially useful in a virtualized enivronment.

```
$ vmstat
```

Mac OS X

`vm_stat` is a Mach-specific version of the UNIX command `vmstat`:

```
$
```

```
  vm_stat

Mach Virtual Memory Statistics: (page size of 4096 bytes)
Pages free:                        5538.
Pages active:                    142583.
Pages inactive:                   76309.
Pages wired down:                 37531.
"Translation faults":           4427933.
Pages copy-on-write:              54807.
Pages zero filled:              3086159.
Pages reactivated:                 6533.
Pageins:                          46357.
Pageouts:                          4453.
Object cache: 66465 hits of 76770 lookups (86% hit rate)
```

polkit

chapter 60

summary

This chapter looks at `polkit`, a Unix (and Linux) command.

`polkit` is used to control system-wide privileges.

running privileged commands

`polkit` can be used to run privileged commands by typing `pkexec`, followed by the intended command:

```
$ pkexec cat README
```

sudo alternative

It is generally recommended to use `sudo` rather than `pkexec`, because `sudo` provides greater flexibity and security.

Mac OS X

While `polkit` is *not* part of the official Mac OS X release from Apple, it can be installed on mac OS X.

security

`pkexec` never does any validation of the ARGUMENTS passed to a program.

graphic programs

X11 (and other graphic) programs won't run by default with `polkit` because the $DISPLAY environment variable is not set. The following information from the `pkexec` manual describes how to set `pkexec` so that it can run a graphic program.

```
To specify what kind of authorization is needed to execute the program
   /usr/bin/pk-example-frobnicate as another user, simply write an action
   definition file like this

       <?xml version="1.0" encoding="UTF-8"?>
       <!DOCTYPE policyconfig PUBLIC
        "-//freedesktop//DTD PolicyKit Policy Configuration 1.0//EN"
        "http://www.freedesktop.org/standards/PolicyKit/1/policyconfig.dtd">
       <policyconfig>

         <vendor>Examples for the PolicyKit Project</vendor>
         <vendor_url>http://hal.freedesktop.org/docs/PolicyKit/</vendor_url>

         <action id="org.freedesktop.policykit.example.pkexec.run-frobnicate">
           <description>Run the PolicyKit example program Frobnicate</description>
           <description xml:lang="da">Kør PolicyKit eksemplet Frobnicate</description>
           <message>Authentication is required to run the PolicyKit example
 program Frobnicate</message>
           <message xml:lang="da">Autorisering er påkrævet for at afvikle
 PolicyKit eksemplet Frobnicate</message>
           <icon_name>audio-x-generic</icon_name>
           <defaults>
             <allow_any>no</allow_any>
             <allow_inactive>no</allow_inactive>
             <allow_active>auth_self_keep</allow_active>
           </defaults>
           <annotate key="org.freedesktop.policykit.exec.path">
/usr/bin/pk-example-frobnicate</annotate>
         </action>

       </policyconfig>

   and drop it in the /usr/share/polkit-1/actions directory under a
   suitable name (e.g. matching the namespace of the action). Note that in
   addition to specifying the program, the authentication message,
   description, icon and defaults can be specified. For example, for the
   action defined above, the following authentication dialog will be
   shown:

       [IMAGE][2]

           +-------------------------------------------------------------+
           |                      Authenticate                       [X] |
           +-------------------------------------------------------------+
           |                                                             |
           |   [Icon]  Authentication is required to run the PolicyKit   |
           |           example program Frobnicate                        |
           |                                                             |
           |           An application is attempting to perform an        |
           |           action that requires privileges. Authentication   |
           |           is required to perform this action.               |
           |                                                             |
           |           Password: [_____________________________________]  |
           |                                                             |
           |  [V] Details:                                               |
           |    Command: /usr/bin/pk-example-frobnicate                  |
           |    Run As:  Super User (root)                               |
           |    Action:  org.fd.pk.example.pkexec.run-frobnicate         |
           |    Vendor:  Examples for the PolicyKit Project              |
           |                                                             |
           |                                  [Cancel] [Authenticate]    |
           +-------------------------------------------------------------+

   If the user is using the da_DK locale, the dialog looks like this:
```

```
      [IMAGE][3]

          +-------------------------------------------------------------+
          |                      Autorisering                      [X] |
          +-------------------------------------------------------------+
          |                                                             |
          |   [Icon]  Autorisering er pŒkr¾vet for at afvikle           |
          |           PolicyKit eksemplet Frobnicate                    |
          |                                                             |
          |           Et program fors¿ger at udf¿re en handling der     |
          |           kr¾ver privilegier. Autorisering er pŒkr¾vet.     |
          |                                                             |
          |           Kodeord: [_____________________________________]  |
          |                                                             |
          |  [V] Detaljer:                                              |
          |   Bruger:   Super User (root)                               |
          |   Program:  /usr/bin/pk-example-frobnicate                  |
          |   Handling: org.fd.pk.example.pkexec.run-frobnicate         |
          |   Vendor:   Examples for the PolicyKit Project              |
          |                                                             |
          |                                  [Annullžr] [Autorisering]  |
          +-------------------------------------------------------------+

    Note that pkexec does no validation of the ARGUMENTS passed to PROGRAM.
    In the normal case (where administrator authentication is required
    every time pkexec is used), this is not a problem since if the user is
    an administrator he might as well just run pkexec bash to get root.

    However, if an action is used for which the user can retain
    authorization (or if the user is implicitly authorized), such as with
    pk-example-frobnicate above, this could be a security hole. Therefore,
    as a rule of thumb, programs for which the default required
    authorization is changed, should never implicitly trust user input
    (e.g. like any other well-written suid program).
```

defaults

chapter 61

summary

This chapter looks at `defaults`, a Mac OS X command.

`defaults` is used to read, write, or delete defaults from a command line shell.

defaults options

Mac OS X programs use the defaults system to record user preferences and other information. Most of the information is more easily accessed through the application’s Preferencess panel, but some information is normally inaccessible to the end user.

Note: Since applications do access the defaults system while they’re running, you shouldn’t modify the defaults of a running application.

All programs share the defaults in **NSGlobalDomain**. If the program doesn’t have its own default, it uses the value from the NSGlobalDomain.

screencapture defaults

The following methods use Terminal to change the default file format and location where the screenshot is saved from the graphic user interface.

In Mac S X 10.4 (Tiger) or more recent, the default screencapture format can be changed in Terminal by using the `defaults` command. In Mac S X 10.4 (Tiger), the new default does not take effect until you logout and log back in (from the entire computer, not just from Terminal — a full restart will also work) unless you also use the `killall` command.

```
$ defaults write com.apple.screencapture type ImageFormat; killall SystemUIServer
```

The *ImageFormat* can be png (Portable Network Graphic), pdf (Portable Document Format), tiff (Tagged Image File Format), jpg or jpeg (Joint Photographic Experts Group), pict (Macintosh QuickDraw Picture), bmp (Microsoft Windows Bitmap), gif (Graphics Interchange Format), psd (Adobe Photoshop Document), sgi (Silicon Graphics File Format), or tga (Truevision Targe File Format).

JPGs are saved at quality 60%.

To change the default location where the screenshot file is saved (the default is Desktop), use the following Terminal command (where *PathName* is the full path to a directory.

```
$ defaults write com.apple.screencapture location PathName; killall SystemUIServer
```

The normal default location would be reset with the following command (where *UserName* is the current account's user name.

```
$ defaults write com.apple.screencapture location /Users/UserName/Desktop; killall
SystemUIServer
```

dock defaults

Use the following commands to change the 3D Dock of Mac OS X Leopard and later back to the 2D look. The `killall` restarts the Dock so that the change takes effect right away.

```
$ defaults write com.apple.dock no-glass -boolean YES; killall Dock
```

Return to the 3D look with the following commands:

```
$ defaults write com.apple.dock no-glass -boolean NO; killall Dock
```

Use the following commands to add a gradient behind an icon in a Dock Stack in Mac OS X:

```
$ defaults write com.apple.dock mouse-over-hilte-stack -boolean YES; killall Dock
```

Use the following commands to add a small Exposé button to the upper right of your Mac OS X screen. Clicking on the Exposé button will show all of the windows from the current application (same as the normal F10 default) and option clicking the Exposé button will show all windows from all programs (same as the normal F9 default). This will work even if you reconfigure the F9 and F10 keys to do something else.

```
$ defaults write com.apple.dock wvous-floater -bool YES; killall Dock
```

Use the following commands to remove the small Exposé button.

```
$ defaults write com.apple.dock wvous-floater -bool NO; killall Dock
```

login defaults

Use the following command to add a message to your Login window. Replace *message* with a short message. Keep the message short. As always with the `sudo` command, type very carefully.

```
$ sudo defaults write /Library/Preferences/com.apple.loginwindow LoginwindowText
"message"
```

To remove the login message, type the following:

```
$ sudo defaults write /Library/Preferences/com.apple.loginwindow LoginwindowText ""
```

showing dot files

The Mac OS X Finder hides all files that start with the period or dot character. This matches the standard behavior of `ls` in UNIX, where a leading dot is used as an indicator of a hidden file.

To have Finder always show all files, including hidden files, type:

```
$ defaults write com.apple.Finder AppleShowAllFiles YES; killall Finder
```

To restore Finder to its default behavior, type:

```
$ defaults write com.apple.Finder AppleShowAllFiles NO; killall Finder
```

.DS_Store files

To prevent Mac OS X from creating .DS_Store files on network shares (and everywhere else), type:

```
$ defaults write com.apple.desktopservices DSDontWriteNetworkStores true
```

To restore Mac OS X from creating .DS_Store files on network shares (and everywhere else), type:

```
$ defaults write com.apple.desktopservices DSDontWriteNetworkStores false
```

Mac Flashback Trojan

You can use `defaults` to determine if your Macintosh has been infected by the Mac Flashback Trojan (which enters your computer through a Java flaw).

Type the following command (copy and paste into Terminal):

```
$ defaults read /Applications/Safari.app/Contents/Info LSEnvironment
```

A clean system will report “The domain/default pair of (/Applications/Safari.app/Contents/Info, LSEnvironment) does not exist”. Any other result indicates your computer is infected.

If your computer passes the first test, type the following command (copy and paste into Terminal):

```
$ defaults read ~/.MacOSX/environment DYLD_INSERT_LIBRARIES
```

A clean system will report “The domain/default pair of (/Users/*user-name*/.MacOSX/environment, DYLD_INSERT_LIBRARIES) does not exist”. Any other result indicates your computer is infected.

If your computer is infected, immediately go to F-Secure at http://www.f-secure.com/v-descs/trojan-downloader_osx_flashback_i.shtml.

Downloading the latest security patches from Apple at http://support.apple.com/kb/HT5228?viewlocale=en_US&locale=en_US will help prevent infection.

init

chapter 62

summary

This chapter looks at `init`, a Linux command.

`init` is used to change the runlevel.

run levels

Linux has run levels (sometimes written as a single word, runlevel). The non-graphic mode is run level 3. the graphic mode is run level 5. Sometimes a Linux server will only boot up to the non-graphic level. You can use this command to temporarily get the graphic interface running (run level 5).

```
$ init 5
```

Edit the `/etc/inittab` file to include `id:5:initdefault:` to make a permanent change.

A chart of the Linux run levels:

RUN LEVEL	DESCRIPTION
0	System is halted
1	Single-user Mode used for system maintenance
2	Multiuser mode withut networking rarely used
3	Standard multiuser mode with networking active
4	not defined
5	Multiuser mode with graphics Like standard multiuser mode with networking active plus the X Window System graphic interface
6	Reboots the system shuts down all system services

sendmail

chapter 63

summary

This chapter looks at `sendmail`, a Unix/Linux utility.

`sendmail` is used to send email.

In 1975 the `Sendmail` email system was added to the BSD version of UNIX. Because email was built into the operating system, email was the method UNIX used for sending notifications to the system administrator of system status, reports, problems, etc.

set-up

Each version of UNIX has its own variation of `sendmail`, but the default installation almost always works correctly without modification. A system administrator rarely needs to know how to set-up or modify `sendmail`.

ifconfig

chapter 64

summary

This chapter looks at `ifconfig`, a Unix (and Linux) command.

`ifconfig` is used to configure network cards and interfaces.

Warning: `ifconfig` will not save changes on AIX after a reboot. On Linux, any changes made with `ifconfig` are saved on a reboot.

view configuration

Type `ifconfig` by itself to view how your network is configured.

```
$ ifconfig
```

static IP address

Use `ifconfig` *`interface IP-address`* to set the static IP address for a particular interface

```
$ ifconfig en0 100.1.1.1
```

arp

chapter 65

summary

This chapter looks at `arp`, a Unix (and Linux) command.

`arp` is used to display and modify the Internet-to-Ethernet address translation tables.

display all

Use the `-a` option to display all of the current ARP entries. Useful in conjunction with `ifconfig` and `route`.

```
$ arp -a
? (192.168.0.1) at 0:11:95:75:8:86 on en1 [ethernet]
? (192.168.0.106) at 0:d:60:d2:55:3f on en1 [ethernet]
? (192.168.0.255) at ff:ff:ff:ff:ff:ff on en1 [ethernet]
$
```

netstat

chapter 66

summary

This chapter looks at `netstat`, a Unix (and Linux) command.

`netstat` is used to symbolically displays the contents of various network-related data structures.

view connections

Type `netstat` to see the list of all sockets and server connections.

```
$ netstat
```

main info

The main information for webserver administration is in the first few lines, so you can pipe to `head`:

```
$ netstat | head
```

routing addresses

Use the `-r` option to display the network routing addresses.

```
$ netstat -r
```

route

chapter 67

summary

This chapter looks at `route`, a Unix (and Linux) command.

`route` is used to manipulate the network routing tables.

view connections

Type `route` followed by the `-v` option to list the routing tables. This is close to the same information reported by `netstat -r`.

```
$ route -v
```

routing commands

The `route` utility can used for `add`, `flush`, `delete`, `change`, `get`, or `monitor`.

The `add` command will add a route.

The `flush` command will remove all routes.

The `delete` command will delete a specific route.

The `change` command will change aspects of a route (such as its gateway).

The `get` command will lookup and display the route for any destination.

The `monitor` command will continuously report any changes to the routing infrmation base, routing lookup misses, or suspected network partitionings.

ping

chapter 68

summary

This chapter looks at `ping`, a Unix (and Linux) command.

`ping` is used to test a server connection.

test packets

`ping` sends test packets to a specific server to see if the server responds properly.

Type `ping` followed by `-c` *`count`* followed by an IP address or domain name.

`ping` will continue forever unless halted or you give a specific count limit.

```
$ ping -c 5 www.osdata.com
PING osdata.com (67.18.4.2): 56 data bytes
64 bytes from 192.168.0.1: icmp_seq=0 ttl=127 time=18.801 ms
64 bytes from 192.168.0.1: icmp_seq=1 ttl=127 time=9.918 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=127 time=17.216 ms
64 bytes from 192.168.0.1: icmp_seq=3 ttl=127 time=6.748 ms
64 bytes from 192.168.0.1: icmp_seq=4 ttl=127 time=30.640 ms
```

```
--- 192.168.0.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 6.748/16.665/30.640/8.295 ms
$
```

`ping` will quickly either return a report or will say *destination host unreachable* on failure.

measuring

You can measure the dropped packet rate and the minimum, mean, maximum, and standard deviation of round-trip times.

When using `ping` for testing, start by pinging local host to verify that the local network interface is up and running. Then ping hosts and gateways further away.

nslookup

chapter 69

summary

This chapter looks at `nslookup`, a Unix (and Linux) command.

`nslookup` is used to obtain domain name and IP information about a server.

information

Type `nslookup` followed by a doman name.

```
$ nslookup www.osdata.com
Server:       192.168.0.1
Address:      192.168.0.1#53

Non-authoritative answer:
www.osdata.com  canonical name = osdata.com.
Name:   osdata.com
Address: 67.18.4.2

$
```

security

You can use `nslookup` to find the DNS information for a particular host IP in your server access logs.

traceroute

chapter 70

summary

This chapter looks at `traceroute`, a Unix (and Linux) command.

`traceroute` is used to obtain the entire path to a server or host.

entire route

Type `traceroute` followed by a doman name.

```
$ traceroute www.osdata.com
```

You can use `traceroute` to diagnose network problems. When some people can access your website and other can't, then `traceroute` can help you find the break or error along the network path.

etiquette

`traceroute` puts a load on every server in the path to the selected host. Therefore, only use `traceroute` when you need the diagnosis.

ftp and sftp

chapter 71

summary

This chapter looks at `ftp` and `sftp`, a Unix (and Linux) command.

`ftp` is FIle Transfer Protocol.

`sftp` is Secure FIle Transfer Protocol.

In general, you should use `sftp` rather than `sftp`. There is a lot of overlap between the two tools and their commands. This chapter will primarily discuss examples as `ftp`. In practice you should substitute `sftp` whenever possible.

There are many graphic FTP tools available. You should consider using these tools for most file transfers. I personally use Fetch, but that's from long time habit. FileZilla is popular, free, and open source, with binaries available for Windows, Linux, and Mac OS X, and source code available for most other platforms.

For serious scripting, you should probably consider the more powerful `cURL` (see next chapter).

You will find many existing scripts that use `ftp`. When you are using the command line, it is often easier and faster to do quick work with `ftp` or `sftp`. If you are at an unfamiliar machine and simply fall back on the command line FTP rather than try to guess what graphic FTP program is available.

Adobe recommends using command line FTP "for testing when troubleshooting problems with the Dreamweaver or UltraDev FTP client.". See web page http://kb2.adobe.com/cps/164/tn_16418.html . This is a recommendation to the mostly artist, non-geek audience for Adobe's creative products. Adobe provides step by step instructions for both Windows and

Mac OS X.

sample session

The following sample session shows how a simple `ftp` session might go. This will give you context for the many specific details that follow.

```
$ ftp example.com
Connected to example.com
220 example FTP Server (Version x.x Mon Jan 20 05:32:40 EDT 2014) ready.
User (example.com:(none)): User-name
331 Password required for user-name
Password: password
230 User user-name logged in.
ftp> cd /directory
250 CWD command successful.
ftp> pwd
257 "/directory" is the current directory.
ftp> ls
random.txt
picture.png
226 transfer completed
ftp> ascii
200 Type set to A.
ftp> put filename.txt
200 PORT command successful.
Opening ASCII mode data connection for filename.txt
226 Transfer complete
ftp> quit
221 Goodbye.
$
```

formatting for this lesson

In the following examples, things you type will be in **bold**. Anything contained inside square braces **[ ]** is optional and anything outside of square braces is required. Braces inside braces indicate options that can be applied to options. The stroke or vertical bar character **|** indicates a choice — enter only one of the choices. Elipses **…** indicate that you can have one, two, or mroe of the indicated item.

standard FTP

To start an FTP session, type `ftp hostname`. The host name can be a named URL or an IP address. REMINDER: whenever possible, substitute `sftp` for `ftp` throughout this chapter.

You will be asked for your userid and password.

To end the FTP session type either `quit` or `bye`.

Enter commands after the `ftp>` prompt. to conduct your FTP session. Type `help` to get a current list of commands available and type `help` followed by a command to get help on a specific command.

All FTP commands are case sensitive and in lower case.

anonymous FTP

Some servers allow anonymous ftp access. Use `anonymous` or `ftp` as your userid. If you are asked for a logon password,

your own email address or `guest` will often be acceptable.

help

`?` will give a list of available commands. If followed by the name of a specific command, it will give a brief description of that command. Works the same way as `help`.

`help` will give a list of available commands. If followed by the name of a specific command, it will give a brief description of that command. Works the same way as `?`.

`rhelp` will get help from the remote server (which may have a different command set than the local machine). `rhelp` is more commonly available than `remotehelp`.

```
ftp> rhelp
```

`remotehelp` will get help from the remote server (which may have a different command set than the local machine).

```
ftp> remotehelp [command-name]
```

connections

`ftp` will connect to a remote FTP server. This command is given at the normal command shell to start FTP. Provide a host name (URL or IP address) and an optional port number.

```
$ ftp host-name [port]
```

`sftp` will connect to a remote FTP server using secure SSH. This command is given at the normal command shell to start SFTP. Provide a host name (URL or IP address) and an optional port number.

```
$ sftp host-name [port]
```

`open` will connect to a specified remote FTP host (by name or IP address). An optional port number may be supplied. If auto-login option is on (the default), then FTP will attempt to automatically log the user in t the FTP server.

```
ftp> open host-name [port]
```

`account` will supply a supplemental password. Sometimes there is a need for an additional password to access specific resources (such as a password-protected directory). If no argument is included, the user willbe prompted for an account password in a non-echoing input mode.

```
ftp> account [password]
```

`close` will terminate the FTP session. You can follow a `close` command with a `open` command to connect to a new host. This is the same as the `disconnect` command. Any defined macors are erased.

```
ftp> close
```

disconnect will terminate the FTP session. You can follow a `disconnect` command with a `open` command to connect to a new host. This is the same as the `close` command.

bye will terminate the FTP session and exit back to the shell. The same as `quit`.

```
ftp> bye
```

quit will terminate the FTP session and exit back to the shell. The same as `bye`.

```
ftp> quit
```

file transfers

ascii will set the FTP session for ASCII file transfers (the alternative is `binary`). This is the default method.

binary will set the FTP session for binary file transfers (the alternative is `ascii`).

type will set the file transfer type.

form will set the file transfer format. The default format is file.

```
ftp> form format
```

mode will set the file transfer mode. The default format is stream.

```
ftp> mode mode-name
```

struct will set file transfer structure to `struct-name`. The default is `stream` structure.

```
ftp> struct struct-mode
```

xferbuf will set the size of the socket send/receive buffer.

```
ftp> xferbuf size
```

get will receive a file from the remote server. Name the file after the `get` command. You may optionally follow with a second name which will assign the second file name to the local downloaded file. The same as `recv`.

recv will receive a remote file. Name the file after the `recv` command. You may optionally follow with a second name which will assign the second file name to the local downloaded file. The same as `get`.

mget will get multiple remote files. If `glob` is on, then you can use wildcards in specifying files. Toggle `prompt` off if you don’t want to be prompted for each file downloaded.

fget will get files using a localfile as the source of the file names. That is, put the list of file names into a local file and

then pass that file name to this command.

```
ftp> fget localfile
```

glob will toggle metacharacter expansion of local file names.

put will send one file to the remote server. Follow the `put` command with the file to upload. Optionally include a second name to assign a different name to the file uploaded to the remote server. It is an error if the file does not exist on your local machine. The same as `send`.

send will send one file to the remote server. Follow the `send` command with the file to upload. Optionally include a second name to assign a different name to the file uploaded to the remote server. It is an error if the file does not exist on your local machine. The same as `put`.

append will append to a file.

```
ftp> append local-file [remote-file]
```

mput will send multiple files to the remote server. If `glob` is on, then you can use wildcards in specifying files. Toggle `prompt` off if you don't want to be prompted for each file uploaded.

delete will delete a remote file.

mdelete will delete multiple remote files.

rename will rename a file on the remote server.

directory control

pwd will print the current working directory on the remote server.

```
ftp> pwd
```

lpwd will print the current *local* working directory.

```
ftp> lpwd
```

cd changes the working directory on the remote server. Use `lcd` to change the local working directory.

ls will list the contents of a remote directory. Works the same as `dir`. The listing will include any system-dependent information the server chooses to include. Most UNIX systems will produce the output of the command `ls -l`. You may name a specific remote directory. Otherwise, the current remote working directory is listed. You may optionally also name a local file and the listing will be stored in the designated local file. If no local file is specified or the local file is "`-`", the output will be sent to the terminal.

```
ftp> ls [remote-path [local-file]]
```

dir will list the contents of the remote directory. Works the same as `ls`.

- `dir -C` lists the files in wide format.
- `dir --1` lists the files in bare format in alphabetic order.
- `dir --S` lists the files in bare format in alphabetic order.
- `dir --r` lists the files in reverse alphabetic order.
- `dir -R` lists all files in the current directory and sub directories.

`lcd` will display the local working directory if typed by itself. If a path name follows the lcd command, it will change the local working directory. Use `cd` to change the remote working directory.

`mlst` will list the contents of the remote directory in a machine parsable form. It defaults to listing the current remote working directory.

```
ftp> mlst [remote-path]
```

`mdir` will list the contents of multiple remote directories. Works the same as `mls`.

`mls` will list the contents of multiple remote directories. Works the same as `mdir`.

`mkdir` will make a directory on the remote server.

`rmdir` will delete or remove a directory on the remote server.

```
ftp> rmdir directory-name
```

`modtime` will show the last modification time and date for a remote file.

```
ftp> modtime remote-file
```

`chmod` will change file permissions on a remote file.

```
ftp> chmod mode remote-file
```

toggles

`cr` will toggle the stripping of carriage returns during ASCII type file retrieval. records are denoted by the sequence of carriage return/line feed sequence during ASCII type file transfer. When `cr` is on, carriage returns are stripped to conform to the UNIX single linefeed record delimiter. This is the default. records on non-UNIX remote systems may contain single linefeeds and when an ASCII type file transfer is made, these linefeeds will only be distinguished from a record delimiter if `cr` is off. The default is for `cr` to be on.

```
ftp> cr [ on | off ]
```

`debug` will toggle the debugging mode. If the optional debug value is provided, it is used to set the debugging level. When debugging is on, FTP prints each command sent to the remote server, preceded by the string "`-->`".

```
ftp> debug [ on | off | debuglevel ]
```

gate will toggle the gate-ftp. Specify the optional host[:port] to change the proxy.

```
ftp> gate [ on | off | gateserver [port] ]
```

glob will toggle metacharacter expansion of local file names.

hash will toggle printing # for each buffer of 1,024 (1 kilobyte) of data is transferred.

passive will toggle the use of passive transfer mode.

```
ftp> passive [ on | off | auto ]
```

sunique will toggle storing of files under unique file names on the remote server. Remote FTP server must support the FTP protocol STOU command for successful completion. The remote server will report unique names. The defauly is off.

```
ftp> sunique [ on | off ]
```

trace will toggle packet tracing.

verbose will toggle the verbose mode. In verbose mode, all responses from the FTP server are displayed to the local user. When a file transfer completes, verbose mode also provides statistics about the effiency of the file transfer. By default, verbose is on.

```
ftp> verbose [ on | off ]
```

pagers

Some versions of command line FTP will allow you to read the contents of remote files in a local pager (such as less or more).

page will view a remote file through your local pager.

```
ftp> page remote-file
```

less will view a remote file through the pager named `less`.

```
ftp> less remote-file
```

more will view a remote file through the pager named `more`.

```
ftp> more remote-file
```

pdir will list the contents of a remote path through your local pager. You can designate a specific remote path or default to the remote server current working directory.

```
ftp> pdir [remote-path]
```

lpage will view a local file through your local pager.

```
ftp> lpage local-file
```

other commands

prompt will force interactive prompting on multiple commands.

system will show the operating system running on the remote server.

```
ftp> system
```

status will show the current status.

rstatus will show the current status of the remote server. If a file name is specified, it will show the status of the file on the remote server.

```
ftp> rstatus [remote-file]
```

remotestatus will show the current status of the remote server. If a file name is specified, it will show the status of the file on the remote server.

```
ftp> remotestatus [remote-file]
```

user will send new user information.

! Escape the the shell. Type `exit` to return to the FTP session. You can follow `!` with a specific command. If you type a command, then all the following text will be considered as arguments for that command.

```
ftp> ! [command [args]]
```

```
ftp> !
$ unix_command
command results
$ exit
ftp>
```

bell will beep when a command is completed.

literal will send an arbitrary FTP command with an expected one line reponse. The same as `quote`.

macro will execute the designated macro `macro-name`. The macro `macro-name` must have been defined with the `macdef` command. Arguments passed to the macro are unglobbed.

```
ftp> $ macro_name [args]
```

quote will send an arbitrary command to the remote server. The same as `literal`. All arguments are sent veratim to the remote FTP server.

```
ftp> quote line-to-send [ arg1 arg2 … ]
```

FileZilla

FileZilla Client is a free, open source FTP client that supports FTP, SFTP, and FTPS (FTP over SSL/TLS). It is available for many operating systems and there are binaries available for Windows, Linux, and Mac OS X. The source code is hosted at http://sourceforge.net/projects/filezilla. The binaries can be found at https://filezilla-project.org/download.php.

While FileZilla is intended as a graphic interface for FTP, the client will accept command line.

Use the command `filezilla [<FTP URL>]` to connect to a server. The form of the URL should be `[protocol://][user[:pass]@]host[:port][/path]`. Valid protocols include **ftp://**, **ftps://**, **ftpes://**, or **sftp://**. The default is ordinary ftp.

Use the option `-c` or `--site=<string>` to connect to the specified Site Manager site.

Use the option `-a` or `--local=<string>` to set the local directory from which you will be uploading files.

Putting this together, the following example will connect to the site example.com by ftp protocol using the user name and password and sets the local folder (directory) to uploads in the current user's home directory:

```
$ filezilla ftp://username:password@ftp.example.com --local="~/uploads"
```

Use the `-l` or `---logontype=(askinteractive)` flag to have FileZilla interactively ask for login information. This option is useful for scripting.

cURL

chapter 72

summary

This chapter looks at `cURL`, a Unix (and Linux) command line tool.

`curl` is used to interact with the internet. Linux also includes the similar `wget` command tool. You can download `wget` for free using `fink`.

cURL is a command line tool and accompanying library for transferring data over the internet. cURL supports a wide variety of internet protocols, including DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, Telnet, and TFTP.

download a file

You can use `curl` to download the content of a URL. The following command displays the HTML for Google as the STDOUT (on your Terminal screen):

```
$ curl http://www.google.com
```

redirect a download

You can redirect the results to file using the standard Unix/Linux redirect. You will be given some status information if you redirect.

```
$ curl http://www.google.com > google.html
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10559    0 10559    0     0  26690      0 --:--:-- --:--:-- --:--:--  323k
```

You can use the `-o` (lower case) to save the downloaded file with the name of your choice. You will see progress meter information as the download progresses. In the example, the download was so fast that only one status report was created.

```
$ curl -o google.303 www.google.com
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   219  100   219    0     0    800      0 --:--:-- --:--:-- --:--:--  0
```

sysstat

chapter 73

summary

This chapter looks at `sysstat`, a server package.

package

`sysstat` is a server toolset originally created by Sebastien Godard.

Download sysstat from http://pagesperso-orange.fr/sebastien.godard/download.html.

The most famous tools ar iostat, pidstat, and sar.

`iostat` reports CPU statistics and input/output statistics for devices, partitions and network filesystems.

`mpstat` reports individual or combined processor related statistics.

`pidstar` reports statistics for Linux tasks (processes) : I/O, CPU, memory, etc.

`sar` collects, reports and saves system activity information (CPU, memory, disks, interrupts, network interfaces, TTY, kernel tables,etc.)

`sadc` is the system activity data collector, used as a backend for sar.

`sa1` collects and stores binary data in the system activity daily data file. It is a front end to sadc designed to be run from cron.

`sa2` writes a summarized daily activity report. It is a front end to sar designed to be run from cron.

`sadf` displays data collected by sar in multiple formats (CSV, XML, etc.) This is useful to load performance data into a database, or import them in a spreadsheet to make graphs.

`nfsiostat` reports input/output statistics for network filesystems (NFS).

`cifsiostat` reports CIFS statistics.

at

chapter 74

summary

This chapter looks at `at`, a Unix (and Linux) command.

`at` is used to schedule a particular job at a particular time.

example

Type `at midnight`, followed by ENTER or RETURN.

```
$ at midnight
```

You may see the `at>` prompt (on Mac OS X, there is no prompt).

Type a single command, followed by ENTER or RETURN.

```
$ who > who.out
```

Type one command per line.

When finished, hold down the CONTROL key and then the D key (written ^d) to exit `at`.

```
job 1 at Fri Jul 13 00:00:00 2012
$
```

You will see a job number and the time it will run. This job will run all of the commands you entered.

removing an at job

Type `atrm` (for at remove), followed by the job number to remove an existing `at` job.

```
$ atrm 1
```

timing

You can name a specific time using the YYMMDDhhmm.SS format.

You can also specify noon, midnight, or teatime (4 p.m.).

If the time has already passed, the next day is assumed.

back ups

chapter 75

summary

This chapter looks at backing up systems and data.

While everyone should perform back ups, when a system crashes, everyone will turn to the system administrator to be rescued. And this is a get yourself fired kind of issue because the system administrator will always get the blame for lack of back ups.

tools

Your basic tools for back ups are `tar`, various compression schemes, and `cron` scripts.

`tar` stands for Tape ARchive, from back in the days when back ups were put on those large tape machines you see in the old James Bond movies.

`cron` is a system for scheduling jobs, including UNIX shell scripts, to run at specific times.

rule of thumb

People are always asking how often they should do their backups, looking for a regualr time period.

The rule of thumb on back-ups is to back up your data before you reach the point where you have too much work to do to replace the data.

If you start typing an email and the system crashes, you are annoyed, but you can type the email over again. On the other hand, if you lose all of your emails, you have a problem.

Hence, the simple rule of thumb to back up as soon as you are about to have too much data to create over again.

Notice that this answer wasn't in the form of a specific time periood (such as daily or weekly).

The reality of any large scale system is that you will be using the cruder method of specific times, such as daily, weekly, and monthly back ups.

incremental and full

A full back up stores everything on a system.

An incremental backup stores only those things that have changed.

Back ups (especially full backups) take a lot of storage space.

The typical policy is to do incremental backups fairly often (daily, hourly, whatever is appropriate for the demands of your system) and to do full backups less often (weekly or even monthly).

When you have a failure, you go back to the last full back up and then run every incremental back up. The more incremental back ups you have to run, the longer it will be before the syetem is restored, with angry users and angry bosses waiting impatiently.

three location storage

There is the rule of a minimum of three copies of your back ups. You have two copies of your back on site and one stored at a secure location far away.

The secure storage copy protects you from fires or eathquakes or floods or other disasters that physically destroy your ability to restore your system. Therefore, the secure back ups need to be kept far enough away that they won't get destroyed in the same disaster.

Of course, this also means that there is travel time for using the secure back up.

The local back ups are usually alternated (or rotated, if you have more than just two).

The first full back up is stored on media A. The next full back up is stored on media B. The next full back up is stored on media A (wiping out the original back up).

tar

chapter 76

summary

This chapter looks at `tar`, a Unix (and Linux) command.

Use `tar` for making backups and for retrieving open source software from the internet.

This command is used to archive directories and extract directories from archives. The `tar` archives can be used for back-up storage and for transfer to other machines.

`tar` files are often used to distribute open source programs.

`tar` is short for Tape ARchive.

Warning: `tar` does *not* work the exact same on all versions of UNIX and Linux. Test and confirm local variations before relying on scripts moved from one platform to another.

create

Creating a new tar archive:

```
$ tar cvf archivename.tar dirname/
```

The c indicates create a new archive, the v indicates verbosely list file names, and the f indicates that the following is the archive file name. The v option may be left out to avoid a long list.

Creating a new gzipped tar archive:

```
$ tar cvzf archivename.tar.gz dirname/
$ tar cvzf archivename.tgz dirname/
```

The z indicates the use of gzip compression. The file extensions .tgz and .tar.gz mean the same thing and are interchangeable.

Creating a new bzip2 tar archive:

```
$ tar cvzj archivename.tar.bz2 dirname/
$ tar cvzj archivename.tbz dirname/
```

The j indicates the use of bzip2 compression. The file extensions .tbz and .tar.bz2 mean the same thing and are interchangeable.

Bzip2 takes longer to compress and decompress than gzip, but also produces a smaller compressed file.

extract

Extracting from an existing tar archive:

```
$ tar xvf archivename.tar
```

The x indicates extract files from an archive, the v indicates verbosely list file names, and the f indicates that the following is the archive file name. The v option may be left out to avoid a long list.

Extracting from a gzipped tar archive:

```
$ tar xvfz archivename.tgz
$ tar xvfz archivename.tar.gz
```

The z option indicates uncompress a gzip tar archive.

Extracting from a bzipped tar archive:

```
$ tar xvfj archivename.tbz
$ tar xvfj archivename.tar.bz2
```

The j option indicates uncompress a bzip2 tar archive.

tarpipe

It is possible to tar a set of directories (and their files and them pipe the tar to an extraction at a new location, making an exact copy of an entire set of directories (including special files) in a new location:

```
$ tar -c "$srcdir" | tar -C "$destdir" -xv
```

tarbomb

A malicious tar archive could overwrite existing files. The use of absolute paths can spread files anywhere, including overwriting existing files (even system files) anywhere. The use of parent directory references can also be used to overwrite existing files.

One can examine the tar archive without actually extracting from it.

The bsdtar program (the default tar on Mac OS X v10.6 or later) refuses to follow either absolute path or parent-directory references.

viewing

Viewing an existing tar archive (without extracting) with any of the following:

```
$ tar tvf archivename.tar
$ tar -tf archivename.tar
$ tar tf archivename.tar
$ tar --list --file=archivename.tar
```

Viewing an existing gzipped tar archive (without extracting) with any of the following:

```
$ tar tvfz archivename.tar.gz
$ tar tvfz archivename.tgz
```

Viewing an existing bzipped tar archive (without extracting) with any of the following:

```
$ tar tvfj archivename.tar.bz2
$ tar tvfj archivename.tbz
```

less

You can pipe the output of a tar file to the less command.

You can directly use the less command to open a tar archive. Less will show you the names of the files, the file sizes, the file permissions, the owner, and the group (the equivalent of `ls -l`).

```
$ less archivename.tar
```

extracting a single file

You can extract a specific single file from a large tar archive.

```
$ tar xvf archivefile.tar /complete/path/to/file
```

You can extract a specific compressed file from a large tar archive.

```
$ tar xvf archivefile.tar.gz /complete/path/to/file
$ tar xvf archivefile.tar.bz2 /complete/path/to/file
```

extracting a single directory

You can extract a specific single directory from a large tar archive.

```
$ tar xvf archivefile.tar /complete/path/to/dir/
```

You can extract a specific compressed directory from a large tar archive.

```
$ tar xvf archivefile.tar.gz /complete/path/to/dir/
$ tar xvf archivefile.tar.bz2 /complete/path/to/dir/
```

extracting a few directories

You can extract specific directories from a large tar archive by giving the name of each of the directories.

```
$ tar xvf archivefile.tar /complete/path/to/dir1/ /complete/path/to/dir2/
```

You can extract specific compressed directories from a large tar archive.

```
$ tar xvf archivefile.tar.gz /complete/path/to/dir1/ /complete/path/to/dir2/
$ tar xvf archivefile.tar.bz2 /complete/path/to/dir1/ /complete/path/to/dir2/
```

extracting a group of files

You can extract a group of files from a large tar archive by using globbing. Specify a pattern (the following example finds all .php files):

```
$ tar xvf archivefile.tar --wildcards '*.php'
```

You may use the gzip and bzip2 options as well.

adding to an existing archive

You can add an additional file to an existing tar archive with the option `-r`:

```
$ tar rvf archivefile.tar newfile
```

You can add an additional directory to an existing tar archive:

```
$ tar rvf archivefile.tar newdir/
```

This does *not* work for adding a file or directory to a compressed archive. You will get the following error message instead:

```
tar: Cannot update compressed archives
```

verifying files

You can verify a new archive file while creating is by using the `-W` option:

```
$ tar cvfW archivefile.tar dir/
```

If you see something similar to the following output, something has changed:

```
Verify 1/filename
1/filename: Mod time differs
1/filename: Size differs
```

If you just see the Verify line by itself, the file or directory is fine:

```
Verify 1/filename
```

The `-W` option does *not* work for compressed files.

For gzip compressed files, find the difference between the gzip archive and the file system:

```
$ tar dfz 1/filename.tgz
```

For bzip2 compressed files, find the difference between the bzip2 archive and the file system:

```
$ tar dfj 1/filename.tbz
```

touch

chapter 77

summary

This chapter looks at `touch`, a Unix (and Linux) command.

Use `touch` to change the date stamp on a file.

The most common use of `touch` is to force `make` to recompile a particular file (or group of files).

how to use

The simple form of `touch` changes a named file's timestamp to the current date and time:

```
$ touch filename
```

multiple files

Use `touch` to change all of the files in a directory:

```
$ touch *
```

specific time

You can use `touch` to set a specific time by using the `-t` option. The following command sets a file to the Solstice at the end of the Mayan calendar (11:12 a.m. December 21, 2012):

```
$ touch -t201212211112.00 filename
```

The following command sets a file to the noon on January 1, 2012:

```
$ touch -t201201011200.00 filename
```

The format is `YYYYMMDDhhmm.ss`, YYYY = four digit year, MM - two digit month, DD = two digit day, hh = two digit hour (24 hour clock), mm = two digit month, and ss = two digit second.

create an empty file

You can use `touch` to create an empty file. Simply issue the `touch` command with the desired file name.

```
$ touch filename
```

This will create an empty file with the designated file name.

Why would you want to do this? One common reason is to create an expected file. Some scripts will check for the existence of particular files and choke if they can't find them. If the script does not need particular data alreazdy stored in the file (for example, it will be appending data intot he file or it is simply checking for the existence of the file), then `touch` is your fast and easy method for creating the expected file (as an empty file).

find

chapter 78

summary

This chapter looks at `find`, a Unix (and Linux) command.

`find` is used to find files.

find a file

You can use `find` to quickly find a specific file. The `-x` option (this is `-xdev` on older systems) limits the search to only those directories on the existing filesystem.

```
$ find / -name filename -type d -x
```

Note that you may want to run this command from root or superuser.

The `locate` command is more recent.

gotchas

Do not use the output of `find` in a BASH `for` loop.

```
$ for i in $(find *.mp3); do # Wrong!
$    command $i # Wrong!
$ done # Wrong!

$ for i in $(find *.mp3) # Wrong!
$ for i in `find *.mp3` # Wrong!
```

If any file name has spaces in it, then it will be word split. So, the MP3 for Mumford & Sons's "The Cave" (number two song on their "Sigh No More" album) might be `02 The Cave`. In that case, your command would run on the files "02", "The", and "Cave", rather than "02 The Cave". Additional errors likely to follow. Listen to the song "The Cave" for free at http://www.thissideofsanity.com/musicbox/musicbox.php?mastersong=48 .

Double quote won't work either because that will cause the entire output of the `find` command to be treated as a single item. Your loop will run once on all of the file names conactenated together.

```
$ for i in "$(ls *.mp3)"; # Wrong!
```

The solution is to run your loop without even using `find`, relying on BASH's filename expansion. Don't forget to check for the possibility that there are no qualifying files in the current directory.

```
$ for i in *.mp3; do # Wrong!
$    [[ -f "$i" ]] || continue
$    command "$i"
$ done
```

text processing

chapter 79

summary

This chapter looks at text processing in Linux/UNIX.

The original approach of UNIX was to move data between small programs as character-based text. Most UNIX commands and tools produce text output as a character stream.

A number of special utilities were developed that allowed a UNIX script to modify the text-based character stream as it passed between programs. That use is still common used in shell scripts.

Some of these tools are also useful for processing and formatting text for humans to read.

text processing tools

There are several text processing tools discussed in this book.

- awk
- basename
- cat
- sed
- tail
- wc

basename

chapter 80

summary

This chapter looks at `basename`, a Unix (and Linux) command.

The `basename` command removes the directory and (optionally) file extension (or suffix) from a file name.

removing directory

Remove the diretcory portion of a filename with this command:

```
$ basename filename
```

Don’t forget to use some form of escaping as needed.

```
$ basename "/home/local/bigfile.txt"
bigfile.txt
$
```

removing file suffix

Remove the file suffix portion of a filename with this command:

```
$ basename filename suffix
```

And an example:

```
$ basename "/home/local/bigfile.txt" ".txt"
bigfile
$
```

Note that the suffix portion can be *any* text. This means you can use this command to remove the last portion of any text:

```
$ basename "Los Angeles, CA" ", CA"
Los Angeles
$
```

sed

chapter 81

summary

This chapter looks at sed, a Unix (and Linux) command.

sed is a command line editor.

fixing end of line

Windows and DOS end every line with the old carriage return and line feed combination (\r\n). You cna use sed to convert Windows/DOS file format to UNIX file format:

```
$ sed 's/.$//' filename
```

adding line numbers

You can use sed to add line numbers to all of the non-empty lines in a file:

```
$ sed '/./=' filename | sed 'N; s/\n/ /'
```

awk

chapter 82

summary

This chapter looks at `awk`, a Unix (and Linux) command.

`awk` is a programming language intended to serve a place between simple shell scripts and full programming languages. It is Turing complete.

remove duplicate lines

You can use `awk` to remove duplicate lines from a file.

```
$ awk '!($0 in array) { array[$0]; print }' filename
```

screencapture

chapter 83

summary

This chapter looks at `screencaptures`, a Mac OS X-only command.

`screencapture` creates an image of the screen or a portion of the screen.

from the graphic user interface

The normal method for obtaining a screen capture is through the graphic user interface. Command-Shift-3 takes a screenshot of the screen and saves it as a file to the desktop under the name of “Picture 1” (or next available number if there are already screenshots saved there).

If you have multiple monitors connected, each monitor is saved as a separate picture, named “Picture 1”, “Picture 1(2)”, “Picture 1(3)”, etc.

With Mac OS X 10.6 (Snow Leopard) the default name changes to “Screen shot YYYY-MM-DD at HH.MM.SS XM”, where YYY=year, MM=month, DD=day, HH=hour, MM=minute, SS=second, and XM = either AM or PM.

The basic screen capture options:

- **Command-Shift-3:** Take a screenshot of the entire screen and save it as a file on the desktop.
- **Command-Shift-4, then select an area:** Take a screenshot of an area and save it as a file on the desktop.
- **Command-Shift-4, then space, then click on a window:** Take a screenshot of a selected window and save it as a file on the desktop.
- **Command-Control-Shift-3:** Take a screenshot of the screen and save it to the clipboard.
- **Command-Control-Shift-4, then select an area:** Take a screenshot of an area and save it to the clipboard.
- **Command-Control-Shift-4, then space, then click on a window:** Take a screenshot of a selected window and save it to the clipboard.

In Mac OS X 5 (Leopard) or more recent, the following keys can be held down when selecting an area (with either Command-Shift-4 or Command-Control-Shift-4):

- **Space:** Used to lock the size of the selected region and move the selected region as the mouse moves.
- **Shift:** Used to resize only one edge of the slected region.
- **Option:** Used to resize the selected region with its center as the anchor point.

Different versions of Mac OS X have different default file formats for saving the screenshot:

- **Mac OS X 10.2 (Jaguar):** jpg
- **Mac OS X 10.3 (Panther):** pdf

- **Mac OS X 10.4 (Tiger):** png

changing defaults

The following methods use Terminal to change the default file format and location where the screenshot is saved from the graphic user interface.

In Mac S X 10.4 (Tiger) or more recent, the default screencapture format can be changed in Terminal by using the `defaults` command. In Mac S X 10.4 (Tiger), the new default does not take effect until you logout and log back in(from the entire computer, not just from Terminal — a full restart will also work) unless you also use the `killall` command.

```
$ defaults write com.apple.screencapture type ImageFormat
$ killall SystemUIServer
```

The *FileFormat* can be png (Portable Network Graphic), pdf (Portable Document Format), tiff (Tagged Image File Format), jpg or jpeg (Joint Photographic Experts Group), pict (Macintosh QuickDraw Picture), bmp (Microsoft Windows Bitmap), gif (Graphics Interchange Format), psd (Adobe Photoshop Document), sgi (Silicon Graphics File Format), or tga (Truevision Targe File Format).

JPGs are saved at quality 60%.

To change the default location where the screenshot file is saved (the default is Desktop), use the following Terminal command (where *pathname* is the full path to a directory.:

```
$ defaults write com.apple.screencapture location PathName
$ killall SystemUIServer
```

The normal default location would be reset with the following command (where *USername* is the current account's user name.

```
$ defaults write com.apple.screencapture location /Users/UserName/Desktop
$ killall SystemUIServer
```

command line screenshots

You can also take screenshots from Terminal.

I needed a screenshot of the CONTROL-TAB selection of a program, but in the graphic user interface, I couldn't simultaneously run Command-Tab and Command-Shift-4, so I used the following command in Terminal to set a 10 second delay and save the screenshot selection:

```
$ screencapture -T 10 -t png controltab.png
```

You can add this command to your Mac OS X scripts.

The format is `screencapture` *`options filenames`*. List more than one file name if you have more than one monitor. You can use the options in any combination.

You can use the *filename* to change the file name from the normal default and to set a relative path to a directory/folder of your choice.

```
$ screencapture [-icMPmwsWxSCUt] [files]
```

The basic use, which takes an immediate screenshot in the default format and stores it with the designated filename (in this case “Picture1”) in the user’s home directory (not the desktop).

```
$ screencapture Picture1
```

Force the screenshot to go to the clipboard (the equivalent of the Command-Shift-Control- choices).

```
$ screencapture -c [files]
```

Capture the cursor as well as the screen. This applies only in non-interactive modes (such as a script).

```
$ screencapture -C [files]
```

Display errors to the user graphically.

```
$ screencapture -d [files]
```

Capture the screenshot interactively by either selection or window (the equivalent of Command-Shift-4). Use the CONTROL key to cause the screenshot to go to the clipboard. Use the SPACE key to toggle between mouse selection and window selection modes. Use the ESCAPE key to cancel the interactive screen shot.

```
$ screencapture -i [file]
```

Use the `-m` option to only capture the main monitor. This does not work if the `-i` option is also set.

```
$ screencapture -m [file]
```

Send the screenshot to a new Mail message.

```
$ screencapture -M [files]
```

Use the `-o` option in window capture mode to only capture the window and to *not* capture the shadow of the window.

```
$ screencapture - o [file]
```

After savng the screenshot, open the screen capture output in Preview.

```
$ screencapture -P [files]
```

Use `-s` to only allow mouse selection mode.

```
$ screencapture -s [files]
```

Use `-w` to only allow window selection mode.

```
$ screencapture -w [file]
```

Use `-W` to start interaction in the window selection mode.

```
$ screencapture -W [file]
```

Use the `-S` option in window capture mode to capture the screen rather than the window.

```
$ screencapture -S [files]
```

Set the format with the `-t` option. The *Format* can be png (Portable Network Graphic), pdf (Portable Document Format), tiff (Tagged Image File Format), jpg or jpeg (Joint Photographic Experts Group), pict (Macintosh QuickDraw Picture), bmp (Microsoft Windows Bitmap), gif (Graphics Interchange Format), psd (Adobe Photoshop Document), sgi (Silicon Graphics File Format), or tga (Truevision Targe File Format)

```
$ screencapture -tFormat [files]
```

Set a delay time in seconds. The default is five seconds.

```
$ screencapture -TSeconds [files]
```

Prevent the playing of sounds (no camera click sound).

```
$ screencapture -x [files]
```

signals

chapter 84

summary

This chapter looks at UNIX and Linux signals.

The first chart shows the signals by name, with their explanation and their numbers by operating system. The second chart shows the signals by number, with the corresponding name by operating system.

signals chart by name

name	meaning	Linux	Mac OS X	Solaris
SIGABRT	Used by abort		6 create core image	6
SIGALRM	Alarm clock, real-time timer expired (POSIX)	14	14 terminate process	14
SIGBUS	BUS error (BSD 4.2)	7	10 create core image	10
SIGCANCEL	Thread cancellation signal used by libthread			36
SIGCHLD	Child process has stopped or exited, Child status change alias (POSIX)	17	20 discard signal	18
SIGCONT	Continue executing, if stopped; Stopped process has been continued (POSIX)	18	19 discard signal	25
SIGEMT	EMT instruction; Emulate instruction executed		7 create core image	7
SIGFPE	Floating point exception (ANSI)	8	8 create core image	8
SIGFREEZE	Special signal used by CPR			34
SIGHUP	Hangup (POSIX)	1	1 terminate process	1
SIGILL	Illegal instruction (ANSI)	4	4 create core image	4

SIGINFO	status request from keyboard		29 terminate process	
SIGINT	Terminal interrupt (ANSI)	2	2 terminate process	2
SIGIO	I/O now possible (BSD 4.2)	29	23 discard signal	22
SIGIOT	IOT Trap (BSD 4.2)	6	6 create core image now called SIGABRT	
SIGKILL	Kill (POSIX) (can't be caught or ignored)	9	9 terminate process	9
SIGLOST	Resource lost			37
SIGLWP	Special signal used by thread library			33
SIGPIPE	Write on a pipe with no reader, Broken pipe (POSIX)	13	13 terminate process	13
SIGPOLL	Pollable event occurred or Socket I/O possible			22
SIGPROF	Profiling alarm clock, profiling timer expired (BSD 4.2)	27	27 terminate process	29
SIGPWR	Power failure restart (System V)	30		19
SIGQUIT	Terminal quit (POSIX)	3	3 create core image	3
SIGRTMAX	Lowest priority real-time signal			45
SIGRTMIN	Highest priority real-time signal			38
SIGSEGV	Invalid memory segment access, Segmentation violation (ANSI)	11	11 create core image	11
SIGSTKFLT	Stack fault	16		
SIGSTOP	Stop executing (POSIX) (can't be caught or ignored)	19	17 stop process	23
SIGSYS	Bad argument to system call; Non-existent system call invoked		12 create core image	12
SIGTERM	Software termination (ANSI)	15	15 terminate process	15
SIGTHAW	Special signal used by CPR			35
SIGTRAP	Trace trap (POSIX)	5	5 create core image	5
SIGTSTP	Terminal stop signal, User stop requested from TTY (POSIX)	20	18 stop process	24
SIGTTIN	Background process trying to read from TTY control terminal (POSIX)	21	21 stop process	26
SIGTTOU	background process trying to write to TTY control terminal (POSIX)	22	22 stop process	27
SIGURG	Urgent condition on socket (BSD 4.2)	23	16 discard signal	21
SIGUSR1	User defined signal 1 (POSIX)	10	30 terminate process	16
SIGUSR2	User defined signal 2 (POSIX)	12	31 terminate process	17
SIGVTALRM	Virtual alarm clock, Virtual timer expired (BSD 4.2)	26	26	28

			terminate process	
SIGWAITING	Process' LWPs are blocked			32
SIGWINCH	Window size change (BSD 4.3, Sun)	28	28 discard signal	20
SIGXCPU	CPU limit exceeded (BSD 4.2)	24	24 terminate process	30
SIGXFSZ	File size limit exceeded (BSD 4.2)	25	25 terminate process	31

signals chart by number

number	Linux	Mac OS X	Solaris
1	SIGHUP	SIGHUP	SIGHUP
2	SIGINT	SIGINT	SIGINT
3	SIGQUIT	SIGQUIT	SIGQUIT
4	SIGILL	SIGILL	SIGILL
5	SIGTRAP	SIGTRAP	SIGTRAP
6	SIGIOT	SIGABRT	SIGABRT
7	SIGBUS	SIGEMT	SIGEMT
8	SIGFPE	SIGFPE	SIGFPE
9	SIGKILL	SIGKILL	SIGKILL
10	SIGUSR1	SIGBUS	SIGBUS
11	SIGSEGV	SIGSEGV	SIGSEGV
12	SIGUSR2	SIGSYS	SIGSYS
13	SIGPIPE	SIGPIPE	SIGPIPE
14	SIGALRM	SIGALRM	SIGALRM
15	SIGTERM	SIGTERM	SIGTERM
16	SIGSTKFLT	SIGURG	SIGUSR1
17	SIGCHLD	SIGSTOP	SIGUSR2
18	SIGCONT	SIGTSTP	SIGCHLD
19	SIGSTOP	SIGCONT	SIGPWR
20	SIGTSTP	SIGCHLD	SIGWINCH
21	SIGTTIN	SIGTTIN	SIGURG
22	SIGTTOU	SIGTTOU	SIGIO SIGPOLL
23	SIGURG	SIGIO	SIGSTOP
24	SIGXCPU	SIGXCPU	SIGTSTP
25	SIGXFSZ	SIGXFSZ	SIGCONT
26	SIGVTALRM	SIGVTALRM	SIGTTIN
27	SIGPROF	SIGPROF	SIGTTOU
28	SIGWINCH	SIGWINCH	SIGVTALRM
29	SIGIO	SIGINFO	SIGPROF
30	SIGPWR	SIGUSR1	SIGXCPU
31		SIGUSR2	SIGXFSZ

32			SIGWAITING
33			SIGLWP
34			SIGFREEZE
35			SIGTHAW
36			SIGCANCEL
37			SIGLOST
38			SIGRTMIN
45			SIGRTMAX

kernel modules

chapter 85

summary

This chapter looks at UNIX and Linux kernel modules.

Loadable Kernel Modules (LKM) go by many different names, including kernel loadable module (kld) in FreeBSD and kernel extension (kext) in Mac OS X, as well as Kernel Loadable Modules (KLM) or Kernel Modules (KMOD).

purpose

Loadable kernel modules allow the flexibility to add additional functionality to an operating system without requiring recompiling and reloading the entire operating system. This same flexibility can be used to choose between various options at start-up time.

Linux

Loadable Kernel Modules (LKMs) are loaded (and unloaded) by the `modprobe` command. Linux LKMs are located in `/lib/modules` and have the file extension `.ko` (for kernel object). Prior to Linux version 2.6 they were identified by the `.o` file extension.

Linux views LKMs as derived works of the kernel and allow symbols to be marked as only available to GNU General Public License (GPL) modules. Loading a proprietary or non-GPL-compatible LKM will set a “taint” flag in the running module.

LKMs run as part of the kernel, so they can corrupt kernel data structures and produce bugs.

FreeBSD

Kernel modules for FreeBSD are stored within `/boot/kernel/` for modules distributed with the OS, or usually `/boot/modules/` for modules installed from FreeBSD ports or FreeBSD packages, or for proprietary or otherwise binary-only modules. FreeBSD kernel modules usually have the extension `.ko`. Once the machine has booted, they may be loaded with the `kldload` command, unloaded with `kldunload`, and listed with `kldstat`. Modules can also be loaded from the loader before the kernel starts, either automatically (through /boot/loader.conf) or by hand.

Mac OS X

Some loadable kernel modules in OS X can be loaded automatically. Loadable kernel modules can also be loaded by the `kextload` command. They can be listed by the `kextstat` command. Loadable kernel modules are located in application

bundles with the extension `.kext`. Modules supplied with the operating system are stored in the `/System/Library/Extensions` directory; modules supplied by third parties are in various other directories.

binary compatibility

Linux does not provide a stable API or ABI for kernel modules. This means that there are differences in internal structure and function between different kernel versions, which can cause compatibility problems. In an attempt to combat those problems, symbol versioning data is placed within the .modinfo section of loadable ELF modules. This versioning information can be compared with that of the running kernel before loading a module; if the versions are incompatible, the module will not be loaded.

Other operating systems, such as Solaris, FreeBSD, Mac OS X, and Windows keep the kernel API and ABI relatively stable, thus avoiding this problem. For example, FreeBSD kernel modules compiled against kernel version 6.0 will work without recompilation on any other FreeBSD 6.x version, e.g. 6.4. However, they are not compatible with other major versions and must be recompiled for use with FreeBSD 7.x, as API and ABI compatibility is maintained only within a branch.

security

While loadable kernel modules are a convenient method of modifying the running kernel, this can be abused by attackers on a compromised system to prevent detection of their processes or files, allowing them to maintain control over the system. Many rootkits make use of LKMs in this way. Note that on most operating systems modules do not help privilege elevation in any way, as elevated privilege is required to load a LKM; they merely make it easier for the attacker to hide the break-in.

Linux allows disabling module loading via `/proc/sys/kernel/modules_disabled`. A `initramfs` system may load specific modules needed for a machine at boot and then disable module loading. This makes the security very similar to a monolithic kernel. If an attacker can change the `initramfs`, they can change the kernel binary.

On Mac OS X, a loadable kernel module in a kernel extension bundle can be loaded by non-root users if the `OSBundleAllowUserLoad` property is set to `True` in the bundle's property list. However, if any of the files in the bundle, including the executable code file, are not owned by root and group wheel, or are writable by the group or "other", the attempt to load the kernel loadable module will fail.

LAMP

chapter 86

summary

This chapter looks at LAMP, which is the combination of Linux-Apache-MySQL-PHP (or Perl or Python).

These are all open source programs that provide the ability to have a working modern production web server.

MAMP

The MAMP project achieves the same capabilities using Mac OS X instead of Linux. The MAMP package can be downloaded and used for free without interfering with any similar built-in Mac OS X services. MAMP is intended for creating a local duplicate of your production server environment so that you can freely test new software before deploying it to your production servers.

MAMP can be found at http://www.mamp.info/en/index.html.

WAMP

The WampServer project achieves the same capabilities using Windows instead of Linux. The WampServer package can be downloaded and used for free without interfering with any similar built-in Windows services. WampServer is intended for creating a local duplicate of your production server environment so that you can freely test new software before deploying it to your production servers.

WampServer can be found at http://www.wampserver.com/en/.

mysql

chapter 87

summary

This chapter looks at `mysql`, a Unix (and Linux) command.

`mysql` is the most common SQL database on Linux.

interactive queries

If you need to run a single interactive query on a particular MySQL database, use the `-e` flag.

```
$ mysql DBname - e "select * from table_foo;"
```

official keywords

The following is the list of the keywords for the official SQL standard:

ALL	DELETE	INTO	REFERENCES
AND	DESC	IS	ROLLBACK
ANY	DISTINCT	KEY	SCHEMA
AS	DOUBLE	LANGUAGE	SECTION
ASC	END	LIKE	SELECT
AUTHORIZATION	ESCAPE	MAX	SET
AVG	EXEC	MIN	SMALLINT
BEGIN	EXISTS	MODULE	SOME
BETWEEN	FETCH	NOT	SQL
BY	FLOAT	NULL	SQLCODE
CHAR	FOR	NUMERIC	SQLERROR
CHARACTER	FOREIGN	OF	SUM
CHECK	FORTRAN	ON	TABLE
CLOSE	FOUND	OPEN	TO
COBOL	FROM	OPTION	UNION
COMMIT	GO	OR	UNIQUE
CONTINUE	GOTO	ORDER	UDATE
COUNT	GRANT	PASCAL	USER
CREATE	GROUP	PLI	VALUES
CURRENT	HAVING	PRECISION	VIEW
CURSOR	IN	PRIMARY	WHENEVER
DEC	INDICATOR	PRIVILEGES	WHERE

DECIMAL	INSERT	PROCEDURE	WITH
DECLARE	INT	PUBLIC	WORK
DEFAULT	INTEGER	REAL	

PHP

chapter 88

summary

This chapter looks at `php`, a Unix (and Linux) command, including both running shell scripts from a web browser and running PHP from the shell.

`php` is used to run a CLI version of PHP.

php script example

This example assumes that you have created the `scripts` directory in your home directory.

Create a php script called `script.php` and save it in the new scripts directory (folder):

```
<?php
echo "Hello World!"
?>
```

Notice that we skip the `chmod` step typical for activating shell scripts.

Run your new script by running the `php` program with your script as the file to execute:

```
$ php ~/scripts/script.php
Hello World!
$
```

Note that the shell does *not* render HTML, so if you run a web scirpt, you will see raw HTML, CSS, and JavaScript as plain text in your terminal window.

You can run perl, ruby, and python scripts in the same manner.

running shell from PHP

You can run shell commands and shell scripts from PHP in Apache by using either the shell_exec function or backticks.

The shell_exec function in PHP takes a shell command as a string, runs that command (or command list) in shell, and returns the rsult as a string. You can assign the result to a string variable and then output those results to a web browser,

with or without additional processing. Use the <pre> tag to have the browser use the UNIX new line characters.

```
<php
$linesofoutput = shell_exec('ls');
echo '<pre>'.$linseofoutput.'</pre>';
?>
```

You can also use the backtick (or back quote) character (`).

```
<php
$linesofoutput = `ls`;
echo '<pre>'.$linseofoutput.'</pre>';
?>
```

You cannot combine both methods. This is useful, because you might actually want to send the backticks to the shell. If you are using backticks to run a shell command or command list, you can escape internal backticks to send them to the shell.

web security

Most system administrators disable shell_exec(). When shell_exec is disbled, the PHP script will be unable to run any shell commands.

If you enable shell access, you must guarantee that web visitor submitted input is never sent to the command shell without thorough scrubbing or crackers will be able to take over your server.

The difficulty with that “guarantee” is that there is probably a lot of PHP you didn’t write, such as WordPress, a PEAR package, off-the-shelf software, etc.

Hence the rule: disable shell_exec() and similar functions for any public facing server. Thanks to Stuart Beeton for emphasizing this point.

command line PHP

PHP is available both as a CGI binary (for Apache) and a CLI binary (for the shell).

Use the -v option to find out how your CLI copy of PHP is set up.

```
$ php -v
PHP 5.3.15 with Suhosin-Patch (cli) (built: Jul 31 2012 14:49:18)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
$
```

Notice that the above output indicated the cli version.

PHP defaults

The CLI version of PHP does not send out any HTTP headers, because the shell doesn’t understand them and instead expects plain text.

The CLI version of PHP does not change the working directory to that of the script. For a web server, you want to find all the files from the current web base. For a script running in the shel, you want the script or operator to make the decision.

This allows a PHP script running in the shell to be generic and simply run in the current working directory.

The CLI version of PHP has several *php.ini* options overridden by default. These changes are enforced after the *php.ini* file is run, so you can't change them in the configuration file, although you can change them during runtime.

- html_errors is FALSE(because the shell is cluttered by uninterpretted HTML tags.
- implicit_flush is TRUE because the shell should receive immediate outputs of print, echo, and similar commands rather than having them held in a buffer.
- max_execution_time is 0 (unlimited) because a shell environment might run very much more complex scripts than those in a typical web-based script.
- register_argc_argc is TRUE because the PHP script will need to process the command line arguments. The PHP variable `$argc` has the number of arguments passed and the variable `$argv` is an array of the actual arguments. These values are also in the `$_SERVER` array, as `$_SERVER['argv']` and `$_SERVER['argc']`.
- output_buffering is hardcoded FALSE< but the output buffering functions are available.
- max_input_time is FALSE because PHP CLI does not support GET, POST, or file uploads.

running a file in PHP

Use the `-f` option to parse and execute a designated file in PHP.

```
$ php -f filename.php
$
```

Note that the `-f` switch is optional. You will get the same result if you leave off the `-f` option and just give the filename. You get the exact same results from:

```
$ php filename.php
$
```

Note that the `.php` extension is for your convenience.The CLI PHP will run any file as if it were a PHP file, without regard to the extension type.

run PHP interactively

Use the `-a` switch to run PHP interactively. You will receive a secondary prompt and each line of PHP code will be run as soon as a `;` or `}` is found. Do not include the script tags <? and ?>.

```
$ php -a
asdf
$
```

run single line of PHP

Use the `-r` switch to run some particular PHP code immediately. This is useful inside of a script. Do not include the script tags <? and ?>.

```
$ php -r 'echo 'hello world\n'
hello world
$
```

You need to carefully watch your use of quoting and escaping to avoid problems with the command line variable substitution done by the shell.

syntax check

Use the `-l` switch to perform a syntax check on the PHP code.

```
$ php filename
No syntax error detected in <filename>
$
```

If there are no errors, you will get the message `No syntax errors detected in <filename >` on standard output and the shell return code is ***0***.

If there are no errors, you will get the message `Errors parsing <filename >` on standard output and the shell return code is ***-1***.

This option does not work with the `-r` option.

use PHP in a pipeline

PHP will accept PHP code from standard input (STDIN). Note this means that the previous commands are creating PHP code, not passing data for a PHP file to use as input.

```
$ some_application | some_filter | php | some_command > final_output.txt
$
```

combinations

You cannot combine any of the three ways of executing PHP code (file, -r, and STDIN). It would be great if there was a way to accept data from STDIN and name a PHP file to process it, with the output continuing as STDOUT to the next commadn in the shell pipeline, but this isn't currently a choice.

arguments

PHP can receive an unlimited number of arguments. Unfortunately, the shell has a limit on the number of total characters which can be passed to a command.

An argument that starts with a - will be used by the PHP interpreter. Use the argument list separator `--` and all of the arguments that follow will be sent to the PHP script through the `$argv` variable.

shebang

You can insert the shebang line with the correct location of the CLI PHP program and set the execution attrbutes of the file. The file needs the normal PHP starting and end tags.

Use the `type` command to find the location of your CLI copy of PHP.

```
$ type php
php is /usr/bin/php
$
```

Add that location to the standard shebang as the first line of the file.

```
#!/usr/bin/php
```

Set the file to execute:

```
$ chmod +x filename.php
$
```

The shebang line has no effect on Windows, but also causes no harm because it is also a valid PHP comment. Therefore include the shebang line for true cross-platform PHP scripts.

shebang example

Create a sample script file.

```
#!/usr/bin/php
<?php
var_dump($argv);
?>
```

Make the script file excutable and then run it.

```
$ chmod +x phptest.php
$ ./phptest.php -h -- foo
array(4) {
    [0]=>
    string(13) "./phptest.php"
    [1]=>
    string(2) "-h"
    [2]=>
    string(2) "--"
    [3]=>
    string(3) "foo"
}
$
```

Perl

chapter 89

summary

This chapter looks at `perl`, a Unix (and Linux) command, including both running shell scripts from a web browser and running Perl from the shell.

`perl` is used to run a CLI version of Perl.

getting started

Check that Perl is actually running on your system by typing `perl -v` from the command line. Your exact results may vary.

```
    $ perl -v

    This is perl 5, version 12, subversion 4 (v5.12.4) built for darwin-thread-multi-
2level
    (with 2 registered patches, see perl -V for more detail)

    Copyright 1987-2010, Larry Wall

    Perl may be copied only under the terms of either the Artistic License or the
    GNU General Public License, which may be found in the Perl 5 source kit.

    Complete documentation for Perl, including FAQ lists, should be found on
    this system using "man perl" or "perldoc perl". If you have access to the
    Internet, point your browser at http://www.perl.org/, the Perl Home Page.

    $
```

Use `which` to determine the location of your local copy of Perl:

```
    $ which perl
    This is perl 5, version 12, subversion 4 (v5.12.4) built for darwin-thread-multi-
2level
    /usr/bin/perl
    $
```

This example assumes that you have created the `scripts` directory in your home directory.

Create a Perl script called `script.pl` and save it in the new scripts directory (folder):

```
#!/usr/bin/perl
print "Hello World!";
```

Notice that we skip the `chmod` step typical for activating shell scripts.

Run your new script by running the `perl` program with your script as the file to execute:

```
$ perl ~/scripts/script.pl
Hello World!
$
```

brief history

Perl created by Larry Wall in 1986. Perl was originally intended as a utility to create reports. Perl stands for Practical Extraction and reporting Language.

Tcl

chapter 90

summary

This chapter looks at `tcl`, a Unix (and Linux) command, including both running shell scripts from a web browser and running Tcl from the shell.

`tcl` is used to run a CLI version of Tcl.

getting started

Use `which` to determine the location of your local copy of Tcl (and to make sure you actually have a copy):

```
$ which tclsh
/usr/bin/tclsh
$
```

This example assumes that you have created the `scripts` directory in your home directory.

Create a tcl script called `script.tcl` and save it in the new scripts directory (folder):

```
#!/usr/bin/tclsh
puts "Hello World!"
```

Run your new script by running the `tclsh` program with your script as the file to execute:

```
$ tclsh ~/scripts/script.tcl
Hello World!
$
```

text-based interactive version

To run a text-only version of `tclsh`, simply type `tclsh` (and return) from the BASH command line and you should get a percent sign (`%`) prompt. This means you have sccessfully entered `tclsh`.

```
$ tclsh
%
```

Now that you are in `tclsh`, type `info tclversion` to see which version.

```
% info tclversion
8.5
%
```

And to see Tcl make use of a simple BASH command, type `ls -l`:

```
% ls -l
a regular listing of files
%
```

And to return back to BASH, type `exit`:

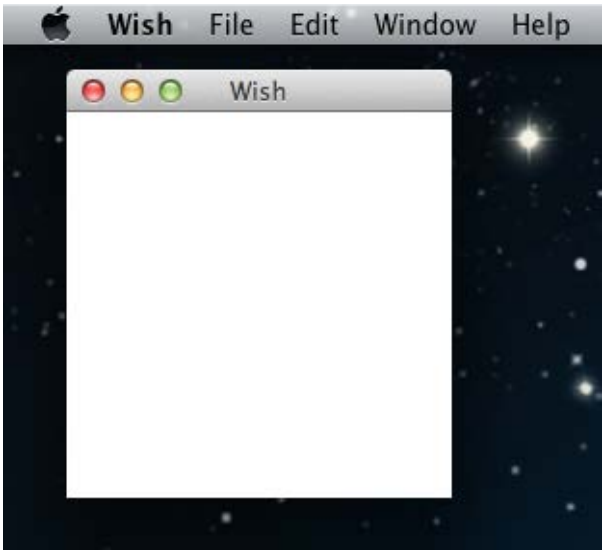
```
% exit
$
```

visual interactive version

Now test out the visual version by typing `wish` (assuming it is installed and you have X-Windows running).

```
$ wish
%
```

The contents of your Terminal window will not change from the text-only version, but you will get an additional display window that is controlled from the command line.



And return to BASH:

```
% exit
$
```

brief history

Tcl was created in the late 1980s by Dr. John Osterhout. Tcl stands for Tool Command Language.

installing software from source code

chapter 91

summary

This chapter looks at installing software from source code.

While it is possible to do a lot with just preinstalled software and precompiled software, those working with large systems will sooner or later need to download the raw source code and use that source code to create their own copy of the latest version of some software on their servers or other computer systems.

gcc

The vast majority of open source software is written in the C programming language. The GNU foundation C compiler, known as `gcc`, is the most commonly used C compiler.

Apple's Mac OS X Developer Tools include a copy of `gcc`, but also include an Apple specific variation that includes specialized features useful for development of Macintosh, iPhone, and iPad software.

Xcode

Xcode is an Apple integrated development environment. It includes the Instruments analysis tool, iOS Simulator (useful for testing your software on the Mac before it is ready for deployment to real devices), and the latest Mac OS X and iOS SDKs.

Fink

Fink (German for the bird finch) is an open source program that automates the process of downloading, configuring, compiling, and installing hundreds of software packages from the original source code.

manual installation

There are two major methods for manually installing software from source: the standard method (used for most Linux software) and the Perl method (used for Perl modules).

programming

chapter 92

summary

This chapter is intended as an introduction to computer programming.

Programming is problem solving and writing instructions for a computer.

The principles of programming are independent of the computer programming language used. Different languages have different strengths and weaknesses, making some kinds of programs easier or more difficult to write, but the basic principles remain the same regardless of language.

A skilled programmer should be able to switch to a new programming language in a few hours.

On the other hand, beginners should pick one language and learn it before attempting a second language.

This free text book includes information on multiple programming languages. Unless instructed otherwise, you should concentrate on the language you are learning and skip over the others. Trying to learn the syntax and semantics of multiple programming languages at the same time as learning the basics of programming is a recipe for utter confusion.

> "As long as programmers cherish their freedom not only to design their own clever software, but also to modify adopted software according to their likings, a proper design discipline remains unlikely. And as long as companies secretly cherish complexity as an effective protection against being copied, there is little hope for dramatic improvements of the state of the art." —Niklaus Wirth

programming as a skill

I would like to strongly suggest that learning the principles of computer programming is far more valuable than learning a specific programming language.

> "Hire a programmer. He's a programmer. You don't need to hire a Java programmer. By the way, you should do that now anyway. Do not hire Java programmers, not because Java programmers are bad. But you should not be asking, "excuse me, what language do you program in? We're only hiring Java programmers here." Learning a language is a matter of a week. Oh, and to get skillful in the platform is a matter of another week or two or three or a month. Finding someone who actually knows how to write code, that's the hard problem, so you don't ask, "you must know Sprint." No, you do not have to know Sprint. "you must know JavaScript." No, you don't have to know JavaScript. "Can you write code?" Show me code in any language. If you can write code, I'll hire you." —Robert C. Martin, *The Last Programming Language*, approximately 37:09

> "Learning how to code in any one particular [computer] language is not going to be worthwhile beyond 10 or 20 years. Learning how to problem-solve using algorithms and how technology works and how it's built is going to last a century at least," —Hadi Partovi, cofounder of code.org, *For some, learning coding is a calculated strategy*, *Los Angeles Times* , Business Section, Saturday, August 2, 2014, p. B1

other

> "7. It is easier to write an incorrect program than understand a correct one." —Alan Perlis, *Epigrams on Programming*, ACM's SIGPLAN Notices Volume 17, No. 9, September 1982, pages 7-13

“93. When someone says ‘I want a programming language in which I need only say what I wish done,’ give him a lollipop.” —Alan Perlis, *Epigrams on Programming*, ACM’s SIGPLAN Notices Volume 17, No. 9, September 1982, pages 7-13

size of programs

chapter 93

summary

The educational goal of this chapter is to introduce the concept of differences in scale or size of programming projects and how the size and complexity of a program impacts the task of programming.

Programs vary in size. Bad habits learned with small or trivial programs might get you into real trouble with medium or large programs.

size of programs

Programs are generally divided into three basic sizes: trivial, small, and large.

Trivial programs are programs that a skilled programmer can write in less than two days of coding.

Small programs are programs that one skilled programmer can write in less than one year of full time work.

Large programs are programs that require more than two to five man-years of labor, normally written by programming teams (which can exceed 1,000 skilled workers).

These estimates are approximate and there are obvious gaps in the gray zone between the sizes. Further, there can be huge differences in individual abilities.

Larry Ellison wrote the first version of Oracle database by himself in about six months. That is a genius exception. Data bases typically take large teams (sometimes hundreds of programmers) at least a year.

Bill Gates, copying and pasting from the source code of three working open source versions, took more than six months to create a bug-filled BASIC compiler and then hired a team of six skilled programmers who spent more than six more months to get rid of enough bugs to make the compiler somewhat usable (a total of more than three man-years). That is an idiot exception. A BASIC compiler typically takes a skilled programmer a few hours to create. Note that Bill Gates takes credit for quickly having created a BASIC compiler, but according to other sources he was sued for having illegally used open source code for commercial purposes, forcing him to spend a great deal of time attempting to do a project that many programmers of the day could successfully finish in hours.

impact on good programming practices

Almost every program assigned in a class setting will be trivial, simply because there isn't enough time in a quarter or semester for longer programs.

Each programming assignment will concentrate on one or a small number of specific programming concepts.

The artificial nature of school programming assignments cause most students to question the utility of modern programming practices, especially the time and effort spent on form and documentation.

These practices are the result of decades of real world programming.

Successful programs tend to have a long lifetime. Programmers will have to look at existing source code, figure out what is going on, and then correctly modify the program to add new features or update existing features to meet changing real world conditions.

Unix style

The Unix philosophy on programming is to encourage a lot of small tools (programs) that work together to complete a task (usually threaded together with a shell script). This bottom-up approach to programming encourages the reuse and sharing of software. The typical Unix (or LINUX) distribution includes hundreds of small standardized tools that allow a skilled Unix prorgammer (or even a skilled Unix administrator or user) to immediately do useful large scale work.

object-oriented programming

Many students openly question why they are going to so much extra work to create object-oriented projects when procedural programming will clearly be sufficient and involve less work.

The simple answer is that object-oriented programming is one of the few proven successful methods for cutting costs and improving reliability of large scale real-world projects.

kinds of programs

chapter 94

summary

The educational goal of this chapter is to introduce the basic different kinds of programming in common use.

The two major kinds of programming are systems programming and applications programming.

list of kinds of programming

There are two basic kinds of programming: system and application.

System programming deals with the use of a computer system. This includes such things as the operating system, device drivers for input and output devices, and systems utilities.

Application programming deals with the programs directly used by most people.

Application programming is generally divided further into scientific and business programming.

Scientific programming is work in the scientific, engineering, and mathematical fields. Often the programmers are the researchers who use the programs.

Business programming is work in the data processing field, including large scale business systems, web-based businesses, and office applications. It is exceedingly rare for these kinds of programs to be created by the person who uses them.

Another large category of programming is programming for **personal or home** use. This includes **games**. Historically, many advances in computer science occurred in the development of computer and video games.

Embedded systems are programs that are built into specific hardware, such as the computer systems in an automobile or microwave oven. These programs combine features of operating systems and application program into a single monolithic system.

Scripting is a simplified version of programming originally intended for use primarily by non-programmers. In practice, most non-programmers have trouble with scripting languages. Some professional programmers have built very useful, sometimes intricate systems using scripting languages, especially those contained in office software (such as word processors or spreadsheets).

programming languages

chapter 95

summary

The educational goals of this chapter are to familiarize the student with the kinds of programming languages and the basic tools for programming. Key terminology is introduced (the student should know the definition of terms in bold). This is followed by some C specific information from the the Stanford CS Education Library.

Programming languages vary from low level assemblers to high level languages.

> "19. A language that doesn't affect the way you think about programming, is not worth knowing." —Alan Perlis, *Epigrams on Programming*, ACM's SIGPLAN Notices Volume 17, No. 9, September 1982, pages 7-13

direct programming

Originally computers were programmed directly in a "language" that the computer understood.

This direct programming could involve directly wiring the program into the computer. In some cases, this involved a soldering iron. In other cases there was some kind of plug-board ot make it easier to change the programmed instructions. This method was known as **hard wiring**.

Large telegraph networks and later large telephone networks became so complex as to essentially be a computer on a system-wide basis. Many of the ideas (especially logic circuits) that were later necessary to create computers were first developed for large scale telegraph and telephone systems.

In some early computers the programming could be accomplished with a set of switches. The use of **front panel** switches (and corresponding indicator lights) continued as an option on many mainframe and minicomputer systems. Some microcomputer systems intended for hobbyists and for dedicated systems also had some kind of front panel switches.

Another method was the use of **punched cards**. This was a technology originally developed for controlling early industrial age factories, particularly large weaving looms. The designs or patterns for the cloth would be programmed using punched cards. This made it easy to switch to new designs. Some of the large looms became so complex that they were essentially computers, although that terminology wasn't used at the time.

machine code and object code

Both the front panel switch and the punched card methods involved the use of numeric codes. Each numeric code indicated a different machine instruction. The numbers used internally are known as **machine code**. The numbers on some external media, such as punched cards (or disk files) are known as **object code**.

assembly and assemblers

One of the early developments was a **symbolic assembler**. Instead of writing down a series of binary numbers, the programmer would write down a list of machine instructions, using human-readable symbols. A special program, the **assembler**, would convert these symbolic instructions into object or machine code.

Assembly languages have the advantage that they are easier to understand than raw machine code, but still give access to all of the power of the computer (as each assembler symbol translates directly into a specific machine instruction).

Assembly languages have the disadvantage that they are still very close to machine language. These can be difficult for a human to follow and understand and time-consuming for a human to write. Also, programs written in assembly are tied to a specific computer hardware and can't be reused on another kind of computer.

The human readable version of assembly code is known as **source code** (it is the source that the assembler converts into object code). All programs written in high level languages are also called source code.

high level languages

High level languages are designed to be easier to understand than assembly languages and allow a program to run on multiple different kinds of computers.

The source code written in high level languages needs to be translated into object code. The two basic approaches are compilers and interpetters. Some programming languages are available in both interpretted and compiled versions.

High level languages have usually been designed to meet the needs of some particular kind of programming. For example, FORTRAN was originally intended for scientific programming. COBOL was originally intended for business programming and data processing. SQL was originally intended for data base queries. C was originally intended for systems programming. LISP was originally intended for list processing. PHP was originally intended for web scripting. Ada was originally intended for embedded systems. BASIC and Pascal were originally intended as teaching languages.

Some high level languages were intended to be general purpose programming languages. Examples include PL/I and Modula-2. Some languages that were originally intended for a specific purpose have turned into general purpose programming languages, such as C and Pascal.

compilers

Compilers convert a finished program (or section of a program) into object code. This is often done in steps. Some compilers convert high level language instructions into assembly language instructions and then an assembler is used to create the finished object code.

Some compilers convert high level language instructions into an **intermediate language**. This intermediate language is platform-independent (it doesn't matter which actual computer hardware is eventually used). The intermediate language is then converted into object code for a specific kind of computer. This approach makes it easier to move (or **port**) a compiler from one kind of computer to another. Only the last step (or steps) need to be rewritten, while the main complier is reused.

Compiled code almost always runs faster than interpretted code. An **optimizing compiler** examines a high level program and figures out ways to optimize the program so that it runs even faster.

C

A C program is considered to be **strictly conforming** to ANSI C if the program only uses features and libraries as they are described in the ANSI standard (with no additional optional features or extensions).

A **conforming hosted implementation** accepts any strictly conforming program. This applies to a program that is intended to run on an operating system.

A **conforming freestanding implementation** accepts any strictly conforming program that doesnt use any library facilities other than those in the header files `float.h`, `limits.h`, `stdarg.h`, and `stdef.h`.. This applies to a program that is intended to run on an embedded system or other environment with minimal operating system support (such as no file system)..

compilers and assemblers

A brief explanation of the difference between compilers and assemblers.

An assembler converts symbolic (human readable text) instructions into their corresponding machine or object instructions. There is generally a one-to-one correspondence between assembler instructions and machine instructions (at the macro-machine level).

A compiler converts a high level language into machine or object code. Typically there are many machine instructions for every high level language instruction. There are some exceptions — some older languages, such as COBOL and FORTRAN, had several instructions that translated directly into a single machine instruction, but even in those cases, most of the useful portions of the language were translated into many machine instructions.

An example from the C programming language:

```
    if (x==0) z=3; /* test to see if x is zero, if x is zero, then set z to 3 */
```

The same example in 8080 assembly language (everything after the semicolon ; is a comment to help you understand):

```
        LXIH $E050 ; point to location of variable x (Load Double Immediate into HL
register pair)
        MOVAM ; load value of x into the accumulator (MOVe to A register from Memory)
        CMPA ; test the value of the accumulator (CoMPare A register with itself)
        JNZ @1 ; if not zero, then skip variable assignment (Jump is Not Zero)
        MVIA #3 ; load three (new value for variable z) into accumulator (MoVe Immediate
into A register the number three)
        LXIH $E060 ; point to location of variable z (Load Double Immediate into HL
register pair)
        MOVMA ; store three into variable z (MOVe to Memory from A register)
    @1  NOP ; drop through or skip to here to continue program (No OPeration)
    DS  $E050 ; reserve memory for variable x (Data Storage)
    DS  $E060 ; reserve memory for variable z (Data Storage)
```

The same example in 8080 machine code (the comments after the semicolon ; wouldn't normally be included, but are added to help you follow along, in a real case of object code it would be just binary/hexadecimal numbers):

```
    21 ; LXIH
    50 ; data address
    E0 ; data address
    7E ; MOVAM
    BF ; CMPA
    C2 ; JNZ
    0D ; code address
    00 ; code address
    1E ; MVIA
    03 ; data
    21 ; LXIH
    60 ; data address
    E0 ; data address
    77; MOVMA
    00 ; NOP
```

and later in memory: the data storage

```
    $E050 ; variable x, unknown contents
    $E060 ; variable y, becomes three (3)
```

You will notice that there is one machine instruction for each assembly instruction (some instructions are followed by data or addresses), while there are many assembly or machine instructions for one C instruction.

linkers

As programs grow in size, requiring teams of programmers, there is a need to break them up into separate files so that different team members can work on their individual assignments without interfering with the work of others. Each file is compiled separately and then combined later.

Linkers are programs that combine the various parts of a large program into a single object program. Linkers also bring in support routines from **libraries**. These libraries contain utility and other support code that is reused over and over for lots of different programs.

Historically, linkers also served additional purposes that are no longer necessary, such as resolving relocatable code on early hardware (so that more than one program could run at the same time).

loaders

A **loader** is a program that loads programs into main memory so that they can be run. In the past, a loader would have to be explicitely run as part of a job. In modern times the loader is hidden away in the operating system and called automatically when needed.

interpreters

Interpreters convert each high level instruction into a series of machine instructions and then immediately run (or execute) those instructions. In some cases, the interpreter has a library of routines and looks up the correct routine from the library to handle each high level instruction.

Interpreters inherently run more slowly than the same software compiled. In the early days of computing this was a serious problem. Since the mid-1980s, computers have become fast enough that interpreters run fine for most purposes.

Most of the scripting languages common on the web and servers are intereted languages. This includes JavaScript, Perl, PHP, Python, Ruby.

Note that some modern programming languages (including Java and Python) translate the raw text into an intermediate numeric code (usually a byte code) for a virtual machine. This method is generally faster than older traditional methods of interpreting scripts and has the advantage of providing a pkatform-independent stored code.

editors

An **editor** is a program that is used to edit (or create) the source files for programming. Editors rarely have the advanced formatting and other features of a regular word processor, but sometimes include special tools and features that are useful for programming.

Two important editors are emacs and vi from the UNIX world. I personally use Tom Bender's Tex-Edit Plus, which is available in multiple different languages (Danish, English, French, German, Italian, Japanese, Spanish).

command line interface

A **command line interface** is an old-style computer interface where the programmer (or other person) controls the computer by typing lines of text. The text lines are used to give instructions (or commands) to the computer. The most famous example of a command line interface is the UNIX shell.

In addition to built-in commands, command line interfaces could be used to run programs. Additional information could be passed to a program, such as names of files to use and various "program switches" that would modify how a program operated.

See the information on how to use the shell.

development environment

A **development environment** is an integrated set of programs (or sometimes one large monolithic program) that is used to support writing computer software. Development environments typically include an editor, compiler (or compilers), linkers, and various additional support tools. Development environments may include their own limited command line interface specifically intended for programmers.

The term “development environment” can also be used to mean the collection of programs used for writing software, even if they aren’t integrated with each other.

Because there are a huge number of different development environments and a complete lack of any standardization, the methods used for actually typing in, compiling, and running a program are *not* covered by this book. Please refer to your local documentation for details.

The development environment for UNIX, Linux, and Mac OS X are discussed in the chapter on shell programming.

Stanford introduction

Stanford CS Education Library This [the following section until marked as end of Stanford University items] is document #101, Essential C, in the Stanford CS Education Library. This and other educational materials are available for free at http://cslibrary.stanford.edu/. This article is free to be used, reproduced, excerpted, retransmitted, or sold so long as this notice is clearly reproduced at its beginning. Copyright 1996-2003, Nick Parlante, nick.parlante@cs.stanford.edu.

The C Language

C is a professional programmer’s language. It was designed to get in one’s way as little as possible. Kernighan and Ritchie wrote the original language definition in their book, *The C Programming Language* (below), as part of their research at AT&T. Unix and C++ emerged from the same labs. For several years I used AT&T as my long distance carrier in appreciation of all that CS research, but hearing “thank you for using AT&T” for the millionth time has used up that good will.

Some languages are forgiving. The programmer needs only a basic sense of how things work. Errors in the code are flagged by the compile-time or run-time system, and the programmer can muddle through and eventually fix things up to work correctly. The C language is not like that.

The C programming model is that the programmer knows exactly what they want to do and how to use the language constructs to achieve that goal. The language lets the expert programmer express what they want in the minimum time by staying out of their way.

C is “simple’ in that the number of components in the language is small-- If two language features accomplish more-or-less the same thing, C will include only one. C’s syntax is terse and the language does not restrict what is “allowed” -- the programmer can pretty much do whatever they want.

C’s type system and error checks exist only at compile-time. The compiled code runs in a stripped down run-time model with no safety checks for bad type casts, bad array indices, or bad pointers. There is no garbage collector to manage memory. Instead the programmer mangages heap memory manually. All this makes C fast but fragile.

Analysis -- Where C Fits

Because of the above features, C is hard for beginners. A feature can work fine in one context, but crash in another. The programmer needs to understand how the features work and use them correctly. On the other hand, the number of features is pretty small.

Like most programmers, I have had some moments of real loathing for the C language. It can be irritatingly obedient -- you type something incorrectly, and it has a way of compiling fine and just doing something you don't expect at run-time. However, as I have become a more experienced C programmer, I have grown to appreciate C's straight-to-the point style. I have learned not to fall into its little traps, and I appreciate its simplicity.

Perhaps the best advice is just to be careful. Don't type things in you don't understand. Debugging takes too much time. Have a mental picture (or a real drawing) of how your C code is using memory. That's good advice in any language, but in C it's critical.

Perl and Java are more "portable" than C (you can run them on different computers without a recompile). Java and C++ are more structured than C. Structure is useful for large projects. C works best for small projects where performance is important and the progammers have the time and skill to make it work in C. In any case, C is a very popular and influential language. This is mainly because of C's clean (if minimal) style, its lack of annoying or regrettable constructs, and the relative ease of writing a C compiler.

Other Resources

The C Programming Language, 2nd ed., by Kernighan and Ritchie. The thin book which for years was the bible for all C programmers. Written by the original designers of the language. The explanations are pretty short, so this book is better as a reference than for beginners.

end of Stanford introduction

standards and variants

chapter 96

summary

The educational goal of this chapter is to make the student aware that there are offcial versions of many programming languages, but that in practice there are a lot of variations (and that a programmer must be able to adapt to changes and variations).

Programming languages can meet official standards or come in variants and dialects.

standards and variants

Programming languages have traditionally been developed either by a single author or by a committee.

Typically after a new programming language is released, new features or modifications, called **variants**, start to pop-up. The different versions of a programming language are called **dialects**. Over time, the most popular of these variants become common place in all the major dialects.

If a programming language is popular enough, some international group or committee will create an official **standard** version of a programming language. The largest of these groups are **ANSI** (Ameican national Standards Institute) and **ISO** (International Orgnaization for Standardization).

While variants and dialects may offer very useful features, the use of the non-standard features will lock the program into a particular development environment or compiler and often will lock the program into a specific operating system or even hardware platform.

Use of official standaards allows for **portability**, which is the ability to move a program from one machine or operating system to another.

While variants were traditionally introduced in an attempt to improve a programming language, Microsoft started the practice of intentionally creating variants to lock developers into using Microsoft products. In some cases the Microsoft variants offered no new features, but merely chaanged from the established standard for the sake of being different. Microsoft lost a lawsuit with Sun Microsystems for purposely creating variants to Java in hopes of killing off Java in favor of Microsoft languages.

test bed

chapter 97

summary

This chapter looks at a test bed for writing this book.

At the time of this writing, this test bed does not yet exist. The author was asked to put together a list of test bed requirements, emphasizing saving money as much as possible. Rather than keep this test bed as a secret, it seems that it would be a good idea to share this information with the readers. It also makes sense to share with the reader the process of installing each component of the test bed.

Thank you.

stages

There are five basic stages of deployment of the test bed.

1. **initial main development computer** This is a low cost used computer to get work started, as well as the additional essential items to support that development.
2. **initial server machines** These are the first and most important machines for testing server deployment.
3. **mobile computing** These are the items needed for creating and testing mobile apps.
4. **content creation** These are the items needed for full content creation. These will be divided into additional stages.
5. **additional servers** These are the additional desired test platforms for server and desktop testing.

initial main development computer

The items in this section are the absolute minimums to be able to do any useful work. This is *not* enough to do all of the important aspects of the job, but is enough to get started and to do very important work.

This is a low cost used computer to get work started, as well as the additional essential items to support that development.

The initial main development computer is older technology chosen to greatly lower costs, yet still provide a useful platform for development.

Hardware: Note that this is available only as a used computer — Apple Power Mac G5 “Cipher”:

- **Processor:** DP DC “Quadcore“ 2.5 GHz Power PC 970MP (G5)
- **Cooling:** Panasonic LCS
- **Graphics:** GeForce 7800 GT with 512 MB of DDR RAM
- **Memory:** 16GB of 533 MHz PC2-4200 DDR2 SDRAM
- **Hard Drive:** Two of 500GB 7200 rpm drives
- **Wireless:** AirPort Extreme with Bluetooth 2.0+EDR combo card
- **Optical Drives:** Two of 16x SuperDrive (DVD+R DL/DVD±RW/CD-RW)
- **OS:** Mac OS X 10.5.8 Leopard
- **Monitor:** Apple Cinema Display (30-inch DVI)
- **Input:**Apple extended keyboard and Mighty Mouse

I will supply a set of USB speakers and a USB 1.0 hub. If you can provide a modern USB 3.0 and FireWire 800 hub, that would be very useful.

The obvious question is why does it have to be a Macintosh, because business persons hate Macintosh.

The answer is that the Macintosh is the only computer that allows access to the UNIX command line and all of the UNIX tools, including a wide array of programming languages and the BASH sheel, and at the same time allows access to the mainstream content creation software. Adobe software is not available for Linux (other than viewers). Final Cut Pro, Logic Pro, and other mainstream content creation software is only available on Macintosh.

The use of the BASH shell and UNIX compilers (and other tools) is absolutely essential for this work. .NET and other Microsoft proprietary software is useless for building open systems. The Microsoft development software is only useful for building on Microsoft proprietary systems.

Photoshop won't run on Linux. The other listed content creation software won't run on Linux. Several of the programs listed won't run on Widnows or Linux, only on Macintosh. GIMP is not an acceptable substitute for everyday graphics work. GIMP is great for automated content creation on a server, but it is very poor for direct human use.

Software: Note that most of this software is only available used.

- Tex-Edit Plus (new-shareware)
- Fetch (new)
- Apple Developer Tools (download free from Apple - you will have to apply and pay a fee to Apple)
- Adobe Creative Suite 4 Design Premium, including Adobe InDesign CS4, Photoshop CS4 Extended, Illustrator CS4, Flash CS4 Professional, Dreamweaver CS4, Fireworks CS4, and Acrobat 9 Pro with additional tools and services (only available used)
- Apple Final Cut Pro 5 (only available used)
- Apple Logic Pro 8 (only available used)
- Apple QuickTime Pro
- Fink (free download)

Note that most of this software is only available used.

Other: These are the additional items that are essential.

A place indoors to work. The place must have a lock. I must be able to get in and out on a 24-hour a day basis, including any holidays. And the place must be within a reasonable walking distance of 17th and Newport in Costa Mesa.

Moving on, I also need electricity.

Internet connections. There is a need for two independent internet connections that come from two unrelated sources. Getting two different internet providers that both lease their lines from the phone company does *not* count. It really needs to be two completely unrleated sources (such as DSL or U-verse from the phone company and cable modem from the cable company. It is vital to have two different sources, because internet connections do invariably go down.

I need a chair to sit in. A folding chair is not acceptable because it causes too much back pain after several hours.

I need a stable flat surface to work on. It does not have to be a desk. It can be a table.

Starting in October I need a reliable $2 an hour (after taxes) for a 40 hour work week. I can work all overtime for free.

I understand that the minimum wage is completely unfair to business owners. I am not asking for the minimum wage.

Optional: It would be useful to have a place to obtain a shower. It would be useful to have the ability to store and cook foods. None of this is essential and it is probably very bad business to spend the extra money, but it would increase productivity. It would also be useful to have a pair of glasses that I can see clearly through.

initial server machines

These are the first and most important machines for testing server deployment.

Note that there is the intent to build a quad boot computer that has Macintosh, FreeBSD, Linux, and Windows all on one hard drive. So, this will keep costs down, because it is one computer rather than four. And the steps for completing this work will be written down and provided to the public as a how-to guide. A friend, Joe, needs to be paid for creating the quad-boot partioning.

FreeBSD computer

A computer set up with FreeBSD.

Linux Mint computer

A computer set up with Linux Mint.

OpenVMS computer

An itanium computer set up with a hobbyist licensed copy of OpenVMS. Alternatively, an old Alpha or old VAX can also be set up with a hobbyist licensed copy of OpenVMS. Given the lower cost for a used machine, the Alpha version is probably the wisest, but if an Alpha or VAX is used, we will still need the modern itanium version for the later step of additional servers.

Windows computer

Need to find someone willing to work in the horrid Windows environment who can check that the software works correctly in that environment. Microsoft’s monopolistic approach may prevent deployment of system-independent software.

mobile computing

These are the items needed for creating and testing mobile apps.

development computer

Hardware: This becomes the primary development computer (although the G5 development computer will remain in use for testing and other purposes). This is a 15-inch MacBook Pro with Retina Display

- **Processor:** 2.7 GHz Quad-core Intel Core 17
- **Memory:** 16GB 1600MHz DDR3L SDRAM
- **Storage:** Internal 768GB Flash Storage; External Promise Pegasus 12TB (6x2TB) R6 RAID System
- **Optical:** External USB SuperDrive
- **Display:** Internal: 15-inch Retina Display; external: Apple 30 inch Cinema HD Display
- **Keyboard:** Backlit Keyboard (English) and User’s Guide (English)
- **Software:** Preinstalled from Apple: Keynote, Pages, Numbers
- **Accessories:** Apple Thunderbolt to Gigabit Ethernet Adaptor; Mini DisplayPort to VGA Adaptor; Apple Thunderbolt cable; Thunderbolt to FireWire 800 adaptor; Apple Mighty Mouse

Software: This is additional important software.

- Tex-Edit Plus (new-shareware)
- Fetch (new)
- Apple Developer Tools (download free from Apple - you will have to apply and pay a fee to Apple)
- Adobe CS6 Master Collection: Photoshop CS6 Extended; Illustrator CS6; InDesign CS6; Acrobat X Pro; Flash Professional CS6; Flash Builder 4.6 Premium Edition; Dreamweaver CS6; Fireworks CS6; Adobe Premiere Pro CS6; After Effects CS6; Adobe Audition CS6; SpeedGrade CS6; Prelude CS6; Encore CS6; Bridge CS6; Media Encoder

CS6
- Apple Final Cut Pro X (with Apple Motion 5 and Apple Compressor 4) and Final Cut Pro 7 (only available used)
- Apple Logic Studio
- Apple QuickTime 7 Pro for Mac OS X
- Apple QuickTime MPEG-2 Playback Component for Mac OS X
- Unsanity FruitMenu
- Unsanity Labels X
- Unsanity MenuMaster
- Unsanity WindowShades
- Fink (free download)

iPhone

An iPhone 5 and development system.

Android

An Android and development system.

iPad

An iPad and development system.

BlackBerry

A BlackBerry and development system.

Windows Phone

A Windows Phone 7 and development system.

Nokia

A Nokia and development system.

content creation

These are the items needed for full content creation. These will be divided into additional stages.

The content creation computer needs to be rented or leased, because a significant upgrade is expected early 2013 and that is when we need to purchase the long term content creation computer. In the meantime, we need to purchase an external FireWire drive and a Thunderbolt converter. We will do our work from the FireWire drive and then be easily able to plug it in to the new computer. Note that this main work drive is in addition to working storage drive(s) and backup drive(s).

additional servers

These are the additional desired test platforms for server and desktop testing. There is no particular order implied at this time, other than getting to Red hat Enterprise, Ubuntu, and Solaris as soon as possible. We also want an IBM Power or Z system (even if it is an old one) with FORTRAN, PL/I, COBOL, and C compilers, as well as IBM’s server and system administration software.

Red Hat Enterprise Linux computer

A computer set up with Red Hat Enterprise Linux.

Ubuntu Linux computer

A computer set up with Ubuntu Linux.

Solaris computer

An UltraSPARC computer set up with Solaris and Oracle Database.

Fedora Linux computer

A computer set up with Fedora Linux.

OpenSUSE Linux computer

A computer set up with OpenSUSE Linux.

CentOS Linux computer

A computer set up with CentOS Linux.

Oracle Linux computer

A computer set up with Oracle Linux and Oracle Database.

Debian Linux computer

A computer set up with Debian Linux.

IBM zEnterprise EC12 computer

An IBM zEnterprise EC12 model HA1 computer running z/OS with Enterprise PL/I, FORTRAN, Enterprise COBOL,

and XL C/C++ compilers and APL interpreter. The HA1 with 101 processor units in 4 books, which include Central Processor (CP), Internal Coupling facility (ICF), Integrated facility for Linux (IFL), Additional System Assist Processor (SAP), System z Application Assist Processor (zAAP), and System z Intergrated Information Processor (zIIP) — the exact mix of the 101 processors to be determined. Water cooled option. 3040 GB of memory. zBX Model 003 attachment (with IBM POWER7 blade running PowerVM Enterprise Edition, IBM POWER7 blade running AIX, IBM BladeCenter HX5 blade running Red Hat Enterprise Linux, and IBM BladeCenter HX5 blade running SUSE Linux Enterprise Server, up to 112 blades). Trusted Key Entry (TKE) workstation. IBM Crypto Express 4S digital signature cryptography and EAL 5+ certification. IBM zAware. IBM DB2 Amalytics Accelerator. IBM SmartCloud Enterprise+ for System z. IBM System Storage DS8000. IBM TS1140 tape drives with encryption.

The zEnterprise EC12 is available in five hardware models: H20, H43, H66, H89, HA1A1. This includes the capability to support over 100 configurable cores.

command summaries

appendix A

This chapter provides a list of UNIX (Linux) command summaries.

shell commands, tools, and utilities

CONTROL

CONTROL-c Kills current process

a

a2ps Fromat an ASCII file for printing on a PostScript printer [Linux]
alias Create a command alias [builtin command in BASH]
apropos Search Help manual pages (man -k)
apt-get Search for and install software packages (Debian/Ubuntu)
aptitude Search for and install software packages (Debian/Ubuntu)
arp Display and modify the Internet-to-Ethernet address translation tables
aspell Spell Checker
at Schedule a job
awk Find and Replace text, database sort/validate/index

b

basename Strip directory and suffix from filenames
bash GNU Bourne-Again SHell
bc Arbitrary precision calculator language
bg Send to background
break Exit from a loop [builtin command in BASH]
builtin Run a shell builtin
bzip2 Compress or decompress file(s)

c

cal Displays a calendar [alternative layout using `ncal`]
case Conditionally perform a command
cat Concatenate and print (display) the contents of files
cd Change Directory
cfdisk Partition table manipulator for Linux
chgrp Change group ownership
chmod Change access permissions
chown Change file owner and group
chroot Run a command with a different root directory
chkconfig System services (runlevel)
cksum Print CRC checksum and byte counts
clear Clears the terminal screen
cmp Compare two files
comm Compare two sorted files line by line
command Run a command - ignoring shell functions [builtin command in BASH]
continue Resume the next iteration of a loop [builtin command in BASH]
cp Copy one or more files to another location
cron Daemon to execute scheduled commands
crontab Schedule a command to run at a later time
csplit Split a file into context-determined pieces
cut Divide a file into several parts

d

date Display or change the date and time
dc Desk Calculator
dd Convert and copy a file, write disk headers, boot records
ddrescue Data recovery tool
declare Declare variables and give them attributes [builtin command in BASH]
defaults Set defaults [mac OS X]
df Display drives and free disk space
dict Look up a word at dict.org [Perl client script]
diff Display the differences between two files
diff3 Show differences among three files
dig DNS lookup
dir Briefly list directory contents
dircolors Color setup for `ls`
dirname Convert a full pathname to just a path
dirs Display list of remembered directories
dmesg Print kernel & driver messages
du Estimate file space usage

e

echo Display message on screen [builtin command in BASH]
egrep Search file(s) for lines that match an extended expression
eject Eject removable media
enable Enable and disable builtin shell commands [builtin command in BASH]
env Environment variables
ethtool Ethernet card settings
eval Evaluate several commands/arguments
exec Execute a command
exit Exit the shell
expect Automate arbitrary applications accessed over a terminal
expand Convert tabs to spaces
export Set an environment variable
expr Evaluate expressions

f

false Do nothing, unsuccessfully
fdformat Low-level format a floppy disk
fdisk Partition table manipulator for Linux
fg Send job to foreground
fgrep Search file(s) for lines that match a fixed string
file Determine file type
find Search for files that meet a desired criteria
fmt Reformat paragraph text
fold Wrap text to fit a specified width
for Loop and expand *words* and execute *commands*
format Format disks or tapes
free Display memory usage
fsck File system consistency check and repair
ftp File Transfer Protocol
function Define Function Macros
fuser Identify/kill the process that is accessing a file

g

gawk Find and Replace text within file(s)
getopts Parse positional parameters
grep Search file(s) for lines that match a given pattern
groupadd Add a user security group
groupdel Delete a group
groupmod Modify a group

groups Print group names a user is in
gzip Compress or decompress named file(s)

h

hash Remember the full pathname of a name argument
head Output the first part of file(s)
help Display help for a built-in command [builtin command in BASH]
history Command History
hostname Print or set system name

i

iconv Convert the character set of a file
id Print user and group IDs
if Conditionally perform a command
ifconfig Configure a network interface
ifdown Stop a network interface
ifup Start a network interface up
import Capture an X server screen and save the image to file
init Set run level on Linux
install Copy files and set attributes

j

jobs List active jobs [builtin command in BASH]
join Join lines on a common field
jot Print numeric sequences [BSD and Mac OS X]

k

kill Stop a process from running
killall Kill processes by name

l

less Display output one screen at a time
let Perform arithmetic on shell variables [builtin command in BASH]
ln Make links between files
local Create variables [builtin command in BASH]
locate Find files
logname Print current login name
logout Exit a login shell [builtin command in BASH]
look Display lines beginning with a given string
lpc Line printer control program
lpr Off line print
lprint Print a file
lprintd Abort a print job
lprintq List the print queue
lprm Remove jobs from the print queue
ls List information about file(s)
lsof List open files

m

make Compile or recompile a group of programs
man Help manual
mkdir make directory(s)
mkfifo Make FIFOs (named pipes)
mkisofs Create an hybrid ISO9660/JOLIET/HFS filesystem
mknod Make block or character special files
mmv Mass Move and rename (files)
more Display output one screen at a time
mount Mount a file system
mtools Manipulate MS-DOS files
mtr Network diagnostics (traceroute/ping)

mv Move or rename files or directories
mysql SQL database

n

nano Very simple editor
ncal> Displays a calendar [alternative layout using `cal`]
netstat Networking information
nice Set the priority of a command or job
nl Number lines and write files
nohup Run a command immune to hangups
notify-send Send desktop notifications
nslookup Query Internet name servers interactively

o

open Open a file in its default application
op Operator access

p

passwd Modify a user password
paste Merge lines of files
pathchk Check file name portability
ping Test a network connection
pkill Stop processes from running
popd Restore the previous value of the current directory
pr Prepare files for printing
printcap Printer capability database
printenv Print environment variables
printf Format and print data [builtin command in BASH]
ps Process Status
pushd Save and then change the current directory
pwd Print Working Directory

q

quota Display disk usage and limits
quotacheck Scan a file system for disk usage
quotactl Set disk quotas

r

ram ram disk device
rcp Remote copy files between two machines
read Read a line from standard input [builtin command in BASH]
readarray Read from stdin into an array variable [builtin command in BASH]
readonly Mark variables/functions as readonly
reboot Reboot the system
rename Rename files
renice Alter priority of running processes
remsync Synchronize remote files via email
return Return from a shell function
rev Reverse lines of a file
rm Remove files
rmdir Remove directory(s)
route Routing commands
rpm Package manager
rsync Remote file copy (Synchronize file trees)

s

screen Multiplex terminal, run remote shells via SSH
screencapture Create screenshotts [Mac OS X]
scp Secure copy (remote file copy)
sdiff Merge two files interactively

sed Stream Editor
select Accept keyboard input
seq Print numeric sequences
service Run init scripts [System V]
set Manipulate shell variables and functions
sftp Secure File Transfer Program
shift Shift positional parameters
shopt Shell Options
shred Secure delete
shutdown Shutdown or restart the system
sleep Delay for a specified time
slocate Find files
sort Sort text files
source Run commands from a file [also .]
split Split a file into fixed-size pieces
ssh Secure Shell client (remote login program)
strace Trace system calls and signals
su Substitute User identity
sudo Execute a command as another user [root]
sum Print a checksum for a file
suspend Suspend execution of this shell [builtin command in BASH]
symlink Make a new name for a file
sync Synchronize data on disk with memory
sysstat Server toolset

t

tail Output the last part of file
tar Tape ARchiver
tee Redirect output to multiple files
test Evaluate a conditional expression
time Measure program running time
times User and system times
touch Change file timestamps
top List top processes running on the system
traceroute Trace Route to Host
trap Run a command when a signal is set (Bourne)
tr Translate, squeeze, and/or delete characters
troff Format text for print
true Do nothing, successfully
tsort Topological sort
tty Print filename of terminal on stdin
type Describe a command [builtin command in BASH]

u

ulimit Limit user resources [builtin command in BASH]
umask Users file creation mask
umount Unmount a device
unalias Remove an alias [builtin command in BASH]
uname Print system information
unexpand Convert spaces to tabs
uniq Uniquify files
units Convert units from one scale to another
unset Remove variable or function names
unshar Unpack shell archive scripts
until Execute commands (until error)
unzip Uncompress .zip files
uptime Show uptime

useradd	Create new user account
userdel	Delete a user account
usermod	Modify user account
users	List users currently logged in
uuencode	Encode a binary file
uudecode	Decode a file created by uuencode

v

v	verbosely list directory contents (`ls -l -b`)
vdir	verbosely list directory contents (`ls -l -b`)
vi	ancient text editor
vim	modern version of ancient text editor
vmstat	Report Virtual Memory STATistics

w

w	Information on what each user is doing
wait	Wait for a process to complete [builtin command in BASH]
watch	Execute/display a program periodically
wc	Print byte, word, and line counts
whatis	Single line summary of a Unix/Linux command
whereis	Search the user's `$path`, man pages and source files for a program
which	Search the user's `$path` for a program file
while	Loop
who	Print all usernames currently logged in
whoami	Print the current user id and name (`id -un`)
wget	Retrieve web pages or files via HTTP, HTTPS or FTP
write	Send a message to another user

x

xargs	Execute utility, passing constructed argument list(s)
xdg-open	Open a file or URL in the user's preferred application

y

yes	Print a string until interrupted
yum	Install, upgrade, or remove the Apache web server

punctuation

.	Run a command script in the current shell
!	Run the last command again
#	Comment or Remark

history of computers

appendix B

Notes on the summary at the end of each year:

Year Year that items were introduced.

Operating Systems Operating systems introduced in that year.

Programming Languages Programming languages introduced. While only a few programming languages are appropriate for operating system work (such as Ada, BLISS, C, FORTRAN, and PL/I, the programming languages available with an operating system greatly influence the kinds of application programs available for an operating system.

Computers Computers and processors introduced. While a few operating systems run on a wide variety of computers (such as UNIX and Linux), most operating systems are closely or even intimately tied to their primary computer hardware. Speed listings in parenthesis are in operations per second (OPS), floating point operatins per second (FLOPS), or clock speed (Hz).

Software Software programs introduced. Some major application programs that became available. Often the choice of operating system and computer was made by the need for specific programs or kinds of programs.

Games Games introduced. It may seem strange to include games in the time line, but many of the advances in computer hardware and software technologies first appeared in games. As one famous example, the roots of UNIX were the porting of an early computer game to new hardware.

Technology Major technology advances.

Theory Major advances in computer science or related theory.

antiquity

The earliest calculating machine was the abacus, believed to have been invented in Babylon around 2400 BCE. The abacus was used by many different cultures and civilizations, including the major advance known as the Chinese abacus from the 2nd Century BCE.

200s BCE

The Antikythera mechanism, discovered in a shipwreck in 1900, is an early mechanical analog computer from between 150 BCE and 100 BCE. The Antikythera mechanism used a system of 37 gears to compute the positions of the sun and the moon through the zodiac on the Egyptian calendar, and possibly also the fixed stars and five planets known in antiquity (Mercury, Venus, Mars, Jupiter, and Saturn) for any time in the future or past. The system of gears added and subtracted angular velocities to compute differentials. The Antikythera mechanism could accurately predict eclipses and could draw up accurate astrological charts for important leaders. It is likely that the Antikythera mechanism was based on an astrological computer created by Archimedes of Syracuse in the 3rd century BCE.

100s BCE

The Chinese developed the South Pointing Chariot in 115 BCE. This device featured a differential gear, later used in modern times to make analog computers in the mid-20th Century.

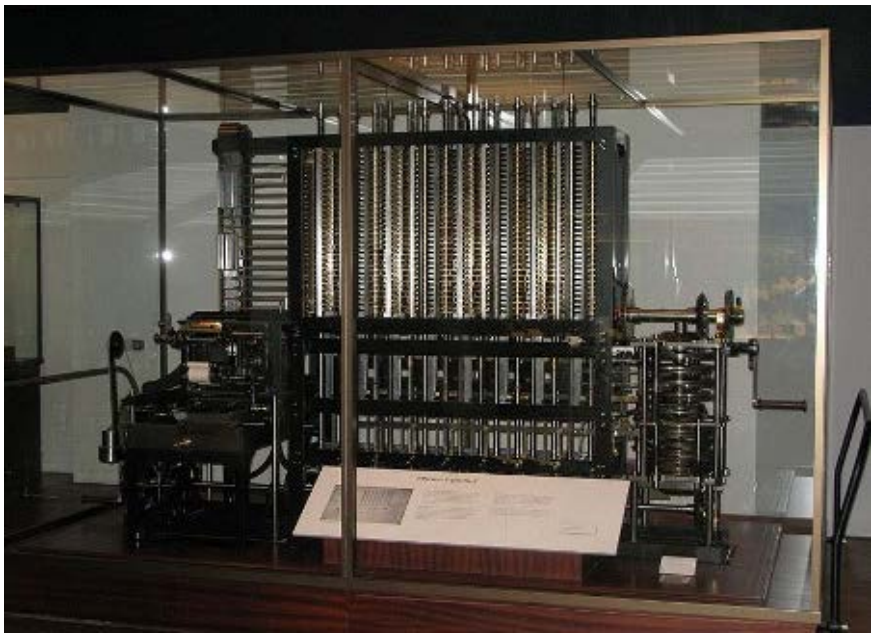
400s

The Indian grammarian Panini wrote the *Ashtadhyayi* in the 5th Century BCE. In this work he created 3,959 rules of grammar for India's Sanskrit language. This important work is the oldest surviving linguistic book and introduced the idea of metarules, transformations, and recursions, all of which have important applications in computer science.

1400s

The Inca created digital computers using giant loom-like wooden structures that tied and untied knots in rope. The knots were digital bits. These computers allowed the central government to keep track of the agricultural and economic details of their far-flung empire. The Spanish conquered the Inca during fighting that stretched from 1532 to 1572. The Spanish destroyed all but one of the Inca computers in the belief that the only way the machines could provide the detailed information was if they were Satanic divination devices. Archaeologists have long known that the Inca used knotted strings woven from cotton, llama wool, or alpaca wool called khipu or quipus to record accounting and census information, and possibly calendar and astronomical data and literature. In recent years archaeologists have figured out that the one remaining device, although in ruins, was clearly a computer.

1800s



Charles Babbage created the difference engine and the analytical engine, often considered to be the first modern computers. Augusta Ada King, the Countess of Lovelace, was the first modern computer programmer.

In the 1800s, the first computers were programmable devices for controlling the weaving machines in the factories of the Industrial Revolution. Created by Charles Babbage, these early computers used Punch cards as data storage (the cards contained the control codes for the various patterns). These cards were very similiar to the famous Hollerinth cards developed later. The first computer programmer was Lady Ada, for whom the Ada programming language is named.

In 1822 Charles Babbage proposed a difference engine for automated calculating. In 1933 Babbage started work on his Analytical Engine, a mechanical computer with all of the elements of a modern computer, including control, arithmetic, and memory, but the technology of the day couldn't produce gears with enough precision or reliability to make his computer possible. The Analytical Engine would have been programmed with Jacquard's punched cards. Babbage designed the Difference Engine No.2. Lady Ada Lovelace wrote a program for the Analytical Engine that would have correctly calculated a sequence of Bernoulli numbers, but was never able to test her program because the machine wasn't built.

George Boole introduced what is now called Boolean algebra in 1854. This branch of mathematics was essential for creating the complex circuits in modern electronic digital computers.

In 1871, James Clerk Maxwell introduced his thought experiment, Maxwell's demon, and a possible violation of the Second Law of Thermodynamics. This thought experiment plays an important role in quantum physics and information theory.

In 1893, Jacques Salomon Hadamard proved his inequaltiy of determinants, which led to the discovery of Hadamard matrices, which are the foundation for the Hadamard Gate in quantum computing.

1900s

In the 1900s, researchers started experimenting with both analog and digital computers using vacuum tubes. Some of the most successful early computers were analog computers, capable of performing advanced calculus problems rather quickly. But the real future of computing was digital rather than analog. Building on the technology and math used for telephone and telegraph switching networks, researchers started building the first electronic digital computers.

Leonardo Torres y Quevedo first proposed floating point arithmetic in Madrid in 1914. Konrad Zuse independently proposed the same idea in Berlin in 1936 and built it into the hardware of his Zuse computer. George Stibitz also indpendently proposed the idea in New Jersey in 1939.

The first modern computer was the German Zuse computer (Z3) in 1941. In 1944 Howard Aiken of Harvard University created the Harvard Mark I and Mark II. The Mark I was primarily mechanical, while the Mark II was primarily based on reed relays. Telephone and telegraph companies had been using reed relays for the logic circuits needed for large scale switching networks.

The first modern electronic computer was the ENIAC in 1946, using 18,000 vacuum tubes. See below for information on Von Neumann's important contributions.

The first solid-state (or transistor) computer was the TRADIC, built at Bell Laboratories in 1954. The transistor had previously been invented at Bell Labs in 1948.

1925

Theory: Wolfgang Pauli invents the Pauli matrices, which are the basis for the Pauli-X, Pauli-Y, and Pauli-Z Gates in quantum computing.

1936

Theory: Alan Turing invents the Turing machine in his paper "On Computable Numbers, with an Application to the *Entscheidungsproblem*".

1938

Computers: Zuse Z1 (Germany, 1 OPS, first mechanical programmable binary computer, storage for a total of 64 numbers stored as 22 bit floating point numbers with 7-bit exponent, 15-bit signifocana [one implicit bit], and sign bit); Konrad Zuse called his floating point hardware "semi-logarithmic notation" and included the ability to handle infinity and undefined.

1941

The first modern computer was the German Zuse computer (Z3) in 1941.

Computers: Atanasoff-Berry Computer; Zuse Z3 (Germany, 20 OPS, added floating point exceptions, plus and minus infinity, and undefined)

1942

Computers: work started on Zuse Z4

1943

In 1943 Howard Aiken of Harvard University created the Harvard Mark I. The Mark I was primarily mechanical.

Computers: Harvard Mark I (U.S.); Colossus 1 (U.K., 5 kOPS)

1944

In 1944 Howard Aiken of Harvard University created the Harvard Mark II. While the Mark I was primarily mechanical, the Mark II was primarily based on reed relays. Telephone and telegraph companies had been using reed relays for the logic circuits needed for large scale switching networks.

Computers: Colossus 2 (U.K., single processor, 25 kOPS); Harvard Mark II and AT&T Bell Labortories' Model V (both relay computers) were the first American computers to include floating point hardware

1945

Plankalkül (Plan Calculus), created by Konrad Zuse for the Z3 computer in Nazi germany, may have been the first programming language (other than assemblers). This was a surprisingly advanced programming language, with many features that didn't appear again until the 1980s.

Programming Languages: Plankalkül
Computers: Zuse Z4 (relay based computer, first commercial computer)

von Neumann architecture

John Louis von Neumann, mathematician (born János von Neumann 28 December 1903 in Budapest, Hungary, died 8 February 1957 in Washington, D.C.), proposed the *stored program concept* while professor of mathemtics (one of the orginal six) at Princeton University's Institute for Advanced Services, in which programs (code) are stored in the same memory as data. The computer knows the difference between code and data by which it is attempting to access at any given moment. When evaluating code, the binary numbers are decoded by some kind of physical logic circuits (later other methods, such as microprogramming, were introduced), and then the instructions are run in hardware. This design is called **von Neumann architecture** and has been used in almost every digital computer ever made.

Von Neumann architecture introduced flexibility to computers. Previous computers had their programming hard wired into the computer. A particular computer could only do one task (at the time, mostly building artillery tables) and had to be physically rewired to do any new task.

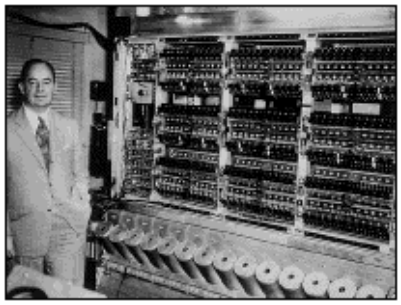
By using numeric codes, von Neumann computers could be reprogrammed for a wide variety of problems, with the decode logic remaining the same.

As processors (especially super computers) get ever faster, the *von Neumann bottleneck* is starting to become an issue. With data and code both being accessed over the same circuit lines, the processor has to wait for one while the other is being fetched (or written). Well designed data and code caches help, but only when the requested access is already loaded into cache. Some researchers are now experimenting with **Harvard architecture** to solve the von Neumann bottleneck. In Harvard arrchitecture, named for Howard Aiken's experimental Harvard Mark I (ASCC) calculator [computer] at Harvard University, a second set of data and address lines along with a second set of memory are set aside for executable code, removing part of the conflict with memory accesses for data.

Von Neumann became an American citizen in 1933 to be eligible to help on top secret work during World War II. There is a story that Oskar Morganstern coached von Neumann and Kurt Gödel on the U.S. Constitution and American history while driving them to their immigration interview. Morganstern asked if they had any questions, and Gödel replied that he had no questions, but had found some logical inconsistencies in the Constitution that he wanted to ask the Immigration officers about. Morganstern recommended that he not ask questions, but just answer them.

Von Neumann occassionally worked with Alan Turing in 1936 through 1938 when Turing was a graduate student at Princeton. Von Neumann was exposed to the concepts of logical design and universal machine proposed in Turing's 1934 paper "On Computable Numbers with an Application to the Entschiedungs-problem".

Von Neumann worked with such early computers as the Harvard Mark I, ENIAC,
EDVAC, and his own IAS computer.



Early research into computers involved doing the computations to create tables, especially artillery firing tables. Von Neumann was convinced that the future of computers involved applied mathematics to solve specific problems rather than mere table generation. Von Neumann was the first person to use computers for mathematical physics and economics, proving the utility of a general purpose computer.

Von Neumann proposed the concept of stored programs in the 1945 paper “First Draft of a Report on the EDVAC”. Influenced by the idea, Maurice Wilkes of the Cambridge University Mathematical Laboratory designed and built the EDSAC, the world’s first operational, production, stored-program computer.

The first stored computer program ran on the Manchester Mark I [computer] on June 21, 1948.

Von Neumann foresaw the advantages of parallelism in computers, but because of construction limitations of the time, he worked on sequential systems.

Von Neumann advocated the adoption of the bit as the measurement of computer memory and solved many of the problems regarding obtaining reliable answers from unreliable computer components.

Interestingly, von Neumann was opposed to the idea of compilers. When shown the idea for FORTRAN in 1954, von Neumann asked “Why would you want more than machine language?”. Von Neumann had graduate students hand assemble programs into binary code for the IAS machine. Donald Gillies, a student at Princeton, created an assembler to do the work. Von Neumann was angry, claiming “It is a waste of a valuable scientific computing instrument to use it to do clerical work”.

Von Neumann also did important work in set theory (including measure theory), the mathematical foundation for quantum theory (including statistical mechanics), self-adjoint algebras of bounded linear operators on a Hilbert space closed in weak operator topology, non-linear partial differential equations, and automata theory (later applied to cmputers). His work in economics included his 1937 paper “A Model of General Economic Equilibrium” on a multi-sectoral growth model and his 1944 book “Theory of Games and Economic Behavior” (co-authored with Morgenstern) on game theory and uncertainty.

I leave the discussion of von Neumann with a couple of quotations:

> “If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.”
>
> “Anyone who considers arithmetical methods of producing random numbers is, of course, in a state of sin.”

bare hardware

In the earliest days of electronic digital computing, everything was done on the bare hardware. Very few computers existed and those that did exist were experimental in nature. The researchers who were making the first computers were also the programmers and the users. They worked directly on the “bare hardware”. There was no operating system. The experimenters wrote their programs in machine or assembly language and a running program had complete control of the entire computer. Often programs and data were entered by hand through the use of toggle switches. Memory locations (both data and programs) could be read by viewing a series of lights (one for each binary digit). Debugging consisted of a combination of fixing both the software and hardware, rewriting the object code and changing the actual computer itself.

The lack of any operating system meant that only one person could use a computer at a time. Even in the research lab, there were many researchers competing for limited computing time. The first solution was a reservation system, with researchers signing up for specific time slots. The earliest billing systems charged for the entire computer and all of its resources (regardless of whether used or not) and was based on outside clock time, being billed from the scheduled start to scheduled end times.

The high cost of early computers meant that it was essential that the rare computers be used as efficiently as possible. The reservation system was not particularly efficient. If a researcher finished work early, the computer sat idle until the next time slot. If the researcher's time ran out, the researcher might have to pack up his or her work in an incomplete state at an awkward moment to make room for the next researcher. Even when things were going well, a lot of the time the computer actually sat idle while the researcher studied the results (or studied memory of a crashed program to figure out what went wrong). Simply loading the programs and data took up some of the scheduled time.

computer operators

One solution to this problem was to have programmers prepare their work off-line on some input medium (often on punched cards, paper tape, or magnetic tape) and then hand the work to a computer operator. The computer operator would load up jobs in the order received (with priority overrides based on politics and other factors). Each job still ran one at a time with complete control of the computer, but as soon as a job finished, the operator would transfer the results to some output medium (punched tape, paper tape, magnetic tape, or printed paper) and deliver the results to the appropriate programmer. If the program ran to completion, the result would be some end data. If the program crashed, memory would be transferred to some output medium for the programmer to study (because some of the early business computing systems used magnetic core memory, these became known as "core dumps").

The concept of computer operators dominated the mainframe era and continues today in large scale operations with large numbers of servers.

device drivers and library functions

Soon after the first successes with digital computer experiments, computers moved out of the lab and into practical use. The first practical application of these experimental digital computers was the generation of artillery tables for the British and American armies. Much of the early research in computers was paid for by the British and American militaries. Business and scientific applications followed.

As computer use increased, programmers noticed that they were duplicating the same efforts.

Every programmer was writing his or her own routines for I/O, such as reading input from a magnetic tape or writing output to a line printer. It made sense to write a common device driver for each input or putput device and then have every programmer share the same device drivers rather than each programmer writing his or her own. Some programmers resisted the use of common device drivers in the belief that they could write "more efficient" or faster or ""better" device drivers of their own.

Additionally each programmer was writing his or her own routines for fairly common and repeated functionality, such as mathematics or string functions. Again, it made sense to share the work instead of everyone repeatedly "reinventing the wheel". These shared functions would be organized into libraries and could be inserted into programs as needed. In the spirit of cooperation among early researchers, these library functions were published and distributed for free, an early example of the power of the open source approach to software development.

Computer manufacturers started to ship a standard library of device drivers and utility routines with their computers. These libraries were often called a **runtime library** because programs connected up to the routines in the library at run time (while the program was running) rather than being compiled as part of the program. The commercialization of code libraries ended the widespread free sharing of software.

Manufacturers were pressured to add security to their I/O libraries in order to prevent tampering or loss of data.

input output control systems

The first programs directly controlled all of the computer's resources, including input and output devices. Each individual program had to include code to control and operate each and every input and/or output device used.

One of the first consolidations was placing common input/output (I/O) routines into a common library that could be shared by all programmers. I/O was separated from processing.

These first rudimentary operating systems were called an Input Output Control System or IOCS.

Computers remained single user devices, with main memory divided into an IOCS and a user section. The user section consisted of program, data, and unused memory.

The user remained responsible for both set up and tear down.

Set up included loading data and program, by front panel switches, punched card, magnetic tapes, paper tapes, disk packs, drum drives, and other early I/O and storage devices. Paper might be loaded into printers, blank cards into card punch mahcines, and blank or formatted tape into tape drives, or other output devices readied.

Tear down would include unmounting tapes, drives, and other media.

The very expensive early computers sat idle during both set up and tear down.

This waste led to the introduction of less expensive I/O computers. While one I/O computer was being set up or torn down, another I/O computer could be communicating a readied job with the main computer.

Some installations might have several different I/O computers connected to a single main computer to keep the expensive main computer in use. This led to the concept of multiple I/O channels.

monitors

As computers spread from the research labs and military uses into the business world, the accountants wanted to keep more accurate counts of time than mere wall clock time.

This led to the concept of the **monitor**. Routines were added to record the start and end times of work using computer clock time. Routines were added to I/O library to keep track of which devices were used and for how long.

With the development of the Input Output Control System, these time keeping routines were centralized.

You will notice that the word monitor appears in the name of some operating systems, such as FORTRAN Monitor System. Even decades later many programmers still refer to the operating system as the monitor.

An important motivation for the creation of a monitor was more accurate billing. The monitor could keep track of actual use of I/O devices and record runtime rather than clock time.

For accurate time keeping the monitor had to keep track of when a program stopped running, regardless of whether it was a normal end of the program or some kind of abnormal termination (such as aa crash).

The monitor reported the end of a program run or error conditions to a computer operator, who could load the next job waiting, rerun a job, or take other actions. The monitor also notified the computer operator of the need to load or unload various I/O devices (such as changing tapes, loading paper into the printer, etc.).

1946

The first modern electronic computer was the ENIAC in 1946, using 18,000 vacuum tubes.

Computers: UPenn Eniac (5 kOPS); Colossus 2 (parallel processor, 50 kOPS)
Technology: electrostatic memory

1948

Computers: IBM SSEC; Manchester SSEM
Technology: random access memory; magnetic drums; transistor

Theory: Claude E. Shannon publishes “A Mathematical Theory of Communication”, including Shannon’s source coding theorem that establishes the limits to possible data compression.

1949

Short Code created in 1949. This programming language was compiled into machine code by hand.

Programming Languages: Short Code
Computers: Manchester Mark 1
Technology: registers

1950s

Some operating systems from the 1950s include: FORTRAN Monitor System, General Motors Operating System, Input Output System, SAGE, and SOS.

SAGE (Semi-Automatic Ground Environment), designed to monitor weapons systems, was the first real time control system.

1951

Grace Hopper starts work on A-0.

Computers: Ferranti Mark 1 (first commercial computer); Leo I (frst business computer); UNIVAC I, Whirlwind

1952

Autocode, a symbolic assembler for the Manchester Mark I computer, was created in 1952 by Alick E. Glennie. Later used on other computers.

A-0 (also known as AT-3), the first compiler, was created in 1952 by Grace Murray Hopper. She later created A-2, ARITH-MATIC, MATH-MATIC, and FLOW-MATIC, as well as being one of the leaders in the development of COBOL.Grace Hopper was working for Remington Rand at the time. Rand released the language as MATH-MATIC in 1957.

According to some sources, first work started on FORTRAN.

Programming Languages: Autocode; A-0; FORTRAN
Computers: UNIVAC 1101; IBM 701
Games: OXO (a graphic version of Tic-Tac-Toe created by A.S. Douglas on the EDSAC computer at the University of Cambridge to demonstrate ideas on human-computer interaction)

1953

Computers: Strela

1954

The first solid-state (or transistor) computer was the TRADIC, built at Bell Laboratories in 1954. The transistor had previously been invented at Bell Labs in 1948.

FORTRAN (FORmula TRANslator) was created in 1954 by John Backus and other researchers at International Business Machines (now IBM). Released in 1957. FORTRAN is the oldest programming language still in common use. Identifiers were limited to six characters. Elegant representation of mathematic expressions, as well as relatively easy input and output. FORTRAN was based on A-0.

"Often referred to as a *scientific language,* FORTRAN was the first *high-level* language, using the first compiler ever developed. Prior to the development of FORTRAN computer programmers were required to program in machine/assembly code, which was an extremely difficult and time consuming task, not to mention the dreadful chore of debugging the code. The objective during its design was to create a programming language that would be: simple to learn, suitable for a wide variety of applications, *machine independent,* and would allow complex mathematical expressions to be stated similarly to regular algebraic notation. While still being almost as efficient in execution as assembly language. Since FORTRAN was so much easier to code, programmers were able to write programs 500% faster than before, while execution efficiency was only reduced by 20%, this allowed them to focus more on the problem solving aspects of a problem, and less on coding.

"FORTRAN was so innovative not only because it was the first high-level language [still in use], but also because of its compiler, which is credited as giving rise to the branch of computer science now known as *compiler theory.* Several years after its release FORTRAN had developed many different *dialects,* (due to special *tweaking* by programmers trying to make it better suit their personal needs) making it very difficult to transfer programs from one machine to another." —Neal Ziring, The Language Guide, University of Michigan

"Some of the more significant features of the language are listed below:" —Neal Ziring, The Language Guide, University of Michigan

- **Simple to learn** - when FORTRAN was design one of the objectives was to write a language that was easy to learn and understand.
- **Machine Independent** - allows for easy transportation of a program from one machine to another.
- **More natural ways to express mathematical functions** - FORTRAN permits even severely complex mathematical functions to be expressed similarly to regular algebraic notation.
- **Problem orientated language**
- **Remains close to and exploits the available hardware**
- **Efficient execution** - there is only an approximate 20% decrease in efficiency as compared to assembly/machine code.
- **Ability to control storage allocation** -programmers were able to easily control the allocation of storage (although this is considered to be a dangerous practice today, it was quite important some time ago due to limited memory.
- **More freedom in code layout** - unlike assembly/machine language, code does not need to be laid out in rigidly defined columns, (though it still must remain within the parameters of the FORTRAN source code form).

"42. You can measure a programmer's perspective by noting his attitude on the continuing vitality of FORTRAN." —Alan Perlis, *Epigrams on Programming,* ACM's SIGPLAN Notices Volume 17, No. 9, September 1982, pages 7-13

Programming Languages: FORTRAN
Computers: IBM 650; IBM 704 (vacuum tube computer with floating point); IBM NORC (67 kOPS)
Technology: magnetic core memory

1955

Operating Systems: GMOS (General Motors OS for IBM 701)
Computers: Harwell CADET

batch systems

Batch systems automated the early approach of having human operators load one program at a time. Instead of having a human operator load each program, software handled the scheduling of jobs. In addition to programmers submitting their jobs, end users could submit requests to run specific programs with specific data sets (usually stored in files or on cards). The operating system would schedule "batches" of related jobs. Output (punched cards, magnetic tapes, printed material,

etc.) would be returned to each user.

General Motors Operating System, created by General Motors Research Laboratories in early 1956 (or late 1955) for thieir IBM 701 mainframe is generally considered to be the first batch operating system and possibly the first “real” operating system.

The operating system would read in a program and its data, run that program to completion (including outputing data), and then load the next program in series as long as there were additional jobs available.

Batch operating systems used a Job Control Language (JCL) to give the operating system instructions. These instructions included designation of which punched cards were data and which were programs, indications of which compiler to use, which centralized utilities were to be run, which I/O devices might be used, estimates of expected run time, and other details.

This type of batch operating system was known as a single stream batch processing system.

Examples of operating systems that were primarily batch-oriented include: BKY, BOS/360, BPS/360, CAL, and Chios.

1956

Researchers at MIT begin experimenting with direct keyboard input into computers.

IPL (Information Processing Language) was created in 1956 by A. Newell, H. Simon, and J.C. Shaw. IPL was a low level list processing language which implemented recursive programming.

Programming Languages: IPL
Operating Systems: GM-NAA I/O
Computers: IBM 305 RAMAC; MIT TX-0 (83 kOPS)
Technology: hard disk

1957

MATH-MATIC was released by the Rand Corporation in 1957. The language was derived from Grace Murray Hopper’s A-0.

FLOW-MATIC, also called B-0, was created in 1957 by Grace Murray Hopper.

The first commercial FORTRAN program was run at Westinghouse. The first compile run produced a missing comma diagnostic. The second attempt was a success.

BESYS OS was created by Bell telephone Laboratories for internal use.

The U.S. government created the Advanced Research Project Group (ARPA) in esponse to the Soviet Union’s launching of Sputnik. ARPA was intended to develop key technology that was too risky for private business to develop.

Programming Languages: FLOW-MATIC; MATH-MATIC
Operating Systems: BESYS OS
Computers: IBM 608
Technology: dot matrix printer

1958

FORTRAN II in 1958 introduces subroutines, functions, loops, and a primitive For loop.

IAL (International Algebraic Logic) started as the project later renamed ALGOL 58. The theoretical definition of the language is published. No compiler.

LISP (LISt Processing) was created n 1958 and released in 1960 by John McCarthy of MIT. LISP is the second oldest programming language still in common use. LISP was intended for writing artificial intelligence programs.

"Interest in artificial intelligence first surfaced in the mid 1950. Linguistics, psychology, and mathematics were only some areas of application for AI. Linguists were concerned with natural language processing, while psychologists were interested in modeling human information and retrieval. Mathematicians were more interested in automating the theorem proving process. The common need among all of these applications was a method to allow computers to process symbolic data in lists.

"IBM was one of the first companies interested in AI in the 1950s. At the same time, the FORTRAN project was still going on. Because of the high cost associated with producing the first FORTRAN compiler, they decided to include the list processing functionality into FORTRAN. The FORTRAN List Processing Language (FLPL) was designed and implemented as an extention to FORTRAN.

"In 1958 John McCarthy took a summer position at the IBM Information Research Department. He was hired to create a set of requirements for doing symbolic computation. The first attempt at this was differentiation of algebraic expressions. This initial experiment produced a list of of language requirements, most notably was recursion and conditional expressions. At the time, not even FORTRAN (the only high-level language in existance) had these functions.

"It was at the 1956 Dartmouth Summer Research Project on Artificial Intelligence that John McCarthy first developed the basics behind Lisp. His motivation was to develop a list processing language for Artificial Intelligence. By 1965 the primary dialect of Lisp was created (version 1.5). By 1970 special-purpose computers known as Lisp Machines, were designed to run Lisp programs. 1980 was the year that object-oriented concepts were integrated into the language. By 1986, the X3J13 group formed to produce a draft for ANSI Common Lisp standard. Finally in 1992, X3J13 group published the American National Standard for Common Lisp." —Neal Ziring, The Language Guide, University of Michigan

"Some of the more significant features of the language are listed below:" —Neal Ziring, The Language Guide, University of Michigan

- **Atoms & Lists** - Lisp uses two different types of data structures, atoms and lists.
- **Atoms** are similar to identifiers, but can also be numeric constants
- **Lists** can be lists of atoms, lists, or any combination of the two
- **Functional Programming Style** - all computation is performed by applying functions to arguments. Variable declarations are rarely used.
- **Uniform Representation of Data and Code** - example: the list (A B C D)
  - a list of four elements (interpreted as data)
  - is the application of the function named A to the three parameters B, C, and D (interpreted as code)
- **Reliance on Recursion** - a strong reliance on recursion has allowed Lisp to be successful in many areas, including Artificial Intelligence.
- **Garbage Collection** - Lisp has built-in garbage collection, so programmers do not need to explicitly free dynamically allocated memory.

"55. A LISP programmer knows the value of everything, but the cost of nothing." —Alan Perlis, *Epigrams on Programming*, ACM's SIGPLAN Notices Volume 17, No. 9, September 1982, pages 7-13

Programming Languages: FORTRAN II; IAL; LISP
Operating Systems: UMES
Computers: UNIVAC II; IBM AN/FSQ-7 (400 kOPS)
Games: Tennis For Two (developed by William Higinnotham using an osciliscope and an analog computer)
Technology: integrated circuit

1959

COBOL (COmmon Business Oriented Language) was created in May 1959 by the Short Range Committee of the U.S. Department of Defense (DoD). The CODASYL committee (COnference on DAta SYstems Languages) worked from May 1959 to April 1960. Official ANSI standards included COBOL-68 (1968), COBOL-74 (1974), COBOL-85 (1985), and COBOL-2002 (2002). COBOL 97 (1997) introduced an object oriented version of COBOL. COBOL programs are divided into four divisions: identification, environment, data, and procedure. The divisions are further divided into sections. Introduced the RECORD data structure. Emphasized a verbose style intended to make it easy for business managers to read programs. Admiral Grace Hopper is recognized as the major contributor to the original COBOl language and as the inventor of compilers.

LISP 1.5 released in 1959.

> " '**DYNAMO** is a computer program for translating mathematical models from an easy-to-understand notation into tabulated and plotted results. … A model written in DYNAMO consists of a number of algebraic relationships that relate the variables one to another.' Although similar to FORTRAN, it is easier to learn and understand. DYNAMO stands for DYNAmic MOdels. It was written by Dr. Phyllis Fox and Alexander L. Pugh, III, and was completed in 1959. It grew out of an earlier language called SIMPLE (for Simulation of Industrial Management Problems with Lots of Equations), written in 1958 by Richard K. Bennett." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

ERMA (Electronic Recording Method of Accounting), a magnetic ink and computer readable font, was created for the Bank of America.

Programming Languages: COBOL; DYNAMO; ERMA; LISP 1.5
Operating Systems: SHARE
Computers: IBM 1401

early 1960s

The early 1960s saw the introduction of time sharing and multi-processing.

Some operating systems from the early 1960s include: Admiral, B1, B2, B3, B4, Basic Executive System, BOS/360, Compatible Timesharing System (CTSS), EXEC I, EXEC II, Honeywell Executive System, IBM 1410/1710 OS, IBSYS, Input Output Control System, Master Control Program, and SABRE.

The first major transaction processing system was SABRE (Semi-Automatic Business Related Environment), developed by IBM and American Airlines.

multiprogramming

There is a huge difference in speed between I/O and running programs. In a single stream system, the processor remains idle for much of the time as it waits for the I/O device to be ready to send or receive the next piece of data.

The obvious solution was to load up multiple programs and their data and switch back and forth between programs or jobs.

When one job idled to wait for input or output, the operating system could automatically switch to another job that was ready.

system calls

The first operating system to introduce system calls was University of Machester's Atlas I Supervisor.

time sharing

The operating system could have additional reasons to rotate through jobs, including giving higher or lower priority to

various jobs (and therefore a larger or smaller share of time and other resources). The Compatible Timesharing System (CTSS), first dmonstrated in 1961, was one of the first attempts at timesharing.

While most of the CTSS operating system was written in assembly language (all previous OSes were written in assembly for efficiency), the scheduler was written in the programming language MAD in order to allow safe and reliable experimentation with different scheduling algorithms. About half of the command programs for CTSS were also written in MAD.

Timesharing is a more advanced version of multiprogramming that gives many users the illusion that they each have complete control of the computer to themselves. The scheduler stops running programs based on a slice of time, moves on to the next program, and eventually returns back to the beginning of the list of programs. In little increments, each program gets their work done in a manner that appears to be simultaneous to the end users.

mid 1960s

Some operating systems from the mid-1960s include: Atlas I Supervisor, DOS/360, Input Output Selector, Master Control Program, and Multics.

The Atlas I Supervisor introduced spooling, interrupts, and virtual memory paging (16 pages) in 1962. Segmentation was introduced on the Burroughs B5000. MIT's Multics combined paging and segmentation.

The Compatible Timesharing System (CTSS) introduced email.

late 1960s

Some operating systems from the late-1960s include: BPS/360, CAL, CHIPPEWA, EXEC 3, EXEC 4, EXEC 8, GECOS III, George 1, George 2, George 3, George 4, IDASYS, MASTER, Master Control Program, OS/MFT, OS/MFT-II, OS/MVT, OS/PCP, and RCA DOS.

1960

ALGOL (ALGOrithmic Language) was released in 1960. Major releases in 1960 (ALGOL 60) and 1968 (ALGOL 68). ALGOL is the first block-structured labguage and is considered to be the first second generation computer language. This was the first programming language that was designed to be machine independent. ALGOL introduced such concepts as: block structure of code (marked by BEGIN and END), scope of variables (local variables inside blocks), BNF (Backus Naur Form) notation for defining syntax, dynamic arrays, reserved words, IF THEN ELSE, FOR, WHILE loop, the := symbol for assignment, SWITCH with GOTOs, and user defined data types. ALGOL became the most popular programming language in Europe in the mid- and late-1960s.

C.A.R. Hoare invents the **Quicksort**in 1960.

Programming Languages: ALGOL
Operating Systems: IBSYS
Computers: DEC PDP-1; CDC 1604; UNIVAC LARC (250 kFLOPS)

1961

The Compatible Timesharing System (CTSS), first dmonstrated in 1961, was one of the first attempts at timesharing.

While most of the CTSS operating system was written in assembly language (all previous OSes were written in assembly for efficiency), the scheduler was written in the programming language MAD in order to allow safe and reliable experimentation with different scheduling algorithms. About half of the command programs for CTSS were also written in MAD.

Operating Systems: CTSS, Burroughs MCP

Computers: IBM 7030 Stretch (1.2 MFLOPS)

1962

APL (A Programming Language) was published in the 1962 book *A Programming Language* by Kenneth E. Iverson and a subset was first released in 1964. The language APL was based on a notation that Iverson invented at Harvard University in 1957. APL was intended for mathematical work and used its own special character set. Particularly good at matrix manipulation. In 1957 it introduced the array. APL used a special character set and required special keyboards, displays, and printers (or printer heads).

FORTRAN IV is released in 1962.

Simula was created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center between 1962 and 1965. A compiler became available in 1964. Simula I and Simula 67 (1967) were the first object-oriented programming languages.

SNOBOL (StroNg Oriented symBOli Language) was created in 1962 by D.J. Farber, R.E. Griswold, and F.P. Polensky at Bell Telephone Laboratories. Intended for processing strings, the language was the first to use associative arrays, indexed by any type of key. Had features for pattern-matching, concatenation, and alternation. Allowed running code stored in strings. Data types: integer, real, array, table, pattern, and user defined types.

SpaceWarI, the first interactive computer game, was created by MIT students Slug Russel, Shag Graetz, and Alan Kotok on DEC's PDP-1.

The first operating system to introduce system calls was University of Machester's Atlas I Supervisor.

The Atlas I Supervisor introduced spooling, interrupts, and virtual memory paging (16 pages) in 1962. Segmentation was introduced on the Burroughs B5000. MIT's Multics combined paging and segmentation.

Operating Systems: GECOS
Programming Languages: APL; FORTRAN IV; Simula; SNOBOL
Computers: ATLAS, UNIVAC 1100/2200 (introduced two floating point formats, single precision and double precision; single precision: 36 bits, 1-bit sign, 8-bit exponent, and 27-bit significand; double precision: 36 bits, 1-bit sign, 11-bit exponent, and 60-bit significand), IBM 7094 (followed the UNIVAC, also had single and double precision numbers)
Games: Spacewar! (created by group of M.I.T. students on the DEC PDP-1)
Technology: RAND Corporation proposes the internet

1963

Work on PL/I starts in 1963.

> "**Data-Text** was the "original and most general problem-oriented computer language for social scientists." It has the ability to handle very complicated data processing problems and extremely intricate statistical analyses. It arose when FORTRAN proved inadequate for such uses. Designed by Couch and others, it was first used in 1963/64, then extensively revised in 1971. The Data-Text System was originally programmed in FAP, later in FORTRAN, and finally its own language was developed." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

Sketchpad, an interactive real time computer drawing system, was created in 1963 by Ivan Sutherland as his doctoral thesis at MIT. The system used a light pen to draw and manipulate geometric figures on a computer screen.

ASCII (American Standard Code for Information Interchange) was introduced in 1963.

Programming Languages: Data-Text
Computers: DEC PDP-6
Software: Sketchpad

Technology: mouse

1964

BASIC (Beginner's All-purpose Symbolic Instruction Code) was designed as a teaching language in 1963 by John George Kemeny and Thomas Eugene Kurtz of Dartmouth College. BASIC was intended to make it easy to learn programming. The first BASIC program was run at 4 a.m. May 1, 1964.

PL/I (Programming Language One) was created in 1964 at IBM's Hursley Laboratories in the United Kingdom. PL/I was intended to combine the scientific abilities of FORTRAN with the business capabilities of COBOL, plus additional facilities for systems programming. Also borrows from ALGOL 60. Originally called NPL, or New Programming Language. Introduces storage classes (automatic, static, controlled, and based), exception processing (On conditions), Select When Otherwise conditional structure, and several variations of the DO loop. Numerous data types, including control over precision.

RPG (Report Program Generator) was created in 1964 by IBM. Intended for creating commercial and business reports.

APL\360 implemented in 1964.

Bell Telephone Laboratories determines that it needs a new operating system to replace its 1957 BESYS OS. This project becomes Multics.

Operating Systems: DTSS, TOPS-10
Programming Languages: APL\360; BASIC; PL/I; RPG
Computers: IBM 360; DEC PDP-8; CDC 6600 (first supercomputer, scalar processor, 3 MFLOPS)
Technology: super computing

1965

Multics The Massachusetts Institute of Technology (MIT), AT&T Bell Labs, and General Electric attempted to create an experimental operating system called Multics for the GE-645 mainframe computer. AT&T intended to offer subscription-based computing services over the phone lines, an idea similar to the modern cloud approach to computing.

While the Multics project had many innovations that went on to become standard approaches for operating systems, the project was too complex for the time.

SNOBOL 3 was released in 1965.

Attribute grammars were created in 1965 by Donald Knuth.

Operating Systems: OS/360; Multics
Programming Languages: SNOBOL 3
Technology: time-sharing; fuzzy logic; packet switching; bulletin board system (BBS); email

1966

ALGOL W was created in 1966 by Niklaus Wirth. ALGOL W included RECORDs, dynamic data structures, CASE, passing parameters by value, and precedence of operators.

Euler was created in 1966 by Niklaus Wirth.

FORTRAN 66 was released in 1966. The language was rarely used.

ISWIM (If You See What I Mean) was described in 1966 in Peter J. Landin's article *The Next 700 Programming Languages* in the Communications of the ACM. ISWIM, the first purely functional language, influenced functional programming languages. The first language to use lazy evaluation.

LISP 2 was released in 1966.

Programming Languages: ALGOL W; Euler; FORTRAN 66; ISWIM
Computers: BESM-6

1967

Logo was created in 1967 (work started in 1966) by Seymour Papert. Intended as a programming language for children. Started as a drawing program. Based on moving a "turtle" on the computer screen.

Simula 67 was created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in 1967. Introduced classes, methods, inheriteance, and objects that are instances of classes.

SNOBOL 4 (StroNg Oriented symBOli Language) was released in 1967.

CPL (Combined Programming Language) was created in 1967 at Cambridge and London Universities. Combined ALGOL 60 and functional language. Used polymorphic testing structures. Included the ANY type, lists, and arrays.

Operating Systems: ITS; CP/CMS; WAITS
Programming Languages: CPL; Logo; Simula 67; SNOBOL 4

1968

In 1968 a group of scientists and engineers from Mitre Corporation (Bedford, Massachusetts) created Viatron Computer company and an intelligent data terminal using an 8-bit LSI microprocessor from PMOS technology.

ALGOL 68 in 1968 introduced the =+ token to combine assignment and add, UNION, and CASTing of types. It included the IF THEN ELIF FI structure, CASE structure, and user-defined operators.

Forth was created by Charles H. Moore in 1968. Stack based language. The name "Forth" was a reference to Moore's claim that he had created a fourth generation programming language.

ALTRAN, a variant of FORTRAN, was released.

ANSI version of COBOL defined.

Edsger Dijkstra wrote a letter to the Communications of the ACM claiming that the use of GOTO was harmful.

Programming Languages: ALTRAN; ALGOL 68; Forth
Computers: DEC PDP-10
Technology: microprocessor; interactive computing (including mouse, windows, hypertext, and fullscreen word processing)

1969

In 1969 Viatron created the 2140, the first 4-bit LSI microprocessor. At the time MOS was used only for a small number of calculators and there simply wasn't enough worldwide manufacturing capacity to build these computers in quantity.

UNIX was created at AT&T Bell Telephone Laboratories (now an independent corporation known as Lucent Technologies) by a group that included Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, Michael Lesk and Joe Ossanna.

This group of researchers, the last of the AT&T employees involved in Multics, decided to attempt the goals of Multics on a much more simple scale.

Unix was originally called UNICS, for Uniplexed Information and Computing Service, a play on words variation of

Multics, Multiplexed Information and Computing Service (*uni-* means “one”, *multi-* means “many”).

“What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication,” according to Dennis Ritchie.

UNIX was originally intended as a programmer’s workbench. The original version was a single-user system. As UNIX spread through the academic community, more general purpose tools were added, turning UNIX into a general purpose operating system.

While Ken Thompson still had access to the Multics environment, he wrote simulations on Multics for UNIX’s file and paging system.

Ken Thompson and Dennis Ritchie led a team of Bell Labs researchers (the team included Rudd Canaday) working on the PDP-7 to develop a hierarchial file system, computer processes, device files, a command-line interpreter, and a few small utility programs.

This version of UNIX was written in the B language.

At the time, AT&T was prohibited from selling computers or software, but was allowed to develop its own software and computers for internal use. AT&T’s consent decree with the U.S. Justice Department on monopoly charges was interpretted as allowing AT&T to release UNIX as an open source operating system for academic use. Ken Thompson, one of the originators of UNIX, took UNIX to the University of California, Berkeley, where students quickly started making improvements and modifications, leading to the world famous Berkeley Software Distribution (BSD) form of UNIX.

UNIX quickly spread throughout the academic world, as it solved the problem of keeping track of many (sometimes dozens) of proprietary operating systems on university computers. With UNIX all of the computers from many different manufacturers could run the same operating system and share the same programs (recompiled on each processor).

BCPL (Basic CPL) was created in 1969 in England. Intended as a simplified version of CPL, includes the control structures For, Loop, If Then, While, Until Repeat, Repeat While, and Switch Case.

> “BCPL was an early computer language. It provided for comments between slashes. The name is condensed from “Basic CPL”; CPL was jointly designed by the universities of Cambridge and London. Officially, the “C” stood first for “Cambridge,” then later for “Combined.” -- Unofficially it was generally accepted as standing for Christopher Strachey, who was the main impetus behind the language.” —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

B (derived from BCPL) developed in 1969 by Ken Thompson of Bell Telephone Laboratories for use in systems programming for UNIX. This was the parent language of C.

SmallTalk was created in 1969 at Xerox PARC by a team led by Alan Kay, Adele Goldberg, Ted Kaehler, and Scott Wallace. Fully object oriented programming language that introduces a graphic environment with windows and a mouse.

RS-232-C standard for serial communication introduced in 1969.

Space Travel Ken Thompson ported the Space Travel computer game from Multics to a Digital Equipment Corporation (DEC) PDP-7 he found at Bell Labs.

ARPA creates **ARPAnet**, the forerunner of the Internet (originally hosted by UCLA, UC Santa Barbara, University of Utah, and Stanford Research Institute).

Operating Systems: ACP; TENEX/TOPS-20; work started on Unix (initially called UNICS)
Programming Languages: B, BCPL; SmallTalk
Computers: CDC 7600 (36 MFLOPS)
Games: Space Travel (written by Jeremy Ben for Multics; when AT&T pulled out of the Multics project, J. Ben ported the program to FORTRAN running on GECOS on the GE 635; then ported by J. Ben and Dennis Ritchie in PDP-7 assembly language; the process of porting the game to the PDP-7 computer was the beginning of Unix)
Technology: ARPANET (military/academic precursor to the Internet); RS-232; networking; laser printer (invented by Gary Starkweather at Xerox)

early 1970s

Some operating systems from the early-1970s include: BKY, Chios, DOS/VS, Master Control Program, OS/VS1, and UNIX.

mid 1970s

Some operating systems from the mid-1970s include: CP/M, Master Control Program.

late 1970s

Some operating systems from the late-1970s include: EMAS 2900, General Comprehensive OS, VMS (later renamed OpenVMS), OS/MVS.

1970

Prolog (PROgramming LOGic) was created in 1972 in France by Alan Colmerauer with Philippe Roussel. Introduces Logic Programming.

Pascal (named for French religious fanatic and mathematician Blaise Pascal) was created in 1970 by Niklaus Wirth on a CDC 6000-series computer. Work started in 1968. Pascl was intended as a teaching language to replace BASIC, but quickly developed into a general purpose programming language. Programs compiled to a platform-independent intermediate P-code. The compiler for pascal was written in Pascal, an influential first in language design.

Forth used to write the program to control the Kitt Peaks telescope.

BLISS was a systems programming language developed by W.A. Wulf, D.B. Russell, and A.N. Habermann at Carnegie Mellon University in 1970. BLISS was a very popular systems programming language until the rise of C. The original compiler was noted for its optimizing of code. Most of the utilities for DEC's VMS operating system were written in BLISS-32. BLISS was a typeless language based on expressions rather than statements. Expressions produced values, and possibly caused other actions, such as modification of storage, transfer of control, or looping. BLISS had powerful macro facilities, conditional execution of statements, subroutines, built-in string functions, arrays, and some automatic data conversions. BLISS lacked I/O instructions on the assumption that systems I/O would actually be built in the language.

UNIX In 1970 Dennis Ritchie and Ken Thompson traded the promise to add text processing capabilities to UNIX for the use of a Digital Equipment Corporation (DEC) PDP-11/20. The initial version of UNIX, a text editor, and a text formatting program called `roff` were all written in PDP-11/20 assembly language.

UNIX was renamed from the original name UNICS. The author of the name change is lost to history.

ROFF Soon afterwards `roff` evolved into `troff` with full typesetting capability. The *UNIX Programmer's Manual* was published on November 3, 1971. The first commercial UNIX system was installed in early 1962 at the New York Telephone Co. Systems Development Center under the direction of Dan Gielan. Neil Groundwater build an Operational Support System in PDP-11/20 assembly language.

Quantum Computing Stephen Wiesner invents conjugate coding.

Operating Systems: Unix; RT-11; RSTS-11
Programming Languages: BLISS; Pascal; Prolog
Computers: Datapoint 2200; DEC PDP-11
Software: ROFF
Technology: dynamic RAM; flight data processor

1971

Manufacturers saw the benefit of MOS, starting with Intel's 1971 release of the 4-bit 4004 as the first commercially available microprocessor.

UNIX shell This first UNIX command shell, called the Thompson shell and abbreviated `sh`, was written by Ken Thompson at AT&T's Bell Labs, was much more simple than the famous UNIX shells that came along later. The Thompson shell was distributed with Versions 1 through 6 of UNIX, from 1971 to 1975.

The first UNIX book, *UNIX Programmer's Manual*, by Ken Thompson and Dennis Ritchie, was published in 1971 and included 60 commands.

Computers: Intel 4004 (four-bit microprocessor)
Software: UNIX shell
Games: Computer Space (first commercial vidoe game)
Technology: floppy disk; first electronic calculator (T1)

1972

In 1972 Rockwell released the PPS-4 microprocessor, Fairchild released the PPS-25 microprocessor, and Intel released the 8-bit 8008 microprocessor. All used PMOS.

C was developed from 1969-1972 by Dennis Ritchie and Ken Thompson of Bell Telephone Laboratories for use in systems programming for UNIX. C was based on the earlier programming language B.

UNIX In 1972 work started on converting UNIX to C. The UNIX kernel was originally written in assembly language, but by 1973 it had been almost completely converted to the C language. At the time it was common belief that operating systems must be written in assembly in order to perform at reasonable speeds. This made Unix the world's first portable operating system, capable of being easily ported (moved) to any hardware. This was a major advantage for Unix and led to its widespread use in the multi-platform environments of colleges and universities. Writing UNIX in C allowed for easy portability to new hardware, which in turn led to the UNIX operating system being used on a wide variety of computers.

In 1972, the pipe concept was invented.

Pong, the first arcade video game, was introduced by Nolan Bushnell in 1972. His company was called Atari.

Operating Systems: VM/CMS; UNIX rewritten in C
Programming Languages: C
Computers: Intel 8008 (microprocessor); Rockwell PPS-4 (microprocessor); Fairchild PPS-25 (microprocessor)
Games: Pong; Magnavox Odysssey (first home video game console)
Technology: game console (Magnavox Odyssey); first scientific calculator (HP); first 32-bit minicomputer; first arcade video game

1973

In 1973 National released the IMP microprocessor using PMOS.

In 1973 Intel released the faster NMOS 8080 8-bit microprocessor, the first in a long series of microprocessors that led to the current Pentium.

Quantum Computing Alexander Holevo publishes a paper showing that *n* quibts cannot carry more than *n* classical bits of information (known as "Holevo's theorem").
Charles H. Bennett shows that computation can be done reversibly.

ML (Meta Language) was created in 1973 by R. Milner of the University of Edinburgh. Functional language implemented in LISP.

Actor is a mathematical model for concurrent computation first published bby Hewitt in 1973.

> "Actor is an object-oriented programming language. It was developed by the Whitewater Group in Evanston, Ill." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

UNIX The UNIX kernal was rewritten in the C programming language.

In 1973, UNIX was used at 16 sites worldwide.

ARPA creates Transmission Control Protocol/Internet Protocol (TCP/IP) to network together computers for ARPAnet.

Programming Languages: ML
Computers: National IMP (microprocessor)
Software: Actor
Technology: TCP/IP; ethernet

1974

In 1974 Motorola released the 6800, which included two accumulators, index registers, and memory-mapped I/O. Monolithic Memories introduced bit-slice microprocessing.

SQL (Standard Query Language) was designed by Donald D. Chamberlin and Raymond F. Boyce of IBM in 1974.

AWK (first letters of the three inventors) was designed by Aho, Weinberger, and Kerninghan in 1974. Word processing language based on regular expressions.

Alphard (named for the brightest star in Hydra) was designed by William Wulf, Mary Shaw, and Ralph London of Carnegie-Mellon University in 1974. A Pascal-like language intended for data abstraction and verification. Make use of the "form", which combined a specification and an implementation, to give the programmer control over the impolementation of abstract data types.

> "Alphard is a computer language designed to support the abstraction and verification techniques required by modern programming methodology. Alphard's constructs allow a programmer to isolate an abstraction, specifying its behavior publicly while localizing knowledge about its implementation. It originated from studies at both Carnegie-Mellon University and the Information Sciences Institute." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

> "CLU began to be developed in 1974; a second version was designed in 1977. It consists of a group of modules. One of the primary goals in its development was to provide clusters which permit user-defined types to be treated similarly to built-in types." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

BSD Work on UNIX at University of California, Berkeley, the initial version of BSD UNIX.

Operating Systems: MVS, BSD UNIX
Programming Languages: Alphard; AWK; CLU; SQL
Computers: Intel 8080 (microprocessor); Motorola 6800 (microprocessor); CDC STAR-100 (100 MFLOPS)
Technology: Telenet (first commercial version of ARPANET)

1975

In 1975 Texas Instruments introduced a 4-bit slice microprocessor and Fairchild introduced the F-8 microprocessor.

Scheme, based on LISP, was created by Guy Lewis Steele Jr. and Gerald Jay Sussman at MIT in 1975.

Tiny BASIC created by Dr. Wong in 1975 runs on Intel 8080 and Zilog Z80 computers.

RATFOR (RATional FORtran) created by Brian Kernigan in 1975. Used as a precompiler for FORTRAN. RATFOR allows C-like control structures in FORTRAN.

TENEX shell (also known as the TOPS shell) was created by Ken Greer by September 1975 and merged into the C shell in December 1983. Ken Greer based the shell on the TENEX operating system (hence, the "t" in `tsch`).

UNIX Fifth Edition licensed to universities for free.

Quantum Computing R.P. Poplavskii publishes "Thermodynamical models of information processing" (in Russian) showng the comnputational infeasibility of simulating quantum systems on clasical computers because of the superposition problem.

Programming Languages: RATFOR; Scheme; TENEX shell; Tiny BASIC
Computers: Altair 880 (first personal computer); Fairchild F-8 (microprocessor); MOS Technology 6502 (microprocessor); Burroughs ILLIAC IV (150 MFLOPS)
Technology: single board computer; laser printer (commercial release by IBM)

1976

Design System Language, a forerunner of PostScript, is created in 1976. The Forth-like language handles three dimensional databases.

SASL (Saint Andrews Static Language) is created by D. Turner in 1976. Intended for teaching functional programming. Based on ISWIM. Unlimited data structures.

CP/M, an operating system for microcomputers, was created by Gary Kildall in 1976.

PWB shell The PWB shell or Mashey shell, abbreviated `sh`, was a variation of the Thompson shell that had been augmented by John Mashey and others at Bell Labs. The Mashey shell was distributed with the Programmer's Workbench UNIX in 1976.

Operating Systems: CP/M
Programming Languages: Design System language; SASIL
Computers: Zilog Z-80 (microprocessor); Cray 1 (250 MFLOPS); Apple I
Software: PWB shell
Technology: inkjet printer; Alan Kay's Xerox NoteTaker developed at Xerox PARC

1977

Icon, based on SNOBOL, was created in 1977 by faculty, staff, and students at the University of Arizona under the direction of Ralph E. Griswold. Icon uses some programming structures similar to pascal and C. Structured types include list, set, and table (dictionary).

OPS5 was created by Charles Forgy in 1977.

FP was presented by John Backus in his 1977 Turing Award lecture *Can Programming be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs*.

Modula (MODUlar LAnguage) was created by Niklaus Wirth, who started work in 1977. Modula-2 was released in 1979.

Bourne shell The Bourne shell, created by Stephen Bourne at AT&T's Bell Labs as a scripting language, was released in 1977 as the default shell in the Version 7 Unix release distributed to colleges and universities. It gained popularity with the publication of *The UNIX Programming Environment* by Brian W. Kernighan and Rob Pike. The book was the first commercially published tutorial on shell programming.

BSD UNIX The University of California, Berkeley, released the Berkeley Software Distribution (BSD) version of UNIX, based on the 6th edition of AT&T UNIX.

Programming Languages: Bourne shell; FP; Icon; Modula; OPS5
Computers: DEC VAX-11; Apple II; TRS-80; Commodore PET; Cray 1A
Games: Atari 2600 (first popular home video game consle)

1978

CSP was created in 1978 by C.A.R. Hoare.

> "C.A.R. Hoare wrote a paper in 1978 about parallel computing in which he included a fragment of a language. Later, this fragment came to be known as CSP. In it, process specifications lead to process creation and coordination. The name stands for Communicating Sequential Processes. Later, the separate computer language Occam was based on CSP." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

C Shell The C shell, abbreviated `csh`, was created by Bill Joy, a graduate student at the University of California, Berkeley. With additional work by Michael Ubell, Eric Allman, Mike O'Brien, and Jim Kulp, it was released in the 2BSD release of BSD Unix in 1978.

It became common to use the C shell for everyday interactive computing and to use the Bourne shell for script programming.

UNIX The System V flavor of UNIX started.

Operating Systems: Apple DOS 3.1; VMS (later renamed OpenVMS)
Programming Languages: C shell; CSP
Computers: Intel 8086 (microprocessor)
Games: Space Invaders (arcade game using raster graphics; so popular in Japan that the government has to quadruple its supply of Yen)
Technology: LaserDisc

1979

Modula-2 was released in 1979. Created by Niklaus Wirth, who started work in 1977.

VisiCalc (VISIble CALculator) was created for the Apple II personal computer in 1979 by Harvard MBA candidate Daniel Bricklin and programmer Robert Frankston.

REXX (REstructured eXtended eXecutor) was designed by Michael Cowlishaw of IBM UK Laboratories. REXX was both an interpretted procedural language and a macro language. As a maco language, REXX can be used in application software.

Work started on **C with Classes**, the language that eventually became C++.

Programming Languages: C with Classes; Modula-2; REXX
Computers: Motorola MC68000 (microprocessor); Intel 8088 (microprocessor)
Software: VisiCalc
Games: Lunar Lander (arcade video game, first to use vector graphics); Asteroids (vector arcade game); Galaxian (raster

arcade game, color screen); first Multi-USer Dungeon (MUD, written by Roy Trubshaw , a student at Essex University, forrunner of modern massively multiplayer games); Warrior (first head-to-head arcade fighting game)
Technology: first spreadsheet (VisiCalc); object oriented programming; compact disk; Usenet discussion groups

1980s

Some operating systems from the 1980s include: AmigaOS, DOS/VSE, HP-UX, Macintosh, MS-DOS, and ULTRIX.

The 1980s saw the commercial release of the graphic user interface, most famously the Apple Macintosh, Commodore Amiga, and Atari ST, followed by MIT's X Window System (X11) and Microsoft's Windows.

1980

dBASE II was created in 1980 by Wayne Ratliff at the Jet Propulsion Laboratories in Pasadena, California. The original version of the language was called **Vulcan**. Note that the first version of dBASE was called dBASE II.

SmallTalk-80 released.

Quantum Computing Yuri Manin proposes the idea of quantum computing.

Operating Systems: OS-9
Computers: Commodore VIC-20; ZX80; Apple III
Software: dBASE II
Games: Battlezone (vector arcade video game, dual joystick controller and periscope-like viewer); Berzerk (raster arcade video game, used primative speech synthesis); Centipede (raster arcade video game, used trackball controller); Missile Command (raster arcade video game, used trackball controller); Defender (raster arcade video game); Pac-Man (raster arcade video game); Phoenix (raster arcade video game, use of musical score); Rally-X (raster arcade video game, first game to have a bonus round); Star Castle (vector arcade video game, color provided by transparent plastic screen overlay); Tempest (vector arcade video game, first color vector game); Wizard of Wor (raster arcade video game)

1981

Relational Language was created in 1981 by Clark and Gregory.

Quantum Computing Tommaso Toffoli introduces the reversible Toffoli gate. TOgether with the NOT and XOR gates providing a universal set for classical computation on a quantum computer.

Operating Systems: MS-DOS; Pilot
Programming Languages: Relational Language
Computers: 8010 Star; ZX81; IBM PC; Osborne 1 (first portable computer); Xerox Star; MIPS I (microprocessor); CDC Cyber 205 (400 MFLOPS)
Games: Donkey Kong (raster arcade video game); Frogger (raster arcade video game); Scramble (raster arcade video game, horizontal scrolling); Galaga (raster arcade video game); Ms. Pac-Man (raster arcade video game); Qix (raster arcade video game); Gorf (raster arcade video game, synthesized speech); Zork (first adventure game)
Technology: portable PC; ISA bus; CGA video card

1982

ANSI C The American National Standards Institute (ANSI) formed a technical subcommittee, X3J11, to create a standard for the C language and its run-time libraries.

InterPress, the forerunner of PostScript, was created in 1982 by John Warnock and Martin Newell at Xerox PARC.

Operating Systems: SunOS
Programming Languages: ANSI C; InterPress

Computers: Cray X-MP; BBC Micro; Commodore C64 (first home computer with a dedicated sound chip); Compaq Portable; ZX Spectrum; Atari 5200; Intel 80286 (microprocessor)
Games: BurgerTime (raster arcade video game); Dig Dug (raster arcade video game); Donkey Kong Junior (raster arcade video game); Joust (raster arcade video game); Moon Patrol (raster arcade video game, first game with parallax scrolling); Pole Position (raster arcade video game); Q*bert (raster arcade video game); Robotron 2084 (raster arcade video game, dual joystick); Time Pilot (raster arcade video game); Tron (raster arcade video game); Xevious (raster arcade video game, first game promoted with a TV commercial); Zaxxon (raster arcade video game, first game to use axonometric projection)
Technology: MIDI; RISC; IBM PC compatibles
Theory: Quantum cryptography proposed

1983

Ada was first released in 1983 (ADA 83), with major releases in 1995 (ADA 95) and 2005 (ADA 2005). Ada was created by the U.S. Department of Defense (DoD), originally intended for embedded systems and later intended for all military computing purposes. Ada is named for Augusta Ada King, the Countess of Lovelace, the first computer programmer inmodern times.

Concurrent Prolog was created in 1983 by Shapiro.

Parlog was created in 1983 by Clark and Gregory.

C++ was developed in 1983 by Bjarne Stroustrup at Bell Telephone Laboratories to extend C for object oriented programming.

Turbo Pascal, a popular Pascal compiler, was released.

The University of California at Berkeley released a version of UNIX that included TCP/IP.

tcsh The improved C shell, abbreviated `tcsh`, created by Ken Greer, was merged into the C shell in December 1983. Ken Greer based the shell on the TENEX operating system (hence, the “t” in `tsch`). Mike Ellis, Paul Placeway, and Christos Zoulas made major contributions to `tcsh`.

Korn shell The Korn shell, abbreviated `ksh`, was created by David Korn at AT&T’s Bell Labs and announced at USENIX on July 14, 1983. Mike Veach and Pat Sullivan were also early contributors. The Korn shell added C shell features to the Bourne shell.

In 1983, Richard Stallman founded the GNU project, which eventually provided the tools to go with Linux for a complete operating system. The GNU project intended to create a complete open source UNIX-like operating system, with GNU Hurd as the kernel. Richard Stallman created the GNU General Public License (GPL).

Operating Systems: Lisa OS
Programming Languages: Ada; C++; Concurrent Prolog; Korn shell; Parlog; tcsh; Turbo Pascal
Computers: Apple IIe; Lisa; IBM XT; IBM PC Jr; ARM (microprocessor); Cray X-MP/4 (941 MFLOPS)
Games: Dragon’s Lair (raster arcade video game, first video game to use laserdisc video; note that the gambling device Quarterhorse used the laserdisc first); Elevator Action (raster arcade video game); Gyruss (raster arcade video game, used the musical score Toccata and Fugue in D minor by Bach); Mappy (raster arcade video game, side scrolling); Mario Bros. (raster arcade video game); Spy Hunter (raster arcade video game, musical score Peter Gunn); Star Wars (vector arcade video game, digitized samples from the movie of actor’s voices); Tapper (raster arcade video game); Lode Runner (Apple][E); Journey (arcade video game includes tape of the song *Separate Ways* leading to licensed music in video games)
Technology: math coprocessor; PC harddisk

1984

Objective C, an extension of C inspired by SmallTalk, was created in 1984 by Brad Cox. Used to write NextStep, the operating system of the Next computer.

Standard ML, based on ML, was created in 1984 by R. Milner of the University of Edinburgh.

PostScript was created in 1984 by John Warnock and Chuck Geschke at Adobe.

When the AT&T broke up in 1984 into “Baby Bells” (the regional companies operating local phone service) and the central company (which had the long distance business and Bell Labs), the U.S. government allowed them to start selling computers and computer software. UNIX broke into the System V (sys-five) and Berkeley Software Distribution (BSD) versions. System V was the for pay version and BSD was the free open source version.

AT&T gave academia a specific deadline to stop using “encumbered code” (that is, any of AT&T’s source code anywhere in their versions of UNIX). This led to the development of free open source projects such as FreeBSD, NetBSD, and OpenBSD, as well as commercial operating systems based on the BSD code.

Meanwhile, AT&T developed its own version of UNIX, called System V. Although AT&T eventually sold off UNIX, this also spawned a group of commercial operating systems known as Sys V UNIXes.

UNIX quickly swept through the commercial world, pushing aside almost all proprietary mainframe operating systems. Only IBM’s MVS and DEC’s OpenVMS survived the UNIX onslaught.

Some of the famous official UNIX versions include Solaris, HP-UX, Sequent, AIX, and Darwin. Darwin is the UNIX kernel for Apple’s OS X, AppleTV, and iOS (used in the iPhone, iPad, and iPod).

The BSD variant started at the University of California, Berkeley, and includes FreeBSD, NetBSD, OpenBSD, and DragonFly BSD.

Most modern versions of UNIX combine ideas and elements from both Sys-V and BSD.

Other UNIX-like operating systems include MINIX and Linux.

In 1984, Jim Gettys of Project Athena and Bob Scheifler of the MIT Laboratory for Computer Science collaborated on a platform-independent graphcis system to link together heterogeneous multi-vendor systems. Project Athena was a joint project between Digital Equipment Corporation (DEC), IBM, and MIT. X version 1 was released in May 1984.

Operating Systems: GNU project started; MacOS 1, X Window System (X11)
Programming Languages: Objective C; PostScript; Standard ML
Computers: Apple Macintosh; IBM AT; Apple IIc; MIPS R2000 (microprocessor); M-13 (U.S.S.R., 2.4 GFLOPS); Cray XMP-22
Software: Apple MacWrite; Apple MacPaint
Games: 1942 (raster arcade video game); Paperboy (raster arcade video game, unusual controllers, high resolution display); Punch-Out (raster arcade video game, digitized voice, dual monitors)
Technology: WYSIWYG word processing; LaserJet printer; DNS (Domain Name Server); IDE interface

1985

Paradox was created in 1985 as a competitor to the dBASE family of relational data base languages.

PageMaker was created for the Apple Macintosh in 1985 by Aldus.

The Intel 80386 was the first Intel x86 microprocessor with a 32-bit instruction set and an MMU with paging.

Quantum Computing David Deutsch suggests the first blueprint for the first universal quantum computer, using quantum gates. The universal quantum computer can simulate any other quantum computer with at most a polynomial slowdown.

Operating Systems: GEM; AmigaOS; AtariOS; WIndows 1.0; Mac OS 2
Programming Languages: Paradox
Computers: Cray-2/8 (3.9 GFLOPS); Atari ST; Commodore Amiga; Apple Macintosh XL; Intel 80386

(microprocessor); Sun SPARC (microprocessor)
Software: Apple Macintosh Office; Aldus PageMaker (Macintosh only)
Games: Super Mario Bros.; Tetris (puzzle game; invented by Russian mathematician Alexey Pajitnov to test equipment that was going to be used for artificial intelligence and speech recognition research); Excitebike (first game with battery backup for saving and loading game play)
Technology: desktop publishing; EGA video card; CD-ROM; expanded memory (for IBM-PC compatibles; Apple LaserWriter

1986

Eiffel (named for Gustave Eiffel, designer of the Eiffel Tower) was released in 1986 by Bertrand Meyer. Work started on September 14, 1985.

> "Eiffel is a computer language in the public domain. Its evolution is controlled by Nonprofit International Consortium for Eiffel (NICE), but it is open to any interested party. It is intended to treat software construction as a serious engineering enterprise, and therefore is named for the French architect, Gustave Eiffel. It aims to help specify, design, implement, and change quality software." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

GAP (Groups, Algorithms, and Programming) was developed in 1986 by Johannes Meier, Werner Nickel, Alice Niemeter, Martin Schönert, and others. Intended to program mathematical algorithms.

CLP(R) was developed in 1986.

In 1986, Maurice J. Bach of AT&T Bell Labs published *The Design of the UNIX Operating System*, which described the System V Release 2 kernel, as well as some new features from release 3 and BSD.

Operating Systems: Mach; AIX; GS-OS; HP-UX; Mac OS 3
Programming Languages: CLP(R); Eiffel; GAP
Computers: Apple IIGS; Apple Macintosh Plus; Amstrad 1512; ARM2 (microprocessor); Cray XMP-48
Software: Apple Macintosh Programmer's Workshop
Games: Metroid (one of first games to have password to save game proogress; first female protagonist in video games, non-linear game play; RPG)
Technology: SCSI

1987

CAML (Categorical Abstract Machine Language) was created by Suarez, Weiss, and Maury in 1987.

Perl (Practical Extracting and Report Language) was created by Larry Wall in 1987. Intended to replace the Unix shell, Sed, and Awk. Used in CGI scripts.

HyperCard was created by William Atkinson in 1987. **HyperTalk** was the scripting language built into HyperCard.

Thomas and John Knoll created the program Display, which eventually became PhotoShop. The program ran on the Apple Macintosh.

Adobe released the first version of Illustrator, running on the Apple Macintosh.

Minix Andrew S. Tanenbaum released MINIX, a simplified version of UNIX intended for academic instruction.

Operating Systems: IRIX; Minix; OS/2; Windows 2.0; MDOS (Myarc Disk Operating System); Mac OS 4; Mac OS 5
Programming Languages: CAML; HyperTalk; Perl
Computers: Apple Macintosh II; Apple Macintosh SE; Acorn Archimedes; Connection Machine (first massive parallel computer); IBM PS/2; Commodore Amiga 500; Nintendo Entertainment System
Software: Apple Hypercard; Apple MultiFinder; Adobe Illustrator (Macintosh only; later ported to NeXT, Silicon

Graphics IRIX, Sun Solaris, and Microsoft Windows); Display (which eventually became ImagePro and then Photoshop; Macintosh only); QuarkXPress (Macintosh only)

Games: Final Fantasy (fantasy RPG); The Legend of Zelda (first free form adventure game); Gran Turismo (auto racing); Mike Tyson's Punch Out (boxing sports game)

Technology: massive parallel computing; VGA video card; sound card for IBM-PC compatibles; Apple Desktop Bus (ADB)

1988

CLOS, an object oriented version of LISP, was developed in 1988.

Mathematica was developed by Stephen Wolfram.

Oberon was created in 1986 by Niklaus Wirth.

Refined version of Design becomes ImagePro for the Apple Macintosh.

Operating Systems: OS/400; Mac OS 6
Programming Languages: CLOS; Mathematica; Oberon
Computers: Cray Y-MP; Apple IIc Plus
Software: ImagePro (which eventually became Photoshop; Macintosh only)
Games: Super Mario Bros. 2; Super Mario Bros 3 (Japanese release); Contra (side-scrolling shooter); Joh Madden Football (football sports game)
Technology: optical chip; EISA bus

1989

ANSI C The American National Standards Institute (ANSI) completed the official ANSI C, called the *American National Standard X3.159-1989.*

HTML was developed in 1989.

Miranda (named for a character by Shakespeare) was created in 1989 by D. Turner. Based on SASL and ML. Lazy evaluation and embedded pattern matching.

Standard Interchange Languagewas developed in 1989.

BASH, which stands for Bourne Again SHell, was created by Brian Fox for the Free Software Foundation and first released on June 7, 1989. `bash` combined features from the Bourne shell, the C shell, and the Korn shell. `bash` is now the primary shell in both Linux and Mac OS X.

Operating Systems: NeXtStep; RISC OS; SCO UNIX
Programming Languages: ANSI C; BASH; HTML; Miranda; Standard Interchage Language
Computers: Intel 80486 (microprocessor); ETA10-G/8 (10.3 GFLOPS)
Software: Microsoft Office; first (pre-Adobe) version of Photoshop (Macintosh only)
Games: Sim; SimCity; Warlords (four-player shooter)
Technology: ATA interface, World Wide Web

1990s

Some operating systems from the 1990s include: BeOS, BSDi, FreeBSD, NeXT, OS/2, Windows 95, Windows 98, and Windows NT.

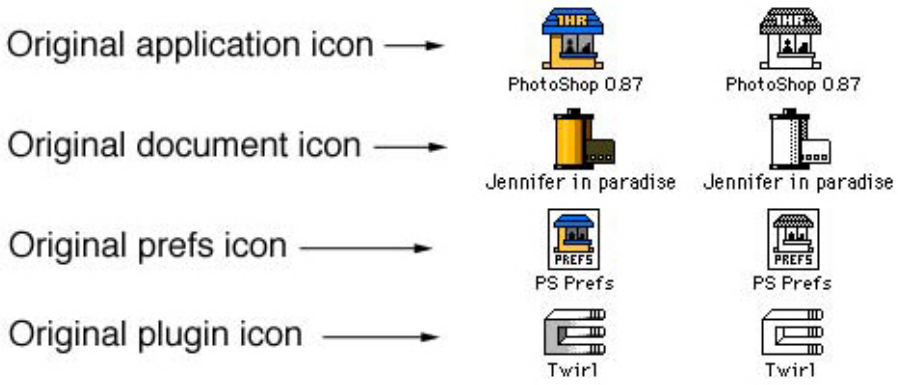
1990

Haskell was developed in 1990.

Tim Berners-Lee of the European CERN laboratory created the World Wide Web on a NeXT computer.

Z shell The Z shell, abbreviated `zsh`, was written by Paul Flastad in 1990 when he was a student at Princton University.

In February of 1990, Adobe released the first version of the program PhotoShop (for the Apple Macintosh). Version 2.0 of Photoshop, code named Fast Eddy, ships in the fall of 1990. Version 1.07 toolbar to right and original icons designed by John Knoll below.



Operating Systems: BeOS; OSF/1
Programming Languages: Haskell; Z shell
Computers: NEC SX-3/44R (Japan, 23.2 GFLOPS); Cray XMS; Cray Y-MP 8/8-64 (first Cray supercomputer to use UNIX); Apple Macintosh Classic; Neo Geo; Super Nintendo Entertainment System
Software: Adobe Photoshop (Macintosh only)
Games: Final Fantasy III released in Japan (fantasy RPG); Super Mario Bros 3 (Japanese release); Wing Commander (space combat game)
Technology: SVGA video card; VESA driver

1991

Python (named for Monty Python Flying Circus) was created in 1991 by Guido van Rossum. A scripting language with dynamic types intended as a replacement for Perl.

Pov-Ray (Persistence of Vision) was created in 1991 by D.B.A. Collins and others. A language for describing 3D images.

Visual BASIC, a popular BASIC compiler, was released in 1991.

Linux operating system was first released on September 17, 1991, by Finnish student Linus Torvalds. With the permission of Andrew S. Tanenbaum, Linus Torvalds started work with MINIX. There is no MINIX source code left in Linux. Linux Torvalds wrote Linux using the GNU C compiler running on MINIX on an Intel 80386 processor.

Linus Torvalds started work on open source Linux as a college student. After Mac OS X, Linux is the most widely used variation of UNIX.

Linux is technically just the kernel (innermost part) of the operating system. The outer layers consist of GNU tools. GNU was started to guarantee a free and open version of UNIX and all of the major tools required to run UNIX.

Operating Systems: Linux kernel; Mac OS 7
Programming Languages: Pov-Ray; Python; Visual BASIC
Computers: Apple PowerBook; PowerPC (microprocessor); PARAM 8000 (India, created by Dr. Vijay Bhatkar, 1GFLOP)
Games: Street Fighter II (shooter); The Legend of Zelda: A Link to the Past (fantasy RPG); Sonic the Hedgehog; Sid

Meyer's Civilization
Technology: CD-i

1992

Dylan was created in 1992 by Apple Computer and others. Dylan was originally intended for use with the Apple Newton, but wasn't finished in time.

Oak, the forerunner of Java, was developed at Sun Microsystems.

In 1992 Unix System Laboratories sued Berkeley Software Design, Inc and the Regents of the University of California to try to stop the distribution of BSD UNIX. The case was settled out of court in 1993 after the judge expressed doubt over the validity of USL's intellectual property.

Operating Systems: Solaris; Windows 3.1; OS/2 2.0; SLS Linux; Tru64 UNIX
Programming Languages: Dylan; Oak
Computers: Cray C90 (1 GFLOP)
Games: Wolfenstein 3D (ffirst fully 3D rendered game engine); Mortal Kombat; NHLPA 93 (multiplayer hockey sports game); Dune II (first real-time strategy game)

1993

> "Vendors such as Sun, IBM, DEC, SCO, and HP modified Unix to differentiate their products. This splintered Unix to a degree, though not quite as much as is usually perceived. Necessity being the mother of invention, programmers have created development tools that help them work around the differences between Unix flavors. As a result, there is a large body of software based on source code that will automatically configure itself to compile on most Unix platforms, including Intel-based Unix.
>
> Regardless, Microsoft would leverage the perception that Unix is splintered beyond hope, and present Windows NT as a more consistent multi-platform alternative." —Nicholas Petreley, "The new Unix alters NT's orbit", NC World

AppleScript, a scripting language for the Macintosh operating system and its application softweare, was released by Apple Computers.

Version 2.5.1 of Photoshop released in 1993. It was one of the first programs to run native on a PowerPC chip. First release of Windows version of Photoshop in April 1993. Toolbar for Photoshop 2.5.1 to the right.

Operating Systems: Windows NT 3.1; Stackware Linux; Debian GNU/Linux; Newton
Programming Languages: AppleScript
Computers: Cray EL90; Cray T3D; Apple Newton; Apple Macintosh TV; Intel Pentium (microprocessor, 66MHz); Thinking Machines CM-5/1024 (59.7 GFLOPS); Fujitsu Numerical Wind Tunnel (Japan, 124.50 GFLOPS); Intel Paragon XP/S 140 (143.40 GFLOPS)
Games: Myst (first puzzle-based computer adventure game; CD-ROM game for Macintosh); Doom (made the first person shooter genre popular, pioneering work in immersive 3D graphics, networked multiplayer gaming, and support for customized additions and modifications; also proved that shareware distribution could work for game distribution); U.S. Senator Joseph Lieberman holds Congressional hearings and attempts to outlaw violent games
Technology: MP3

1994

Work continued on **Java** with a version designed for the internet.

Version 3.0 of Photoshop released in 1994. It included Layers.

Operating Systems: Red Hat Linux

Programming Languages: Java
Computers: Cray J90; Apple Power Macintosh; Sony PlayStation; Fujitsu Numerical Wind Tunnel (Japan, 170.40 GFLOPS)
Games: Super Metroid (RPG)
Technology: DNA computing
Theory: Peter Shor publishes Shor's algorithm for integer factorization wit h a quantum computer.

1995

Java (named for coffee) was created by James Gosling and others at Sun Microsystems and released for applets in 1995. Original work started in 1991 as an interactive language under the name Oak. Rewritten for the internet in 1994.

JavaScript (originally called LiveScript) was created by Brendan Elch at Netscape in 1995. A scripting language for web pages.

PHP (PHP Hypertext Processor) was created by Rasmus Lerdorf in 1995.

Ruby was created in 1995 by Yukihiro Matsumoto. Alternative to Perl and Python.

Delphi, a variation of Object Pacal, was released by Borland

Operating Systems: OpenBSD; OS/390; Windows 95
Programming Languages: Delphi; Java; JavaScript; PHP; Ruby
Computers: BeBox; Cray T3E; Cray T90; Intel Pentium Pro (microprocessor); Sun UltraSPARC (microprocessor)
Software: Microsoft Bob
Games: Chrono Trigger (fantasy RPG); Command & Conquer (real time strategy game)
Technology: DVD; wikis

1996

UML (Unified Modeling Language) was created by Grady Booch, Jim Rumbaugh, and Ivar Jacobson in 1996 by combining the three modeling languages of each of the authors.

Version 4.0 of Photoshop released in 1996. It included a controversial change in the key commands.

Operating Systems: MkLinux
Programming Languages: UML
Computers: Hitachi SR2201/1024 (Japan, 220.4 GFLOPS); Hitachi/Tsukuba CP=PACS/2048 (Japan, 368.2 GFLOPS)
Games: Tomb Raider and Lara Croft, Pokémon; Quake (first person shooter using new 3D rendering on daughter boards); Super Mario 64 (3D rendering)
Technology: USB

1997

REBOL (Relative Expression Based Object language) was created by Carl SassenRath in 1997. Extensible scripting language for internet and distributed computing. Has 45 types that use the same operators.

ECMAScript (named for the European standards group E.C.M.A.) was created in 1997.

Alloy, a structural modelling language, was developed at M.I.T.

Rhapsody The first developer version of Mac OS X released.on August 31, 1997.

To date, the most widely used desktop version of UNIX is Apple's Mac OS X, combining the ground breaking object oriented NeXT with some of the user interface of the Macintosh.

Operating Systems: Mac OS 8; Rhapsody
Programming Languages: Alley; ECMAScript; REBOL
Computers: AMD K6 (microprocessor); Intel Pentium II (microprocessor); PalmPilot; Intel ASCI Red/9152 (1.338 TFLOPS)
Software: AOL Instant Messenger
Games: Final fantasy VII; Goldeneye 007 (one of the few successful movie to game transitions; based on 1995 James Bond movie *Goldeneye*); Castlevania: Symphony of the Night (2D fantasy RPG)
Technology: web blogging

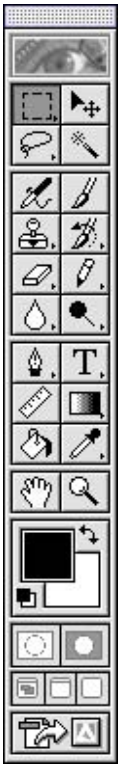
1998

Version 5.0 of Photoshop released in 1998. It included the History palette.

Operating Systems: Windows 98
Computers: Apple iMac; Apple iMac G3; Intel Xeon (microprocessor)
Games: The Legend of Zelda: Ocarina of Time (fantasy RPG); Metal Gear Solid; Crash Bandicoot: Warped

1999

Mac OS X Mac OS X Server 1.0, intended as a server operating system and not for general desktop use, and Mac OS X Developer Preview were released on March 16, 1999.



Version 5.5 of Photoshop released in 1999. It was the first web ready version of Photoshop. Toolbar for Photoshop 5.5 to the right.

Operating Systems: Mac OS 9; Mac OS X Server
Computers: PowerMac; AMD Athlon (microprocessor); Intel Pentium III (microprocessor); BlackBerry; Apple iBook; TiVo; Intel ASCI Red/9632 (2.3796 TFLOPS)

2000s

Some operating systems from the 2000s include: Mac OS X, Syllable, Windows 2000, Windows Server 2003, Windows ME, and Windows XP.

2000

D, designed by Walter Bright, is an object-oriented language based on C++, Java, Eiffel, and Python.

C# was created by Anders Hajlsberg of Microsoft in 2000. The main language of Microsoft's .NET.

RELAX (REgular LAnguage description for XML) was designed by Murata Makoto.

Mac OS X The Public Beta version of Mac OS X was released on September 13, 2000.

Operating Systems: Mac OS 9; Windows ME; Windows 2000
Programming Languages: C#; D; RELAX
Computers: Intel Pentium 4 (microprocessor, over 1 GHz); Sony PlayStation 2; IBM ASCI White (7.226 TFLOPS)
Games: Tony Hawk's Pro Skater 2 (skateboarding sports game); Madden NFL 2001 (football sports game)
Technology: USB flash drive

2001

AspectJ (Aspect for Java) was created at the Palo Alto Research Center in 2001.

Scriptol (Scriptwriter Oriented Language) was created by Dennis G. Sureau in 2001. New control structuress include for

in, while let, and scan by. Variables and literals are objects. Supports XML as data structure.

Mac OS X 10.0 Mac OS X 10.0 Cheetah was released on March 24, 2001.

Mac OS X 10.1 Mac OS X 10.1 Puma was released on September 25, 2001. It included DVD playback.

Operating Systems: Mac OS X 10.0 Cheetah; Mac OS X 10.1 Puma; Windows XP; z/OS
Programming Languages: AspectJ; Scriptol
Computers: Nintendo GameCube; Apple iPod; Intel Itanium (microprocessor); Xbox
Games: Halo
Technology: blade server

2002

Mac OS X 10.2 Mac OS X 10.2 Jaguar was released on August 24, 2002. It included the Quartz Extreme graphics engine, a system-wide repository for contact information in the Address Book, and an instant messaging client iChat.

Operating Systems: Mac OS X 10.2 Jaguar
Computers: Apple eMac; Apple iMac G4; Apple XServe; NEC Earth Simulator (Japan, 35.86 TFLOPS)

2003

Mac OS X 10.3 Mac OS X 10.3 Panther was released on October 24, 2003. It included a brushed metal look, fast user switching, Exposé (a Window manager), Filevault, Safari web browser, iChat AV (which added videoconferencing), imporved PDF rendering, and better Microsoft Windows interoperability.

Operating Systems: Windows Server 2003; Mac OS X 10.3 Panther
Computers: PowerPC G5; AMD Athlon 64 (microprocessor); Intel Pentium M (microprocessor)
Games: Final Fantasy Crystal Chronicles

2004

Scala was created February 2004 by Ecole Polytechnique Federale de Lausanne. Object oriented language that implements Python features in a Java syntax.

Programming Languages: Scala
Computers: Apple iPod Mini; Apple iMac G5; Sony PlayStation Portable; IBM Blue Gene/L (70.72 TFLOPS)
Games: Fable
Technology: DualDisc; PCI Express; USB FlashCard

2005

Job Submission Description Language.

Mac OS X 10.4 Mac OS X 10.4 Tiger was released on April 29, 2005. It included Spotlight (a find routine), Dashboard, Smart Folders, updated Mail program with Smart mailboxes, QuickTime 7, Safari 2, Automator, VoiceOver, Core Image, and Core Video.

Operating Systems: Mac OS X 10.4 Tiger

Programming Languages: Job Submission Description Language
Computers: IBM System z9; Apple iPod Nano; Apple Mac Mini; Intel Pentium D (microprocessor); Sun UltraSPARC IV (microprocessor); Xbox 360
Games: Lego Star Wars

2006

Computers: Apple Intel-based iMac; Intel Core 2 (microprocessor); Sony PlayStation 3
Technology: Blu-Ray Disc

2007

Mac OS X 10.5 Mac OS X 10.5 Leopard was released on October 26, 2007. It included an updated Finder, Time Machine, Spaces, Boot Camp, full support for 64-bit applications, new features in Mail and iChat, and new security features.

Operating Systems: Apple iOS (for iPhone); Mac OS X 10.5 Leopard; Windows Vista
Computers: AMD K10 (microprocessor), Apple TV; Apple iPhone; Apple iPod Touch; Amazon Kindle

2008

Operating Systems: Google Android
Computers: Android Dev Phone 1; BlackBerry Storm; Intel Atom (microprocessor); MacBook Air; IBM RoadRunner (1.026 PFLOPS); Dhruva (India, 6 TFLOPS)

2009

Mac OS X 10.6 Mac OS X 10.6 Snow Leopard was released on August 28, 2009. it included the SquirrelFish javaScript interpreter, a rewrite of Finder in the Cocoa API, faster Time Machine backups, more reliable disk ejects, a more powerful version on Preview, a faster version of Safari, Microsoft Exchange Server support for Mail, iCal, and Address Book, QuickTime X, Grand Central Dispatch for using multi-core processors, and OpenCL support.

Operating Systems: Windows 7; Mac OS X 10.6 Snow Leopard
Computers: Motorola Driod; Palm Pre; Cray XT5 Jaguar (1.759 PFLOPS)

2010

Computers: Apple iPad; IBM z196 (microprocessor); Apple iPhone 4; Kobo eReader; HTC Evo 4G

2011

Mac OS X 10.7 Mac OS X 10.7 Lion was released on July 20, 2011. it included Launchpad, auto-hiding scrollbars, and Mission Control.

Operating Systems: Mac OS X 10.7 Lion

2012

Mac OS X 10.8 Mac OS X 10.8 Mountain Lion was released on July 25, 2012. It included Game Center, Notification Center, iCloud support, and more Chinese features.

Operating Systems: Mac OS X 10.8 Mountain Lion, Windows 8

2013

Mac OS X 10.9 Mac OS X 10.9 Mavericks was released on October 22, 2013.

Operating Systems: Mac OS X 10.9 Mavericks, iOS7, WIndows 8.1
Games: Sony PlayStation 4

critical reasoning

appendix C

summary

This appendix looks at critical reasoning.

This is the beginning of the much-promised appendix on critical reasoning. At the moment I am attempting to get the basic structure laid down. This presentation will be too fast paced to be useful to a student, but will help organize the themes that must be discussed to adequately present the material.

Students who want to learn this material can use this as an outline. Look up the appropriate topics in Wikipedia and you will find a brief discussion on each topic. Wikipedia isn't organized for teaching, but using this appendix as a study outline and Wikipedia for information should allow you to cobble together your own lessons.

Important Note: While critical reasoning is a great tool for computers, it clashes with most social institutions, including business, law, and government. Business persons and government officials do not normally react well to critical reasoning and science. Do not confuse the two realms. Use critical reasoning and mathematcis in your work, but switch to social skills for all interactions with those in positions of power or authority.

tinkering and thought

The vast majority of programmers, system administrators, data base administrators, and other tech personnel are tinkerers.

Tinkerers have no real understanding of the objects they work on. Tinkerers use trial and error until things work better.

The best tinkerers are successful because they have a good intuition about what guesses might work and over time they have collected a set of superstitutions that work most of the time for the most common problems.

The very best programmers, system administrators, data base administrators, and other tech personnel use rational thought and critical reasoning to understand how a system works and fails and uses a combination of knowledge, skill, and imagination to craft appropriate solutions.

This chapter is about how to use critical reasoning. The vast majority of readers can safely skip over this material.

Engineering and Business

Engineering and business are fundamentally different. This web page is intended to help business persons successfully make use of engineers.

I will describe the fundamental difference, outline the basic web services, and discuss how to get engineers to deliver what your business needs

fundamental difference

The fundamental difference is that business is a **social** activity and engineering is a **mathematical** activity. The computer is a pure mathematical machine.

In engineering (as well as science and mathematics), two plus two (2+2) always equals four (4), for all times and places, regardless of culture. The answer is the same for Stephen Hawking in modern Great Britain and Euclid in Egypt some two thousaand years ago.

In business (as well as law and government), two plus two might be equal to five-and-a-quarter, trending up. But if it reaches six, consider that a bubble and sell before it falls. If someone is rich or powerful enough, they can demand that two plus two be whatever they need it to be — and nobody dare risk their wrath for failure to obey.

Engineering is **absolute** and business is **relative**.

A business person needs to be aware that engineers think in a completely different manner and that their machines do **not** answer to money or power. Computers answer only to the laws of mathematics.

The reason that business persons often see disconcerting, strange characters among the geeks is because they are highly skilled at mathematics and mathematics doesn't take into account fashion or custom.

business and government

Business, government, law, and most social institutions are all subjective perception.

The clash between subjective business perceptions and the harsh absolutism of mathematics and computers is often confounding to programmers and other technical staff.

It makes sense to base business on perception because consumerism is based not on actual needs but on selling wants and desires.

It makes sense to base government and law on perception because that allows those in power to continually adjust their rulings to meet their own needs.

perception and proof

Reality is perceived. Mathematics is proven.

The existence of optical illusions illustrates the inherent lack of reliability of our perceptions of reality. Different testmony by witnesses to the same event highlight differences in perception by different observers. Some philosphers have questioned whether an objective reality even exists.

Mathematics is pure in abstraction. Mathematics has clear definitions and axioms, from which theorems can be proven true or false.

Philosophers have long examined this distinction.

Imhotep, the chief architect of the first two true pyramids in ancient Egypt, inventor of the scientific method, inventor of medical triage, and inventor of trigonometry, claimed that mathematics were the fundamental divine laws of physical reality.

Modern applied mathematics is based on the idea that mathematics models reality.

Computers simulate pure mathematics. Unlike pure math, physical hardware is subject to momentary glitches, such as electrical spikes, cosmic rays, and distant earthquakes.

objectivity and subjectivity

The goal in critical reasoning is objectivity.

The goal in artwork is subjectivity.

Both are valid goals. Kung Fu-Tze (Confucious), Aristotle, and Buddha all proposed some variation of the belief that humans need balance.

A few hundred years ago, European artists attempted to match exactly what they saw, which was called realism. Impressionism moved to attempting to capture the essential impression of a scene. Surrealism bent and warped reality. Abstract expressionism attempted to capture the abstract essence. Numerous other art movements employed new and different ways to look at the world and express artistic creativity.

Objectivity calls for a fair examination, an attempt to determine a greater truth.

Objective journalism asks six questions; who, what, when, where, why, and how.

classifications

summary

This chapter looks at classifications.

Classifying is the act of placing things into groups.

A famous example is the Dewey Decimal Classification used by some libraries.

concepts and referents

A **concept** is a category.

A **referent** is aspecific member of a category.

Fido is a referent in the concept of dogs. Cats and dogs are referents in the concept of mammals. Mammals and reptiles are referents in the concept of animals.

species and genus

A **species** is a smaller, more specific group, while a **genus** is a larger, more general group.

Using the above example, cats and dogs are species of the genus mammals, while mammals and reptiles are species of the genus animals.

Note that these older philosophical concepts of species and genus predate the more complex modern biological system of categories.

abstract and concrete

Each higher layer of genus becomes more abstract. Each lower layer of species becomes more concrete.

Abstract is a relative term.

Concrete is not limited to physical items. For example, specific emotions (sad, happy) are more concrete than emotional states.

classes and objects

You see this same kind of grouping in object oriented programming. A class is a genus (concept) and the various objects are the species (referent).

A **class** defines the characteristics (data) and behaviors (methods) of a group.

An **object** is a specific individual (instance) of a group.

rules of classification

There are two basic rules for good classifications:

(1) A set of classifications must be used according to a consistent principle that is mutually exclusive and jointly exhaustive.

(2) Classifications must be made on essential attributes.

mutually exclusive

Mutually exclusive means that any particular species can only be in a single genus.

Green characters and Muppets are not mutually exclusive, because Kermit the Frog qualifies for both genuses.

jointly exhaustive

Jointly exhaustive means that all of the species taken together (jointly) will make up the complete (exhaustive) set of the genus.

The species felines (cats) and canines (dogs) are not jointly exhaustive for the genus mammals, because there are still many additional creatures that qualify as mammals, but aren't felines or canines.

consistent principle

It is important to have a consistent principle for organizing your classifications.

The species of (1) green things and (2) big things and (3) mammals are not consistent because the first measure is color, the second size, and the third a biological group.

The species of red, green, and blue are consistent because they are all colors.

essential attributes

It is important that the choice of rule for classification be based on an essential attribute.

Organizing books by the colors of the covers would create a bizarre library, while organizing books by subject (such as the Dewey Decimal Classification) creates a library that is easy to use.

This is because the subject matter (or author) is an essential characteristic of a book, while the color of the cover isn't. The color of the cover can change from printing to printing.

levels of organization

The levels of ogranization need to make sense.

As an example, the Dewey Decimal Classification is made up of ten classes. Each class is divided into ten divisions.

Each division is divided into ten sections.

Class 0 is Computer science, information & general works. Division 00 is Computer science, information & general works. Section 004 is Data processing & computer science, 005 is Computer programming, programs & data, and 006 is Special computer methods.

- 000 Computer science, information & general works
  - 000 Computer science, information & general works
    - 000 Computer science, knowledge & general works
    - 001 Knowledge
    - 002 The book (i.e. Meta writings about books)
    - 003 Systems
    - 004 Data processing & computer science
    - 005 Computer programming, programs & data
    - 006 Special computer methods
    - 007 (Unassigned)
    - 008 (Unassigned)
    - 009 (Unassigned)
  - 010 Bibliographies
  - 020 Library & information sciences
  - 030 Encyclopedias & books of facts
  - 040 Biographies
  - 050 magazines, journals & serials
  - 060 Associaitons, organizations & museums
  - 070 News media, journalism & publishing
  - 080 General collections
  - 090 Manuscripts & rare books
- 100 Philosophy and psychology
- 200 Religion
- 300 Social sciences
- 400 Language
- 500 Science
- 600 technology
- 700 Arts & recreation
- 800 Literature
- 900 History & geography

Make sure that your categories have a species-genus relationship. For example, an enzyme is a kind of protein and therefore they have a species-genus relationship, but an amino acid is is a component part of a protein and therefore they don’t have a species-genus relationship.

outlines

The outlines that you were taught in school are an example of a proper organization of classifications. Outlines developed specifically because they are useful tools for organizing a well-reasoned argument, presentation, or essay.

I. first main topic
   A. first subtopic
      i. first inner topic
         a. first innermost topic
         b. second innermost topic
      ii. second inner topic
         a. first innermost topic
         b. second innermost topic
   B. second subtopic
      i. first inner topic

            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
    C. third subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
II. second main topic
    A. first subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
    B. second subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
    C. third subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
III. third main topic
    A. first subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
    B. second subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic
    C. third subtopic
        i. first inner topic
            a. first innermost topic
            b. second innermost topic
        ii. second inner topic
            a. first innermost topic
            b. second innermost topic

elements

In Western antiquity, the ancient Egyptians and the Greeks developed the idea of four or five basic elements: fire, water, air, earth, and spirit. The Chinese developed the five basic elements of wood, fire, earth, metal, and water.

The Western elements represented the basic underlying principles of natures: three states of matter (earth = solid, water = liquid, and air = gas) and energy (fire = energy). The spirit or soul was sometimes included as a fifth element.

In modern Western civilization the word element was purposely reused to describe the fundamental chemical elements that make up matter.

definition

summary

This chapter looks at definitions.

The first rule of philosophy is to define your terms. The reason is to create clarity and make sure that everyone is discussing the same thing.

Churchill

There is a famous example from World War II. In American use, the term “table” means to put a matter aside. In the rest of the English-speaking world, the term “table” means to begin discussion on a matter.

Winston Churchill describes how this became a source of confusion during World War II in his book *The Second World War, Volume 3: The Grand Alliance*.

> The enjoyment of a common language was of course a supreme advantage in all British and American discussions. … The British Staff prepared a paper which they wished to raise as a matter of urgency, and informed their American colleagues that they wished to “table it.” To the American Staff “tabling” a paper meant putting it away in a drawer and forgetting it. A long and even acrimonious argument ensued before both parties realized that they were agreed on the merits and wanted the same thing.

dictionaries

Why not just use a dictionary?

Dictionaries serve a different purpose. The definitions in dictionaries are about words. The definitions in philosophy are about concepts.

Even though we have a different purpose for definition in critical reasoning, a dictionary definition can be a good starting point for coming up with a useful philosophical definition.

A **Webster’s** style dictionary (there are many different brands, not one single version) attempts to record how the common person uses the language. The first was Noah Webster’s 1928 edition of the *American Dictionary of the English Language*.

An **authorative** dictionary attempts to record expert use of the language (such as the works of great writers). Originally authorative dictionaries recorded how the king or queen wanted the language to be used. The famous example is the *Oxford English Dictionary*, which is printed on behalf of British royalty.

A **lexicon** is a specialized dictionary that records the words and phrases for a particular use, such as a law or science dictionary. The famous example is *Black's Law Dictionary*.

boundaries

One of the most important aspects of a good definition is setting boundaries. A definition divides the universe into that which fits the definition and that which doesn't fit the definition.

It is generallly easy to deal with cases that are clearly within a definition and cases that are clearly outside of a definition. The problem is dealing well with the cases that are near the borders. In computer science, these are sometimes called the "corner cases".

precision and accuracy

Precision is how exacting something is. For example, 3.14159 is more precise than three.

Accuracy is how correct something is.

The classic example is:

> "2 + 2 = 3.9999" is very precise, but inaccurate.
>
> "Two plus two is less than ten" is accurate, but imprecise.

We want any definition to be accurate.

There are always degrees of precision. In philosophy we want to be as precise as necessary. We want to be precise enough to avoid confusion, but have no need to go to extremes for precision's own sake.

clarity

The basic function of a definition in critical reasoning is to be clear. Our goal is to avoid confusion.

Working towards great clarity in definitions actually helps us better understand the topics we are thinking about.

In the last half century or so there has been a lot of work on how language and words powerfully control our thoughts and even what we are able to think of.

summary

A good definition should summarize the concept.

genus and differentia

A classic test of a good definition for critical reasoning is that it includes a genus and a differentia.

The genus let's us know the class of concepts.

The differentia (difference) let's us know how this particular concept difefrs from other similar concepts.

In normal speech it is common to leave the genus implied. For good critical reasoning, it is important to actually state the

genus.

In some cases the genus is essential for distinguishing between multiple meanings of a word.

classical logic

summary

This chapter looks at classical or Aristotelian logic.

Aristotle was a Greek philosopher and scientist who lived from 384 to 322 B.C.E. in Athens, Greece. He was a puil of Plato, who in turn was a pupil an dfriend of Socrates.

logical argument

Aristotle claimed that all logical argument could be reduced to two premises (a major premise and a minor premise) and a conclusion.

three laws

Aristotle created three basic laws or principles of logical reasoning:

1. **The principle of identity**
   A thing is itself: *A* is *A*.
2. **The principle of the excluded middle**
   A proposition is either true or false: either *A* or *not A*.
3. **The principle of contradiction**
   No proposition can be both true and false: *A* cannot be both *A* and *not A*.

engineering

summary

This chapter looks at engineering.

engineering method

There are numerous variations of the engineering method. There is no official version. Most have four steps, although five and six step versions also exist. In general, all of the versions include the same basic stuff, but divide it up differently or express it differently.

1. Define problem.
2. Brainstorm solutions.
3. Evaluate and choose one method
4. Plan implementation

1. The first step is to identify the actual problem that is being solved. The vast majority of projects, especially those created by government or business, start instead with a statement of a particular solution.

In engineering, it is best to start with a clear, neutral definition of the problem that must be solved.

2. The second step is to look at all possible solutions. This includes examining the existing record to see how others have solved the same or similar problems.

3. The third step is to evaluate the risks, benefits, costs, advantages, and disadvantages of the possible solutions and determine which solution best meets the needs and budget.

4. The fourth and final step is to come up with a plan to implement the solution. In computer science, this centers on the software design.

Forth-like routines for Unix/Linux shell

appendix D

summary

This chapter looks at a set of Forth-like routines for use in the BASH UNIX (Linux, Mac OS X) shell.

You can obtain the latest copy of these functions at http://www.osdata.com/programming/shell/forthfunctions place the file in an appropriate directory.

warning

At the time this paragraph was written the www.osdata.com was hosted on HostGator and at the time this paragraph was written HostGator tech support used the last four digits of a credit card as their security method for identifying the account holder. This is a well-known security flaw. The security hole was made famous when a hacker used this hole at Apple and Amazon to get Mat Honan's account information and wipe out his hard drive. Both Apple and Amazon quickly changed their security procedures, but for some reason HostGator (at the time this was written) hasn't bothered to change their security procedures. See http://www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking/ for details.

For this reason, there is the distinct possibility that the forthfunctions file might have been hacked. Please take the time to read through it and make sure it looks correct and doesn't have any suspicious or malicious code inserted. I have a distinctive programming style, so something written by someone else might stand out even if you don't understand what I have written. Of course, it is better if you understand the code and can spot things that look wrong.

I apologize for this security hole, but my website is hosted for free for me and I have no control over the hosting company chosen. The person providing me with the free hosting doesn't believe me that this is a security flaw. This is one of the many hassles of poverty.

basic approach

This supplementary material describes a method for creating Forth-like routines for use in the BASH shell.

The data and return stacks will be stored using a pair of shell variables, one of which is an array and the other is an ordinary scalar variable.

The Forth-like routines will be created as shell functions.

Everything will be stored in a shell function file that can be loaded manually or loaded as part of the normal shell start-up.

Eventually I intend to write two sets of C source code, one that people can compile into their own copy of the official BASH source code and one that is part of an entirely new open source shell project that doesn't have the GNU infecting license. This alternative shell will be designed so that it can run on bare metal, allowing it to be used to deploy embedded systems.

stacks

Forth uses two stacks: the main data stack and the return stack.

For the purposes of the BASH shell, we will implement both stacks with a combination of a top of stack

pointer variable and an array for the stack. The top of stack pointer will hold the array index of the top of the stack. The stack will build up from the array indexed as one. An empty stack is indicated by having the top of stack pointer variable point to the zero index of the stack array variable. The bottom of the stack will actually be the one index of the stack array variable.

```
$ export DataStackPointer=0
$ declare -a DataStack; export DataStack
$ export ReturnStackPointer=0
$ declare -a ReturnStack; export ReturnStack
$
```

That's it for the required variables! And, yes, those could all be placed on one line, but I spread them over four lines for additional clarity.

One very important difference between the real Forth stacks and the Forth-like function stacks is that the real Forth stacks deal with binary data and the Forth-like function stacks deal with character strings. Each item on a real Forth stack is a binary number (or binary bit string). Even characters are stored by their binary representation. Every item on a Forth-like function stack is a character string. Numbers are stored by their character string representation. And an individual item on a Forth-like stack can be a character string of arbitrary length, even an entire book.

direct random access

While Forth insists on the stack always being accessed as a stack, there is nothing that prevents direct random access to Forth-like stack elements.

A shell script mixing BASH and the Forth-like functions can directly read or write any individual element in the data or return stacks. For that matter, a mixed shell script can do any manipulations to the stacks. It is the programmer's responsibility to maintain stack integrity.

One particular gotcha is that I am not currently emptying out stack locations that are no longer used. This means that when the stack shrinks, there will be entries in the array that are no longer valid. Don't be fooled by their existence. Modify my scripts to delete this information if it presents a security risk of any kind.

functions

These are the functions for a Forth-like working environment. Not all of these functions exist in a real Forth. Most of these functions are have modifications from standard Forth. Many standard Forth functions are missing.

In particular, this approach treats the stack as a stack of objects. No matter how much memory space an object takes up, it only takes up a single stack slot. In a real Forth, data is stored on the stack as raw binary strings, characters, or numbers and the programmer is responsble for keeping track of how much memory every item actually uses.

stack functions

The double number versions of Forth stack functions are purposely left out. Our stack elements are variable size objects and can be any size number, as well as strings or characters.

ShowDataStack

The `ShowDataStack` function shows the current contents of the data stack in the order of top of stack to

bottom of stack. This function is built for debugging purposes, but I am making it available for anyone to use.

```
$ ShowDataStack ()
>{
>    echo "stack size of " $DataStackPointer;
>    if [ "$DataStackPointer" -gt "0" ] ; then
>        loopcounter=$DataStackPointer
>        while [ $loopcounter -gt 0 ] ;
>        do
>            echo ${DataStack["$loopcounter"]}
>            let "loopcounter=$loopcounter - 1"
>        done
>    else
>        echo "the stack is empty"
>    fi
>    unset loopcounter #clear out variable
>} # end ShowDataStack
$
```

push

The `push` function pushes a single object onto the data stack. Forth doesn't have this word because simply typing a number is sufficient to push the number onto the stack. This function will be used by other functions to manipulate the data stack.

NOTE: This function does *not* yet have error checking. Need to check for exactly one item to push.

```
$ push ()
>{
>    let "DataStackPointer=$DataStackPointer + 1"; #preincrement
>    DataStack["$DataStackPointer"]="$1"; #push data item on top of stack
>} # end push
$
```

It is very important to notice that the item to be pushed onto the stack comes *after* the Forth-like function `push`, which is very different than the normal order for Forth. This variation appears again in other Forth-like functions and is done to work within the limits of the expectations of the BASH shell.

Note that `push` will fail inside a series of piped commands or any other situation with a subshell.

pop

The `pop` function pops a single object from the data stack. Forth has a different function for this functionality, but this function is intended for internal use by other functions to manipulate the data stack.

NOTE: This function does *not* yet have error checking. Need to check for empty stack.

```
$ pop ()
>{
>    let "DataStackPointer=$DataStackPointer - 1" #postdecrement
>} # end pop
$
```

popout

The `popout` function pops a single object from the data stack and prints it out for use in a pipe. Forth has a different function for this functionality, but this function is intended for internal use by other functions to manipulate the data stack.

Note that the value is printed without a new line character, so it might seem to "disappear" into the beginning of the prompt.

Use the `DOT` function to pop and print with a new line.

NOTE: This function does *not* yet have error checking. Need to check for empty stack.

```
$ popout ()
>{
>    printf ${DataStack["$DataStackPointer"]} #pop data item from top of
stack and print
>    let "DataStackPointer=$DataStackPointer - 1" #postdecrement
>} # end popout
$
```

the power of Forth-like functions

For those who don't already know the power of Forth, let me use the shell commands `pushd`, `popd`, and `dirs` as an example. For those who already know the power of Forth, you can either read along and nod about how great Forth is, or skip ahead to actual implementation of Forth-like functions for the BASH shell.

directory stack

You may have noticed that the C shell and BASH include the `pushd`, `popd`, and `dirs` command to manipulate a stack of directories.

In case you never noticed these commands or have forgotten what they do, a quick summary:

`pushd` allows the user or a script to push the current working directory onto a stack of directories and change to a new named directory. The command acts slightly differently if you leave out a directory name. We won't discuss that case because it takes us on a tangent from our current discussion.

You can simulate the normal use of the shell command `pushd` (named directory) with the Forth-like function `push` and shell command `cd`.

```
$ push `pwd`; cd testdir
$
```

You have to explicitly place the current working directory onto the stack (using `pwd` and backticks) and you have to issue a second command to actually change the directory. Don't worry about the extra steps, because you will continue to use `pushd` for its intended purpose. We are showing the value of Forth through analogy to something you already use to save time and effort.

`popd` allows the user or a script to pop the directory stack and change to that directory.

You can simulate the normal use of the shell command `popd` with the Forth-like function `pop` and shell command `cd`.

```
$ cd `pop`
$
```

Notice that we used the results of the `pop` function as the parameter for the shell command `cd`.

`dirs` allows the user or a script to see all of the directories on the directory stack.

You can simulate the normal use of the shell command `dirs` with the Forth-like function `ShowDataStack`.

```
$ ShowDataStack
stack size of  1
/Users/admin
$
```

taking it to the next level

At this point, there is no advantage to using our Forth-like functions rather than using the already existing shell commands. It is a good idea to continue to use the shell commands for their intended use. Less typing.

You could also use named variables for saving directory names. By using short variable names, you can save a lot of typing over typing out the full path names. And you don't have to memorize a bunch of long path names. Of course, you have to memorize all of your named variables.

The shell commands `pushd`, `popd`, and `dirs` were developed because there are a lot of advantages of a stack over a bunch of named variables.

With a stack, you don't have to memorize any alternate short variable names for directory paths. And you can use the same directory stack for work that moves from an old set of directories to a new set of directories.

There is a lot of power in having a stack and three simple commands for manipulating the stack.

more sophisticated directory stack use

The shell commands `pushd` and `popd` only work with the single directory that is at the top of the stack. Yes, you can see the entire stack with the `dirs` command and use one of several alternate techniques for picking one directory out of the middle of the stack.

What if you could easily swap the directory that was on the top of the stack with the directory that was one down? You could then use the `popd` command to switch to the directory that was formerly one down from the top of the stack and the former top of the stack would still be in the stack, ready for use.

As it turns out, Forth has a command called `SWAP` that does exactly that operation. Forth has a rich set of stack operation words, that can do things like move (or pick) an aribitrary item from deep in the stack to the top of the stack, make a duplicate (or dup) of the top of the stack, delete (or drop) the item on the top of the stack, and other cool things.

The Forth-like functions bring variations of these stack operation words to the shell.

arithmetic computations

Both Forth and the Forth-like functions work on much more than just directories.

A common use of Forth is computations.

Many of the scientific and engineering calculators (which used to be stand-alone devices, but are now commonly apps) use a system called Reverse Polish Notation or RPN

An RPN calculator is a very powerful and efficient way of performing calculations.

For ordinary calculations, such as adding a series of numbers, RPN only saves a single keystroke. That's not enough to justify learning an entirely different method of calculating.

But when the calculations start to become complicated, RPN really shines.

With standard algebraic notation, you suddenly have to start typing a whole bunch of open and close parenthesis and figure out all of your grouping and keep track of how many layers of parenthesis are currently still open so that at the end of your calculation all of the open and close parenthesis balance out. Anyone who has done any C programming knows about this hassle.

With RPN, there are no parenthesis. Once you learn to think in RPN, any arbitrarily complex algebraic computation has a straight-forward and simple RPN expression!

You dramatically save typing, you avoid the headache of balancing nested parenthesis, and you have confidence in the correctness of your straight-forward expression.

Well, both Forth and the Forth-like functions use the RPN system for computing.

You bring the power of scientific and engineering calculators to the BASH shell. Well, actually, you will need to write a few additional functions of your own to have all of the features of your favorite scientific or engineering calculator, but you have all of the tools available to finish out that task.

This means that you can use RPN calculations in all of your shell scripts.

strings

Both Forth and the Forth-like functions can also do manipulations of characters and character strings.

The Forth-like functions store all information as character strings, even numbers. This is different than real Forth, but has its own advantages and disadvantages.

BASH and all of the other shells work with streams of characters. Storing everything (even numbers) on a special string-oriented stack allows our Forth-like functions to play nice with the BASH shell.

This allows us to combine all of the power of the BASH shell with much of the power of Forth.

the power of Forth

The power of Forth lies in three characteristics:

(1) Forth has powerful stack-based processing. The Forth-like functions bring this power to the BASH shell in a manner compatible with BASH's expectations.

(2) Forth is extensible. You can build new Forth-like functions by threading together other Forth-like functions. You won't have quite the flexibility of real Forth and you won't have the Forth advantage of all new Forth words being treated the same as all built-in threaded words. But with modern processors, you will be pretty close to the power of real Forth.

(3) Forth can run on its own without an operating system. This one can't be done with the BASH shell. I do intend to eventually write the code to combine a stripped down shell and Forth as native code for use in embedded systems. The Forth-like functions are intended as a demo to prove that adding Forth to the shell is a

dramatic improvement in both power and flexiibilty.

Once you learn how to use the Forth-like functions in your shell scripts (and even at the interactive command line), you will gain a huge amount of added power and possibilities for your shell scripts.

The one big hassle is that you can't simply import existing Forth programs, which is unfortunate given the huge library of very useful open source and public domain Forth programs.

I do intend to write a translator, which will convert existing Forth programs into their BASH shell Forth-like equivalent.

With that introduction to the power of Forth and Forth-like functions, let's take a look at the actual Forth-like functions.

and now, the functions

dot

The `dot` function pops a single object from the top of the data stack and echos the result. Forth uses the period (`.`) for this word, but the period is already used for a different BASH command, creating a namespace conflict. Hence, the name `dot`.

NOTE: This function does *not* yet have error checking. Need to check for empty stack.

```
    $ dot ()
    >{
    >    echo ${DataStack["$DataStackPointer"]} #pop data item from top of
stack and print
    >    let "DataStackPointer=$DataStackPointer - 1" #postdecrement
    >} # end dot
    $
```

DUP

The `DUP` function is nearly the same functionality as the Forth word `DUP` (the difference is that this function works with character strings). The top item on the stack is duplicated, increasing the size of the stack by one.

NOTE: This function does *not* yet have error checking. Need to check for empty stack.

```
    $ DUP ()
    >{
    >    temporary=${DataStack["$DataStackPointer"]} #make temporary copy of
data item from top of stack
    >    let "DataStackPointer=$DataStackPointer + 1" #preincrement
    >    DataStack["$DataStackPointer"]=$temporary #store duplicate
    >    unset temporary #clear out variable
    >} # end DUP
    $
```

qDUP

The `qDUP` function is nearly the same functionality as the Forth word `?DUP` (the difference is that this function works with character strings). The top item on the stack is duplicated only if it is non-zero, increasing the size

of the stack by one if the item on the top of the stack is non-zero.

NOTE: This function does *not* yet have error checking. Need to check for empty stack and non-number object on top of stack.

```
    $ qDUP ()
    >{
    >    if [ ${DataStack["$DataStackPointer"]} -eq 0 ] ; then #test for zero
string on top of stack
    >        return 0 #if zero, take no action, return OK
    >    fi
    >    temporary=${DataStack["$DataStackPointer"]} #make temporary copy of
data item from top of stack
    >    let "DataStackPointer=$DataStackPointer + 1" #preincrement
    >    DataStack["$DataStackPointer"]=$temporary #store duplicate
    >    unset temporary #clear out variable
    >} # end qDUP
    $
```

OVER

The `OVER` function is nearly the same functionality as the Forth word `OVER` (the difference is that this function works with character strings). The second item down from the top of the stack is duplicated, increasing the size of the stack by one.

This function uses all caps, `OVER`, to prevent conflict with the already existing `over` UNIX tool for pretty printing and scrolling source code on a terminal.

NOTE: This function does *not* yet have error checking. Need to check for empty stack or stack with only one item in it.

```
    $ OVER ()
    >{
    >    let "TemporaryPointer=$DataStackPointer - 1" #create pointer down
one from top of stack
    >    temporary=${DataStack["$TemporaryPointer"]} #make temporary copy of
data item one down in stack
    >    let "DataStackPointer=$DataStackPointer + 1" #preincrement
    >    DataStack["$DataStackPointer"]=$temporary #store duplicate
    >    unset temporary #clear out variable
    >    unset TemporaryPointer #clear out variable
    >} # end OVER
    $
```

PICK

The `PICK` function is nearly the same functionality as the Forth word `PICK` (the difference is that this function works with character strings). The n^{th} item down from the top of the stack is duplicated, increasing the size of the stack by one.

NOTE: This function does *not* yet have error checking. Need to check that a number was entered. Need to check for stack that is deep enough.

```
    $ PICK ()
    >{
```

```
    >    let "TemporaryPointer=$DataStackPointer - $1" #create pointer down
one from top of stack
    >    temporary=${DataStack["$TemporaryPointer"]} #make temporary copy of
data item one down in stack
    >    let "DataStackPointer=$DataStackPointer + 1" #preincrement
    >    DataStack["$DataStackPointer"]=$temporary #store duplicate
    >    unset temporary #clear out variable
    >    unset TemporaryPointer #clear out variable
    >} # end PICK
    $
```

The `PICK` function and the Forth word `PICK` should rarely be used. If you find yourself using this word often, then you should refactor your code to be more simple.

DROP

The `DROP` function is nearly the same functionality as the Forth word `DROP` (the difference is that this function works with character strings). Pops a single object from the top of the data stack.

NOTE: This function does *not* yet have error checking. Need to check for empty stack.

```
    $ DROP ()
    >{
    >    let "DataStackPointer=$DataStackPointer - 1" #postdecrement
    >} # end DROP
    $
```

SWAP

The `SWAP` function is nearly the same functionality as the Forth word `SWAP` (the difference is that this function works with character strings). The item on the top of the stack is swapped with the item one down from the top of the stack.

NOTE: This function does *not* yet have error checking. Need to check that the stack has at least two items.

```
    $ SWAP ()
    >{
    >    let "TemporaryPointer=$DataStackPointer - 1" #create pointer down
one from top of stack
    >    temporary=${DataStack["$TemporaryPointer"]} #make temporary copy of
data item one down in stack
    >    DataStack["$TemporaryPointer"]=${DataStack["$DataStackPointer"]}
#move the top item to 2nd location down
    >    DataStack["$DataStackPointer"]=$temporary #put former 2nd down on
top of stack
    >    unset temporary #clear out variable
    >    unset TemporaryPointer #clear out variable
    >} # end SWAP
    $
```

ROT

The `ROT` function is nearly the same functionality as the Forth word `ROT` (the difference is that this function works with character strings). Rotate third item to top of stack.

NOTE: This function does *not* yet have error checking. Need to check that the stack has at least three items.

```
    $ ROT ()
    >{
    >    let "TemporaryPointer=$DataStackPointer - 2" #create pointer down
two from top of stack
    >    let "SecondPointer=$DataStackPointer - 1" #create pointer down one
from top of stack
    >    temporary=${DataStack["$TemporaryPointer"]} #make temporary copy of
data item one down in stack
    >    DataStack["$TemporaryPointer"]=${DataStack["$SecondPointer"]} #move
the 2nd down item to 3rd location down
    >    DataStack["$SecondPointer"]=${DataStack["$DataStackPointer"]} #move
the top item to 2nd location down
    >    DataStack["$DataStackPointer"]=$temporary #put former 3rd down on
top of stack
    >    unset temporary #clear out variable
    >    unset TemporaryPointer #clear out variable
    >    unset SecondPointer #clear out variable
    >} # end ROT
    $
```

ROLL

The `ROLL` function is nearly the same functionality as the Forth word `ROLL` (the difference is that this function works with character strings). Rotates the nth object to the top of the stack.

NOTE: This function does *not* yet have error checking. Need to check for no user input.

```
    $ ROLL ()
    >{
    >    if [ $DataStackPointer -gt "$1" ] ; then #check to make sure enough
items on stack
    >        let "DestinationPointer=$DataStackPointer - $1"
    >        SavedItem=${DataStack["$DestinationPointer"]} #save old item
    >        let "SourcePointer =$DestinationPointer + 1"
    >        LoopCounter=$1
    >        while [ $LoopCounter -gt 0 ] ; #move data loop
    >        do
    >
DataStack["$DestinationPointer"]=${DataStack["$SourcePointer"]} #move data
    >            let "DestinationPointer=$DestinationPointer + 1" #post
increment
    >            let "SourcePointer=$SourcePointer + 1" #post increment
    >            let "LoopCounter=$LoopCounter - 1"
    >        done
    >        DataStack["$DataStackPointer"]=$SavedItem
    >        unset LoopCounter #clear out variable
    >        unset SavedItem #clear out variable
    >        unset SourcePointer #clear out variable
    >        unset DestinationPointer #clear out variable
    >    fi # end if good input
    >} # end ROLL
    $
```

The `ROLL` function and the Forth word `ROLL` should rarely be used. If you find yourself using this word often, then you should refactor your code to be more simple.

DEPTH

The `DEPTH` function is nearly the same functionality as the Forth word `DEPTH` (the difference is that this function works with character strings). Find existing size of stack and then put that number on top of the stack.

NOTE: This function does *not* yet have error checking.

```
    $ DEPTH ()
    >{
    >    temporary=$DataStackPointer #save the existing depth of the stack
    >    let "DataStackPointer=$DataStackPointer + 1" #preincrement
    >    DataStack["$DataStackPointer"]=$temporary #push depth on top of
stack
    >} # end DEPTH
    $
```

archive copy

You can obtain the latest copy of these functions at http://www.osdata.com/programming/shell/forthfunctions place the file in an appropriate directory.

UNIX and Linux System Administration and Shell Programming

version history

This is a work in progress. For the most up to date version, visit the website http://www.osdata.com/unixbook.html and http://www.osdata.com/programming/shell/unixbook.pdf — Please add links from your website or Facebook page.

56: August 12, 2014. (tail, programming)
55: August 10, 2014. (basename, choice of shells, engineering method, kill, mySQL, programming)
54: July 23, 2014. (touch, LAMP, continue to manually adjust the source file for Adobe Acrobat Pro -- a task that computers can handle much better than humans)
53: July 14, 2014. (modified for Adobe Acrobat Pro -- for some bizarre reason Adobe falsely claims that they can't have both links and proper page breaks in the same document. Obvious bullshit to cover up their own laziness and/or lack of programming skill -- so have to do the work around of manually handling all of the page breaks with BR tags)
52: June 24, 2014. (added internal links)
51: June 8, 2014. (classical logic, connecting to a shell, polkit, vmstat)
50: June 6, 2014. (ftp, ls, mkdir, Perl, shell basics, shell levels, Tcl, UNIX history)
49: January 15, 2014. (spelling corrections, mysql, NFS, rm, scripts, shells, sudo)
48: December 25, 2013. (cool tricks, connecting, date, login, shell basics)
47: August 31, 2013. (cat, cp, ls, mkdir)
46: August 28, 2013. (cd, ls, pwd)
45: August 28, 2013. (flow control)

44: August 27, 2013. (php)
43: August 26, 2013. (substitutions)
42: August 21, 2013. (file system basics, df, mv)
41: August 20, 2013. (scripts, cat, cp, ls, wc)
40: August 18, 2013. (tail, scripts, various parts)
39: March 5, 2013. (kernel modules)
38: February 28, 2013. (curl)
37: November 20, 2012. (management tools, signals)
36: October 29, 2012. (minor changes)
35: October 8, 2012. (NFS, processes, kill, nice, critical reasoning)
34: October 3, 2012. (unix hstory; advanced file system)
33: September 28, 2012. (unix history; sendmail)
32: September 25, 2012. (Forth-like routines)
31: September 23, 2012. (Forth-like routines)
30: September 18, 2012. (Forth-like routines, shells)
29: September 11, 2012. (Forth-like routines; lsof)
28: September 10, 2012. (mkdir)
27: September 10, 2012. (Forth-like routines; uname)
26: September 6, 2012. (Forth-like routines; alias; chmod; chown)
25: September 5, 2012. (command summary)
24: September 4, 2012. (Forth-like routines)
23: September 3, 2012. (Forth-like routines; cat)
22: September 3, 2012. (Forth-like routines)
21: September 2, 2012. (cd)
20: September 2, 2012. (built-in commands; computer basics)
19: September 1, 2012. (cool shell tricks)
18: September 1, 2012. (cool shell tricks)
17: August 31, 2012. (cool shell tricks)
16: August 30, 2012. (defaults)
15: August 30, 2012. (less)
14: August 29, 2012. (history)
13: August 28, 2012. (less)
12: August 27, 2012. (test bed; command separator; installing software from source)
11: August 26, 2012. (quick tour)
10: August 25, 2012. (root, sudo)
9: August 25, 2012. (quick tour, shred, init, command structure)
8: August 24, 2012. (sudo, screencapture)
7: August 24, 2012. (tar, command structure)
6: August 23, 2012. (shells)
5: August 22, 2012. (passwd; cat; file system basics)
4: August 22, 2012. (login/logout; man)
3: August 21, 2012. (login/logout)
2: August 21, 2012. (shell basics)
1: August 19, 2012. (imported amd formatted already existing work from the website)

This book includes material from the http://www.osdata.com/ website and the text book on computer programming.

This book handcrafted on Macintosh computers  using Tom Bender's Tex-Edit Plus  .