Unix Users's Guide

October 11, 2022

Jason W. Bacon

Contents

List of Figures

List of Tables

Chapter 1

Using Unix

Before You Begin
If you think the word "Unix" refers to Sumerian servants specially "trained" to guard a harem, you've come to the right place. This chapter is designed as a tutorial for users with little or no Unix experience.
If you are following this guide as part of an ungraded workshop, please feel free to work together on the exercises in this text. It would be very helpful if experienced users could assist less experienced users during the "practice breaks" in order to keep the class moving forward and avoid leaving anyone behind.

1.1 KISS: Keep It Simple, Stupid

Most people make most things far more complicated than they need to be. Engineers and scientists, especially so.

Aside
To the engineer, all matter in the universe can be placed into one of two categories:

1. Things that need to be fixed
2. Things that will need to be fixed after I've had a few minutes to play with them

Engineers like to solve problems. If there are no problems available, they will create their own problems. Normal people don't understand this concept; they believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet.
No engineer can look at a television remote control without wondering what it would take to turn it into a stun gun. No engineer can take a shower without wondering whether some sort of Teflon coating would make showering unnecessary. To the engineer, the world is a toy box full of sub-optimized and feature-poor toys.
-- The Engineer Identification Test (Anonymous)

Avoid people who tend to look for the most "sophisticated" solution to a problem. Those who look for the simplest solution are more productive, more rational, and more fun to have a beer with. For more amusement on the subject, look up the story of the king's toaster.

Simplicity is the ultimate sophistication. We achieve more when we make things simple for ourselves. We achieve less when we make things complicated. Most people choose the latter. Complexity is the product of carelessness or ego, and simplicity is the product of a wisdom and clarity of thought.

The original Unix designers were an example of wisdom and clarity. Unix is designed to be as simple and elegant as possible. Some things may not seem intuitive at first, but this is probably because the first idea you came up with is not as elegant as the Unix way. The Unix developers had the wisdom to constantly look for simpler ways to implement solutions instead going with

what seemed intuitive at first glance. Learning the Unix way will therefore make you a wiser and happier computer user. I speak from experience.

Unix is not hard to learn. You may have gotten the impression that it's a complicated system meant for geniuses while listening to geniuses talk about it. Don't let them fool you, though. The genius ego compels every genius to make things sound really hard, so you'll think they're smarter than you.

Another challenge with learning anything these days is filtering out the noise on the Internet. Most tutorials on any given subject are incomplete and contain misinformation or bad advice. As a result, new users are often led in the wrong direction and hit a dead end before long. One of the goals of this guide is a show a simple, sustainable, portable, and expandable approach to using Unix systems. This will reduce your learning curve by an order of magnitude.

Most researchers don't know enough about Unix. As a result, their productivity suffers dramatically. Unix has grown immensely since it was created, but the reality is, you don't need to know a lot in order to use Unix effectively. You can become more sophisticated over time if you want, but most Unix users don't really need to. It may be better to stick to the KISS principal (Keep It Simple, Stupid) and focus on learning to use the basic tools *well* rather than a learning a huge collection of tools and using them poorly. It's quality vs quantity. Knowledge is not wisdom. Wisdom is knowing how to apply it effectively.

Aside Einstein was once asked how many feet are in a mile. His reply: "I don't know. Why should I fill my brain with facts I can find in two minutes in any standard reference book?"

Many martial arts students like to collect "forms" (choreographed sequences of moves), for the sake of bragging rights. Knowing more forms does not improve one's Kung Fu, however. The term Kung Fu essentially means "skill". A master is someone who can demonstrate mastery of a few forms, not knowledge of many. This depth of understanding does far more for both self-defense capability and personal development than a shallow knowledge of many forms. Develop your Unix Kung Fu in the same way. Aim to become a master rather than an encyclopedia.

Unix is designed to be as simple as possible and to allow you to work as fast as possible, by staying out of your way. Many other systems will slow you down by requiring you to use cumbersome user interfaces or spend time learning new proprietary methods. As you become a master of Unix, your productivity will be limited only by the speed of the hardware and programs you run.

If something is proving difficult to do under Unix, you're probably going about it the wrong way. There is almost always an easier way, and if there isn't, then you probably shouldn't be trying to do what you're trying to do. If it were a wise thing to do, some Unix developer would have invented an elegant solution by now. Adapt to the wisdom of those who traveled this road before you, and life will become simpler.

1.1.1 Practice

1. What's the engineer's motto regarding things that ain't broke?
2. Why do many people believe that Unix is hard to learn?
3. Is it better to accumulate vast amounts of knowledge or to become highly skilled using the fundamentals? Why?
4. What is the core principle of Unix design? Explain.

1.2 What is Unix?

1.2.1 Aw, man... I Have to Learn Another System?

Well, yeah, but it's the last time, I promise. As you'll see in the sections that follow, once you've learned to use Unix, you'll be able to use your new skills on *virtually any computer*. Over time you'll get better and better at it, and never have to start over from scratch again.

With rare exceptions, if you plan to do computational research, you have two choices:

- Learn to use Unix.
- Rely on the charity of others.

Most scientific software runs only on Unix and very little of it will ever have a graphical or other user interface that allows you to run it without knowing Unix.

The vast majority of high performance computing (HPC) clusters run Unix. You will need basic Unix skills to utilize HPC and HPC clusters generally do not offer a graphical interface. Some HPC administrators attempt to provide for people intent on avoiding Unix, but the results are severely limiting at best.

There have been many attempts to provide access to scientific software via web interfaces, but most of them are abandoned after a short time. People create them with good intentions, but without realizing that they will need to pour effort into maintenance for many years to come. Writing software is like adopting a puppy: It's fun and rewarding, but you need to be committed for the long-term.

In order to be independent in your research computing, you must know how to use Unix in the traditional way. This is the reality of research computing. It's much easier to adapt yourself to reality than to adapt reality to yourself. This chapter will help you become proficient enough to survive and even flourish on your own.

Unix began as the trade name of an operating system developed at AT&T Bell Labs around 1970. It quickly became the model on which most subsequent operating systems have been based. Eventually, "Unix" came into common use to refer to any operating system mimicking the original Unix, much like "Band-Aid" is now used to refer to any adhesive bandage purchased in a drug store.

Over time, formal standards were developed to promote compatibility between the various Unix-like operating systems, and eventually, Unix ceased to be a trade name. Today, the name Unix officially refers to a set of standards to which most operating systems conform.

Look around the room and you will see many standards that make our lives easier. (Wall outlets, keyboards, USB ports, light bulb sockets, etc.) All of these standards make it possible to buy interchangeable devices from competing companies. This competition forces the companies to offer better value. They need to offer a lower price and/or better quality than their competition in order to stay in business.

The Unix standards serve the same purpose as all standards; to foster collaboration, give the consumer freedom of choice, reduce unnecessary learning time, and annoy developers who would rather ignore what everyone else is doing and reinvent the wheel at their employer's expense to gratify their own egos. They allow us to become operating system agnostic nomads, readily switching from one Unix system to another as our needs or situations dictate.

In a nutshell, Unix is every operating system you're likely to use except Microsoft Windows. Table 1.1 provides links to many Unix-compatible operating systems. This is not a comprehensive list. Many more Unix-like systems can be found by searching the web.

Note Apple's Mac OS X has many proprietary extensions, including Apple's own user interface, but is almost fully Unix-compatible and can be used much like any other Unix system by simply choosing not to use the Apple extensions. It is largely based on FreeBSD and other BSD-based components like the Mach kernel.

Note
When you develop programs for any Unix-compatible operating system, those programs can be easily used by people running any other Unix-compatible system. Most Unix programs can even be used on a Microsoft Windows system with the aid of a *compatibility layer* such as Cygwin (Section 1.4.1).
Once you've learned to use one Unix system, you're ready to use any of them. Hence, Unix is the last system you'll ever need to learn!

Unix systems run on everything from your cell phone to the world's largest supercomputers. Unix is the basis for Apple's iOS, the Android mobile OS, embedded systems such as networking equipment and robotics controllers, most PC operating systems, and many large mainframe systems. Many Unix systems are completely free (as in free beer) and can run tens of thousands of high quality free software packages. As an extreme example, NetBSD runs on dozens of different CPU architectures, including some hobbyist systems such as Commodore Amigas, 68k-based Macs, etc.

It's a good idea to regularly use more than one Unix system. This will make you aware of how much they all have in common and what the subtle differences are.

Name	Type	URL
AIX (IBM)	Commercial	https://en.wikipedia.org/wiki/IBM_AIX
CentOS GNU/Linux	Free	https://en.wikipedia.org/wiki/CentOS
Debian GNU/Linux	Free	https://en.wikipedia.org/wiki/Debian
DragonFly BSD	Free	https://en.wikipedia.org/wiki/DragonFly_BSD
FreeBSD	Free	https://en.wikipedia.org/wiki/FreeBSD
GhostBSD	Free	https://en.wikipedia.org/wiki/GhostBSD
HP-UX	Commercial	https://en.wikipedia.org/wiki/HP-UX
JunOS (Juniper Networks)	Commercial	https://en.wikipedia.org/wiki/Junos
Linux Mint	Free	https://en.wikipedia.org/wiki/Linux_Mint
MidnightBSD	Free	https://en.wikipedia.org/wiki/MirOS_BSD
NetBSD	Free	https://en.wikipedia.org/wiki/NetBSD
OpenBSD	Free	https://en.wikipedia.org/wiki/OpenBSD
OpenIndiana	Free	https://en.wikipedia.org/wiki/OpenIndiana
OS X (Apple Macintosh)	Commercial	https://en.wikipedia.org/wiki/OS_X
Redhat Enterprise Linux	Commercial	https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux
Slackware Linux	Free	https://en.wikipedia.org/wiki/Slackware
SmartOS	Free	https://en.wikipedia.org/wiki/SmartOS
Solaris	Commercial	https://en.wikipedia.org/wiki/Solaris_(operating_system)
SUSE Enterprise Linux	Commercial	https://en.wikipedia.org/wiki/SUSE_Linux_Enterprise_Desktop
Ubuntu Linux (See also Kubuntu, Lubuntu, Xubuntu)	Free	https://en.wikipedia.org/wiki/Ubuntu_(operating_system)

Table 1.1: Partial List of Unix Operating Systems

1.2.2 Operating System or Religion?

Aside
Keep the company of those who seek the truth, and run from those who have found it.
-- Vaclav Havel

The more confident someone is in their views, the less they probably know about the subject. As we gain life experience and wisdom, we become less certain about everything and more comfortable with that uncertainty. What looks like confidence is usually a symptom of ignorance of our own ignorance, generally fueled by ego.

If you discuss operating systems at length with most people, you will discover, as the ancient philosopher Socrates did while discussing many topics with "experts", that their views are not based on broad knowledge and objective comparison. Before taking advice from anyone, it's a good idea to find out how much they really know and what role emotion and ego play in their preferences. This process of questioning has become known as a "Socratic examination". Note, however, that if you embarrass the wrong people, it may get you executed, as it did Socrates.

The whole point of the Unix standard, like any other standard, is freedom of choice. However, you won't have any trouble finding evangelists for a particular brand of Unix-compatible operating system on whom this point is lost. "Discussions" about the merits of various Unix implementations often involve arrogant pontification and emotional outbursts, possibly involving some cussing.

If you step back and ask yourself what kind of person gets emotionally attached to a piece of software, you'll realize whose advice you should value and whose you should not. Rational people will keep an open mind and calmly discuss the objective measures of an OS, such as performance, reliability, security, easy of maintenance, specific capabilities, etc. They will also back up their opinions with facts rather than try to bully you into validating their views.

If someone tells you that a particular operating system "isn't worth using", "is way behind the times", or "sucks wads", rather than asking you what you need and objectively discussing alternatives, this is someone whose advice you can safely ignore. They are not interested in helping you. They need you to validate their opinions, because those opinions are not supported by facts.

Aside
We're all capable of rational thought, but sometimes we only use it to rationalize what we want to believe, despite obvious evidence to the contrary.
"I don't understand why some people wash their bath towels. When I get out of the shower, I'm the cleanest object in my house. In theory, those towels should be getting cleaner every time they touch me. By the way, are towels supposed to bend?"
-- Wally (Dilbert)

Evangelists are easy to spot. They will instantly assess your needs without asking you a single question and proceed to explain (often aggressively) why you should be using their favorite operating system or programming language. They invariably have limited or no experience with other alternatives. This is easy to expose with a few simple questions. "How many years of experience to you have with it?" The answer is usually close to 0. "What are the specific advantages and disadvantages?" The response to this will usually be stuttering, silence, or double-talk. Ask them to clarify further and it won't take long to expose their ignorance.

Ultimately, the system that most easily runs your programs to your satisfaction is the best one for you. That could turn out to be BSD, Cygwin, Linux, Mac OS X, OpenIndiana, or any other. Someone who knows what they're doing and truly wants to help you will always begin by asking questions in order to better understand your needs. "What program(s) do you need to run?", "Do they require any special hardware?", "Do you need to run any commercial software, or just open source?", etc. They will then consider multiple alternatives and inform you about the capabilities of each one that might match your needs.

There is another reason besides ego that people often choose inappropriate solutions to a problem; the desire to use what they know instead of being open to learning a better approach.

Aside When all you have is a hammer, everything looks like a nail.

I regularly experiment with various Unix variants to evaluate their ease of use, reliability, and resource requirements. This is easy to do using virtual machines (See Chapter 7.) My personal preference for running Unix software (for now, these could change in the distant future) are listed below. All of these systems are somewhat interchangeable with each other and the many other Unix based systems available, so deviating from these recommendations will generally not lead to catastrophe.

More details on choosing a Unix platform are provided in Chapter 4.

- Servers running mostly open source software: FreeBSD.

  FreeBSD is extremely fast, reliable, and secure. It is known as a "set-and-forget" operating system, since it requires very little attention after initial installation and configuration. Software management is very easy with FreeBSD ports, which offers over 30,000 software packages (not counting different builds of the same software). The ports system supports installation via either generic binary packages, or you can just as easily build from source with custom options or optimizations for your specific CPU. With the Linux compatibility module, FreeBSD can directly run most Linux closed-source programs with no performance penalty and a little added effort and resources.

FreeBSD with the Lumina desktop environment

- Servers running mainly or commercial applications or CUDA GPU software: Enterprise Linux (AlmaLinux, CentOS, RHEL, Rocky Linux, Scientific Linux, SUSE).

  These systems are designed for better reliability, security, and long-term binary compatibility than bleeding-edge Linux systems. They are the only platforms besides MS Windows and Mac OS X supported by many commercial software vendors. While you may be able to get some commercial engineering software running on Ubuntu or Mint, it is often difficult and the company will not provide support. Packages in the native Yum repository of enterprise Linux are generally outdated, but more recent open source software can be installed using a separate add-on package manager such as pkgsrc.

- An average Joe who wants to browse the web, use a word processor, etc.: Debian, GhostBSD, Ubuntu, or similar open source Unix system with graphical installer and management tools, or Macintosh.

  These systems make it easy to install software packages and system updates, with minimal risk of breakage that Joe would not know how to fix.

Debian Linux

- Someone who uses mostly Windows-based software, but needs a basic Unix environment for software development or connecting to other Unix systems: A Windows PC with Cygwin.

  Cygwin is free, entirely open source, and very easy to install in about 10 minutes on most Windows systems. It has some performance bottlenecks, fewer packages than a real Unix system running on the same machine, and a few other limitations, but it's more than adequate for the needs of many typical users. See Section 1.4.1 for details.

1.2.3 The Unix Standard API

Programmer time is expensive. Writing a program twice costs twice as much. Unix standards solve this problem.

Unix systems adhere to an *application program interface* (API) standard, which means that programs written for one Unix-based system can be run on any other with little or no modification, even on completely different hardware. For example, programs written for an Intel/AMD-based Linux system will also run an ARM-based Mac, or FreeBSD on an ARM, Power, or RISC-V processor.

An API defines a set of functions (subprograms) used to request services from the operating system, such as opening a file, allocating memory, running another program, etc. These functions are the same on all Unix systems, but some of them are different on Windows and other non-standard systems. For example, to open a file in a C program on any Unix system, one would typically use the fopen() function:

```
FILE *fopen(const char *filename, const char *mode);
```

Microsoft compilers support fopen() as well, but also provide another function for the same purpose that only works on Windows:

```
errno_t fopen_s(FILE** pFile, const char *filename, const char *mode);
```

Note Microsoft claims that fopen_s() is more secure, which is debatable. Note however, that even if this is true, the existing fopen() function itself could have been made more secure rather than creating a separate, non-portable function that does the same thing. Non-standard functions like fopen_s() mainly benefit the vendor by making it harder to port software to a competing platform.

Here are a few other standard Unix functions that can be used in programs written in C and most other compiled languages. These functions can be used on any Unix system, regardless of the type of hardware running it. Some of these may also work in Windows, but for others, Windows uses a completely different function to achieve the same goal.

```
chdir()          // Change current working directory
execl()          // Load and run another program
mkdir()          // Create a directory
unlink()         // Remove a file
sleep()          // Pause execution of the process
DisplayWidth()   // Get the width of the graphical screen
```

Because the Unix API is platform-independent, it is also possible to compile and run most Unix programs on Windows with the aid of a *compatibility layer*, software that bridges the difference between two platforms. (See Section 1.4.1 for details.) It is not generally possible to compile and run Windows software on Unix, however, because Windows has many features specific to PC hardware.

Since programs written for Unix can be run on almost any computer, including Windows computers, they will never have to be rewritten in order to run somewhere else. Programs written for non-Unix platforms will only run on that platform, and will have to be rewritten (at least partially) in order to run on any other system. This leads to an enormous waste of man-hours that could have gone into creating something new. They may also become obsolete as they proprietary systems for which they were written evolve. For example, most programs written for MS DOS and Windows 3.x are no longer in use today, while programs written for Unix around that same time will still work on modern Unix systems.

1.2.4 Shake Out the Bugs

Another advantage of programming on standardized platforms is the ability to easily do more thorough testing. Compiling and running a program on multiple operating systems and with multiple compilers will almost always expose bugs that you were unaware of while running it on the original development system. The same bug will have different effects on different operating systems, with different compilers or interpreters, or with different compile options (e.g. with and without optimization).

For example, an errant array subscript or pointer might cause corruption in a non-critical memory location in some environments, while causing the program to crash in others.

A program may seem to be fine when you compile it with Clang and run it on your Mac, but may not compile, or may crash when compiled with GCC on a Linux machine.

Finding bugs *now* may save you from the stressful situation of tracking them down under time pressure later, with an approaching grant deadline. A bug that was invisible on your Mac for the test cases you've used could also show up on your Mac later, when you run the program with different inputs.

Developing for the Unix API makes it easy to test on various operating systems and with different compilers. There are many free BSD and Linux based systems, as well as free compilers such as Clang and GCC. Most of them can be run in a virtual machine (Chapter 7), so you don't even need another computer for the sake of program testing. Take advantage of this easy opportunity to stay ahead of program bugs, so they don't lead to missed deadlines down the road.

1.2.5 The Unix Standard UI

The Unix standards not only make programs portable, they make our knowledge as users portable as well. All Unix systems support the same basic set of commands, which conform to standards so that they behave the same way everywhere. So, if you learn to use FreeBSD, most of that knowledge will directly apply to Linux, Mac OS X, Solaris, etc.

Another part of the original Unix design philosophy was to do everything in the simplest way possible. As you learn Unix, you will likely find some of its features befuddling at first. However, upon closer examination, you will often come to appreciate the

elegance of the Unix solution to a difficult problem. If you're observant enough, you'll learn to apply this Zen-like simplicity to your own work, and maybe even your everyday life.

You will also gradually recognize a great deal of consistency between various Unix commands and functions. For example, many Unix commands support a `-v` (verbose) flag to indicate more verbose output, as well as a `-q` (quiet) flag to indicate no unnecessary output. Over time, you will develop an intuitive feel for Unix commands, become adept at correctly guessing how things work, and feel almost God-like at times.

Unix documentation also follows a few standard formats, which users quickly get used to, making it easier to learn new things about commands on any Unix system.

In a nutshell, the time and effort you spend learning any Unix system will make it easy to use any other in the future. You need only learn Unix once, and you'll be proficient with many different implementations such as FreeBSD, Linux, and Mac OS X.

1.2.6 Fast, Stable and Secure

Since Unix systems compete directly with each other to win and retain users running the same programs, developers are highly motivated to optimize objective measures of the system such as performance, stability, and security.

Most Unix systems operate near the maximum speed of the hardware on which they run. Unix systems typically respond faster than other systems on the same hardware and run intensive programs in less time. Many Unix systems require far fewer resources than non-Unix systems, leaving more disk and memory for use by your programs.

Unix systems tend to be very reliable and may run for months or even years without being rebooted. I managed a particular FreeBSD HPC cluster for eight years. Except for some problems in the first few months that were traced to a Dell firmware bug, none of the servers in this cluster ever crashed.

Unlike Windows, software installations almost never require a reboot, and even most security updates can be applied without rebooting. Reboots are typically only needed following a kernel update.

Stability is critical for research computing, where computational models may run for weeks or months. Users of non-Unix operating systems often have to choose between killing a process that has been running for weeks and neglecting critical security updates that require a reboot.

Very few viruses or other malware programs exist for Unix systems. This is in part due to the inherently better security of Unix systems and in part due to a strong tradition in the Unix community of discouraging users from engaging in risky practices such as running programs under an administrator account and installing software from pop-ups on the web.

1.2.7 Sharing Resources

Your mom probably told you that it's nice to share, but did you know it's also more efficient?

One of the major problems for researchers in computational science is managing their own computers. Most researchers aren't very good at installing operating systems, managing software, apply security updates, etc., nor do they want to be. Unfortunately, they often have to do these things in order to conduct computational research. Computers managed by a tag-team of researchers usually end up full of junk software, out-of-date, full of security issues, and infected with malware.

Since Unix is designed from the ground up to be accessed remotely, Unix creates an opportunity to serve researchers' needs far more cost-effectively than individual computers for each researcher. A single Unix machine on a modern PC can support dozens or even hundreds of users at the same time, depending how demanding their software is.

1.2.8 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. After learning Unix, on what operating systems will you be able to use your new skills?
2. What is the major design goal of the Unix standards?
3. What is the alternative to learning Unix for computational scientists? Why?
4. Why does most scientific software lack a convenient graphical or web interface?
5. Is Unix an operating system? Why or why not?
6. What is the advantage of open standards?
7. How many different Unix-compatible operating systems exist? What does this mean for Unix users?
8. Which mainstream operating systems are Unix-compatible and which are not?
9. What types of computer hardware run Unix?
10. How much does Unix cost?
11. Which Unix operating system is the best one?
12. How should we go about choosing a Unix system? What if we make the wrong choice?
13. How do we spot evangelists who are likely to give us irrational advice?
14. What is an API?
15. What is the advantage of the Unix API over the APIs of non-Unix operating systems? What problem does it solve?
16. Can software written for Unix be run on Windows? How?

17. How does the Unix API help us proactively eliminate software bugs?
18. What is a UI? What are three advantages of the Unix UI over the UIs of non-Unix operating systems?
19. Why are Unix-compatible operating systems faster, more stable, and more secure than many non-Unix platforms?
20. How does the inherent remote access capabilities of Unix help researchers?

1.3 Unix User Interfaces

A *user interface*, or *UI*, refers to the software that allows a person to interact with the computer. The UI provides the look and feel of the system, and determines how easily and efficiently it can be used. (Note that ease of use and efficiency are not the same!)

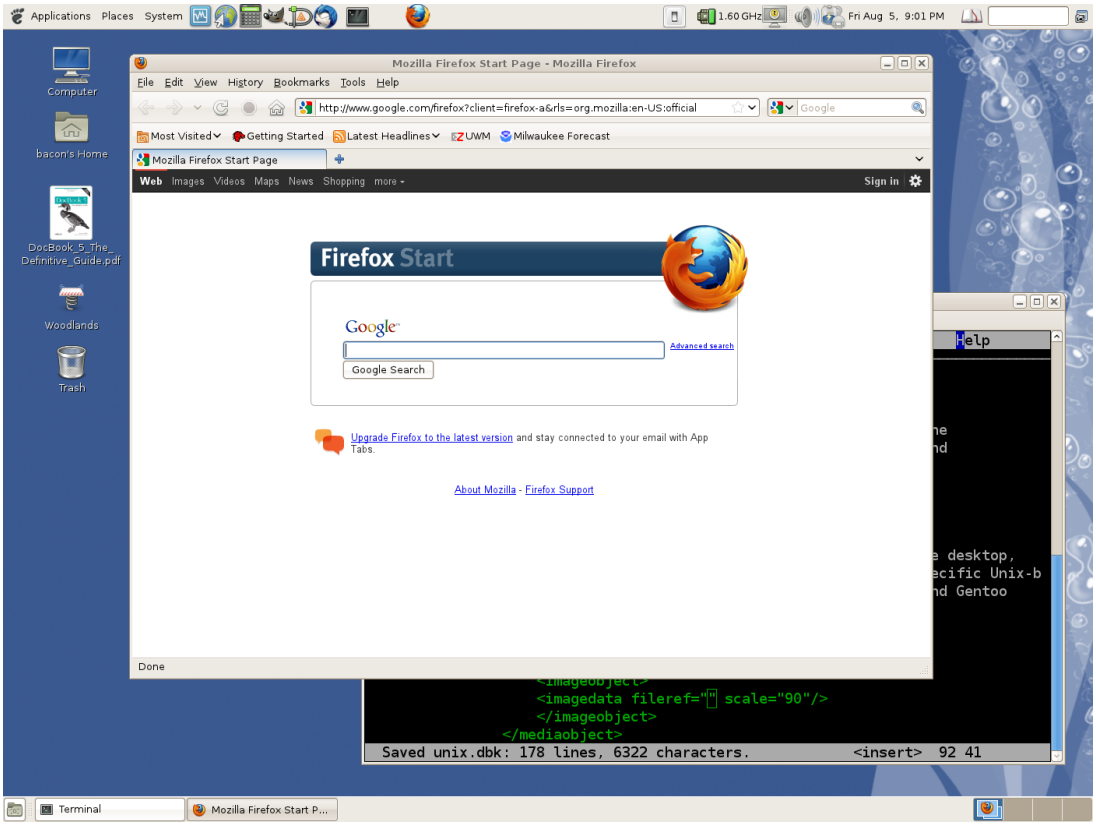
The term "MS Windows" refers to a specific proprietary operating system, and implies all of the features of that system including the API and the UI. When people think of Windows, they think of the Start menu, the Control Panel, etc. Likewise, "Macintosh" refers to a specific product and invokes images of the "Dock" and a menu bar at the top of the screen rather than attached to a window.

The term "Unix", on the other hand, implies an API, but does not imply a specific UI. There are many UIs available for Unix systems. In fact, a computer running Unix can have multiple UIs installed, and each user can choose the one they want when the log in.

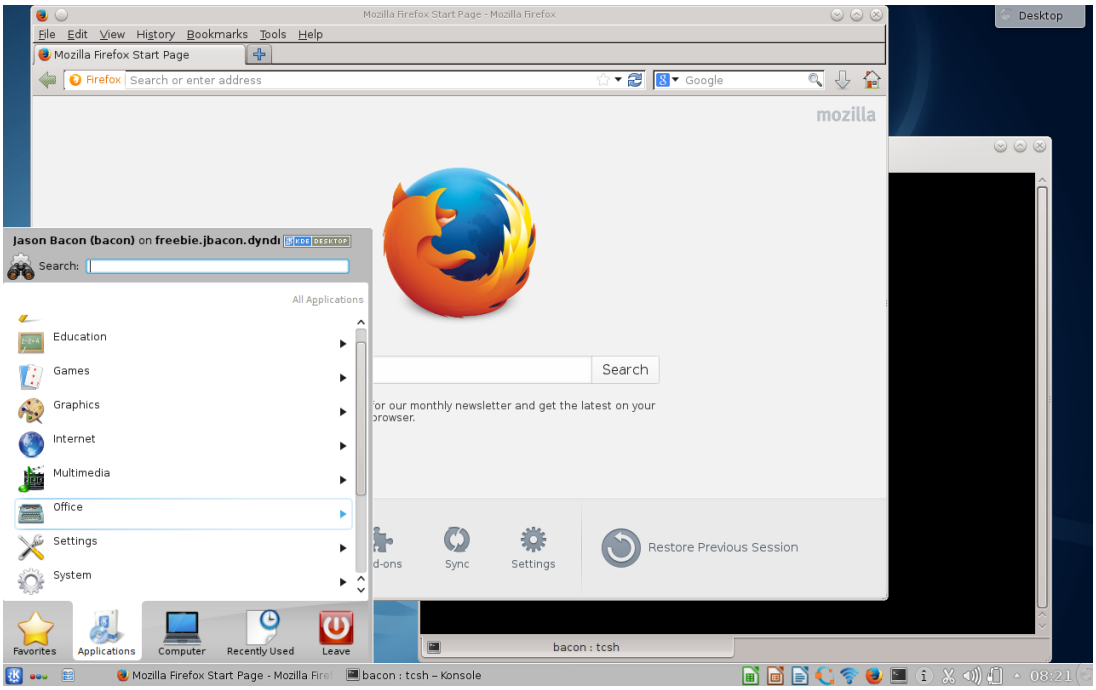
1.3.1 Graphical User Interfaces (GUIs)

A *Graphical User Interface*, or *GUI* (pronounced goo-ee), is a user interface with a graphical screen, and icons and menus we can select using a mouse or a touch screen.

There are many different GUIs available for Unix. Some of the more popular ones include KDE, Gnome, XFCE, LXDE, OpenBox, CDE, and Java Desktop.

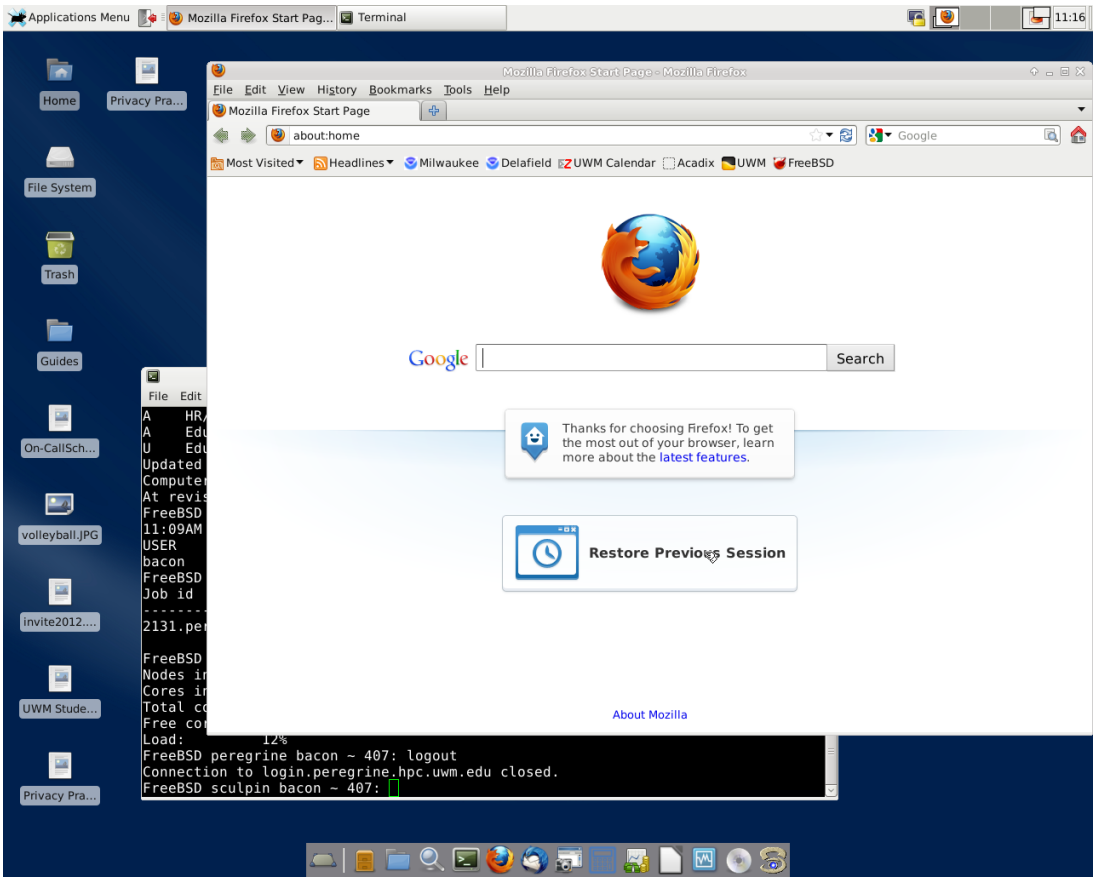


A FreeBSD system running Gnome desktop.



A FreeBSD system running KDE desktop.

A FreeBSD system running Lumina desktop.



A FreeBSD system running XFCE desktop.

Practice Break

If you have access to a Unix GUI, log into your Unix system via the GUI interface now.

All Unix GUIs are built on top of the X11 networked graphics API. As a result, all Unix systems have the inherent ability to display graphics on other Unix systems over a network. I.e., you can remotely log into another Unix computer over a network and run graphical programs that display output wherever you're sitting.

This is not the same as a *remote desktop* system, which mirrors the console display on a remote system. Unix systems allow multiple users in different locations to run graphical programs independent of each other. In other words, Unix supports multiple independent graphical displays on remote computers.

It is also not the same as a *terminal server*, which opens an entire desktop environment on a remote display.

With Unix and X11, we can have individual applications running on multiple remote computers displayed on the same desktop. Doing so is easy and requires no additional software to be installed.

Most Unix GUIs support multiple *virtual desktops*, also known as *workspaces*. Virtual desktops allow a single monitor to support multiple separate desktop images. It's like having multiple monitors without the expense and clutter. The user can switch between virtual desktops by clicking on a panel of thumbnail images, or in some cases by simply moving the mouse over the edge of the screen.

1.3.2 X11 on Mac OS X

Mac OS X is Unix compatible, derived largely from FreeBSD and the Mach kernel project, with components from GNU and other Unix-based projects.

It differs from more traditional Unix systems like BSD and Linux, though, in that it runs Apple's proprietary graphical API and GUI. Native OS X programs don't use the X11 API, but OS X can also run X11-based programs with the XQuartz add-on. See Section 1.19 for instructions on enabling X11 for Mac.

1.3.3 Command Line Interfaces (CLIs): Unix Shells

There are two basic types of user interfaces:

- Menu-driven, where choices are displayed on the screen and the user selects one.
- Command-driven, where the user types commands that they have memorized.

A GUI is a type of menu-driven interface, where the menu items may be text or graphical icons. Some menu systems are simply text selected by entering a number on the keyboard.

```
                 ___          __            ___        __           _
                /   | __  __/ /_____      /   | _____/ /___ ____  (_)___
               / /| |/ / / / __/ __ \    / /| |/ __  / __ `__ \/ / __ \
              / ___ / /_/ / /_/ /_/ /   / ___ / /_/ / / / / / / / / / /
             /_/  |_\__,_/\__/\____/   /_/  |_\__,_/_/ /_/ /_/_/_/ /_/
            ____________________________________________________________

                   Portable Command-line Systems Management
                       https://acadix.biz/auto-admin.php
            ____________________________________________________________

This menu system encompasses only a small fraction of the total auto-admin
functionality.  To see what else is available via the command-line, choose
"List available auto-admin scripts" below.

Full documentation is in the works and will be included in a future release.

1.. Update system
2.. User management
3.. Software management
4.. Network management
5.. Power management
6.. File system actions and settings
7.. Security settings
8.. System settings
9.. Services manager
10.. List available auto-admin scripts
Q.. Quit

Selection?
```

While all modern Unix systems have GUIs, much Unix work is still done via the *command line interface* (*CLI*). A CLI requires the user to type in commands, rather than select them from a menu, much like short-answer questions vs multiple choice.

Menu-driven systems are easier to use if the system has limited functionality and you're new to the system or use it infrequently. However, menus are cumbersome where there is too much functionality to offer in a simple menu. Even simple menu systems can become cumbersome for everyday use.

If a user needs access to dozens or hundreds of features, they cannot all be displayed on the screen at the same time. Hence, it will be necessary to navigate through multiple levels of menus or screens to find the functionality you need. Doing this repeatedly becomes annoying rather quickly. A command line interface, on the other hand, provides instant access to an unlimited number of commands.

An ATM (automatic teller machine) is a good candidate for a menu interface. It has only a few functions and people don't use it every day. An ATM with a command-driven interface would likely be unpopular among banking customers.

You might be surprised to learn that CAD (Computer Aided Design) systems have CLIs. While CAD is inherently graphical in nature, CAD users cannot efficiently access their vast functionality through menus. Most CAD users quickly learn to use the CLI to draw, move, and edit objects via keyboard commands.

Because menu systems slow us down, most support *hot keys*, special key combinations that can be used to access certain features without navigating the menus. Hot keys are often shown in menus alongside the features they activate. For example, Command+q can be used on macOS and Ctrl+q on Windows and most Unix GUIs to terminate many graphical applications, as shown in Figure 1.1.

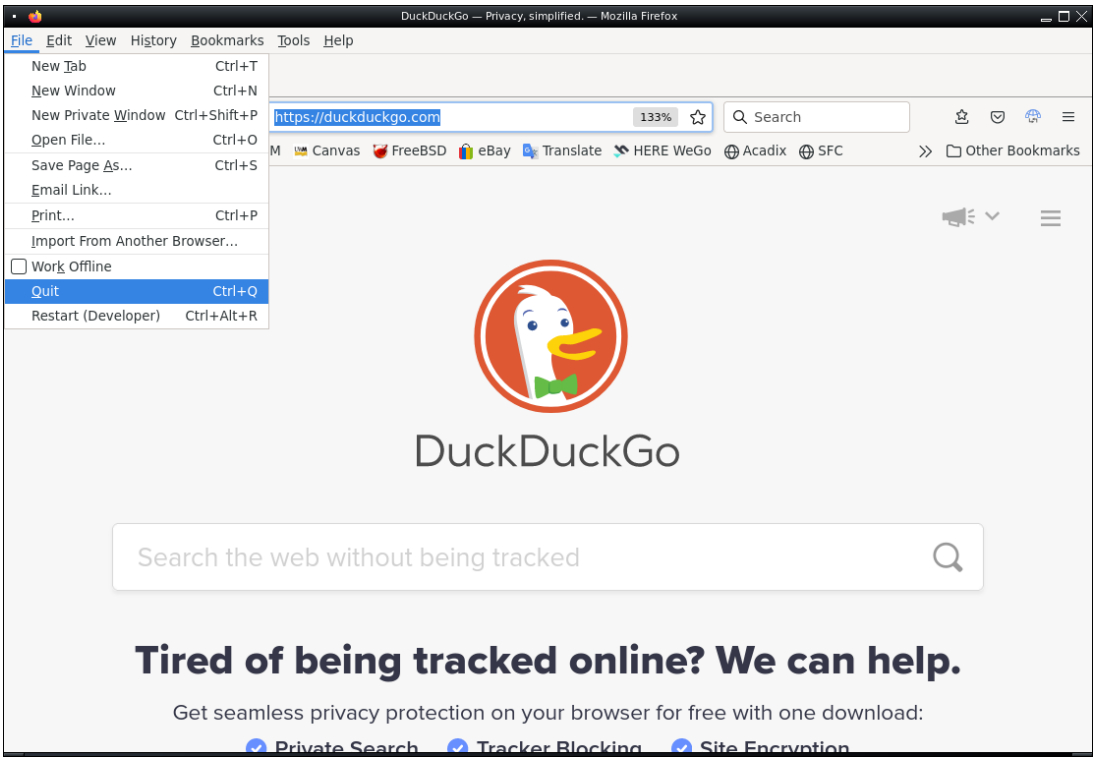


Figure 1.1: Hot Keys

It is also difficult to automate tasks in a menu-driven system. Some systems have this capability, but most do not, and the method of automating is different for each system. Command-driven interfaces are easy to automate by placing commands in a *script*, a simple text file containing a sequence of commands that might otherwise be run directly via the keyboard. Scripting is covered in Chapter 2.

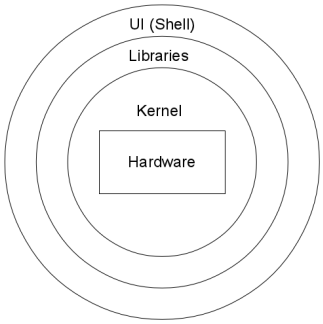
Perhaps the most important drawback of menu-driven systems is non-existence. Programming a menu system, and especially a GUI, requires a lot of grunt-work and testing. As a result, the vast majority of open source software does not and never will have a GUI interface. Open source developers generally don't have the time or programming skills to build and maintain a comprehensive GUI interface.

Caution
If you lack command-line skills, you will be limited to using a small fraction of available open source software. In the tight competition for research grants, those who can use the command-line more often win.

The small investment in learning a command line interface can have a huge payoff, and yet many people try to avoid it. The result is usually an enormous amount of wasted effort dealing with limited and poorly designed custom user interfaces before eventually realizing that things would have been much easier had they learned to use the command line in the first place. It's amazing how much effort people put into avoiding effort...

A *shell* is a program that provides the command line interface. It inputs commands from the user, interprets them, and executes them. Using a shell, you type a command, press enter, and the command is immediately executed.

The word "shell" comes from the view of Unix as three layers of software wrapped around the hardware:



A 3-layer Model of Unix

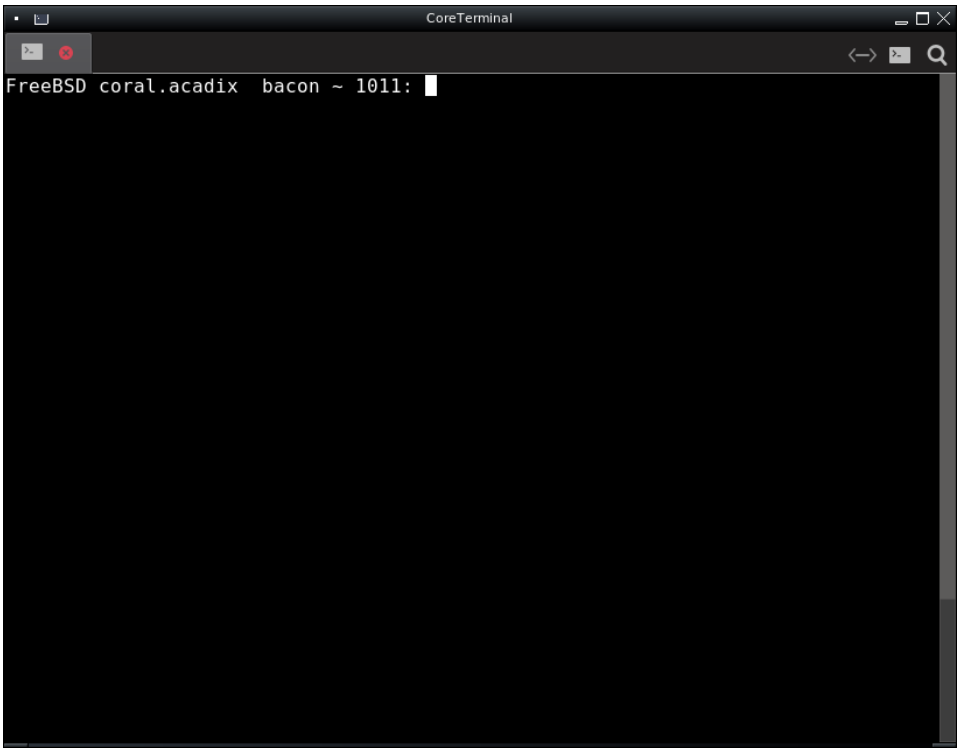
- The innermost layer, which handles all hardware interaction for Unix programs, is called the *kernel*, named after the core of a seed. The Unix kernel effectively hides the hardware from user programs and provides a standard API. This is what allows Unix programs to run on different kinds of hardware without modification. Application programs never "see" the hardware interface. They only see the kernel interface, which is the same regardless of hardware.
- The middle layer, the libraries, provide a wealth of standard functionality for Unix programmers to utilize. The libraries are like a huge box of Legos that can be used to build all kinds of sophisticated programs. They include basic input/output functions, math functions, character string functions, graphics functions, etc.
- The outermost layer, the CLI, is called a shell.

1.3.4 Terminals

All that is needed to use a Unix shell is a keyboard and a screen. In the olden days, these were provided by a simple hardware device called a *terminal*, which connected a keyboard and screen to the system through a simple communication cable. These terminals typically did not have a mouse or any graphics capabilities. They usually had a text-only screen of 80 columns by 24 lines, and offered limited capabilities such as moving the cursor, scrolling the screen, and perhaps a limited number of colors, usually 8 or 16.

Digital VT320 terminal

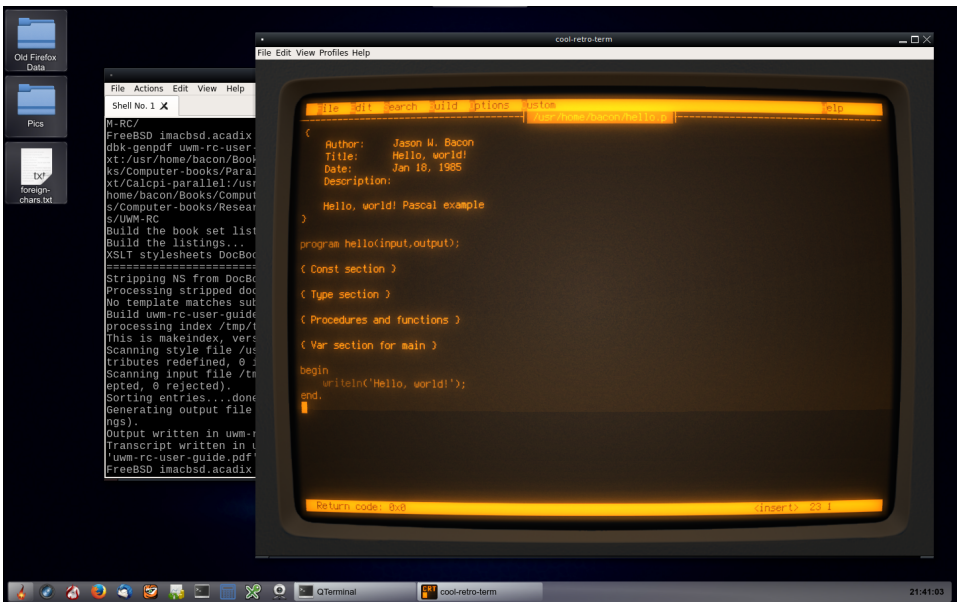
Hardware terminals lost popularity with the advent of cheap personal computers, which can perform the functions of a terminal, as well as running programs of their own. Terminals have been largely replaced by *terminal emulators*. A terminal emulator is a simple program that emulates an old style terminal within a window on your desktop.



A Terminal emulator.

All Unix systems come with a terminal emulator program. There are also free terminal emulators for Windows, which are discussed in Section 1.5.

For purists who *really* want to emulate a terminal, there's *Cool Retro Terminal* (CRT for short, which also happens to stand for cathode ray tube). This emulator comes complete with screen distortion and jitter to provide a genuine nostalgic 1970s experience.



Cool Retro Terminal

1.3.5 Basic Shell Use

Once you're logged in and have a shell running in your terminal window, you're ready to start entering Unix commands.

The shell displays a *shell prompt*, such as "FreeBSD coral.acadix bacon ~ 1011:" in the image above, to indicate that it's waiting for you to enter the next command. The shell prompt can be customized by each user, so it may be different on each Unix system you use.

Note
For clarity, we primarily use the following to indicate a shell prompt in this text:

```
shell-prompt:
```

To enter a Unix command, you type the command on a single line, edit if necessary (using arrow keys to move around), and press **Enter** or **Return**.

We can also enter multiple Unix commands on the same line separated by semicolons.

Modern Unix shells allow commands to be extensively edited. Assuming your terminal type is properly identified by the Unix system, you can use the left and right arrow keys to move around, backspace and delete to remove characters (Ctrl+h serves as a backspace in some cases), and other key combinations to remove words, the rest of the line, etc. Learning the editing capabilities of your shell will make you a much faster Unix user, so it's a great investment of a small amount of time.

If you have access to a Unix system now, do the practice break below. This practice break is offered again in Section 1.5 for those who will be using a remote Unix system.

Practice Break

Remotely log into another Unix system using the **ssh** command or PuTTY, or open a shell on your Mac or other Unix system. Then try the commands shown below.

Unix commands are preceded by the shell prompt "shell-prompt: ". Other text below refers to input to the program (command) currently running. You must exit that program before running another Unix command.

Lines beginning with '#' are comments, and not to be types.

```
# List files in the current working directory (folder)
shell-prompt: ls
shell-prompt: ls -al

# Two commands on the same line
shell-prompt: ls; ls /etc

# List files in the root directory
shell-prompt: ls /

# List commands in the /bin directory
shell-prompt: ls /bin

# Create a subdirectory
shell-prompt: mkdir -p Data/IRC

# Change the current working directory to the new subdirectory
shell-prompt: cd Data/IRC

# Print the current working directory
shell-prompt: pwd

# See if the nano editor is installed
# nano is a simple text editor (like Notepad on Windows)
shell-prompt: which nano

    If this does not report "command not found", then do the following:

# Try the nano editor
shell-prompt: nano sample.txt

# Type in the following text:

This is a text file called sample.txt.
I created it using the nano text editor on Unix.

# Then save the file (press Ctrl+o), and exit nano (press Ctrl+x).
# You should now be back at the Unix shell prompt.

# Try the "vi" editor
# vi is standard editor on all Unix system.  It is more complex than nano.
shell-prompt: vi sample.txt

    Type 'i' to go into insert mode
    Type in some text
    Type Esc to exit insert mode and go back to command mode
    Type :w to save
    Type :q to quit

shell-prompt: ls

# Echo (concatenate) the contents of the new file to the terminal
shell-prompt: cat sample.txt

# Count lines, words, and characters in the file
shell-prompt: wc sample.txt

# Change the current working directory to your home directory
shell-prompt: cd
shell-prompt: pwd

# Show your login name
```

1.3.6 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What is a UI?
2. What is a GUI?
3. What is the difference between Unix and other operating systems with respect to the GUI?
4. How is Unix + X11 different from remote desktop systems and terminal servers?
5. What is a virtual desktop?
6. What are the two basic types of user interfaces? Which type is a GUI?
7. What is a CLI?
8. What types of applications are better suited for a menu-driven interface? Why?
9. What types of applications are better suited for a command-driven interface?
10. Which is easier to automate, a menu-driven system or a CLI? Why?
11. How many scientific programs offer a menu-driven interface? Why?
12. What is a shell?
13. What is a kernel?
14. What are libraries? What kinds of functionality do they provide?

15. What is a terminal?
16. What is a terminal emulator?
17. Do people still use hardware terminals today? Explain.
18. What is a shell prompt?

1.4 Still Need Windows? Don't Panic!

For those who need to run software that is only available for Windows, or those who simply haven't tried anything else yet, there are options for getting to know Unix while still using Windows for your daily work.

One option is to remotely log into a Unix system using a terminal application such as *PuTTY* on your Windows machine.

There are virtual machines (see Chapter 7) that allow us to run Windows and Unix on the same computer, at the same time. This is the best option for those who need a fully functional Unix environment on a Windows machine.

There are also compatibility layers such as Cygwin and Windows Subsystem for Linux (WSL), that allow Unix software to be compiled and run on Windows. A compatibility layer is generally easier to install, but as of this writing, both Cygwin and WSL have performance limitations in some areas. Purely computational software will run about as fast as it would on a real Unix system, but software that performs a lot of file input/output or other system calls can be much slower than a real Unix system, even one running in a virtual machine.

For example, installing the pkgsrc package manager from scratch, which involves running many Unix scripts and programs, required the times shown in Table 1.2. WSL, Cygwin, and the Hyper-V virtual machine were all run on the same Windows 10 host with a 2.6 GHz Core i7 processor and 4 GiB RAM. The native FreeBSD and Linux builds were run on identical 3.0 GHz Xeon servers with 16 GiB RAM, much older than the Core i7 Windows machine.

Platform	**Time**
WSL	104 minutes
Cygwin	71 minutes
FreeBSD Virtual Machine under Hyper-V	21 minutes
CentOS Linux (3.0 GHz Xeon)	6 minutes, 16 seconds
FreeBSD (3.0 GHz Xeon)	5 minutes, 57 seconds

Table 1.2: Pkgsrc Build Times

I highly recommend Cygwin as a light-duty Unix environment under Windows, for connecting to other Unix systems or developing small Unix programs. For serious Unix development or heavy computation, obtaining a real Unix system, even under a virtual machine, will produce better results.

1.4.1 Cygwin: Try This First

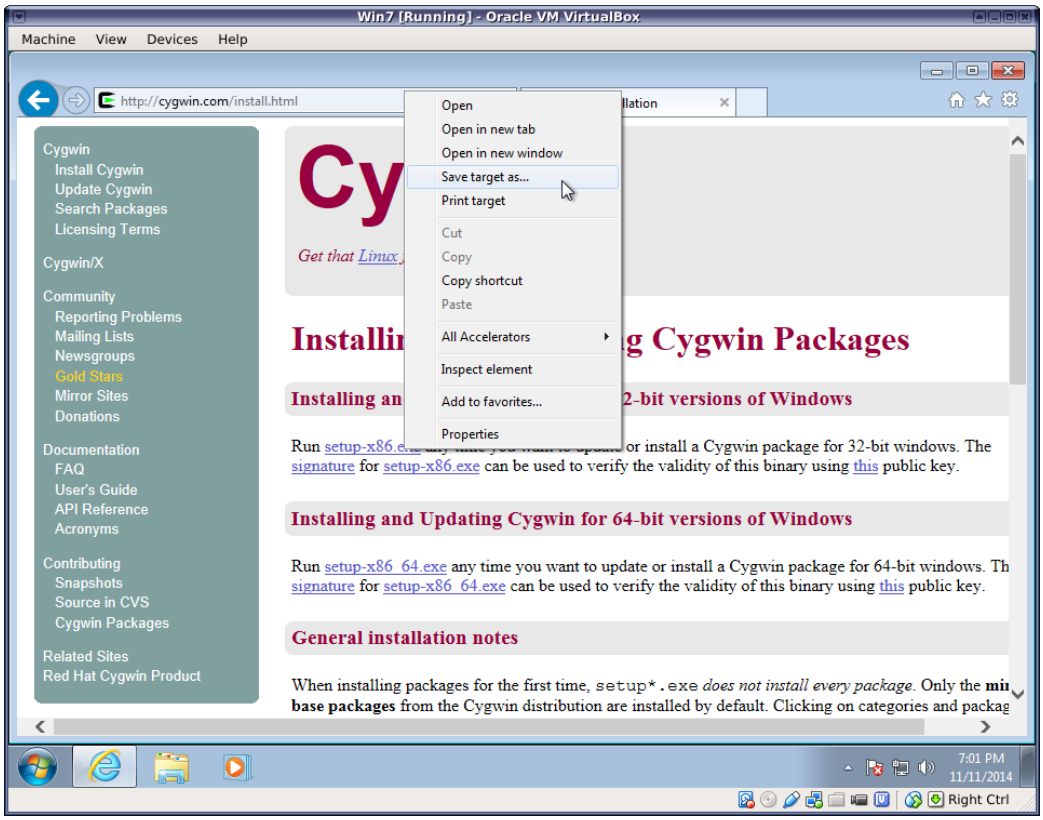
Cygwin is a free collection of Unix software, including many system tools from Linux and other Unix-compatible systems, ported to Windows. It can be installed on any typical Windows machine in about 10 minutes and allows users to experience a Unix user interface as well as run many popular Unix programs right on the Windows desktop.

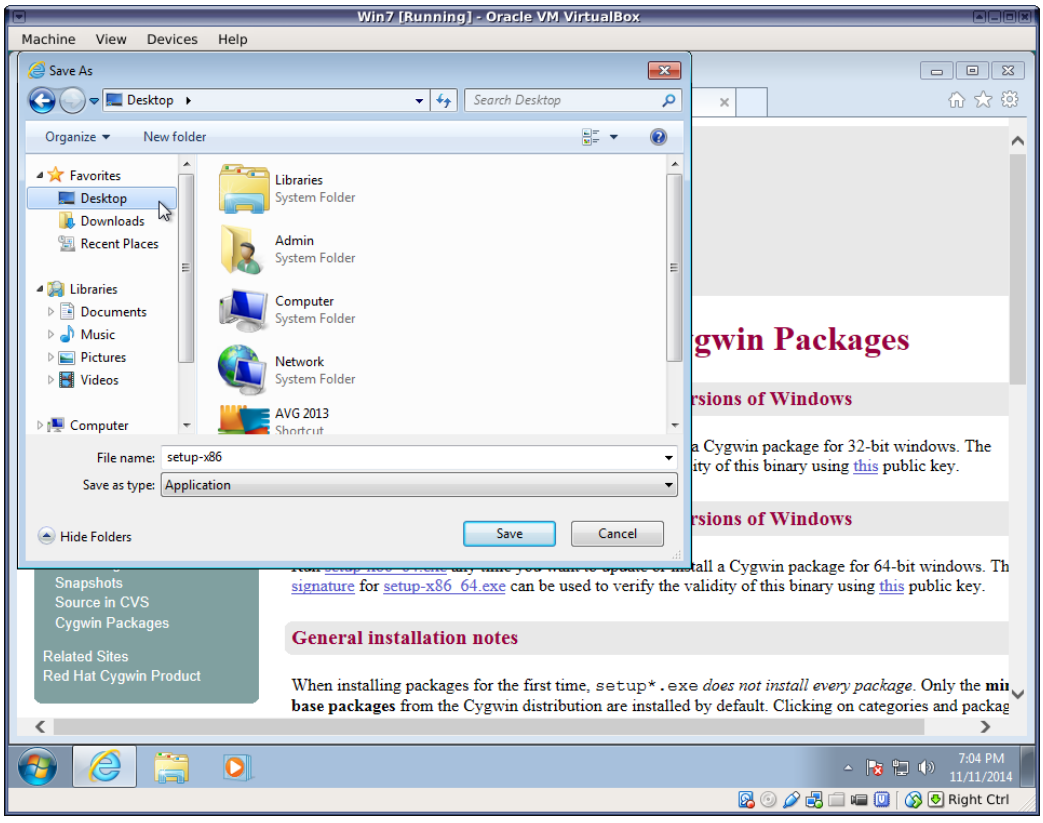
Cygwin is a *compatibility layer*, another layer of software on top of Windows that translates the Unix API to the Windows API. As such, performance is not as good as a native Unix system on the same hardware, but it's more than adequate for many purposes. Cygwin may not be ideal for heavy-duty data analysis where optimal performance is required, but it is an excellent system for basic development and testing of Unix code and for interfacing with other Unix systems.

Cygwin won't break your Windows configuration, since it is completely self-contained in its own directory. Given that it's so easy to install and free of risk, there's no point wasting time wondering whether you should use Cygwin, a virtual machine, or some other method to get a Unix environment on your Windows PC. Try Cygwin first and if it fails to meet your needs, try something else.

Installing Cygwin is quick and easy:

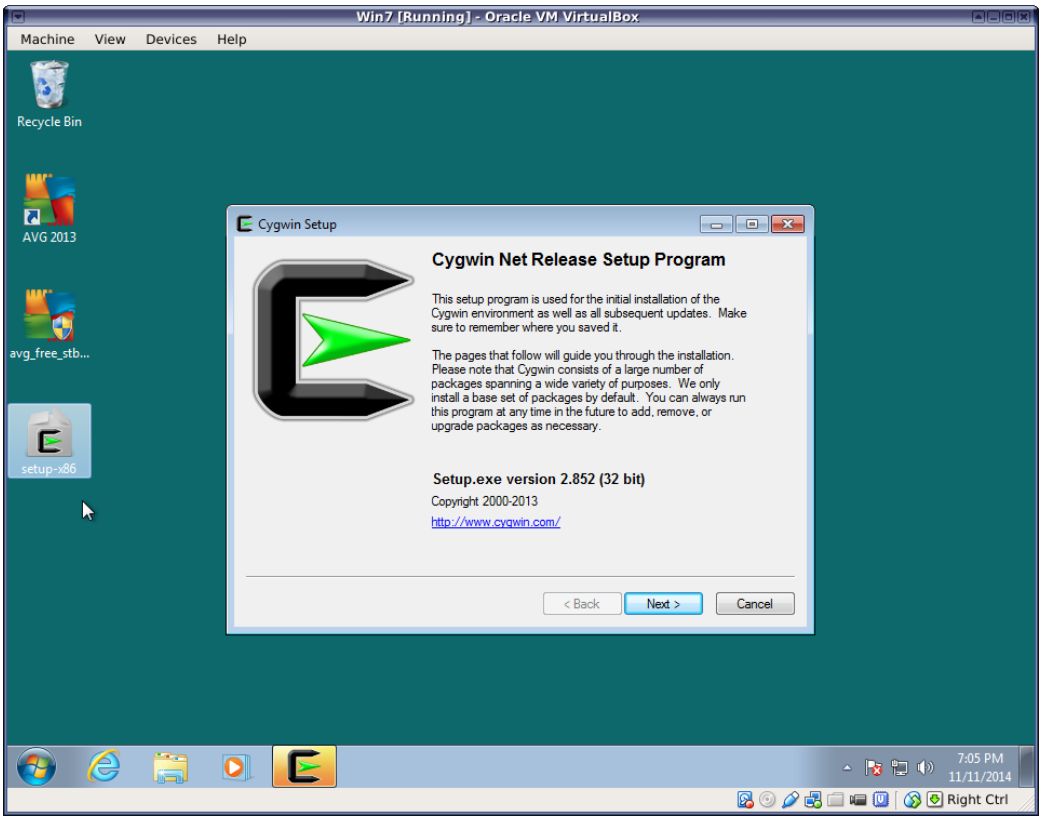
1. Download **setup-x86_64.exe** from https://www.cygwin.com and save a copy on your desktop or some other convenient location. You will need this program to install additional packages in the future.



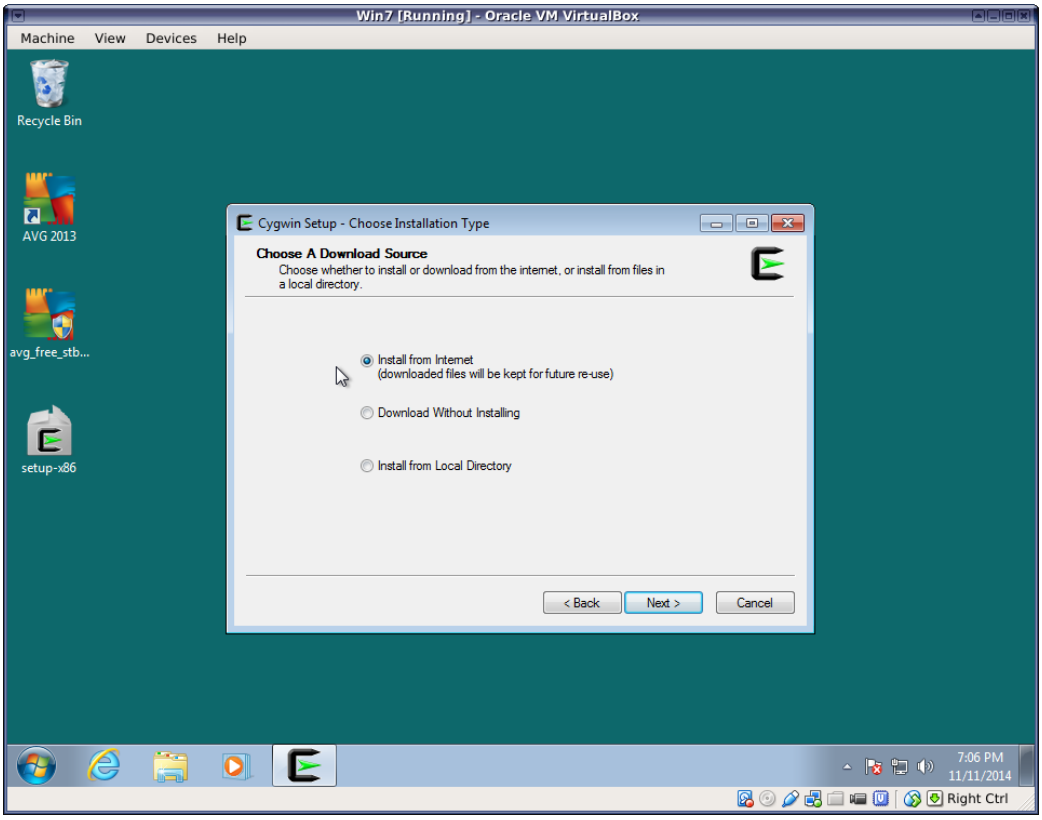


2. Run **setup-x86_64.exe** and follow the instructions on the screen.

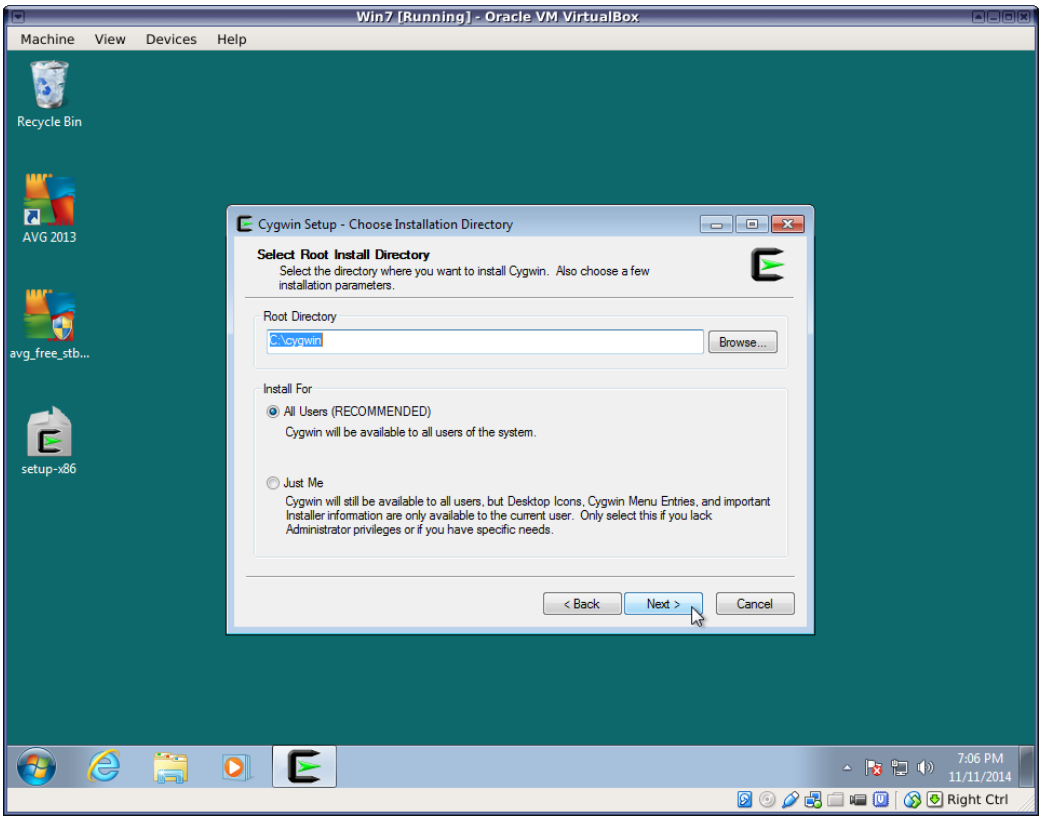
   Unless you know what you're doing, accept the default answers to most questions. Some exceptions are noted below.



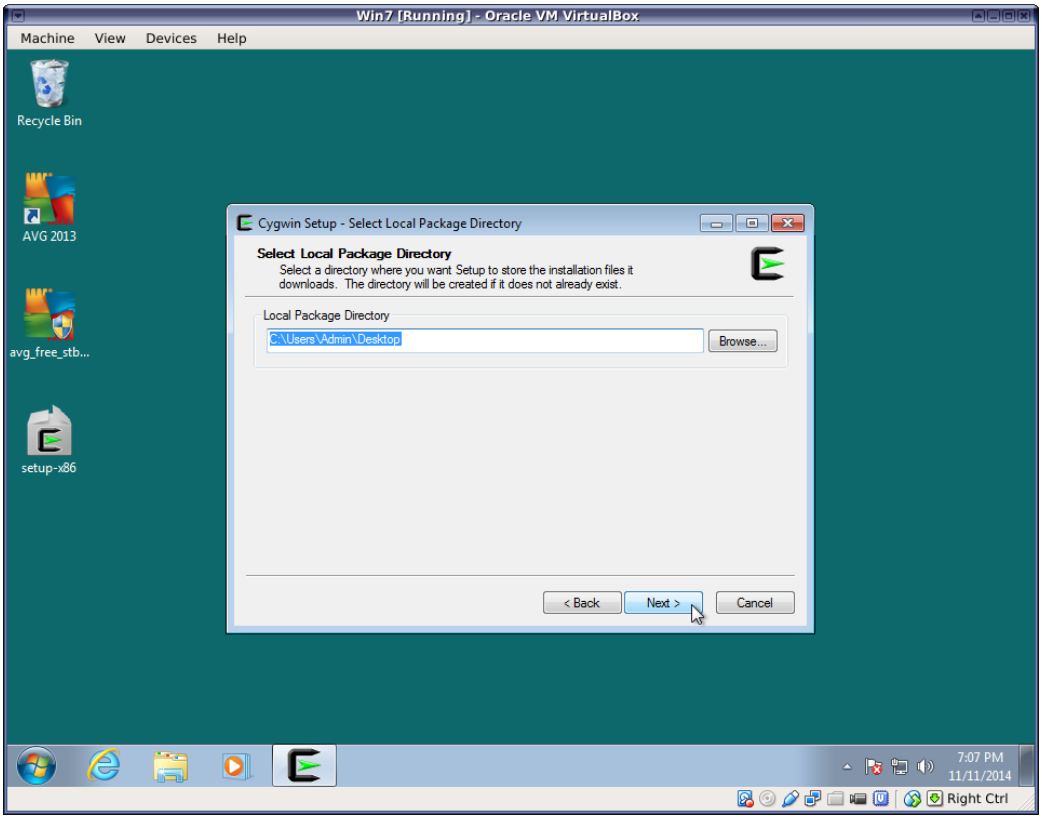
3. Unless you know what you're doing, simply choose "Install from Internet".



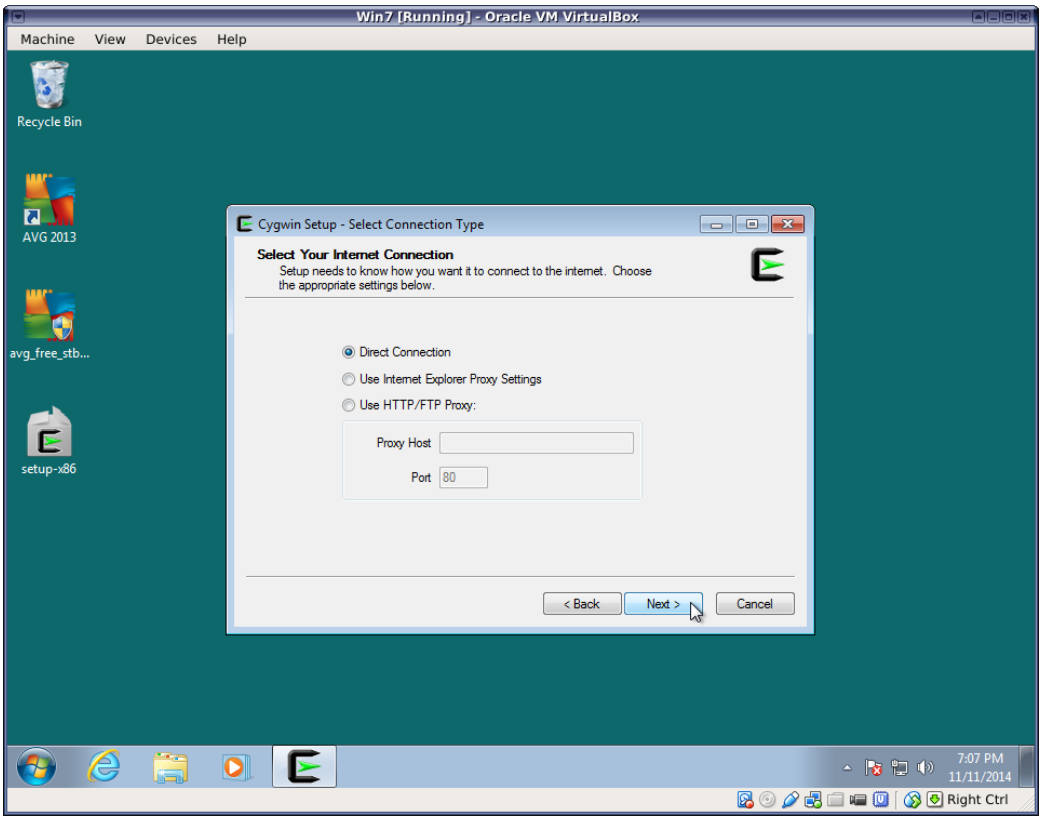
4. Select where you want to install the Cygwin files and whether to install for all users of this Windows machine.



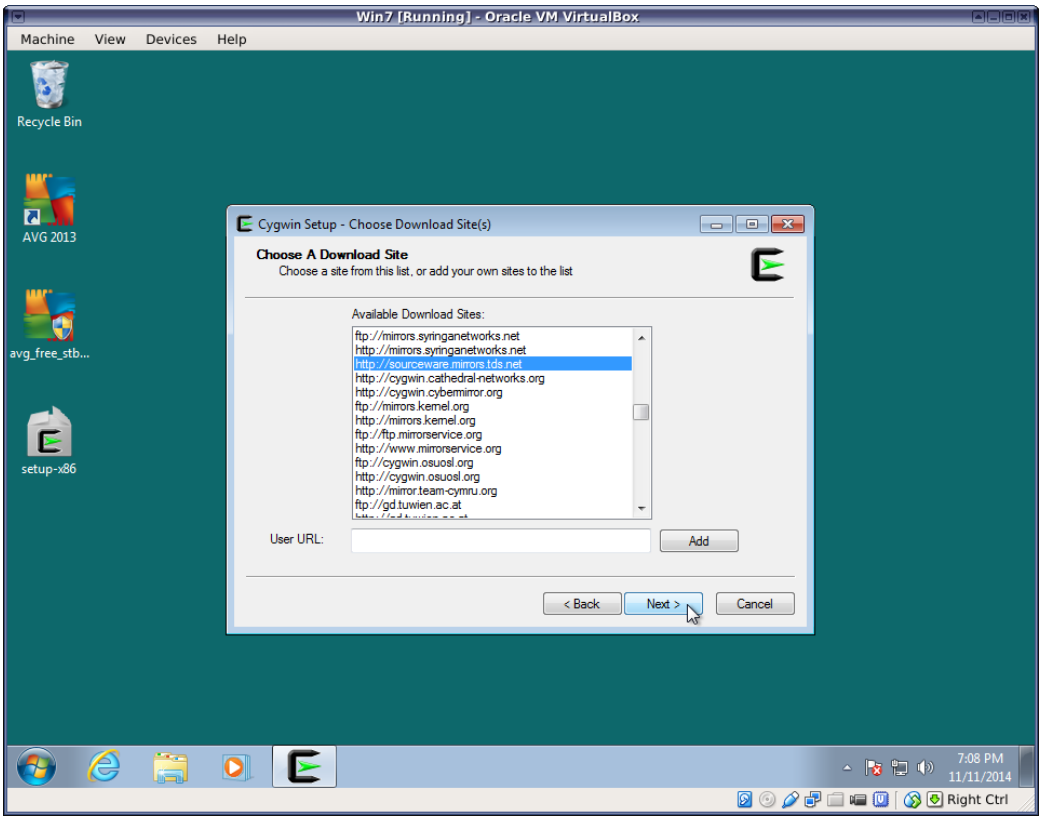
5. Select where to save downloaded packages. Again, the default location should work for most users.



6. Select a network connection type.



7. Select a download site. It is very important here to select a site near you. Choosing a site far away can cause downloads to be incredibly slow. You may have to search the web to determine the location of each URL. This information is unfortunately not presented by the setup utility.



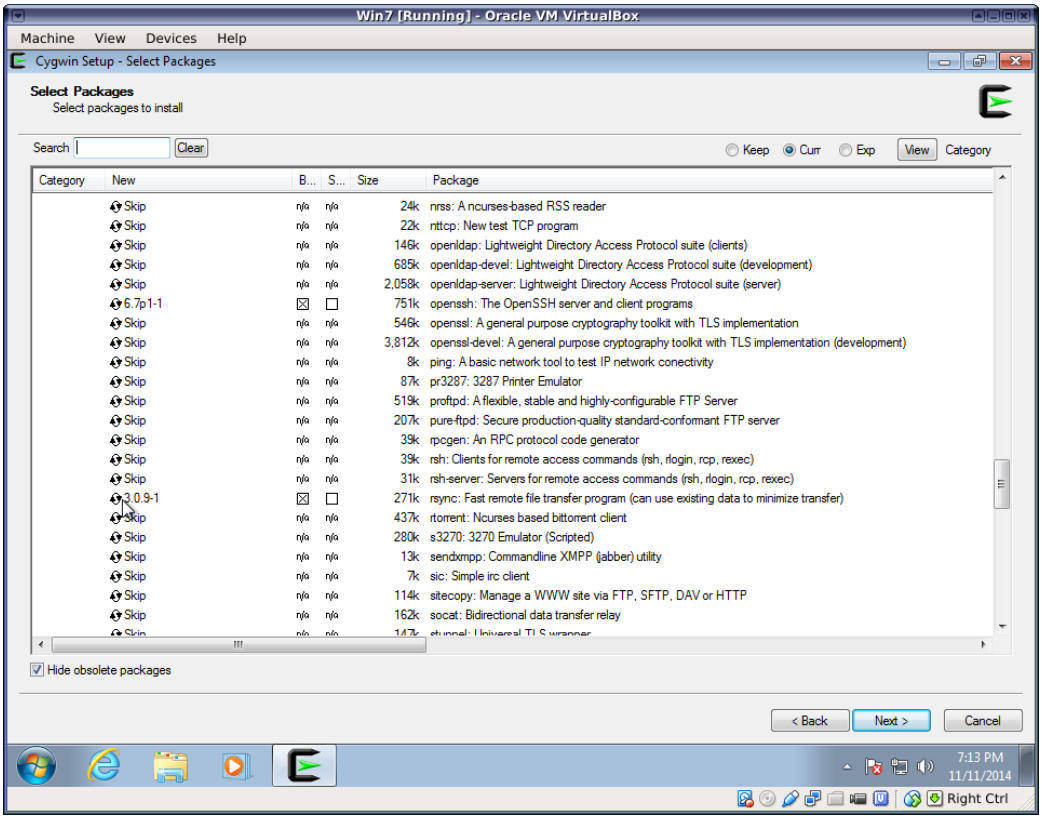
8. When you reach the package selection screen, select at least the following packages in addition to the basic installation:

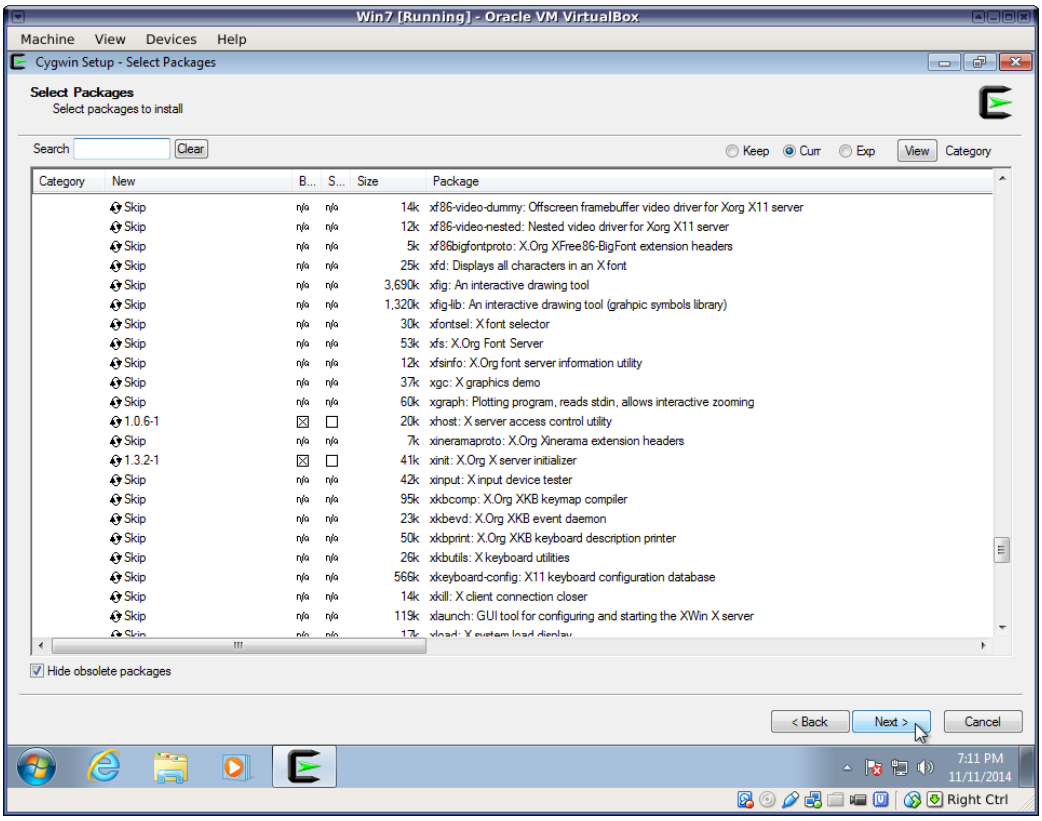
- net/openssh
- net/rsync
- x11/xhost
- x11/xinit

This will install the **ssh** command as well as an X11 server, which will allow you to run graphical Unix programs on your Windows desktop. You may not need graphical capabilities immediately, but they will likely come in handy down the road.

The rsync package is especially useful if you'll be transferring large amounts of data back and forth between your Windows machine and remote servers.

Click on the package categories displayed in order to expand them and see the packages under them.



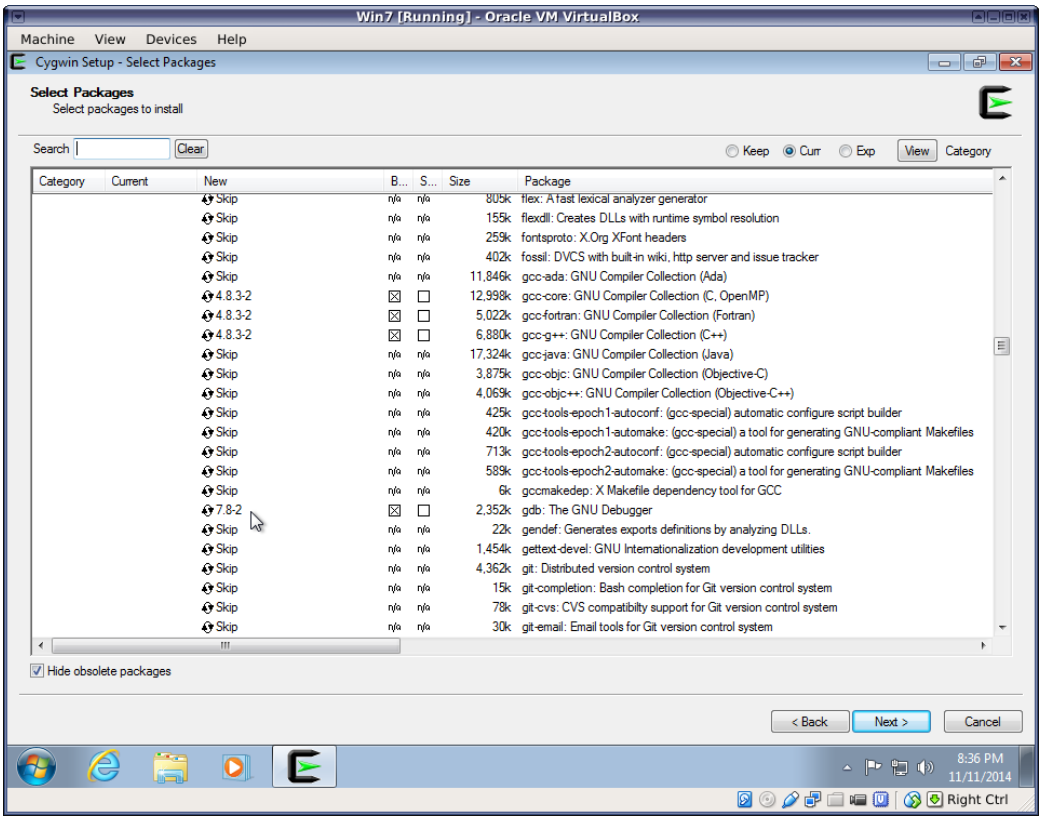


Cygwin can also enable you to do Unix program development on your Windows machine. There are many packages providing Unix development tools such as compilers and editors, as well as libraries. The following is a small sample of common development packages:

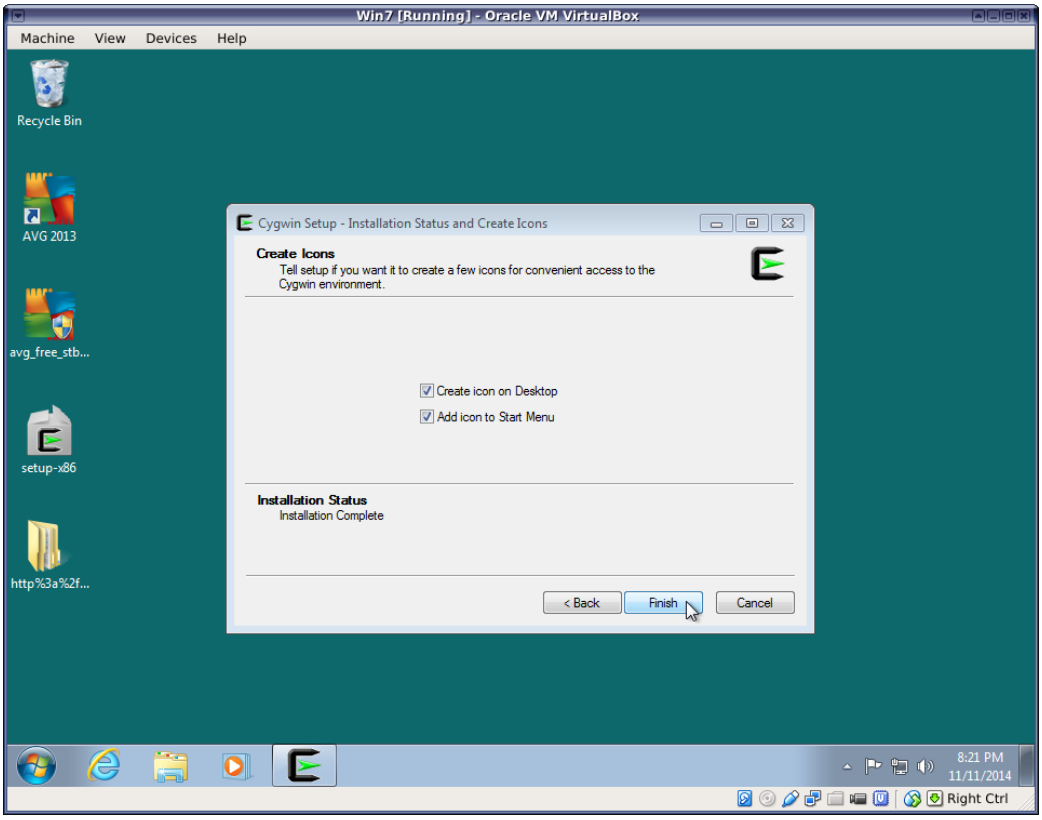
Note

Many of these programs are easier to install and update than their counterparts with a standard Windows interface. By running them under Cygwin, you are also practicing use of the Unix interface, which will make things easy for you when need to run them on a cluster or other Unix host that is more powerful than your PC.

- devel/clang (C/C++/ObjC compiler)
- devel/clang-analyzer (Development and debugging tool)
- devel/gcc-core (GNU Compiler Collection C compiler)
- devel/gcc-g++
- devel/gcc-gfortran
- devel/make (GNU make utility)
- editors/emacs (Text editor)
- editors/gvim (Text editor)
- editors/nano (Text editor)
- libs/openmpi (Distributed parallel programming tools)
- math/libopenblas (Basic Linear Algebra System libraries)
- math/lapack (Linear Algebra PACKage libraries)
- math/octave (Open source linear algebra system compatible with Matlab(r))
- math/R (Open source statistical language)

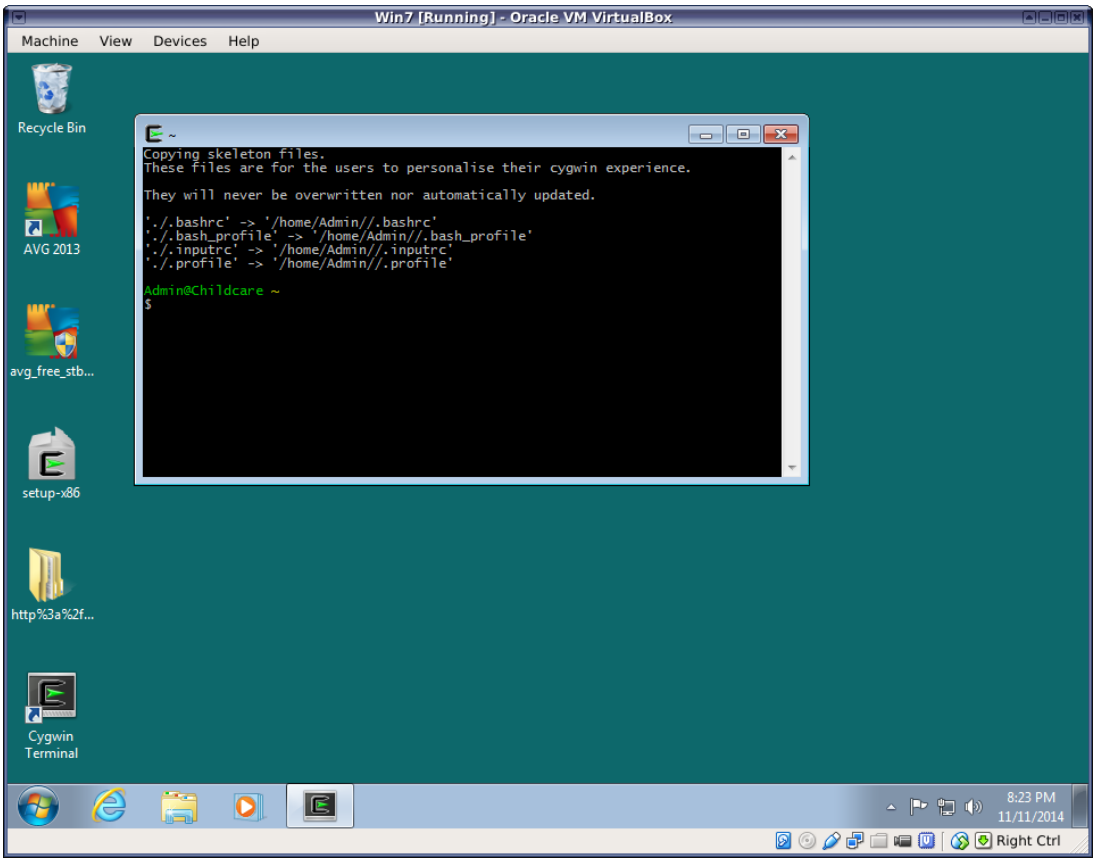


9. Most users will want to accept the default action of adding an icon to their desktop and to the Windows Start menu.

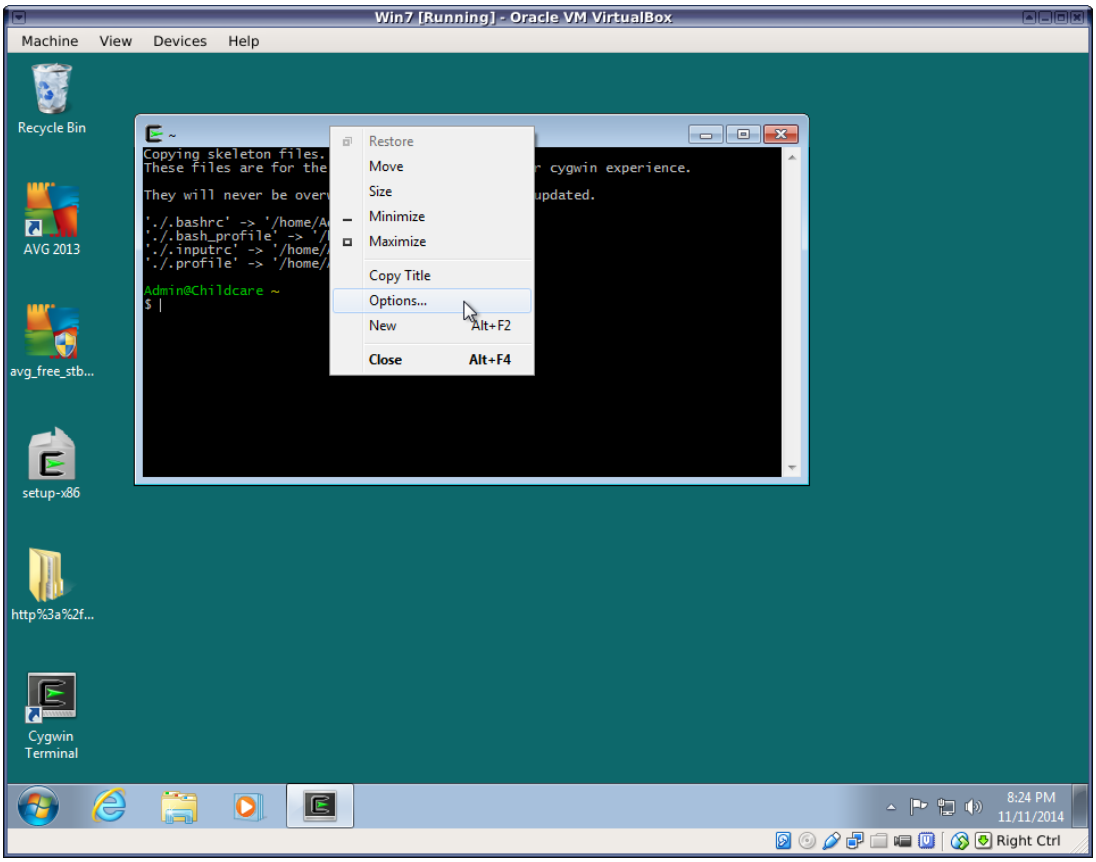


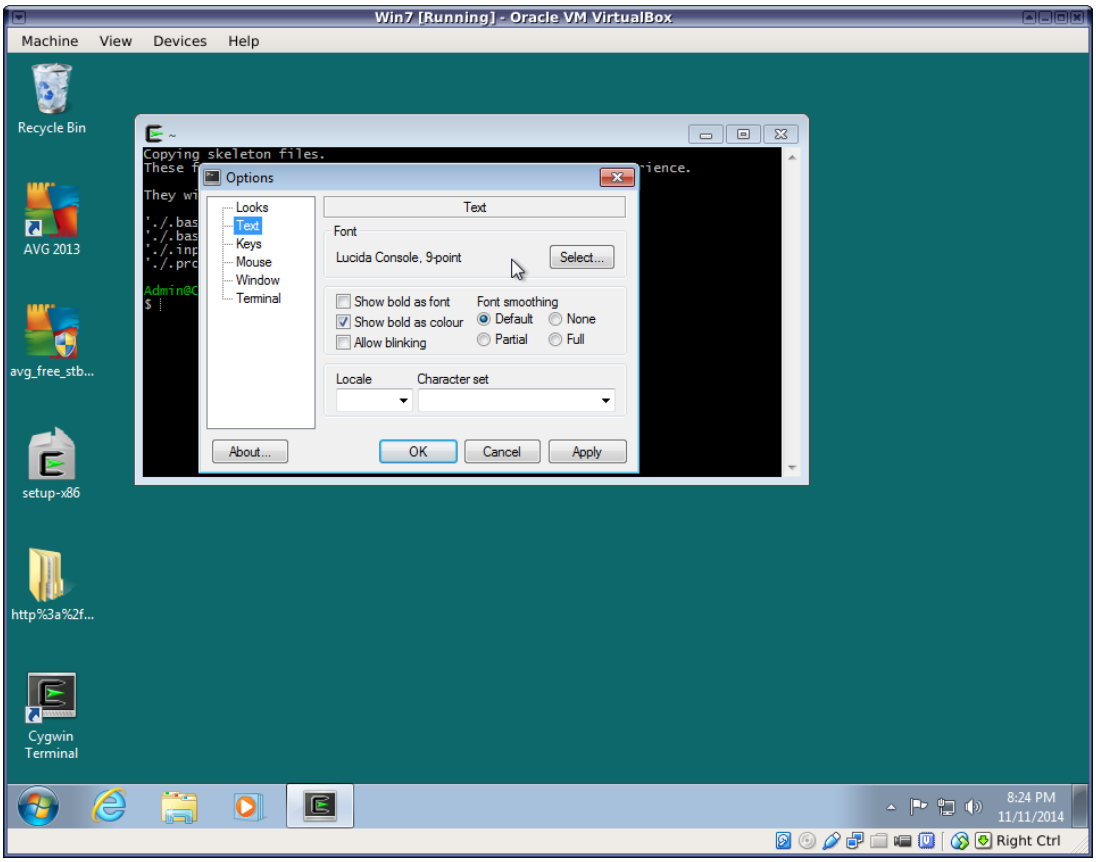
When the installation is complete, you will find Cygwin and Cygwin/X folders in your Windows program menu.

For a basic Terminal emulator, just run the Cygwin terminal:



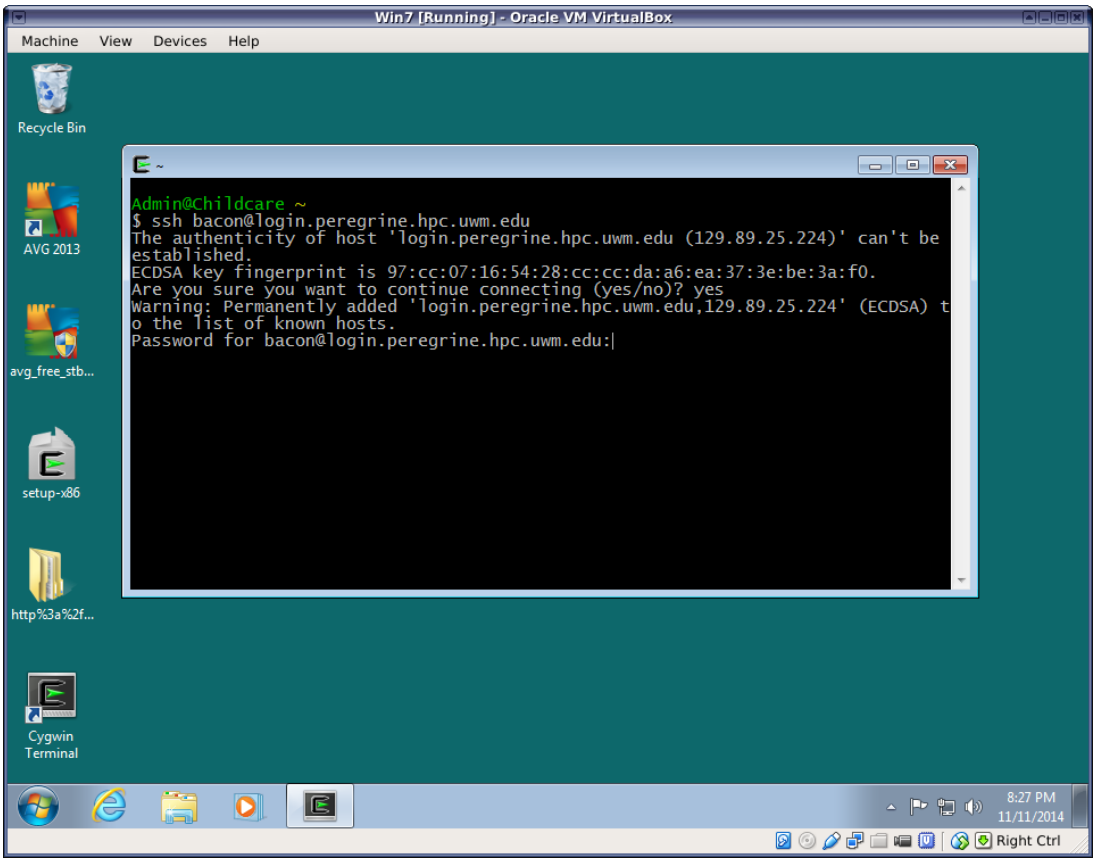
If you'd like to change the font size or colors of the Cygwin terminal emulator, just right-click on the title bar of the window:





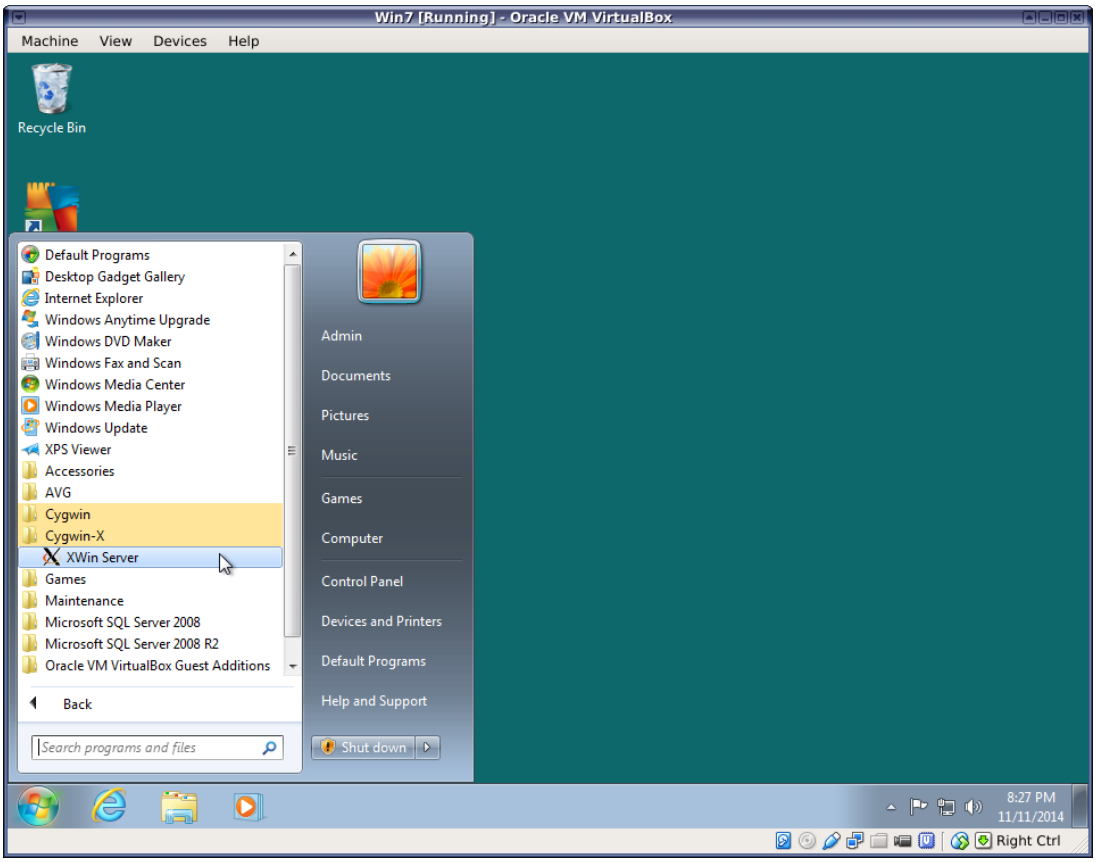
Within the Cygwin terminal window, you are now running a "bash" Unix shell and can run most common Unix commands such as "ls", "pwd", etc.

If you selected the openssh package during the Cygwin installation, you can now remotely log into other Unix machines, such as the clusters, over the network:



Note If you forgot to select the openssh package, just run the Cygwin setup program again. The packages you select when running it again will be added to your current installation.

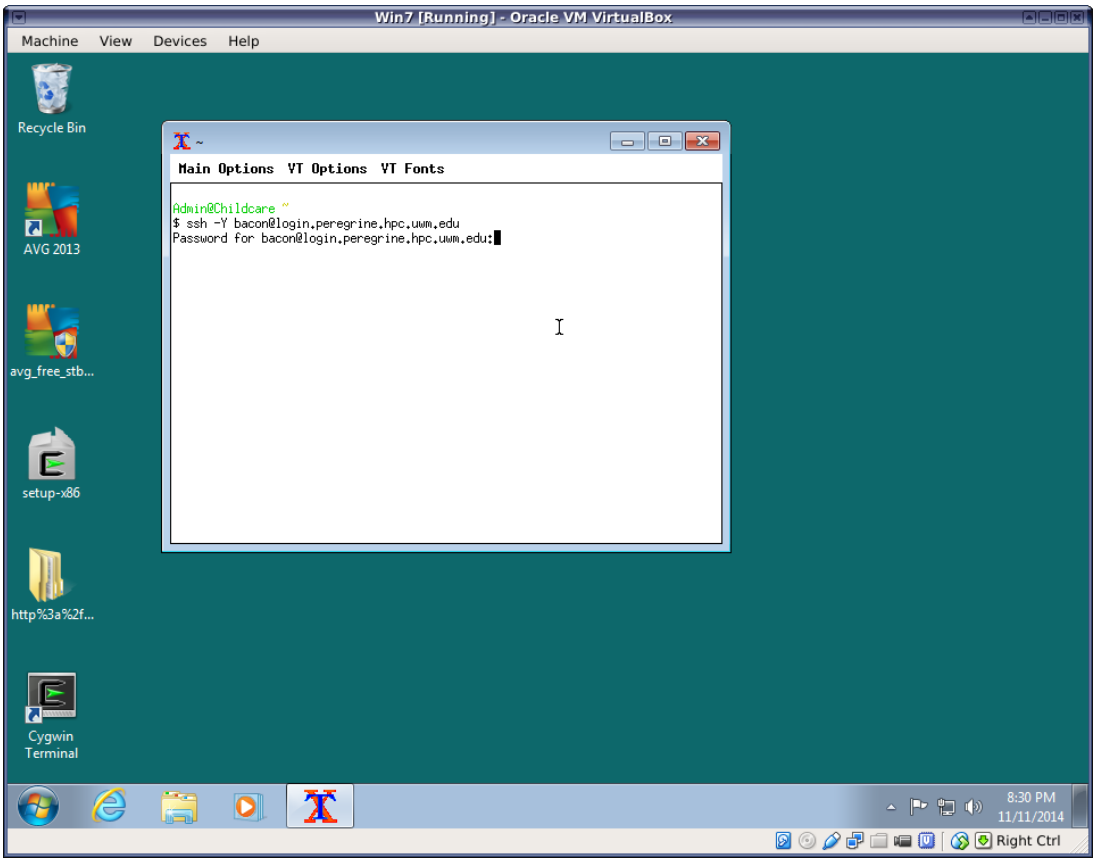
If you want to run Unix graphical applications, either on your Windows machine or on a remote Unix system, run the Cygwin/X application:



Note Doing graphics over a network may require a fast connection. If you are logging in from home or over a wireless connection, you may experience very sluggish rendering of windows from the remote host.

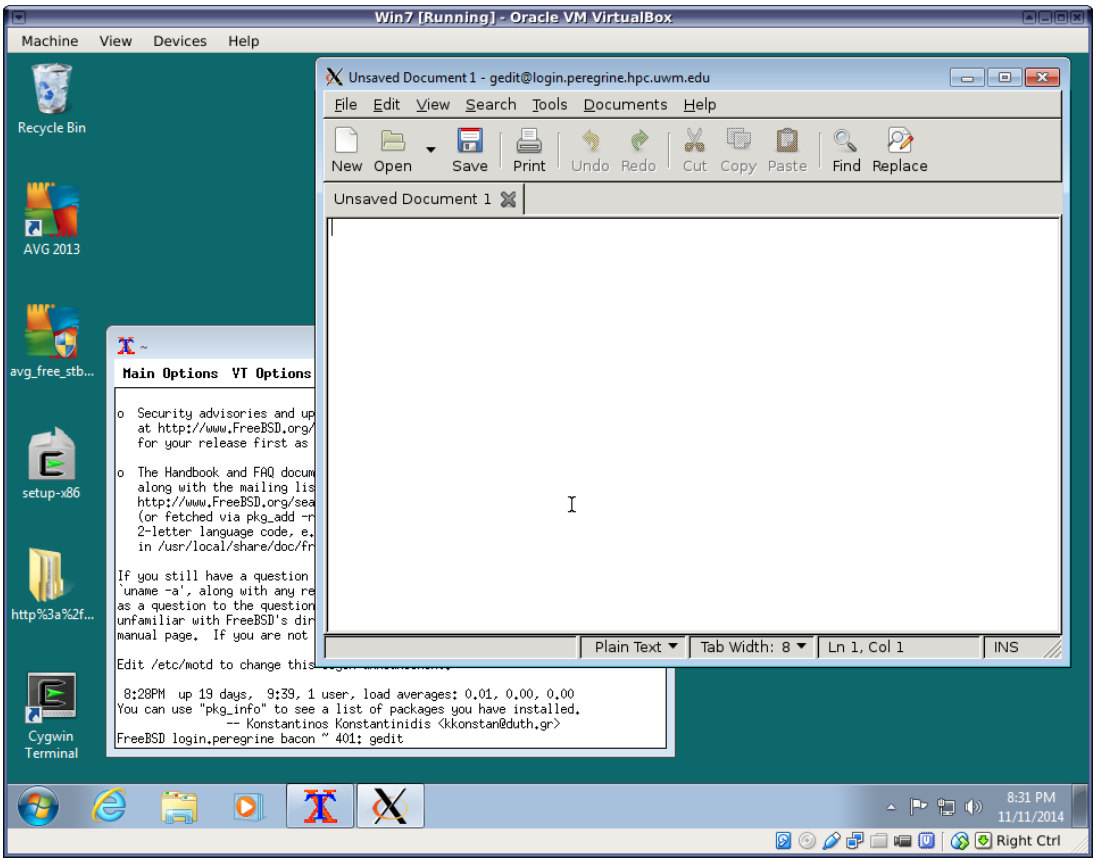
Depending on your Cygwin setup, this might automatically open a terminal emulator called "xterm", which is essentially the same as the standard Cygwin terminal, although it has a different appearance. You can use it to run all the same commands you would in the standard Cygwin terminal, including ssh. You may need to use the -X or -Y flag with ssh to enable some remote graphical programs.

Unlike Cygwin Terminal, the xterm supplied with Cygwin/X is preconfigured to support graphical applications. See Section 1.19.2 for details.



Caution Use of the -X and -Y flags could compromise the security of your Windows system by allowing malicious programs on the remote host to display phony windows on your PC. Use them *only* when logging into a trusted host.

Once you are logged into the remote host from the Cygwin/X xterm, you should be able to run graphical Unix programs.



You can also run graphical applications from the standard Cygwin terminal if you update your start up script. If you are using bash (the Cygwin default shell), add the following line to your .bashrc file:

```
export DISPLAY=unix:0.0
```

You will need to run **source .bashrc** or restart your bash shell after making this change.

If you are using T-shell, the line should read as follows in your .cshrc or .tcshrc:

```
setenv DISPLAY unix:0.0
```

Again, Cygwin is not the ideal way to run Unix programs on or from a Windows machine, but it is a very quick and easy way to gain access to a basic Unix environment and many Unix tools. Subsequent sections provide information about other options besides Cygwin for those with more sophisticated needs.

1.4.2 Windows Subsystem for Linux: Another Compatibility Layer

Windows Subsystem for Linux (WSL) is the latest in a chain of Unix compatibility systems provided by Microsoft. It allows Windows to run a subset of a Linux environment. As of this writing, the user can choose from a few different Linux distributions such as Ubuntu, Debian, SUSE, or Kali.

Differences from Cygwin:

- WSL runs actual Linux binaries (executables), whereas Cygwin allows the user to compile Unix programs into native Windows executables. Programs build under WSL can be run on a compatible Linux distribution and vice-versa. They cannot be run on Windows outside WSL. Programs compiled under Cygwin can in some cases be run under Windows outside Cygwin, but Cygwin cannot run binaries from a real Linux system. Which one you prefer depends on your specific goals. For many people, including most of us who just want to develop or run scientific programs, it makes no difference.
- WSL provides direct access to the native package collection of the chosen Linux distribution. For example, WSL users running the Debian app can install software directly from the Debian project using **apt-get**, just as they would on a real Debian system.

The Debian package collection is much larger than Cygwin's, so if Cygwin does not have a package for software you need, WSL might be a good option.

- WSL is a virtual machine, based on Microsoft Hyper-V. It requires a substantial amount of memory and requires that virtualization features are enabled in the PC BIOS. If your Windows installation is running in a virtual machine, you must also have nested virtualization installed, and WSL performance will suffer.
- Cygwin is an independent open source project, while WSL is a Microsoft product. There are pros and cons to each. Microsoft could change or even terminate support for WSL, as it has done with previous Unix compatibility products, if it no longer appears to be in the company's interest to support it. Support for Cygwin will continue as long as the user community is willing to contribute.

1.4.3 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. Describe three ways we can use Unix software on a Windows machine.
2. What is the advantage of Cygwin over a virtual machine?
3. What is the risk of using Cygwin?
4. What are two advantages of a virtual machine over Cygwin and WSL? Explain.
5. What is an advantage of Cygwin over WSL?

1.5 Logging In Remotely

Virtually all Unix systems allow users to log in and run programs over a network from other locations. This feature is intrinsic to Unix systems, and only disabled on certain proprietary or embedded installations. It is possible to use both GUIs and CLIs in this

fashion, although GUIs may not work well over slow connections such as a slower home Internet services. Different graphical programs have vastly different bandwidth demands. Some will work fine over a DSL, cable, or WiFi connection, while others require a fast wired connection.

The command line interface, on the other hand, works comfortably on even the slowest network connections.

Logging into a Unix CLI from a remote location is usually done using *Secure Shell (SSH)*.

Caution Older protocols such as rlogin, rsh, and telnet, should no longer be used due to their lack of security. These protocols transport passwords over the Internet in unencrypted form, so people who manage the gateway computers they pass through can easily read them.

1.5.1 Unix to Unix

If you want to remotely log in from one Unix system to another, you can simply use the **ssh** command from the command line. The general syntax of the **ssh** command is:

```
ssh [flags] login-id@hostname
```

The login-id portion is your login name on the remote host. If you are logging into a campus-managed server, this is likely the same campus login ID used to log into other services such as VPN, email, Canvas, etc.

The first time you connect to each remote host, you will be asked to verify that you trust it. You must enter the full word "yes" to continue:

```
The authenticity of host 'unixdev1.ceas.uwm.edu (129.89.25.223)' can't be established.
ED25519 key fingerprint is SHA256:askjdkj2ksjfdfamnmnmw5lka7jdkjka,mksjdkssfj.
No matching host key fingerprint found in DNS.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
```

If the connection is successful, you will be asked for your password. Note that nothing will echo to your screen as you type the password. Login panels that echo a '*' or a dot for each character are less secure, since someone looking over your shoulder can see exactly how long your password is. Knowing the length reduces the parameter space they would have to search in order to guess the password.

If you plan to run graphical programs on the remote Unix system, you may need to include the `-X` (enable X11 forwarding) or `-Y` (enable trusted X11 forwarding) flag in your **ssh** command. Run **man ssh** for full details.

Caution Use `-X` or `-Y` only when connecting to trusted computers, i.e. those managed by you or someone you trust. These options allow the remote system to access your display, which can pose a security risk. For example, a hacker on the remote system could display a fake login panel on your screen in order to steal your login and password.

Caution Only ssh should be used to log into remote systems. Older commands such as rsh and telnet lack encryption and are not secure. If anyone tells you to use rsh or telnet, they should not be trusted regarding any computing issues.

Examples:

```
shell-prompt: ssh joe@unixdev1.ceas.uwm.edu
```

Note For licensing reasons, **ssh** may not be included in basic Linux installations, but it can be very easily added via the package management system of most Linux distributions.

If you have X11 capabilities and used -X or -Y with your ssh command, you can easily open additional terminals from the command-line if you know the name of the terminal emulator. Simply type the name of the terminal emulator, followed by an '&' to put it in the background. (See Section 1.18.3 for a full explanation of background jobs.) Some common terminal emulators are coreterminal, konsole, urxvt, and xterm.

```
shell-prompt: coreterminal &
```

1.5.2 Windows to Unix

If you're connecting to a Unix system from a Windows system, you will need to install some additional software.

Cygwin

The Cygwin Unix-compatibility system is free, quick and easy to install, and equips a Windows computer with most common Unix commands, including a Unix-style Terminal emulator. Once Cygwin is installed, you can open a Cygwin terminal on your Windows desktop and use the **ssh** command as shown above.

The Cygwin installation is very quick and easy and is described in Section 1.4.1.

PuTTY

A more limited method for remotely accessing Unix systems is to install a stand-alone terminal emulator, such as PuTTY, https://www.chiark.greenend.org.uk/~sgtatham/putty/. PuTTY has a built-in **ssh** client, and a graphical dialog box for connecting to a remote machine. For more information, see the PuTTY documentation.

1.5.3 Terminal Types

In rare cases, you may be asked to specify a terminal type when you log in:

```
TERM=(unknown)
```

Terminal features such as cursor movement and color changes are triggered by sending special codes (characters or character combinations called *magic sequences*) to the terminal. Pressing keys on the terminal sends codes from the terminal to the computer.

Different types of terminals use different magic sequences. PuTTY and most other terminal emulators emulate an "xterm" terminal, so if asked, just type the string "xterm" (without the quotes).

If you fail to set the terminal type, some programs such as text editors will not function. They may garble the screen and fail to recognize special keys such as arrows, page-up, etc.

You can set the terminal type after logging in, but the methods for doing this vary according to which shell you use, so you may just want to log out and remember to set the terminal type when you log back in.

Practice Break

Remotely log into another Unix system using the **ssh** command or PuTTY, or open a shell on your Mac or other Unix system. Then try the commands shown below.

Unix commands are preceded by the shell prompt "shell-prompt: ". Other text below refers to input to the program (command) currently running. You must exit that program before running another Unix command.

Lines beginning with '#' are comments, and not to be types.

```
# List files in the current working directory (folder)
shell-prompt: ls
shell-prompt: ls -al

# Two commands on the same line
shell-prompt: ls; ls /etc

# List files in the root directory
shell-prompt: ls /

# List commands in the /bin directory
shell-prompt: ls /bin

# Create a subdirectory
shell-prompt: mkdir -p Data/IRC

# Change the current working directory to the new subdirectory
shell-prompt: cd Data/IRC

# Print the current working directory
shell-prompt: pwd

# See if the nano editor is installed
# nano is a simple text editor (like Notepad on Windows)
shell-prompt: which nano

    If this does not report "command not found", then do the following:

# Try the nano editor
shell-prompt: nano sample.txt

# Type in the following text:

This is a text file called sample.txt.
I created it using the nano text editor on Unix.

# Then save the file (press Ctrl+o), and exit nano (press Ctrl+x).
# You should now be back at the Unix shell prompt.

# Try the "vi" editor
# vi is standard editor on all Unix system.  It is more complex than nano.
shell-prompt: vi sample.txt

    Type 'i' to go into insert mode
    Type in some text
    Type Esc to exit insert mode and go back to command mode
    Type :w to save
    Type :q to quit

shell-prompt: ls

# Echo (concatenate) the contents of the new file to the terminal
shell-prompt: cat sample.txt

# Count lines, words, and characters in the file
shell-prompt: wc sample.txt

# Change the current working directory to your home directory
shell-prompt: cd
shell-prompt: pwd

# Show your login name
```

1.5.4 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What must be added to Unix to allow remote access?
2. Can we run graphical programs on remote Unix systems? Elaborate.
3. Does the CLI require a fast connection for remote operation?
4. What command would you use to log into a remote system with host name "myserver.mydomain.edu" using the user name "joe", assuming you want to run a graphical X11 application?
5. What should you do if someone advises you to use rsh or telnet?
6. How can Windows users add an ssh command like the one used on Unix systems?
7. What is the purpose of the TERM environment variable? What will happen if it is not set correctly?

1.6 Unix Command Basics

A Unix command is built from a command name and optionally one or more command line *arguments*. Arguments can be either *flags* or data.

```
ls -a -l /etc /var
^^ ^^^^^ ^^^^^^^^^
|   |    |
|   |    Data Arguments
|   Flags
Command name
```

- The *command name* is either the filename of a program or a command built into the shell. For example, the **ls** command is a program that lists the contents of a directory. The **cd** command is part of the shell.
- Most commands have optional *flags* (sometimes called *options*) that control the behavior of the command. By convention, flags begin with a '-' character.

  Note Unix systems do not enforce this, but very few commands violate it. Unix programmers tend to understand the benefits of conventions and don't have to be coerced to follow them.

  The flags in the example above have the following meaning:

  `-a`: tells **ls** to show "hidden" files (files whose names begin with '.', which ls would not normally list).

  `-l`: tells **ls** to do a "long listing", which is to show lots of information about each file and directory instead of just the name.

  Single-letter flags can usually be combined, e.g. `-a -l` can be abbreviated as `-al`.

  Most newer Unix commands also support long flag names, mainly to improve readability of commands used in scripts. For example, in the Unix **zip** command, `-C` and `--preserve-case` are synonymous. Using -C saves typing, but --preserve-case is more easily understood.
- Many commands also accept one or more data arguments, which provide input data to the command, or instruct it where to send output. Such arguments may be the actual input data or they may be the names of files or directories that contain input or receive output. The `/etc` and `/var` arguments above are directories to be listed by **ls**. If no data arguments are given to **ls**, it lists the current working directory (described in Section 1.8).

For many Unix commands, the flags must come before the data arguments. A few commands require that certain flags appear in a specific order. Some commands allow flags and data arguments to appear in any order. Unix systems do not enforce any rules regarding arguments. How they behave is entirely up to the programmer writing the command. However, the vast majority of commands follow conventions, so there is a great deal of consistency in Unix command syntax.

The components of a Unix command are separated by white space (space or tab characters). Hence, if an argument contains any white space, it must be enclosed in quotes (single or double) so that it will not be interpreted as multiple separate arguments.

Example 1.1 White space in an Argument

Suppose you have a directory called `My Programs`, and you want to see what's in it. You might try the following:

```
shell-prompt: ls My Programs
```

The above command fails because "My" and "Programs" are interpreted as two separate arguments. The **ls** command will look for two separate files or directories called "My" and "Programs". In this case, we must use quotes to bind the parts of the directory name together into a single argument. Either single or double quotes are accepted by all common Unix shells. The difference between single and double quotes is covered in Chapter 2.

```
shell-prompt: ls 'My Programs'
shell-prompt: ls "My Programs"
```

As an alternative to using quotes, we can *escape* the space by preceding it with a backslash ('\') character. This will save one keystroke if there is only one character to be escaped in the text.

```
shell-prompt: ls My\ Programs
```

Practice Break
Try the following commands:

```
shell-prompt: ls
shell-prompt: ls -al
shell-prompt: ls /etc
shell-prompt: ls -al /etc
shell-prompt: mkdir My Programs
shell-prompt: ls
shell-prompt: rmdir My
shell-prompt: rmdir Programs
shell-prompt: mkdir 'My Programs'
shell-prompt: ls
shell-prompt: ls My Programs
shell-prompt: ls "My Programs"
```

1.6.1 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

1. What are the three major components of a Unix command?
2. What are the two sources of the command name?
3. How do we know whether an argument is a flag or data?
4. What is the advantage of short flags and the advantage of long flags?
5. What do data argument represent?
6. What rules does Unix enforce regarding the order of arguments?

7. What separates one Unix argument from the next?
8. Can an argument contain whitespace? If so, how?

1.7 Basic Shell Tools

1.7.1 Common Unix Shells

There are many different shells available for Unix systems. This might sound daunting if you're new to Unix, but fortunately, like most Unix tools, all the common shells adhere to certain standards. All of the common shells are derived from one of two early ancestors:

- Bourne shell (sh) is the de facto basic shell on all Unix systems, and is derived from the original Unix shell developed at AT&T.
- C shell (csh) offers mostly the same features as Bourne shell, but the two differ in the syntax of their scripting languages, which are discussed in Chapter 2. The C shell syntax is designed to be more intuitive and similar to the C language.

Most Unix commands are exactly the same regardless of which shell you are using. Differences will only become apparent when using more advanced command features or writing shell scripts, both of which we will cover later.

Common shells derived from Bourne shell include the following:

- Almquist shell (ash), used as the Bourne shell on some BSD systems.
- Korn shell (ksh), an extended Bourne shell with many added features for user-friendliness.
- Bourne again shell (bash) another extended Bourne shell from the GNU project with many added features for user-friendliness. Used as the Bourne shell on some Linux systems.
- Debian Almquist shell (dash), a reincarnation of ash which is used as the Bourne shell on Debian based Linux systems.

Common shells derived from C shell include the following:

- T shell (tcsh), and extended C shell with many added features for user-friendliness.
- Hamilton C shell, an extended C shell used primarily on Microsoft Windows.

Unix systems differ in which shells are included in the base installation, but most shells can be easily added to any Unix system using the system's package manager.

1.7.2 Command History

Most shells remember a configurable number of recent commands. This command history is saved both in memory and to disk, so that you can still recall this session's commands next time you log in. The exact mechanisms for recalling those commands varies from shell to shell, but some of the features common to all shells are described below.

Most modern shells support scrolling through recent commands by using the up-arrow and down-arrow keys. Only very early shells like lack this capability.

Note This feature may not work if your TERM variable is not set properly, since the arrow keys send magic sequences that may differ among terminal types.

The **history** command lists the commands that the shell currently has in memory.

```
shell-prompt: history
```

A command consisting of an exclamation point (!) followed by any character string causes the shell to search for the most recently executed command that began with that string. This is particularly useful when you want to repeat a complicated command.

```
shell-prompt: find Programs -name '*.o' -exec rm -i '{}' \;
shell-prompt: !find
```

An exclamation point followed by a number runs the command with that history index:

```
shell-prompt: history
   385  13;42   more output.txt
   386  13:54   ls
   387  13:54   cat /etc/hosts
shell-prompt: !386
ls
Avi-admin/            Materials-Studio/     iperf-bsd
Backup@               New-cluster/          notes
Books/                Peregrine-admin/      octave-workspace
```

Tantalizing sneak preview: We can check the history for a particular pattern such as "find" as follows:

```
shell-prompt: history | grep find
```

More on the "| find" in Section 1.13.

1.7.3 Auto-completion

In most Unix shells, you need only type enough of a command or argument filename to uniquely identify it. At that point, pressing the TAB key will automatically fill in the rest for you. Try the following:

```
shell-prompt: touch sample.txt
shell-prompt: cat sam<Press the TAB key now>
```

If there are other files in your directory that begin with "sam", you may need to type a few additional characters before the TAB, like 'p' and 'l' before auto-completion will work.

1.7.4 Command-line Editing

Modern shells allow extensive editing of the command currently being entered. The key bindings for different editing features depend on the shell you are using and the current settings. Some shells offer a selection of different key bindings that correspond to Unix editors such as **vi** or **Emacs**.

See the documentation for your shell for full details. Below are some example default key bindings for shells such as **bash** and **tcsh**.

Key	**Action**
Left arrow	Move left
Right arrow	Move right
Ctrl+a	Beginning of line
Ctrl+e	End of line
Backspace or Ctrl+h	Delete left
Ctrl+d	Delete current

Table 1.3: Default Key Bindings in some Shells

1.7.5 Globbing (File Specifications)

There is often a need to specify a large number of files as command line arguments. Typing all of them would be tedious, so Unix shells provide a mechanism called *globbing* that allows short, simple patterns to match many file names. This allows us to type a brief specification that represents a large number (a glob) of files.

These patterns are built from literal text and/or special symbols called *wild cards* as shown in Table 1.4.

Symbol	**Matches**
*	Any sequence of characters (including none) except a '.' in the first character of the filename.
?	Any single character, except a '.' in the first character of the filename.
[string]	Any character in string
[c1-c2]	Any character from c1 to c2, inclusive
{thing1,thing2}	Thing1 or thing2

Table 1.4: Globbing Symbols

Normally, the shell handles these special characters, expanding globbing patterns to a list of matching file names *before* the command is executed.

If you want an argument containing special globbing characters to be sent to a command in its raw form, it must be enclosed in quotes, or each special character must be escaped (preceded by a backslash, \).

Certain commands, such as **find** need to receive the pattern as an argument and attempt to do the matching themselves rather than have it done for them by the shell. Therefore, patterns to be used by the **find** command must be enclosed in quotes.

```
shell-prompt: ls *.txt      # Lists all files ending in ".txt"
shell-prompt: ls "*.txt"    # Fails, unless there is a file called '*.txt'
shell-prompt: ls '*.txt'    # Fails, unless there is a file called '*.txt'
shell-prompt: ls .*.txt     # Lists hidden files ending in ".txt"
shell-prompt: ls [A-Za-z]* # Lists all files and directories
                           # whose name begins with a letter
shell-prompt: find . -name *.txt    # Fails
shell-prompt: find . -name '*.txt'  # List .txt files in all subdirectories
shell-prompt: ls *.{c,c++,f90}
```

Caution The exact behavior of character ranges such as [A-Z] may be affected by locale environment variables such as LANG, LC_COLLATE, and LC_ALL. See Section 1.16 for general information about the environment. Information about locale and collation can be found online. Behavior depends on these settings as well as which Unix shell you are using and the shell's configuration settings. Setting LANG and the LC_ variables to C or C.UTF-8 will usually ensure the behavior described above.

1.7.6 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What is the de facto standard shell on Unix systems?
2. How do most Unix commands differ when run under one shell such as C shell as opposed to running under another such as Bourne shell or Bourne again shell?
3. What if a shell we like or need is not present on our Unix installation?
4. How can we quickly rerun the previous command in most Unix shells?
5. How can we list all recent commands executed from this shell?
6. How can we run the last command that began with "ls"?
7. Given the shell history shown below, how can we run the last command used to log into unixdev1?

```
984  16:48   vi .ssh/known_hosts
985  16:50   ssh -X bacon@unixdev1.ceas.uwm.edu
986  16:52   ssh -X -C bacon@unixdev1.ceas.uwm.edu
987  16:58   ape
988  16:59   ssh -X -C bacon@unixdev1.ceas.uwm.edu
```

8. How can we avoid typing a long file name or command name in most shells?
9. How can we instantly move to the beginning of the command we are currently typing?
10. How do we list all the non-hidden files in the current directory ending in ".txt"?
11. How do we list all the hidden files in the current directory ending in ".txt"?
12. How do we list all the files in /etc beginning with "hosts"?

13. How do we list all the files in the current directory starting with a lower case letter and ending in ".txt"?
14. How do we list all the files in the current directory starting with any letter and ending in ".txt"?
15. How do we list all the non-hidden files in the current directory ending with ".pdf" or ".txt"?
16. How do we list all the files in /etc and all other directories under /etc with names ending in ".conf"?
17. When are globbing patterns normally expanded to a list of files?
18. How can we include a file name in a command if the file name contains a special character such as '*' or '['?

1.8 Processes

A program is a file containing statements or commands in the form of source code or machine code. A *process*, in Unix terminology, is the *execution of a program*. By this we mean the running of a program, not a blindfold, a cigarette, and firing squad. A program is an object. A process is an action utilizing that object. A grocery list is like a program. A trip to the grocery store to buy what is on the list is like a process.

Unix is a multitasking system, which means that many processes can be running at any given moment, i.e. there can be many active processes.

When you log in, the system creates a new process to run your shell program. The same happens when other people log in. Hence, while everyone may be using the same shell *program*, they are all running different shell *processes*.

When you run a program (a command) from the shell, the shell creates a new process to run the program. Hence, you now have two processes running: the shell process and the command's process. The shell then normally waits for that child process to complete before printing the shell prompt again and accepting another command.

The process created by the shell to run your command is called a *child process* of the shell process. Naturally, the shell process is then called the *parent process* of the command process.

Each process is uniquely identified by an integer serial number called the *process ID*, or *PID*.

Unix systems also keep track of each process's status and resource usage, such as memory, CPU time, etc. Information about your currently running processes can be viewed using the **ps** (process status) command:

```
shell-prompt: ps
  PID TTY          TIME CMD
 7147 ttys000    0:00.14 -tcsh
 7438 ttys000    0:01.13 ape notes.dbk unix.dbk
 7736 ttys001    0:00.13 -tcsh
```

Practice Break

Run the **ps** command. What processes do you have running?

```
shell-prompt: ps
```

What if we want to see all the processes on the system, instead of just our own? On most systems, we can add the `-a` (include other peoples' processes) and `-x` (include processes not started from a terminal) flags.

```
shell-prompt: ps -ax
```

Another useful tool is the **top** command, which monitors all processes in a system and displays system statistics and the top (most active) processes every few seconds. Note that since **top** is a full-terminal command, it will not function properly unless the TERM environment variable is set correctly.

Practice Break
Run the **top** command. What processes are using the most CPU time? Type 'q' to quit **top**.

```
shell-prompt: top
```

1.8.1 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.
   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

1. How does a process differ from a program?
2. If 10 people are logged in and using the same Unix shell, how many shell programs are there? How many shell processes?
3. What normally happens when you run a program from the shell?
4. How are processes identified in Unix?
5. How can we list all the processes currently running on the system?
6. How can we monitor which processes are using the most CPU and memory resources?

1.9 The Unix File System

1.9.1 Unix Files

A Unix *file* is simply a sequence of bytes (8-bit values) stored on a disk and given a unique name. The bytes in a file may be printable characters such as letters, digits, punctuation symbols, invisible *control characters* (which cause a printer or terminal

to perform actions such as backspacing or scrolling), part of a number (a typical integer or floating point number consists of 8 bytes), or other non-character, non-numeric data.

This is how Unix sees all files. It takes no interest whatsoever in the meaning of the bytes within a file. The meaning of the content is determined solely by the programs using the file.

Text vs Binary Files

Files are often classified as either *text* or *binary* files. All of the bytes in a text file are interpreted as ASCII/ISO characters by the programs that read or write the file, while binary files may contain both character and non-character data.

Again, Unix does not make a distinction between text and binary files. This is left to the programs that use the files.

Practice Break
Try the following commands:

```
shell-prompt: cat /etc/hosts
```

What do you see? The `/etc/hosts` file is a text file, and **cat** is used here to echo (concatenate) it to the terminal output. Now try the following:

```
shell-prompt: cat /bin/ls
```

What do you see? The file `/bin/ls` is not a text file. It contains binary program code, not characters. The **cat** command assumes that the file is a text file and sends each byte to your terminal. The terminal tries to interpret each byte as an ASCII/ISO character and display it on the screen. Since the file does not contain a sequence of characters, it appears as nonsense on your terminal. Some of the bytes sent to the terminal may even knock it out of whack, causing it to behave strangely. If this happens, run the **reset** command to restore your terminal to its default state.

Unix vs. Windows Text Files

While it is the programs that interpret the contents of a file, there are some conventions regarding text file format that all Unix programs follow, so that they can all manipulate the same files. Unfortunately, Windows programs follow different conventions. Unix programs assume that text files terminate each line with a control character known as a *line feed* (also known as a *newline* or *NL* for short), which is the 10th character in the standard ASCII/ISO character sets. Windows programs use both a *carriage return* or *CR* (13th character) and NL.

Text files created on Windows will contain both a CR and NL at the end of each line. Text files created on Unix will have only an NL. This can cause problems for programs on either Unix or Windows. Hence, it is not a good idea to use a Windows editor to write code for Unix systems or vice-versa.

The **dos2unix** and **unix2dos** commands can be used to clean up files that have been transferred between Unix and Windows. These programs convert text files between the Windows and Unix standards. If you've edited a text file on a non-Unix system, and are now using it on a Unix system, you can clean it up by running:

```
shell-prompt: dos2unix filename
```

The **dos2unix** and **unix2dos** commands are not standard with most Unix systems, but they are free programs that can easily be added via most package managers.

Caution Note that **dos2unix** and **unix2dos** should only be used on text files. They should never be used on binary files, since the contents of a binary file are not meant to be interpreted as characters such as line feeds and carriage returns.

1.9.2 File system Organization

Basic Concepts

A Unix file system contains *files* and *directories*. A file is like a document, and a directory is like a folder that contains documents and/or other directories. The terms "directory" and "folder" are interchangeable, but "directory" is the standard term used in Unix.

Directories are so called because they serve the same purpose as the directory you might find in the lobby of an office building: They are listings that keep track of what files and other directories are called and where they are located on the disk.

Note

Unix file systems use case-sensitive file and directory names. I.e., `Temp` is not the same as `temp`, and both can coexist in the same directory.

Mac OS X is the only mainstream Unix system that violates this convention. The standard OS X file systems is case-preserving, but not case-sensitive. This means that if you call a file `Temp`, it will remember that the T is capital, but it can also be referred to as `temp`, `tEmp`, etc. Only one of these files can exist in a given directory at any one time.

A Unix file system can be visualized as a tree, with each file and directory contained within another directory. Figure 1.2 shows a small portion of a typical Unix file system. On a real Unix system, there are usually thousands of files and directories. Directories are shown in green and files are in yellow.

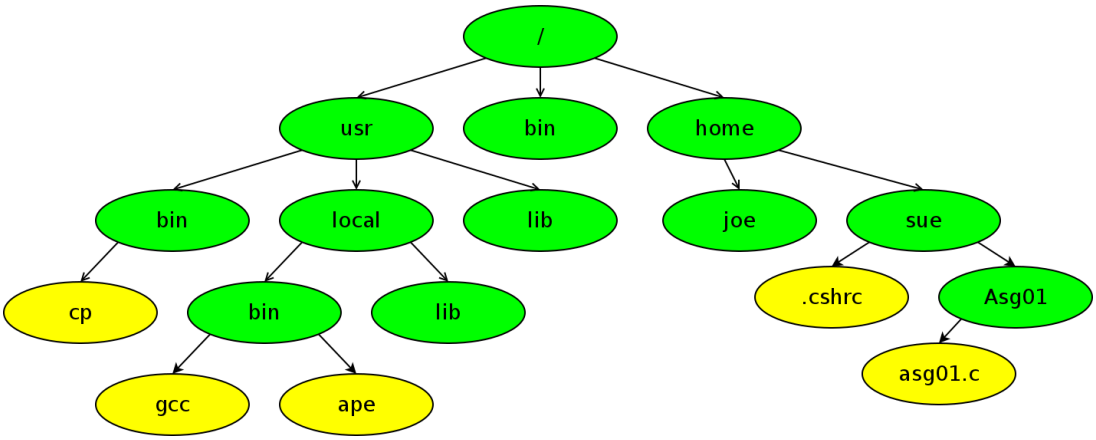


Figure 1.2: Sample of a Unix File system

Unix uses a forward slash (/) to separate directory and file names while Windows uses a backslash (\).

The one directory that is not contained within any other is known as the *root directory*, whose name under Unix is `/`. There is exactly one root directory on every Unix system. Windows systems, on the other hand, have a root directory for each disk partition such as C:\ and D:\.

The Cygwin compatibility layer works around the separate drive letters of Windows by unifying them under a common parent directory called /cygdrive. Hence, for Unix commands run under Cygwin, /cygdrive/c is equivalent to c:\, /cygdrive/d is equivalent to d:\, and so on. This allows Cygwin users to do things like search multiple Windows drive letters with a single command starting in /cygdrive.

Unix file system trees are fairly standardized, but most have some variation. For instance, all Unix systems have a /bin and a /usr/bin, which contain standard Unix commands. Not all of them have /home or /usr/local. Many Linux systems install commands from add-on packages into /usr/bin, mixing them with the standard Unix commands that are essential to the basic functioning of the system. Other systems such as most BSDs keep them separated in /usr/local/bin or /usr/pkg/bin.

The root directory is the *parent* of /bin and /home and an *ancestor* of all other files and directories.

The /bin and /home directories are *subdirectories*, or *children* of /. Likewise, /home/joe and /home/sue are subdirectories of /home, and grandchildren of /.

All of the files in and under /home comprise a *subtree* of /home.

The children of a directory, all of its children, and so on, are known as *descendants* of the directory. All files and directories on a Unix system, except /, are descendants of /.

Each user has a *home directory*, which can be arbitrarily assigned, but is generally a child of /home on many Unix systems or of /Users on macOS. Most or all of a user's files and subdirectories are found under their home directory. In the example above, /home/joe is the home directory for user joe, and /home/sue is the home directory for user sue.

In some situations, a home directory can be referred to as ~ or ~user. For example, user joe can refer to his home directory as ~, ~/, or ~joe, while he can only refer to sue's home directory as ~sue.

Absolute Path Names

The *absolute path name*, also known as *full path name*, of a file or directory denotes the complete path from / (the root directory) to the file or directory of interest. For example, the absolute path name of Sue's .cshrc file is /home/sue/.cshrc, and the absolute path name of the ape command is /usr/local/bin/ape.

The absolute path name is the only way to uniquely identify a file or directory in the file system.

Note An absolute path name always begins with '/' or a '~', noting that '~' is shorthand for a path that begins with a '/' such as /home/joe or /Users/joe.

Practice Break
Try the following commands:

```
shell-prompt: ls
shell-prompt: ls /etc
shell-prompt: cat /etc/hosts
shell-prompt: ls ~
```

Current Working Directory

Every Unix *process* has an attribute called the *current working directory*, or *CWD*. This is the directory that the process is currently "in". When you first log into a Unix system, the shell process's CWD is set to your home directory.

Note It is important to understand that the CWD is a property of each *process*, not of a user or a program.

The **pwd** command prints the CWD of the shell process. The **cd** command changes the CWD of the shell process. Running **cd** with no arguments sets the CWD to your home directory, much like clicking your heels together three times to get back to Kansas.

Practice Break
Try the following commands:

```
shell-prompt: pwd
shell-prompt: cd /
shell-prompt: pwd
shell-prompt: cd
shell-prompt: pwd
```

Many commands, such as **ls**, use the CWD as a default if you don't provide a directory name on the command line. For example, if the CWD is `/home/joe`, then the following commands are the same:

```
shell-prompt: ls
shell-prompt: ls /home/joe
shell-prompt: ls ~joe
```

Relative Path Names

Whereas an absolute path name denotes the path from / to a file or directory, the *relative path name* denotes the path from the CWD to a file or directory.

Any path name that does not begin with a '/' or '~' is interpreted as a relative path name. The absolute path name is then derived by appending the relative path name to the CWD. For example, if the CWD is `/etc`, then the relative path name `hosts` refers to the absolute path name `/etc/hosts`.

absolute path name = CWD + "/" + relative path name

Note Since the CWD is a property of each process, a relative path name is not unique. It may have different meaning to different processes, or different meaning to the same process before and after it changes its CWD. For example the meaning of the relative path name "bin" depends on whether CWD is / or /usr.

Note Relative path names are handled at the lowest level of the operating system, by the Unix kernel. This means that they can be used anywhere: in shell commands, in C or Fortran programs, etc.

When you run a program from the shell, the new process inherits the CWD from the shell. Hence, you can use relative path names as arguments in any Unix command, and they will use the CWD inherited from the shell. For example, the two **cat** commands below have the same effect.

```
shell-prompt: cd /etc       # Set shell's CWD to /etc
shell-prompt: cat hosts      # Inherits CWD from shell, so hosts = /etc/hosts
shell-prompt: cat /etc/hosts # Same effect as above
```

Wasting Time
The **cd** command is one of the most overused Unix commands. Many people use it where it is completely unnecessary and actually results in significantly more typing than needed. Don't use **cd** where you could have used the directory with another command. For example, consider the sequence of commands:

```
shell-prompt: cd /etc
shell-prompt: more hosts
shell-prompt: cd
```

The same effect could have been achieved much more easily using the following single command:

```
shell-prompt: more /etc/hosts
```

Note In almost all cases, absolute path names and relative path names are interchangeable. You can use either type of path name as a command line argument, or within a program.

Practice Break
Try to predict the results of the following commands before running them:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cd /etc
shell-prompt: pwd
shell-prompt: cat hosts
shell-prompt: cat /etc/hosts
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cat hosts
```

Why does the last command result in an error?

Avoid Absolute Path Names

The relative path name is potentially much shorter than the equivalent absolute path name. Using relative path names also makes code more portable.

Suppose you have a project contained in the directory /Users/joe/Thesis on your Mac. Now suppose you want to work on the same project on a cluster, where there is no /Users directory and you have to store it in /share1/joe/Thesis.

The absolute path name of every file and directory will be different on the cluster than it is on your Mac. This can cause major problems if you were using absolute path names in your scripts, programs, and makefiles. Statements like the following will have to be changed in order to run the program on a different computer.

```
infile = fopen("/Users/joe/Thesis/Inputs/input1.txt", "r");
```

```
sort /Users/joe/Thesis/Inputs/names.txt
```

No program should ever have to altered just to make it run on a different computer. Changes like these are a source of *regressions* (new program bugs).

While the absolute path names change when you move the Thesis directory, the path names relative to the Thesis directory remain the same. For this reason, absolute path names should be avoided unless they are guaranteed to be portable, which is very rare.

The statements below will work on any computer as long as the program or script is running with Thesis as the CWD. It does not matter where the Thesis directory is located, so long as the Inputs directory is its child.

```
infile = fopen("Inputs/input1.txt", "r");
```

```
sort Inputs/names.txt
```

Special Directory Names

In addition to absolute path names and relative path names, there are a few special symbols for directories that are commonly referenced:

Symbol	Refers to
.	The current working directory
..	The parent of the current working directory
~	Your home directory
~user	user's home directory

Table 1.5: Special Directory Symbols

The '.' notation for CWD is useful for copying files to CWD and other commands that require a target directory name.

It is also useful if a mishap occurs, leading to the creation of a file whose name begins with a special character such as '-' or '~'. If we have a file called "-file.txt", we cannot remove it with **rm -file.txt**, since the **rm** command with think the '-' indicates a flag argument. To get around this, we simply need to make the argument not begin with a '-'. We can either use the absolute path name of the file, e.g. `/home/joe/-file.txt` or `./-file.txt`.

Practice Break
Try the following commands and see what they do:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: ls
shell-prompt: ls ~
shell-prompt: ls .
shell-prompt: mkdir Data Scripts
shell-prompt: cp /etc/hosts .
shell-prompt: mv hosts Data
shell-prompt: ls Data
shell-prompt: cd Data
shell-prompt: cd ../Scripts
shell-prompt: ls ..
shell-prompt: ls ../Data
shell-prompt: more ../Data/hosts
shell-prompt: rm ../Data/hosts
shell-prompt: ls ~/Data
shell-prompt: ls /bin
shell-prompt: cd ..
shell-prompt: pwd
```

1.9.3 Ownership and Permissions

Overview

Every file and directory on a Unix system has inherent access control features based on a simple system:

- Every file and directory belongs to an individual user and to a group of users.
- There are 3 types of permissions which are controlled separately from each other:
  - Read
  - Write (modify)
  - Execute (e.g. run a file if it's a program)
- Read, write, and execute permissions can be granted or denied separately for each of the following:
  - The individual who owns the file (user)
  - The group that owns the file (group)
  - All other users on the system (a hypothetical group known as "world" (other)

Execute permissions on a file mean that the file can be executed as a script or a program by typing its name. It does not mean that the file actually contains a script or a program: It is up to the owner of the file to set the execute permissions appropriately for each file.

Execute permissions on a directory mean that permitted users can **cd** into it. Users only need read permissions on a directory to list it or access a file within it, but they need execute permissions in order for their processes to make it the CWD.

Unix systems provide this access using 9 on/off switches (bits) associated with each file.

Viewing Permissions

If you do a long listing of a file or directory, you will see the ownership and permissions:

```
shell-prompt: ls -l
drwx------   2 joe    users      512 Aug  7 07:52 Desktop/
drwxr-x---  39 joe    users     1536 Aug  9 22:21 Documents/
drwxr-xr-x   2 joe    users      512 Aug  9 22:25 Downloads/
-rw-r--r--   1 joe    users    82118 Aug  2 09:47 bootcamp.pdf
```

The leftmost column shows the type of object and the permissions for each user category.

A '-' in the leftmost character means a regular file, 'd' means a directory, 'l' means a link. etc. Running **man ls** will reveal all the codes.

The next three characters are, in order, read, write and execute permissions for the owner (joe).

The next three after that are permissions for members of the owning group (users).

The next three are permissions for world (other).

A '-' in a permission bit column means that the permission is denied for that user or set of users and an 'r', 'w', or 'x' means that read, write, or execute is permitted.

The next three columns show the number of links (different path names for the same file), the individual and group ownership of the file or directory. The remaining columns show the size, the date and time it was last modified, and name. In addition to the 'd' in the first column, directory names are followed by a '/' if the **ls** is so configured.

You can see above that Joe's `Desktop` directory is readable, writable, and executable for Joe, and completely inaccessible to everyone else.

Joe's `Documents` directory is readable, writable and executable for Joe, and readable and executable for members of the group "users". Users not in the group "users" cannot access the Documents directory at all.

Joe's `Downloads` directory is readable and executable to anyone who can log into the system.

The file `bootcamp.pdf` is readable by group and world, but only writable by Joe. It is not executable by anyone, which makes sense because a PDF file is not a program.

Setting Permissions

Users cannot change individual ownership on a file, since this would allow them to subvert disk quotas and do other malicious acts by placing their files under someone else's name. Only the *superuser* (the system administrator) can change the individual ownership of a file or directory.

Every user has a primary group and may also be a member of supplementary groups. Users can change the group ownership of a file to any group that they belong to using the **chgrp** command, which requires a group name as the second argument and one or more path names following the group:

```
shell-prompt: chgrp group path [path ...]
```

All sharing of files on Unix systems is done by controlling group ownership and file permissions.

File permissions are changed using the **chmod** command:

```
shell-prompt: chmod permission-specification path [path ...]
```

The permission specification has a symbolic form, and a raw form, which is an octal number.

The symbolic form consists of any of the three user categories 'u' (user/owner), 'g' (group), and 'o' (other/world) followed by a '+' (grant) or '-' (revoke), and finally one of the three permissions 'r', 'w', or 'x'.

To add read and execute (cd) permissions for group and world on the Documents directory:

```
shell-prompt: chmod go+rx Documents
```

Sometimes it is impossible to express the changes we want to make in one simple specification. In that case, we can use a compound specification, two or more basic specs separated by commas. Remember that white space indicates the end of an argument, so we cannot have any white space next to the comma.

To revoke all permissions for world on the Documents directory and grant read permission for the group:

```
shell-prompt: chmod o-rwx,g+r Documents
```

Disable write permission for everyone, including the owner, on bootcamp.pdf. This can be used to prevent the owner from accidentally deleting an important file.

```
shell-prompt: chmod ugo-w bootcamp.pdf
```

Run **man chmod** for additional information.

The raw form for permissions uses a 3-digit octal number to represent the 9 permission bits. This is a quick and convenient method for computer nerds who can do octal/binary conversions in their head.

```
shell-prompt: chmod 644 bootcamp.pdf   # 644 = 110100100 = rw-r--r--
shell-prompt: chmod 750 Documents      # 750 = 111101000 = rwxr-x---
```

Caution NEVER make any file or directory world-writable. Doing so allows any other user to modify it, which is a serious security risk. A malicious user could use this to install a Trojan Horse program under your name, for example.

By default, new files you create are owned by you and your primary group. If you are a member of more than one group and wish to share a directory with one of your supplementary groups, it may also be helpful to set a special flag on the directory so that new files created in it will have the same group as the directory, rather than your primary group. Then you won't have to remember to chmod every new file you create.

```
shell-prompt: chmod g+s Shared-research
```

Practice Break

Try the following commands, and try to predict the output of each **ls** before you run it.

```
shell-prompt: touch testfile
shell-prompt: ls -l
shell-prompt: chmod go-rwx testfile
shell-prompt: ls -l
shell-prompt: chmod o+rw testfile
shell-prompt: ls -l
shell-prompt: chmod g+rwx testfile
shell-prompt: ls -l
shell-prompt: rm testfile
```

Now set permissions on testfile so that it is readable, writable, and executable by you, only readable by the group, and inaccessible to everyone else.

1.9.4 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What is a file in the viewpoint of Unix?
2. What is the difference between a text file and a binary file?
3. What will happen if you echo a binary file to your terminal?
4. What is the difference between Windows and Unix text files?
5. How can we convert text files between the Unix and Windows standards?
6. What is a directory?
7. What does it mean that Unix filenames are case-sensitive?
8. What is a root directory?
9. How many root directories does a Unix system have? How many does Windows have?
10. What is contained in the /bin and /usr/bin directories?
11. What is a subdirectory?
12. What is a home directory?
13. What is an absolute path name and how do we recognize one?
14. What is the absolute path name of Sue's asg01.c in the tree diagram in this section?
15. Of what is the CWD a property?
16. How can we find out the CWD of a shell process?

17. How can we set the CWD of a shell process to /tmp?
18. How can we set the CWD of a shell process to our home directory?
19. What is a relative path name and how to we recognize one?
20. Is a relative path name unique? Prove your answer with an example.
21. How does Unix determine the absolute path name from a relative path name?
22. If the CWD of a process is /usr/local, what is the absolute path name of "bin/ape"?
23. Where does a new process get its initial CWD?
24. Why should we avoid using absolute path names in programs and scripts?
25. How can we list the contents of the parent directory of CWD?
26. If the CWD of a process is /home/bob/Programs, what is the relative path name of /home/bob/Data/input1.txt?
27. How do we remove a file called "~sue" in the CWD?
28. What are the three user categories that can be granted permissions on a file or directory?
29. What does it mean to set execute permission on a file? On a directory?
30. Given the following ls -l output, who can do what to bootcamp.pdf?

```
-rw-r-----   1 joe    users    82118 Aug  2 09:47 bootcamp.pdf
```

31. How would we allow users who are not in the owning group to read bootcamp.pdf?
32. How would we allow members of the group to read and execute the program "simulation" and at the same time revoke all access to other users?
33. How can we make the directory "MyScripts" world writable?
34. How can we change the group ownership of the directory "Research" to the group "smithlab".
35. Assuming your primary group is "joe", how can we configure the directory Research form the previous question so that new files you create in it will be owned by "smithlab" instead of "joe"?

1.10 Unix Commands and the Shell

Before You Begin You should have a basic understanding of Unix processes, files, and directories. These topics are covered in Section 1.8 and Section 1.9.

Unix commands fall into one of two categories:

- Internal commands are part of the shell.

  No new process is created when you execute an internal command. The shell simply carries out the execution of internal commands by itself.

- External commands are programs separate from the shell. The command name of an external command is actually the name of an *executable file*, i.e. a file containing the program or script. For example, when you run the **ls** command, you are executing the program contained in the file `/bin/ls`.

  When you run an external command, the shell locates the program file, loads the program into memory, and creates a new (child) process to execute the program. The shell then normally waits for the child process to end before prompting you for the next command.

1.10.1 Internal Commands

Commands are implemented internally only when it is necessary or when there is a substantial benefit. If all commands were part of the shell, the shell would be enormous and require too much memory.

One command that must be internal is the **cd** command, which changes the CWD of the shell process. The **cd** command cannot be implemented as an external command, since the CWD is a property of the process, as described in Section 1.9.2.

We can prove this using *Proof by Contradiction*. If the **cd** command were external, it would run as a child process of the shell. Hence, running **cd** would create a child process, which would inherit CWD from the shell process, alter its copy of CWD, and then terminate. The CWD of the parent, the shell process, would be unaffected.

Expecting an external command to change your CWD for you would be akin to asking one of your children to go to take a shower for you. Neither is capable of affecting the desired change. Likewise, any command that alters the state of the shell process must be implemented as an internal command.

1.10.2 External Commands

Most commands are external, i.e. programs separate from the shell. As a result, they behave the same way regardless of which shell we use to run them.

The executable files containing external commands are kept in certain directories, most of which are called `bin` (short for "binary", since most executable files are binary files containing machine code). The most essential commands required for the Unix system to function are kept in `/bin` and `/usr/bin`. The location of optional add-on commands varies, but a typical location is `/usr/local/bin`. Debian and Redhat Linux mix add-on commands with core system commands in `/usr/bin`. BSD systems keep them separate directories such as `/usr/local/bin` or `/usr/pkg/bin`.

Practice Break

1. Use **which** under C shell family shells to find out whether the following commands are internal or external. Use **type** under Bourne family shells (bash, ksh, dash, zsh). You can use either command under either shell, but will get better results if you follow the advice above. (Try both and see what happens.)

```
shell-prompt: which cd
shell-prompt: which cp
shell-prompt: which exit
shell-prompt: which ls
shell-prompt: which pwd
```

2. Use **ls** to find out what commands are located in `/bin` and `/usr/bin`.

1.10.3 Getting Help

In the dark ages before Unix, when programmers wanted to look up a command or function, they actually had to get out of their chairs and walk somewhere to pick up a typically ring-bound printed manual to flip through.

The Unix designers saw the injustice of this situation and set out to rectify it. They imagined a Utopian world where they could sit in the same chair for ten hours straight without ever taking our eyes off the monitor or their fingers off the keyboard, happily subsisting on coffee and potato chips.

Aside

If there is one trait that best defines an engineer it is the ability to concentrate on one subject to the complete exclusion of everything else in the environment. This sometimes causes engineers to be pronounced dead prematurely. Some funeral homes in high-tech areas have started checking resumes before processing the bodies. Anybody with a degree in electrical engineering or experience in computer programming is propped up in the lounge for a few days just to see if he or she snaps out of it.

-- The Engineer Identification Test (Anonymous)

And so, online documentation was born. On Unix systems, all common Unix commands are documented in detail on the Unix system itself, and the documentation is accessible via the command line (you do not need a GUI to view it, which is important when using a dumb terminal to access a remote system). Whenever you want to know more about a particular Unix command, you can find out by typing **man command-name**. For example, to learn all about the **ls** command, type:

```
shell-prompt: man ls
```

The **man** covers virtually every common command, as well as other topics. It even covers itself:

```
shell-prompt: man man
```

The **man** command displays a nicely formatted document known as a *man page*. It uses a file viewing program called **more**, which can be used to browse through text files very quickly. Table 1.6 shows the most common keystrokes used to navigate a man page. For complete information on navigation, run:

```
shell-prompt: man more
```

Key	Action
h	Show key commands
Space bar	Forward one page
Enter/Return	Forward one line
b	Back one page
/	Search

Table 1.6: Common hot keys in **more**

Man pages include a number of standard sections, such as SYNOPSIS, DESCRIPTION, and SEE ALSO, which helps you identify other commands that might be of use.

Man pages do not always make good tutorials. Sometimes they contain too much detail, and they are often not well-written for novice users. If you're learning a new command for the first time, you might want to consult a Unix book or the WEB. The man pages will provide the most detailed and complete reference information on most commands, however.

The **apropos** command is used to search the man page headings for a given topic. It is equivalent to **man -k**. For example, to find out what man pages exist regarding Fortran, we might try the following:

```
shell-prompt: apropos fortran
```

or

```
shell-prompt: man -k fortran
```

The **whatis** is similar to **apropos** in that it lists short descriptions of commands. However, **whatis** only lists those commands with the search string in their name or short description, whereas **apropos** attempts to list everything related to the string.

The **info** command is an alternative to man that uses a non-graphical hypertext system instead of flat files. This allows the user to navigate extensive documentation more efficiently. The **info** command has a fairly high learning curve, but it is very powerful, and is often the best option for documentation on a given topic. Some open source software ships documentation in info format and provides a man page (converted from the info files) that actually has less information in it.

```
shell-prompt: info gcc
```

Practice Break

1. Find out how to display a '/' after each directory name and a '*' after each executable file when running **ls**.
2. Use **apropos** to find out what Unix commands to use with bzip files.

1.10.4 A Basic Set of Unix Commands

Most Unix commands have short names which are abbreviations or acronyms for what they do. (pwd = print working directory, cd = change directory, ls = list, ...) Unix was originally designed for people with good memories and poor typing skills. Some of the most commonly used Unix commands are described below.

Note This section is meant to serve as a quick reference, and to inform new readers about which commands they should learn. There is much more to know about these commands than we can cover here. For full details about any of the commands described here, consult the **man** pages, **info** pages, or the WEB.

This section uses the same notation conventions as the Unix man pages:

- Optional arguments are shown inside [].
- The 'or' symbol (|) between two items means one or the other.
- An ellipses (...) means optionally more of the same.
- "file" means a filename is required and a directory name is not allowed. "directory" means a directory name is required, and a filename is not allowed. "path" means either a filename or directory name is acceptable.

File and Directory Management

cp copies one or more files.

```
shell-prompt: cp source-file destination-file
shell-prompt: cp source-file [source-file ...] destination-directory
```

If there is only one source filename, then destination can be either a filename or a directory. If there are multiple source files, then destination must be a directory. If destination is a filename, and the file exists, it will be overwritten.

```
shell-prompt: cp file file.bak     # Make a backup copy
shell-prompt: cp file file.bak ~   # Copy files to home directory
```

ls lists files in CWD or a specified file or directory.

```
shell-prompt: ls [path ...]
```

```
shell-prompt: ls           # List CWD
shell-prompt: ls /etc      # List /etc directory
```

mv moves or renames files or directories.

```
shell-prompt: mv source destination
shell-prompt: mv source [source ...] destination-directory
```

If multiple sources are given, destination must be a directory.

```
shell-prompt: mv prog1.c Programs
```

ln link files or directories.

```
shell-prompt: ln source-file destination-file
shell-prompt: ln -s source destination
```

The **ln** command creates another path name for the same file. Both names refer to the same file, and changes made through one appear in the other.

Without `-s`, a standard directory entry, known as a *hard link* is created. A hard link is a directory entry that points to the first block of data in the file. Every file must have at least one hard link to it. If only one path name exists for a file, it is a hard link. For this reason, removing a file is also known as "unlinking". To create a second hard link, the source and destination path names must be in the same file system. File systems under Windows appear as different drive letters, such as C: or D:. Under Unix, all file systems are merged into a single directory tree under /. The **df** will list file systems and their location within the directory tree. There is no harm in trying to create a hard link. If it fails, you can do a soft link instead.

With `-s`, a *symbolic link*, or *soft link* is created. A symbolic link is not a standard directory entry, but a pointer to another path name. It is a directory entry that points to another directory entry rather than the content of the file. Only symbolic links can be used for directories, and symbolic links to not have to be in the same file system as the source.

```
shell-prompt: ln -s /etc/hosts ~    # Make a convenient link to hosts
```

rm removes one or more files.

```
shell-prompt: rm file [file ...]
```

```
shell-prompt: rm temp.txt core a.out
```

Caution Removing files with **rm** is not like dragging them to the trash. Once files are removed by **rm**, they cannot be recovered.

If there are multiple hard links to a file, removing one of them only removes the link, and remaining links are still valid.

Caution Removing the path name to which a symbolic link points will render the symbolic link invalid. It will become a *dangling link*.

srm (secure rm) removes files securely, erasing the file content and directory entry so that the file cannot be recovered. Use this to remove files that contain sensitive data. This is not a standard Unix command, but a free program that can be easily installed on most systems via a package manager.

mkdir creates one or more directories.

```
shell-prompt: mkdir [-p] path name [path name ...]
```

The `-p` flag indicates that mkdir should attempt to create any parent directories in the path that don't already exist. If not used, **mkdir** will fail unless all but the last component of the path already exist.

```
shell-prompt: mkdir Programs
shell-prompt: mkdir -p Programs/C/MPI
```

rmdir removes one or more empty directories.

```
shell-prompt: rmdir directory [directory ...]
```

rmdir will fail if a directory is not completely empty. You may also need to check for hidden files using **ls -a directory**. To remove a directory and everything under it, use **rm -r directory**.

```
shell-prompt: rmdir Programs/C/MPI
```

find locates files within a subtree using a wide variety of possible criteria.

```
shell-prompt: find start-directory criteria [action]
```

find is a very powerful and complex command that can be used to not only find files, but run commands on the files matching the search criteria.

Find can process globbing patterns like the shell, but note that we need to prevent the shell from processing them before running find by enclosing them in quotes.

```
# Find all core files (names end with "core")
shell-prompt: find . -name '*core'

# Remove cores
shell-prompt: find . -name '*core' -exec rm '{}' \;

# Remove multiple cores with each rm command (much faster)
shell-prompt: find . -name '*core' -exec rm '{}' +
```

df shows the free disk space on all currently mounted partitions.

```
shell-prompt: df
```

du reports the disk usage of a directory and everything under it.

```
shell-prompt: du [-s] [-h] path
```

The `-s` (summary) flag suppresses output about each file in the subtree, so that only the total disk usage of the directory is shown. The `-h` asks for human-readable output with gigabytes followed by a G, megabytes by an M, etc.

```
shell-prompt: du -sh Qemu
6.8G    Qemu/
```

Shell Internal Commands

As mentioned previously, internal commands are part of the shell, and serve to control the shell itself. Below are some of the most common internal commands.

cd changes the current working directory of the shell process. It is described in more detail in Section 1.9.2.

```
shell-prompt: cd [directory]
```

pushd changes CWD and saves the old CWD on a stack so that we can easily return.

```
shell-prompt: pushd directory
```

Users often encounter the need to temporarily go to another directory, run a few commands, and then come back to the current directory.

The **pushd** command is a very useful alternative to **cd** that helps in this situation. It performs the same operation as **cd**, but it records the starting CWD by adding it to the top of a stack of CWDs. You can then return to where the last **pushd** command was invoked using **popd**. This saves you from having to retype the path name of the directory to which you want to return. This is like leaving a trail of bread crumbs in the woods to retrace your path back home, except the pushd stack will not get eaten by birds and squirrels, and you won't end up in a witch's soup pot.

Practice Break
Try the following sequence of commands:

```
shell-prompt: pwd          # Check starting point
shell-prompt: pushd /etc
shell-prompt: more hosts
shell-prompt: pushd /home
shell-prompt: ls
shell-prompt: popd         # Back to /etc
shell-prompt: pwd
shell-prompt: more hosts
shell-prompt: popd         # Back to starting point
shell-prompt: pwd
```

exit terminates the shell process.

```
shell-prompt: exit
```

This is the most reliable way to exit a shell. In some situations you could also type **logout** or simply press Ctrl+d, which sends an EOT character (end of transmission, ASCII/ISO character 4) to the shell.

Simple Text File Processing

cat echoes the contents of one or more text files.

```
shell-prompt: cat file [file ...]
```

```
shell-prompt: cat /etc/hosts
```

The **vis** and **cat -v** commands display invisible characters in a visible way. For example, carriage return characters present in Windows files are normally not shown by most Unix commands. The vis and cat -v commands will show them as '^M' (representing Control+M, which is what you would type to produce this character).

```
shell-prompt: cat sample.txt
This line contains a carriage return.
shell-prompt: vis sample.txt
This line contains a carriage return.\^M
shell-prompt: cat -v sample.txt
This line contains a carriage return.^M
```

head shows the top N lines of one or more text files.

```
shell-prompt: head -n # file [file ...]
```

If a flag consisting of a - followed by an integer number N is given, the top N lines are shown instead of the default of 10.

```
shell-prompt: head -n 5 prog1.c
```

The **head** command can also be useful for generating small test inputs. Suppose you're developing a new program or script that processes genomic sequence files in FASTA format. Real FASTA files can contain millions of sequences and take a great deal of time to process. For testing new code, we don't need much data, and we want the test to complete in a few seconds rather than hours. We can use **head** to extract a small number of sequences from a large FASTA file for quick testing. Since FASTA files have alternating header and sequence lines, we must always choose a multiple of 2 lines. We use the output redirection operator (>) to send the head output to a file instead of the terminal screen. Redirection is covered in Section 1.13.

```
shell-prompt: head -n 1000 really-big.fasta > small-test.fasta
```

tail shows the bottom N lines of one or more text files.

```
shell-prompt: tail -n # file [file ...]
```

Tail is especially useful for viewing the end of a large file that would be cumbersome to view with **more**.

If a flag consisting of a - followed by an integer number N is given, the bottom N lines are shown instead of the default of 10.

```
shell-prompt: tail -n 5 output.txt
```

The **diff** command shows the differences between two text files. This is most useful for comparing two versions of the same file to see what has changed. Also see **cdiff**, a specialized version of **diff**, for comparing C source code.

The `-u` flag asks for *unified diff* output, which shows the removed text (text in the first file by not the second) preceded by '-', the added text (text in the second file but not the first) preceded by a '+', and some unchanged lines for context. Most people find this easier to read than the default output format.

```
shell-prompt: diff -u input1.txt input2.txt
```

Text Editors

There are more text editors available for Unix systems than any one person is aware of. Some are terminal-based, some are graphical, and some have both types of interfaces.

All Unix systems support running graphical programs from remote locations, but many graphical programs require a fast connection (100 megabits/sec) or more to function comfortably.

Knowing how to use a terminal-based text editor is therefore a very good idea, so that you're prepared to work on a remote Unix system over a slow connection if necessary. Some of the more common terminal-based editors are described below.

vi (visual editor) is the standard text editor for all Unix systems. Most users either love or hate the vi interface, but it's a good editor to know since it is available on every Unix system.

nano is an extremely simplistic text editor that is ideal for beginners. It is a rewrite of the **pico** editor, which is known to have many bugs and security issues. Neither editor is standard on Unix systems, but both are free and easy to install. These editors entail little or no learning curve, but are not sophisticated enough for extensive programming or scripting.

emacs (Edit MACroS) is a more sophisticated editor used by many programmers. It is known for being hard to learn, but very powerful. It is not standard on most Unix systems, but is free and easy to install.

ape is a menu-driven, user-friendly IDE (integrated development environment), i.e. programmer's editor. It has an interface similar to PC and Mac programs, but works on a standard Unix terminal. It is not standard on most Unix systems, but is free and easy to install. **ape** has a small learning curve, and advanced features to make programming much faster.

Eclipse is a popular open-source graphical IDE written in Java, with support for many languages. It is sluggish over a slow connection, so it may not work well on remote systems over ssh.

Networking

hostname prints the network name of the machine.

```
shell-prompt: hostname
```

This is often useful when you are working on multiple Unix machines at the same time (e.g. via **ssh**), and forgot which window applies to each machine.

ssh is used to remotely log into another machine on the network and start a shell.

```
ssh [name@]hostname
```

```
shell-prompt: ssh joe@unixdev1.ceas.uwm.edu
```

Network commands for transferring files are discussed in Section 1.15.

Identity and Access Management

passwd changes your password. It asks for your old password once, and the new one twice (to ensure that you don't accidentally set your password to something you don't know because your finger slipped). Unlike many graphical password programs, **passwd** does not echo anything for each character typed. Even allowing someone to see the length of your password is a bad idea from a security standpoint.

```
shell-prompt: passwd
```

The **passwd** command is generally only used for setting local passwords on the Unix machine itself. Many Unix systems are configured to authenticate users via a remote service such as *Lightweight Directory Access Protocol* (*LDAP*) or *Active Directory* (*AD*). Changing LDAP or AD passwords may require using a web portal to the LDAP or AD server instead of the **passwd** command.

Terminal Control

clear clears your terminal screen (assuming the TERM environment variable is properly set).

```
shell-prompt: clear
```

reset resets your terminal to its default state. This is useful when your terminal has been corrupted by bad output, such as when attempting to view a binary file with **cat**.

Terminals are controlled by *magic sequences*, sequences of invisible control characters sent from the host computer to the terminal amid the normal output. Magic sequences move the cursor, change the color, change the international character set, etc. Binary files contain random data that sometimes by chance contain magic sequences that could alter the mode of your terminal. If this happens, running **reset** will usually correct the problem. If not, you will need to log out and log back in.

```
shell-prompt: reset
```

Table 1.7 provides a quick reference for looking up common Unix commands. For details on any of these commands, run **man command** (or **info command** on some systems).

Synopsis	Description
ls [fileldirectory]	List file(s)
cp source-file destination-file	Copy a file
cp source-file [source-file ...] directory	Copy multiple files to a directory
mv source-file destination-file	Rename a file
mv source-file [source-file ...] directory	Move multiple files to a directory
ln source-file destination-file	Create another name for the same file. (source and destination must be in the same file system)
ln -s source destination	Create a symbolic link to a file or directory
rm file [file ...]	Remove one or more files
rm -r directory	Recursively remove a directory and all of its contents
srm file [file ...]	Securely erase and remove one or more files
mkdir directory	Create a directory
rmdir directory	Remove a directory (the directory must be empty)
find start-directory criteria	Find files/directories based on flexible criteria
make	Rebuild a file based on one or more other files
od/hexdump	Show the contents of a file in octal/hexadecimal
awk	Process tabular data from a text file
sed	Stream editor. Echo files, making changes to contents.
sort	Sort text files based on flexible criteria
uniq	Echo files, eliminating adjacent duplicate lines.
diff	Show differences between text files.
cmp	Detect differences between binary files.
cdiff	Show differences between C programs.
cut	Extract substrings from text.
m4	Process text files containing m4 mark-up.
chfn	Change finger info (personal identity).
chsh	Change login shell.
su	Substitute user.
cc/clang/gcc/icc	Compile C programs.
f77/f90/gfortran/ifort	Compile Fortran programs.
ar	Create static object libraries.
indent	Beautify C programs.
astyle	Beautify C, C++, C#, and Java programs.
tar	Pack a directory tree into a single file.
gzip	Compress files.
gunzip	Uncompress gzipped files.
bzip2	Compress files better (and slower).
bunzip2	Uncompress bzipped files.
zcat/zmore/zgrep/bzcat/bzmore/bzgrep	Process compressed files.
exec command	Replace shell process with command.
date	Show the current date and time.
cal	Print a calendar for any month of any year.
bc	Unlimited precision calculator.
printenv	Print environment variables.

Table 1.7: Unix Commands

1.10.5 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What types of commands have to be internal to the shell? Give one example and explain why it must be internal.
2. How can you find a list of the basic Unix commands available on your system?
3. How can you find out whether the **grep** command is internal or external, and where it is located?
4. What kind of suffering did computer users have to endure in order to read documentation before the Unix renaissance? How did Unix put an end to such suffering?
5. How can we learn about all the command-line flags available for the **tail** command?
6. How can we copy the file /tmp/sample.txt to the CWD?
7. How can we copy all files whose names begin with "sample" and end with ".txt" to the CWD?
8. How can we move all the files whose names end with ".py" to a subdirectory of the CWD called "Python"?
9. How can we create another filename "./test-input.txt" for the file "./Data/input.txt"?
10. What is a hard link?
11. What is a symbolic link?
12. What do we get when we remove the path name to which a symbolic link points?
13. How do we create a new directory /home/joe/Data/Project1 if the Data directory does not exist and the CWD is /home/joe?
14. How do we remove the directory ./Data if it is empty? If it is not empty?
15. How can we find out how much disk space is available in each file system?

16. How can we find out how much space is used by the Data directory?
17. How can we change CWD to /tmp, then to /etc and then return to the original CWD?
18. How do we exit the shell?
19. How can we see if there are carriage returns in graph.py?
20. How can we see the first 20 lines of output.txt?
21. How can we see the last 20 lines of output.txt?
22. How can we see what has changed between analysis.c.old and analysis.c?
23. Which text editor is available on all Unix systems?
24. How can we find out the name of the machine running our shell?
25. How can user joe log into the remote server unixdev1.ceas.uwm.edu to run commands on it?
26. How do we change our local password on a Unix system?
27. How do we change our password for a Unix system that relies on LDAP or AD?
28. How do we clear the terminal display?
29. How do we reset the terminal mode to defaults?

1.11 POSIX and Extensions

Unix-compatible systems generally conform to standards published by the International Organization for Standardization (ISO), the Open Group, and the IEEE Computer Society.

The primary standard used for this purpose is *POSIX*, the Portable Operating System standard based on UnIx. Programs and commands that conform to the POSIX standard will work on any Unix system. Therefore, developing your programs and scripts according to POSIX will prevent the need for even minor changes when porting from one Unix variant to another.

Nevertheless, many common Unix programs have been enhanced beyond the POSIX standard to provide conveniences. Fortunately, most such programs are open source and can therefore be easily installed on most Unix systems. Features that do not conform to the POSIX standard are known as *extensions*. Extensions are often described according to their source, e.g. BSD extensions that come from BSD Unix variants or GNU extensions that come from the GNU software project.

Many standard commands such as awk, make, and sed, may contain extensions that depend on the specific operating system. For example, BSD systems use the BSD versions of awk, make, and sed, which contain BSD extensions, while GNU/Linux systems use the GNU versions of awk, make, and sed, which contain GNU extensions.

When installing GNU software on BSD systems, the GNU version of the command is usually prefixed with a 'g', to distinguish it from the native BSD command. For example, on FreeBSD, "make" and "awk" are the BSD implementations and "gmake" and "gawk" would be the GNU implementations. Likewise, on GNU/Linux systems, BSD commands would generally be prefixed with a 'b' or 'bsd'. The "make" and "tar" commands on GNU/Linux would refer to GNU versions and the BSD versions would be "bmake" and "bsdtar".

All of them will support POSIX features, so if you use only POSIX features, they will behave the same way. If you use GNU or other extensions, you should use the GNU command, e.g. gawk instead of awk.

Program	**Example of extensions**
BSD Tar	Support for extracting ISO and Apple DMG files
GNU Make	Various "shortcut" rules for compiling multiple source files
GNU Awk	Additional built-in functions

Table 1.8: Common Extensions

1.11.1 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What is POSIX and why is it important?
2. What is an extension?
3. Does the use of extensions always prevent things from working on other Unix systems?

1.12 Subshells

Commands placed between parentheses are executed in a new child shell process rather than the shell process that received the commands as input.

This can be useful if you want a command to run in a different directory or with other alterations to its environment, without affecting the current shell process.

```
shell-prompt: (cd /etc; ls)
```

Since the commands above are executed in a new shell process, the shell process that printed "shell-prompt: " will not have its current working directory changed. This command has the same net effect as the following:

```
shell-prompt: pushd /etc
shell-prompt: ls
shell-prompt: popd
```

1.12.1 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. Show a single Unix command that runs pwd and produces the output "/etc", without changing the CWD of the shell process.

1.13 Redirection and Pipes

1.13.1 Device Independence

Many operating systems that came before Unix treated each input or output device differently. Each time a new device became available, programs would have to be modified in order to access it. This is intuitive, since the devices all look different and perform different functions.

The Unix designers realized that this is actually unnecessary and a waste of programming effort, so they developed the concept of *device independence*. What this means is that *Unix treats virtually every input and output device exactly like an ordinary file.* All input and output, whether to/from a file on a disk, a keyboard, a mouse, a scanner, or a printer, is simply a stream of bytes to be input or output *using the same tools*.

Most I/O devices are actually accessible as a *device file* in `/dev`. For example, the primary CD-ROM might be `/dev/cd0`, the main disk `/dev/ad0`, the keyboard `/dev/kbd0`, and the mouse `/dev/sysmouse`.

A user with sufficient permissions can view input coming from these devices using the same Unix commands we use to view a file:

```
shell-prompt: cat /dev/kbd0
shell-prompt: more /dev/cd0
```

In fact, data are often recovered from corrupted file systems or accidentally deleted files by searching the raw disk partition as a file using standard Unix commands such as grep!

```
shell-prompt: grep string /dev/ad0s1f
```

A keyboard sends text data, so `/dev/kbd0` is like a text file. Many other devices send binary data, so using **cat** to view them would output gibberish. To see the raw input from a mouse as it is being moved, we could instead use **hexdump**, which displays the bytes of input as numbers rather than characters:

```
shell-prompt: hexdump /dev/sysmouse
```

Some years ago while mentoring my son's robotics team, as part of a side project, I reverse-engineered a USB game pad so I could control a Lego robot via Bluetooth from a laptop. Thanks to device-independence, no special software was needed to figure out the game pad's communication protocol.

After plugging the game pad into my FreeBSD laptop, the system creates a new UHID (USB Human Interface Device) under `/dev`. The **dmesg** command shows the name of the new device file:

```
ugen1.2: <vendor 0x046d product 0xc216> at usbus1
uhid0 on uhub3
uhid0: <vendor 0x046d product 0xc216, class 0/0, rev 1.10/3.00, addr 2> on usbus1
```

One can then view the input from the game pad using **hexdump**:

```
FreeBSD manatee.acadix  bacon ~ 410: hexdump /dev/uhid0
0000000 807f 7d80 0008 fc04 807f 7b80 0008 fc04
0000010 807f 7780 0008 fc04 807f 6780 0008 fc04
0000020 807f 5080 0008 fc04 807f 3080 0008 fc04
0000030 807f 0d80 0008 fc04 807f 0080 0008 fc04
0000060 807f 005e 0008 fc04 807f 005d 0008 fc04
0000070 807f 0060 0008 fc04 807f 0063 0008 fc04
0000080 807f 006c 0008 fc04 807f 0075 0008 fc04
0000090 807f 0476 0008 fc04 807f 1978 0008 fc04
00000a0 807f 4078 0008 fc04 807f 8c7f 0008 fc04
00000b0 807f 807f 0008 fc04 807f 7f7f 0008 fc04
00000c0 807f 827f 0008 fc04 807f 847f 0008 fc04
00000d0 807f 897f 0008 fc04 807f 967f 0008 fc04
00000e0 807f a77f 0008 fc04 807f be80 0008 fc04
00000f0 807f d980 0008 fc04 807f f780 0008 fc04
0000100 807f ff80 0008 fc04 807f ff83 0008 fc04
0000110 807f ff8f 0008 fc04 807f ff93 0008 fc04
```

To understand these numbers, we need to know a little about hexadecimal, base 16. This is covered in detail in [?]. In short, it works the same as decimal, but we multiply by powers of 16 rather than 10, and digits go up to 15 rather than 9. Digits for 10 through 15 are A, B, C, D, E, and F. The largest possible 4-digit number is therefore FFFF_16. 8000_16 is in the middle of the range.

```
0000_16 =  0 * 16^3 +  0 * 16^2 +  0 * 16^1 +  0 * 16^0 = 0_10
8000_16 =  8 * 16^3 +  0 * 16^2 +  0 * 16^1 +  0 * 16^0 = 32,678_10
FFFF_16 = 15 * 16^3 + 15 * 16^2 + 15 * 16^1 + 15 * 16^0 = 65,535_10
```

It was easy to see that moving the right joystick up resulted in lower numbers in the 3rd and 7th columns, while moving down increased the values. Center position sends a value around 8000 (hexadecimal), fully up is around 0, fully down is ffff.

It was then easy to write a small program to read the joystick position from the game pad (by simply opening /dev/uhid0 like any other file) and send commands over Bluetooth to the robot, adjusting motor speeds accordingly. The Bluetooth interface is simply treated as an output file.

1.13.2 Redirection

Since I/O devices and files are interchangeable, Unix shells can provide a facility called *redirection* to easily interchange them for *any process* without the process even knowing it.

Redirection depends on the notion of a *file stream*. You can think of a file stream as a hose connecting a program to a particular file or device. Redirection simply disconnects the hose from the default file or device (such as the keyboard or terminal screen) and connects it to another file or device chosen by the user.

Every Unix process has three standard streams that are open from the moment the process is born. The standard streams for a shell process are normally connected to the terminal, as shown in Table 1.9.

Stream	Purpose	Default Connection
Standard Input	User input	Terminal keyboard
Standard Output	Normal output	Terminal screen
Standard Error	Errors and warnings	Terminal screen

Table 1.9: Standard Streams

Redirection in the shell allows any or all of the three standard streams to be disconnected from the terminal and connected to a file or other I/O device. It uses special operator characters within the commands to indicate which stream(s) to redirect and where. The basic redirection operators shells are shown in Table 1.10.

Operator	Shells	Redirection type
<	All	Standard Input
>	All	Standard Output (overwrite)
>>	All	Standard Output (append)
2>	Bourne-based	Standard Error (overwrite)
2>>	Bourne-based	Standard Error (append)
>&	C shell-based	Standard Output and Standard Error (overwrite)
>>&	C shell-based	Standard Output and Standard Error (append)

Table 1.10: Redirection Operators

Note Memory trick: A redirection operator is an arrow that points in the direction of data flow.

```
shell-prompt: ls > listing.txt          # Overwrite with listing of .
shell-prompt: ls /etc >> listing.txt    # Append listing of /etc
```

In the examples above, the **ls** process sends its output to `listing.txt` instead of the standard output. However, the filename `listing.txt` is *not* an argument to the **ls** process. The **ls** process never even knows about this output file.

The redirection is handled by the shell and the shell removes "> listing.txt" and ">> listing.txt" from these commands *before executing them*. So, the first **ls** receives no arguments, and the second receives only `/etc`. Most programs have no idea whether their output is going to a file, a terminal, or some other device. They don't need to know and they don't care.

Caution

Using output redirection (>, 2>, or >&) in a command will normally overwrite (clobber) the file that you're redirecting to, even if the command itself fails. Be very careful not to use output redirection accidentally. This most commonly occurs when a careless user meant to use input redirection, but pressed the wrong key.

The moment you press Enter after typing a command containing "> filename", filename will be erased! Remember that the shell performs redirection, not the command, so filename is clobbered before the command is even executed.

If `noclobber` is set for the shell, output redirection to a file that already exists will result in an error. The `noclobber` option can be overridden by appending a ! to the redirection operator in C shell derivatives or a | in Bourne shell derivatives. For example, >! can be used to force overwriting a file in csh or tcsh, and >| can be used in sh, ksh, or bash.

More often than not, we want to redirect both normal output and error messages to the same place. This is why C shell and its derivatives use a combined operator that redirects both at once.

```
shell-prompt: find /etc -name >& all-output.txt
```

The same effect can be achieved with Bourne-shell derivatives using another operator that redirects one stream to another stream. In particular, we redirect the standard output (stream 1) to a file (or device) and at the same time redirect the standard error (stream 2) to stream 1.

```
shell-prompt: find /etc > all-output.txt 2>&1
```

In Bourne family shells, we can separately redirect the standard output with > and the standard error with 2>:

```
shell-prompt: find /etc > list.txt 2> errors.txt
```

If we want to separate standard output and standard error in a C shell or T shell session, we can use a subshell under which the **find** command redirects only the standard output. The output from the subshell process will then only contain the standard error left over from **find**, which we can redirect with &>:

```
shell-prompt: (find /etc > list.txt) >& errors.txt
```

If a program takes input from the standard input, we can redirect input from a file as follows:

```
shell-prompt: command < input-file
```

For example, consider the "bc" (binary calculator) command is an arbitrary-precision calculator which inputs numerical expressions from the standard input and writes the results to the standard output. It's a good idea to use the `--mathlib` flag with **bc** for more complete functionality.

```
shell-prompt: bc --mathlib
3.14159265359 * 4.2 ^ 2 + sqrt(30)
60.89491440932
quit
```

In the example above, the user entered "3.14159265359 * 4.2 ^ 2 + sqrt(30)" and "quit" and the bc program output "60.89491440932". We could instead place the input shown above in a file using any text editor, such as nano or vi, or even using **cat** with keyboard input and output redirection as a primitive editor:

```
shell-prompt: cat > bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
(Type Ctrl+d to signal the end of input to the cat process)
shell-prompt: cat bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
```

Now that we have the input in a file, we can feed it to the **bc** process using input redirection instead of retyping it on the keyboard:

```
shell-prompt: bc --mathlib < bc-input.txt
60.29203070318
```

1.13.3 Special Files in /dev

The standard streams themselves are represented as device files on Unix systems. This allows us to redirect one stream to another without modifying a program, by appending the stream to one of the device files `/dev/stdout` or `/dev/stderr`. For example, if a program sends output to the standard output and we want to send it instead to the standard error, we could do something like the following:

```
shell-prompt: printf "Oops!" >> /dev/stderr
```

If we would like to simply discard output sent to the standard output or standard error, we can redirect it to `/dev/null`. For example, to see only error messages (standard error) from myprog, we could do the following:

```
shell-prompt: ./myprog > /dev/null
```

To see only normal output and not error messages, assuming Bourne shell family:

```
shell-prompt: ./myprog 2> /dev/null
```

In C shell family:

```
shell-prompt: ( find /etc > output.txt ) >& /dev/null ; cat output.txt
```

The device `/dev/zero` is a readable file that produces a stream of zero bytes.

The device `/dev/random` is a readable file that produces a stream of random integers in binary format. We can use the **dd** command, a bit copy program, to copy a fixed number of bytes from one file to another. We specify the input file with "if=", output with "of=", block size with "bs=", and the number of blocks with "count=". Total data copied will be block-size * count.

```
shell-prompt: dd if=/dev/random of=random-data bs=1000000 count=1
```

1.13.4 Pipes

Very often, we want to use the output of one program as input to another. Such a thing could be done using redirection, as shown below:

```
shell-prompt: ls > listing.txt
shell-prompt: more listing.txt
```

The same task can be accomplished in one command using a *pipe*. A pipe redirects one of the standard streams, just as redirection does, but to or from another process instead of a file or device. In other words, we can use a pipe to send the standard output and/or standard error of one process directly to the standard input of another process.

A pipe is constructed by placing the pipe operator (|) between two commands. The whole chain of commands connected by pipes is called a *pipeline*.

Example 1.2 Simple Pipe

The command below uses a pipe to redirect the standard output of an **ls** process directly to the standard input of a **more** process.

```
shell-prompt: ls | more
```

Since a pipe runs multiple processes in the same shell, it is necessary to understand the concept of *foreground* and *background* processes, which are covered in detail in Section 1.18.

Multiple processes can output to a terminal at the same time, although the results would obviously be chaos in most cases.

In contrast to output, only one process can receiving input from the keyboard, however. It would be a remarkable coincidence if the same input made sense to two different programs.

The *foreground process* running under a given shell process is defined as the process that receives the input from the standard input device (usually the keyboard). This is the only difference between a foreground process and a background process.

When running a pipeline command, the *last* command in the pipeline becomes the foreground process. All others run in the background, i.e. do not use the standard input device inherited from the shell process. Hence, when we run:

```
shell-prompt: ls | more
```

It is the **more** command that receives input from the keyboard. The more command has its standard input redirected from the standard output of **ls**, and the standard input of the **ls** command is effectively disabled.

Note The **more** command is somewhat special: Since its standard input is used to receive input from the pipe, it opens another stream to connect to the keyboard so that it can still get user input, such as pressing the space bar for another screen, etc.

This is such a common practice that Unix has defined the term *filter* to apply to programs that can be used in this way. A filter is any command that can receive input from the standard input and send output to the standard output. Many Unix commands are designed to accept a file name as an argument, but to use the standard input and/or standard output if no filename arguments are provided.

Example 1.3 Filters

The **more** command is commonly used as a filter. It can read a file whose name is provided as an argument, but will use the standard input if no argument is provided. Hence, the following two commands have the same effect:

```
shell-prompt: more names.txt
shell-prompt: more < names.txt
```

The only difference between these two commands is that in the first, the **more** process receives `names.txt` as a command line argument, opens the file itself (creating a new file stream), and reads from the new stream (not the standard input stream). In the second instance, the *shell* process opens `names.txt` and connects the standard input stream of the **more** process to it. The **more** process then uses another stream to read user input from the keyboard.
Using the filtering capability of more, we can paginate the output of any command:

```
shell-prompt: ls | more
shell-prompt: find . -name '*.c' | more
shell-prompt: sort names.txt | more
```

We can string any number of commands together using pipes. The only limitations are imposed by the memory requirements of the processes in the pipeline. For example, the following pipeline sorts the names in names.txt, removes duplicates, filters out all names not beginning with 'B', and shows the first 100 results one page at a time.

```
shell-prompt: sort names.txt | uniq | grep '^B' | head -n 100 | more
```

To see lines 101 through 200 of a file output.txt:

```
shell-prompt: head -n 200 output.txt | tail -n 100
```

One more useful tool worth mentioning is the **tee** command. The **tee** command is a simple program that reads from its standard input and writes to both the standard output and to one or more files whose names are provided on the command line. This allows you to view the output of a program on the screen and save it to a file at the same time.

```
shell-prompt: ls | tee listing.txt
```

Recall that Bourne-shell derivatives do not have combined operators for redirecting standard output and standard error at the same time. Instead, we redirect the standard output to a file or device, and redirect the standard error to the standard output using `2>&1`.

We can use the same technique with a pipe, but there is one more condition: For technical reasons, the `2>&1` must come *before* the pipe.

```
shell-prompt: ls | tee listing.txt 2>&1    # Won't work
shell-prompt: ls 2>&1 | tee listing.txt    # Will work
```

The **yes** command (much like Jim Carrey in "Yes Man") produces a stream of y's followed by newlines. It is meant to be piped into a program that prompts for y's or n's in response to yes/no questions, so that the program will receive a yes answer to all of its prompts and run without user input.

```
shell-prompt: yes | ./myprog
```

The **yes** command can actually print any response we want, via a command line argument. To answer 'n' to every prompt, we could do the following:

```
shell-prompt: yes n | ./myprog
```

In cases where the response isn't always the same, we can feed a program any sequence of responses using redirection or pipes. Be sure to add a newline (\n) after each response to simulate pressing the Enter key:

```
shell-prompt: printf "y\nn\ny\n" | ./myprog
```

Or, to save the responses to a file for repeated use:

```
shell-prompt: printf "y\nn\ny\n" > responses.txt
shell-prompt: ./myprog < responses.txt
```

1.13.5 Misusing Pipes

Aside
It's important to learn from the mistakes of others, because we don't have time to make them all ourselves.

Users who don't fully understand Unix and processes often fall into bad habits that can potentially be costly. There are far too many such habits to cover here: One could write a separate 1,000-page volume called "Favorite Bad Habits of Unix Users". As a less painful alternative, we'll explore one common bad habit in detail and try to help you understand how to spot others. Our feature habit of the day is the use of the **cat** command at the head of a pipeline:

```
shell-prompt: cat names.txt | sort | uniq > outfile
```

So what's the alternative, what's wrong with using **cat** this way, what's the big deal, why do people do it, and how do we know it's a problem?

1. The alternative:

   Most commands used downstream of **cat** in situations like this (e.g. **sort**, **grep**, **more**, etc.) are capable of reading a file directly if given the filename as an argument:

   ```
   shell-prompt: sort names.txt | uniq > outfile
   ```

   Even if they don't take a filename argument, we can always use simple redirection instead of a pipe:

   ```
   shell-prompt: sort < names.txt | uniq > outfile
   ```

2. The problem:

   - Using **cat** this way just adds overhead in exchange for no benefit. Pipes are helpful when you have to perform multiple processing steps in sequence. By running multiple processes at the same time instead of one after the other, we can improve resource utilization. For example, while **sort** is waiting for disk input, **uniq** can use the CPU. Better yet, on a computer with multiple cores, the processes can utilize two cores at the same time.

     However, the **cat** command doesn't do any processing at all. It just reads the file and feeds the bytes into the first pipe.

     In using **cat** this way, here's what happens:

     (a) The **cat** command reads blocks from the file into a file input buffer.

(b) It then copies the input buffer, one byte at a time, to its standard output buffer, without processing the data in any way. It just senselessly moves data (through a proverbial straw) from one memory buffer to another.
(c) When the standard output buffer is full, it is copied to the pipe, which is yet another memory buffer.
(d) Characters in the pipe buffer are copied to the standard input buffer of the next command (e.g. **sort**).
(e) The **sort** can finally begin processing the data.

This is like pouring a drink into a glass, then moving it to a second glass using an eye dropper, then pouring it into a third class and finally a fourth glass before actually drinking it.

It's much simpler and less wasteful for the **sort** command to read directly from the file.

- Using a pipe this way also prevents the downstream command from optimizing disk access. A program such as **sort** might use a larger input buffer size to reduce the number of disk reads. Reading fewer, larger blocks from disk can keep the latency incurred for each disk operation from adding up, thereby reducing run time. This is not possible when reading from a pipe, which is a fixed-size memory buffer.

3. What's the big deal?

   Usually, this is not much of a problem. Wasting a few seconds or minutes on your laptop won't hurt anyone. However, sometimes mistakes like this one are incorporated into HPC cluster jobs using hundreds of cores for weeks at a time. In that case, it could increase run time by several days, delaying the work of other users who have jobs waiting in the queue, as well as your own. Not to mention, the wasted electricity could cost the organization hundreds of dollars and create additional pollution.

4. Why do people do things like this?

   By far the most common response I get when asking people about this sort of thing is: "[Shrug] I copied this from an example on the web. Didn't really think about it."

   Occasionally, someone might think they are being clever by doing this. They believe that this speeds up processing by splitting the task into two processes, hence utilizing multiple cores, one running **cat** to handle the disk input and another dedicated to **sort** or whatever command is downstream. However, this strategy only helps if both processes are *CPU-bound*, i.e. they spend more time using the CPU than performing input and output. This is not the case for the **cat** command.

   One might also think it helps by overlapping disk input and CPU processing, i.e. **cat** can read the next block of data while **sort** is processing the current one. This may have worked a long time ago using slow disks and unsophisticated operating systems, but it only backfires with modern disks and modern Unix systems that have sophisticated disk buffering.

   In reality, this strategy only increases the amount of CPU time used, and almost always increases run time.

5. Detection:

   Detecting performance issues is pretty easy. The most common tool is the **time** command.

```
shell-prompt: time fgrep GGTAGGTGAGGGGCGCCTCTAGATCGGAAGAGCACACGTCTGAACTCCAGTCA test.vcf ↩
     > /dev/null
2.539u 6.348s 0:09.86 89.9% 92+173k 35519+0io 0pf+0w
```

   We have to be careful when using **time** with a pipeline, however. Depending on the shell and the time command used (some shells have in internal implementation), it may not work as expected. We can ensure proper function by wrapping the pipeline in a separate shell process, which is then timed:

```
shell-prompt: time sh -c "cat test.vcf | fgrep  ↩
    GGTAGGTGAGGGGCGCCTCTAGATCGGAAGAGCACACGTCTGAACTCCAGTCA > /dev/null"
2.873u 17.008s 0:13.68 145.2%   33+155k 33317+0io 0pf+0w
```

   Table 1.11 compares the run times (wall time) and CPU time of the direct **fgrep** and piped **fgrep** shown above three different operating systems.

   All runs were performed on otherwise idle system. Several trials were run to ensure reliable results. Times from the first read of `test.vcf` were discarded, since subsequent runs benefit from disk buffering (file contents still in memory from the previous read). The wall time varied significantly on the CentOS system, with the piped command running in less wall time for a small fraction of the trials. The times shown in the table are typical. Times for FreeBSD and MacOS were fairly consistent.

   Note that there is significant variability between platforms which should not be taken too seriously. These tests were not run on identical hardware, so they do not tell us anything about relative operating system performance.

We can also collect other data using tools such as **top** to monitor CPU and memory use and **iostat** to monitor disk activity. These commands are covered in more detail in Section 1.14.15 and Section 1.14.16.

System specs	Pipe wall time	No pipe wall time	Pipe CPU time	No pipe CPU time
CentOS 7 i7 2.8GHz	33.43	29.50	13.59	8.45
FreeBSD Phenom 3.2GHz	13.01	8.90	18.76	8.43
MacBook i5 2.7GHz	81.09	81.35	84.02	81.20

Table 1.11: Run times of pipes with cat

1.13.6 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. How does device independence simplify life for Unix users? Give an example.
2. Show an example Unix command that displays the input from a mouse as it is being moved or clicked.
3. What are the standard streams associated with every Unix process? To what file or device are they connected by default?
4. Show a Unix command that saves the output of **ls -l** to a file called long-list.txt.
5. Show a Unix command that appends the output of **ls -l /etc** to a file called long-list.txt.
6. Show a Unix command that discards the normal output of **ls -l /etc** and shows the error messages on the terminal screen.
7. Show a Bourne shell command that saves the output of **ls -al /etc** to output.txt and any error messages to errors.txt.
8. Show a C shell command that saves the output and errors of **ls -al /etc** to all-output.txt.

9. How does **more list.txt** differ from **more < list.txt**?
10. Show a Unix command that creates a 1 gigabyte file called `new-image` filled with 0 bytes.
11. What are two major advantages of pipes over redirecting to a file and then reading it?
12. Show a Unix command that lists all the files in and under /etc, sorts them, and paginates the output.
13. What is a foreground process?
14. Which program in the following pipeline runs in the foreground?

```
shell-prompt: find /etc | sort | more
```

15. What is a filter program?
16. What is the maximum number of commands allowed in a Unix pipeline?
17. Show a Unix command that prints a long listing of /usr/local/bin to the terminal and at the same time saves it to the file local-bin.txt.
18. Do the same as above, but include any error messages in the file as well. Show the command for both C shell and Bourne shell.
19. Is it a good idea to feed files into a pipe using **cat**, rather than have the next command read them directly? Why or why not?

```
Example: Why is better?
cat file.txt | sort | uniq
sort file.txt | uniq
```

1.14 Power Tools for Data Processing

1.14.1 Introduction

Congratulations on reaching the holy land of Unix data processing. It has often been said that if you know Unix well, you may never need to write a program. The tools provided by Unix often contain all the functionality you need to process your data. They are like a box of Legos from which we can construct a machine to perform almost any data analysis imaginable from the Unix shell.

Most of these tools function as filters, so they can be incorporated into pipelines. Most also accept filenames as command-line arguments for simpler use cases.

In this section, we'll introduce some of the most powerful tools that are heavily used by researchers to process data files. This will certainly reduce, if not eliminate, the need to write your own programs for many projects. This is only an introduction to make you aware of the available tools and the power they can give you.

For more detailed information, consult the man pages and other sources. Some tools, such as **awk** and **sed**, have entire books written about them, in case you want to explore in-depth.

However, do not set out to learn as much as you can about these tools. Set out to learn as juch as you *need*. The ability to show off your vast knowledge is not the ability to achieve. Knowledge is not wisdom. Cleverness is not wisdom. Wisdom is doing. Learn what you need to accomplish today's goals as elegantly as possible, and then do it. You will learn more from this doing than from any amount of studying. You will develop problem solving skills and instincts, which are far more valuable than encyclopedic knowledge.

Never stop wondering if there might be an even more elegant solution. Albert Einstein was once asked what was his goal in life. His response: "To simplify." Use the tools presented here to simplify your research and by extension, your life. With this approach can achieve great things without great effort and spend your time savoring the wonders and mysteries of your work rather than memorizing facts that might come in handy one day.

1.14.2 grep

grep shows lines in one or more text streams that match a given *regular expression* (RE). It is an acronym for Global Regular Expression Print (or Pattern or Parser if you prefer).

```
shell-prompt: grep expression [file ...]
```

The expression is often a simple string, but can represent RE patterns as described in detail by **man re_format** on FreeBSD. There are also numerous web pages describing REs.

Using simple strings or REs, we can search any file stream for lines containing information of interest. By knowing how to construct REs that represent the information you seek, you can easily identify patterns in your data.

REs resemble globbing patterns, but they are not the same. For example, '*' by itself in a globbing pattern means any sequence of 0 or more characters. In an RE, '*' means 0 or more of the preceding character. '*' in globbing is expressed as '.*' in an RE. Some of the most common RE patterns are shown in Table 1.12.

Pattern	Meaning
.	Any single character
*	0 or more of the preceding character
+	1 or more of the preceding character
[]	One character in the set or range of the enclosed characters (same as globbing)
^	Beginning of the line
$	End of the line
.*	0 or more of any character
.+	1 or more of any character
[a-z]*	0 or more lower-case letters

Table 1.12: RE Patterns

The command below shows all lines containing a call to the printf() function in prog1.c.

```
shell-prompt: grep printf prog1.c
```

We might also wish to show all lines containing variable names in prog1.c. Since we are looking for any variable name rather than one particular variable, we cannot use a simple string and must construct a regular expression. Variable names begin with a letter or underscore and may contain any number of letters, underscores, or digits after that. So our RE must require a letter or underscore for the first character and then accept zero or more letters, digits, or underscores after that.

```
shell-prompt: grep '[a-zA-Z_][a-zA-Z0-9_]*' prog1.c
```

The following shows lines that have a '#' in the first column:

```
shell-prompt: grep '^#' prog1.c
```

Note Since REs share many special characters with globbing patterns, we must enclose the RE in quotes to prevent the shell from treating it as a globbing pattern.

Note If we want to literally match a special character such as '.' or '*' literally, we must escape it (preceded it with a '\'). For example, to locate method calls in a Java program, which have the form object.method(arguments);, we could use the following:

```
shell-prompt: grep '[a-zA-Z_][a-zA-Z0-9_]*\.[a-zA-Z]*\(.*\);' prog1.java
```

As an example of searching data files, rather than program code, suppose we would like to find all the lines containing contractions in text file. This would consist of some letters, followed by an apostrophe, followed by more letters. Since the apostrophe is the same character as the single quotes we might use to enclose the RE, we either need to escape it (with a '\') or use double quotes to enclose the RE.

```
shell-prompt: grep '[a-zA-Z]+\'[a-zA-Z]+'
shell-prompt: grep "[a-zA-Z]+'[a-zA-Z]+"
```

Another example would be searching for DNA sequences in a genome. We might use this to locate adapters, artificial sequences added to the ends of DNA fragments for the sequencing process, in our sequence data. Sequences are usually stored one per line in a text file in FASTA format. A common adapter sequence is "CTGTCTCTTATA".

Note We can speed up processing by using **grep --fixed-strings** or **fgrep** instead of a regular grep. This uses a more efficient simple string comparison instead of the more complex regular expression matching.

```
shell-prompt: fgrep CTGTCTCTTATA file.fasta
GCGGCCAACACCTTGCCTGTATTGGCATCCATGATGAAATGGGCGTAACCCTGTCTCTTATACACATCTCCGAG
AAAGGCCTGTATGATAAGTTGGCAAATTTCCTCAAGATTGTTTACTTGATACACCTGTCTCTTATACACATCTC
GACCGAGGCACTCGCCGCGCTTGAGCTCGAGATCGATGCCGTCGACCTGTCTCTTATACACATCTCCGAGCCCA
AAAAAATCCCTCCGAAGCATTGTAGGTTTCCATGCTGTCTCTTATACACATCTCCGAGCCCACGAGACTCCTGA
```

It's hard to see the pattern we were looking for in this output. To solve this problem, we can colorize any matched patterns using the `--color` flag as shown in Figure 1.3.

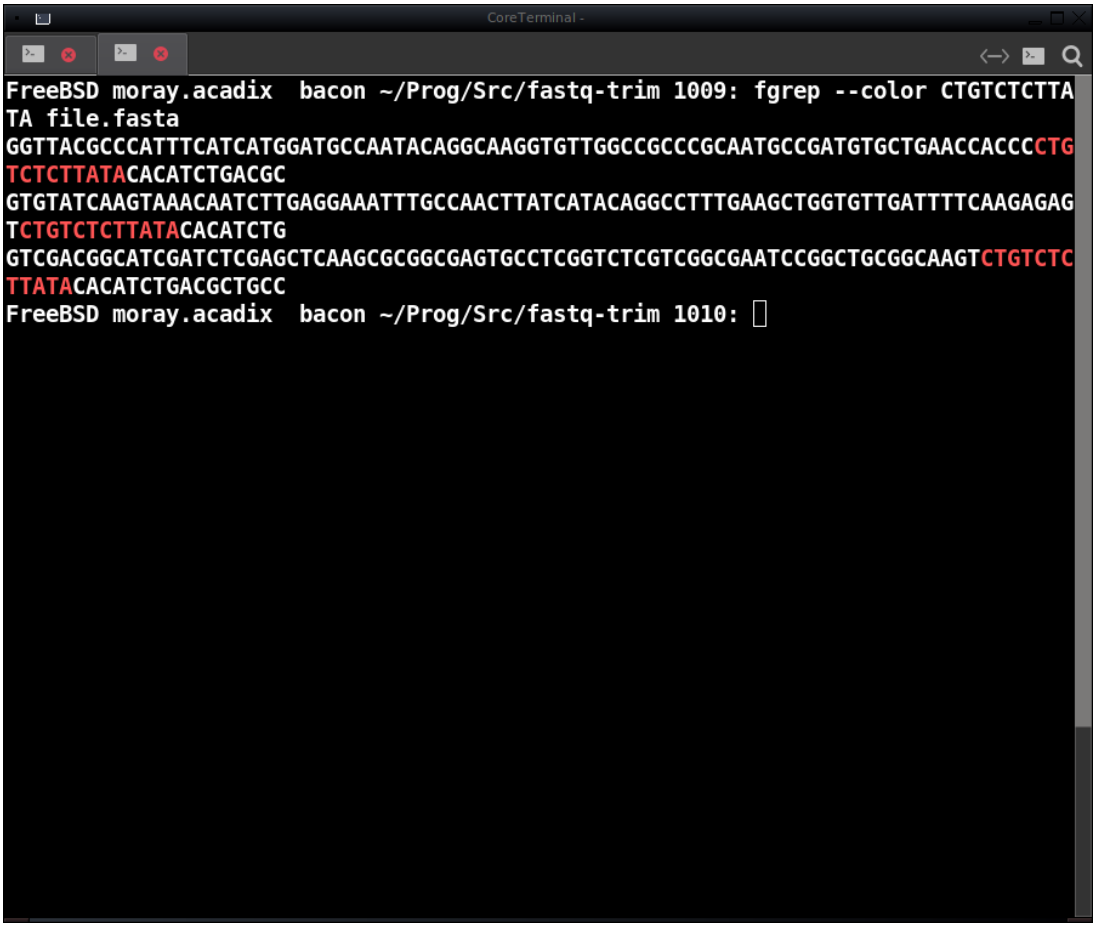


Figure 1.3: Colorized grep output

There is an extended version of regular expressions that is not supported by the normal **grep** command. Extended REs include things like alternative strings, which are separated by a '|'. For example, we might want to search for either of two adapter sequences. To enable extended REs, we use **egrep** or **grep --extended-regexp**.

```
shell-prompt: egrep 'CTGTCTCTTATA|AGATCGGAAGAG' file.fasta
```

1.14.3 awk

AWK, an acronym for Aho, Weinberger, and Kernighan (the original developers of the program), is an extremely powerful tool for processing tabular data. Like **grep**, it supports RE matching, but unlike **grep**, it can process individual columns, called *fields*, in the data. It also includes a flexible scripting language that closely resembles the C language, so we can perform highly sophisticated processing of whole lines or individual fields.

Awk can be used to automate many of the same tasks that researchers often perform manually in a spreadsheet program such as LibreOffice Calc or MS Excel.

There are multiple implementations of awk. The most common are "The one true awk", evolved from the original awk code and used on many BSD systems. Gawk, the GNU project implementation, is used on most Linux systems. Mawk is an independent implementation that tends to outperform the others. It is available in most package managers. Awka is an awk-to-C translator that can convert most awk scripts to C for maximize performance.

Fields by default are separated by white space, i.e. space or tab characters. However, **awk** allows us to specify any set of separators using an RE following the `-F` flag or embedded in the script, so we can process tab-separated (.tsv) files, comma-separated (.csv) files, or any other data that can be broken down into columns.

An awk script consists of one or more lines containing a *pattern* and an *action*. The action is enclosed in curly braces, like a C code block.

```
pattern { action }
```

The pattern is used to select lines from the input, usually using a relational expression such as those found in an **if** statement. The action determines what to do when a line is selected. If no pattern is given, the action is applied to every line of input. If no action is given, the default is to print the line.

In both the pattern and the action, we can refer to the entire line as $0. $1 is the first field: all text up to but not including the first separator. $2 is the second field: all text between the first and second separators. And so on...

It is very common to use awk "one-liners" on the command-line, without actually creating an awk script file. In this case, the awk script is the first argument to **awk**, usually enclosed in quotes to allow for white space and special characters. The second argument is the input file to be processed by the script.

For example, the file `/etc/passwd` contains colon-separated fields including the username ($1), user ID ($3), primary group ID ($4), full name ($5), home directory ($6), and the user's shell program ($7). To see a list of full names for every line, we could use the following simple command, which has no pattern (so it processes every line) and an action of printing the fifth field:

```
shell-prompt: awk -F : '{ print $5 }' /etc/passwd
Jason Bacon
D-BUS Daemon User
TCG Software Stack user
Avahi Daemon User
...
```

To see a list of usernames and shells:

```
shell-prompt: awk -F : '{ print $1, $6 }' /etc/passwd
bacon /bin/tcsh
messagebus /usr/sbin/nologin
_tss /usr/sbin/nologin
avahi /usr/sbin/nologin
...
```

Many data files used in research computing are tabular, with one of the most popular formats being TSV (tab-separated value) files. The *General Feature Format*, or *GFF* file is a TSV file format for describing features of a genome. The first field contains the sequence ID (such as a chromosome number) on which the feature resides. The third field contains the feature type, such as "gene" or "exon". The fourth and fifth fields contain the starting and ending positions withing the sequence. The ninth field contains "attributes", such as the globally unique feature ID and possibly the feature name and other information, separated by semicolons. If we just want to see the locations and attributes of all the genes in a genome and their names, we could use the following:

```
shell-prompt:  awk '$3 == "gene" { print $1, $4, $5, $9 }' file.gff3
1 3073253 3074322 ID=gene:ENSMUSG00000102693;Name=4933401J01Rik
1 3205901 3671498 ID=gene:ENSMUSG00000051951;Name=Xkr4
...
```

Suppose we want to extract specific attributes from the semicolon-separated attributes field, such as the gene ID and gene name, as well as count the number of genes in the input. This will require a few more awk features.

The gene ID is always the first attribute in the field, assuming the feature is a gene. Not every gene has a name, so we will need to scan the attributes for this information. Awk makes this easy. We can break the attributes field into an array of strings using the `split()` function. We can then use a loop to search the attributes for one beginning with "Name=".

To count the genes in the input, we need to initialize a count variable before we begin processing the file, increment it for each gene found, and print it after processing is finished. For this we can use the special patterns BEGIN and END, which allow us to run an action before and after processing the input.

We will use the C-like `printf()` function to format the output. The basic `print` statement always adds a newline, so it does not allow us to print part of a line and finish it with an subsequent print statement.

Since this is a multiline script, we will save it in a file called `gene-info.awk` and run it using the `-f` flag, which tells awk to get the script from a file rather than the command-line.

```
shell-prompt: awk -f gene-info.awk file.gff3
```

Caution Awk can be finicky about the placement of curly braces. To avoid problems, always place the opening brace ({) for an action on the same line as the pattern.

```
BEGIN {
    gene_count = 0;
}

$3 == "gene" {
    # Separate attributes into an array
    split($9, attributes, ";");

    # Print location and feature ID
    printf("%s %s %s %s", $1, $4, $5, attributes[1]);

    # Look for a name attribute and print it if it exists
    # With the for-in loop, c gets the SUBSCRIPT of each element in the
    # attributes array
    for ( c in attributes )
    {
        # See if first 5 characters of the attribute are "Name="
        if ( substr(attributes[c], 1, 5) == "Name=" )
            printf(" %s", attributes[c]);
    }

    # Terminate the output line
```

```
    printf("\n");

    # Count this gene
    ++gene_count;
}

END {
    printf("\nGenes found = %d\n", gene_count);
}
```

As we can see, we can do some fairly sophisticated data processing with a very short **awk** script. There is very little that awk cannot do conveniently with tabular data. If a particular task seems like it will be difficult to do with awk, don't give up too easily. Chances are, with a little thought and effort, you can come up with an elegant awk script to get the job done.

That said, there are always other options for processing tabular data. Perl is a scripting language especially well suited to text processing, with its powerful RE handling capabilities and numerous features. Python has also become popular for such tasks in recent years.

Awk is highly efficient, and processing steps performed with it are rarely a bottleneck in an analysis pipeline. If you do need better performance than awk provides, there are C libraries that can be used to easily parse tabular data, such as libxtend. Libxtend includes a set of DSV (delimiter-separated-value) processing functions that make it easy to read fields from files in formats like TSV, CSV, etc. Once you have read a line or an individual field using libxtend's DSV functions, you now have the full power and performance of C at your disposal to process it in minimal time.

Full coverage of awk's capabilities is far beyond the scope of this text. Readers are encouraged to explore it further via the awk man page and one of the many books available on the language.

1.14.4 cut

The **cut** command is used to select columns from a file, either by byte position, character position, or like **awk**, delimiter-separated columns. Note that characters in the modern world may be more than one byte, so bytes and characters are distinguished here.

To extract columns by byte or character position, we use the `-b` or `-c` option followed by a list of positions. The list is comma-separated and may contain individual positions or ranges denoted with a '-'. For example, to extract character positions 1 through 10 and 21 through 26 from every line of file.txt, we could use the following:

```
shell-prompt: cut -c 1-10,21-26 file.txt
```

For delimiter-separated columns, we use `-d` to indicate the delimiter. The default is a tab character alone, not just any white space. The `-w` flag tells cut to accept any white space (tab or space) as the delimiter. The `-f` is then used to indicate the fields to extract, much like `-c` is used for character positions. Output is separated by the same delimiter as the input.

For example, to extract the username, userid, groupid, and full name (fields 1, 3, 4, and 5) from /etc/passwd, we could use the following:

```
shell-prompt: cut -d : -f 1,3-5 /etc/passwd
...
ganglia:102:102:Ganglia User
nagios:181:181:Nagios pseudo-user
webcamd:145:145:Webcamd user
```

The above is equivalent to the following awk command:

```
shell-prompt: awk -F : '{ printf("%s:%s:%s:%s\n", $1, $3, $4, $5); }' /etc/passwd
```

1.14.5 sed

The **sed** command is a stream editor. It makes changes to a file stream with no interaction from the user. It is probably most often used to make simple text substitutions, though it can also do insertions and deletions of lines and parts of lines, even selecting lines by number or based on pattern matching much like **grep** and **awk**. A basic substitution command takes the following format:

```
sed -e 's|pattern|replacement|g' input-file
```

Pattern is any regular expression, like those used in **grep** or **awk**. Replacement can be a fixed string, but also takes some special characters, such as &, which represents the string matched by pattern. It can also be empty if you simply want to remove occurrences of pattern from the text.

The characters enclosing pattern and replacement are arbitrary. The 'l' character is often used because it stands out among most other characters. If either pattern or replacement contains a 'l', simply use a different separator, such as '/'. The 'g' after the pattern means "global". Without it, **sed** will only replace the first occurrence of pattern in each line. With it, all matches are replaced.

```
shell-prompt: cat fox.txt
The quick brown fox jumped over the lazy dog.
shell-prompt: sed -e 's|fox|worm|g' fox.txt
The quick brown worm jumped over the lazy dog.
shell-prompt: sed -e 's/brown //g' -e 's|fox|&y worm|g' fox.txt
The quick foxy worm jumped over the lazy dog.
```

Using `-E` in place of `-e` causes **sed** to support extended regular expressions.

By default, sed sends output to the standard output stream. The `-i` flag tells **sed** to edit the file in-place, i.e. replace the original file with the edited text. This flag should be followed by a filename extension, such as ".bak". The original file will then be saved to filename.bak, so that you can reverse the changes if you make a mistake. The extension can be an empty string, e.g. '' if you are sure you don't need a backup of the original.

Caution
There is a rare portability issue with **sed**. GNU **sed** requires that the extension be nestled against the `-i`:

```
shell-prompt: sed -i.bak -e 's|pattern|replacement|g' file.txt
```

Some other implementations require a space between the `-i` and the extension, which is more orthodox among Unix commands:

```
shell-prompt: sed -i .bak -e 's|pattern|replacement|g' file.txt
```

FreeBSD's **sed** accepts either form. You must be aware of this in order to ensure that scripts using **sed** are portable. The safest approach is not to use the `-i` flag, but simply save the output to a temporary file and then move it:

```
shell-prompt: sed -e 's|pattern|replacement|g' file.txt > file.txt.tmp
shell-prompt: mv file.txt.tmp file.txt
```

This way, it won't matter which implementation of **sed** is present when someone runs your script.

Sed is a powerful and complex tool that is beyond the scope of this text. Readers are encouraged to consult books and other documentation to explore further.

1.14.6 sort

The **sort** command sorts text data line by line according to one or more *keys*. A key indicates a *field* (usually a column separated by white space or some other delimiter) and the type of comparison, such as lexical (like alphabetical, but including non-letters) or numeric.

If no keys are specified, sort compares entire lines lexically. The `--key` followed by a field number restricts comparison to that field. Fields are numbered starting with 1. This can be used in conjunction with the `--field-separator` flag to specify a separator other than the default white space. The `--numeric-sort` flag must be used to perform integer comparison rather than lexical. The `--general-numeric-sort` flag must be used to compare real numbers.

```
Shell-prompt: cat ages.txt
Bob Vila        23
Joe Piscopo     27
Al Gore         19
Ingrid Bergman  26
Mohammad Ali    22
Ram Das         9
Joe Montana     25

Shell-prompt: sort ages.txt
Al Gore         19
Bob Vila        23
Ingrid Bergman  26
Joe Montana     25
Joe Piscopo     27
Mohammad Ali    22
Ram Das         9

Shell-prompt: sort --key 2 ages.txt
Mohammad Ali    22
Ingrid Bergman  26
Ram Das         9
Al Gore         19
Joe Montana     25
Joe Piscopo     27
Bob Vila        23

Shell-prompt: sort --key 3 --numeric-sort ages.txt
Ram Das         9
Al Gore         19
Mohammad Ali    22
Bob Vila        23
Joe Montana     25
Ingrid Bergman  26
Joe Piscopo     27
```

The **sort** command can process files of any size, regardless of available memory. If a file is too large to fit in memory, it is broken into smaller pieces, which are sorted separately and saved to temporary files. The sorted temporary files are then merged.

The **uniq** command, which removes adjacent lines that are identical, is often used after sorting to remove redundancy from data. Note that the **sort** command also has a `--unique` flag, but it does not behave the same as the **uniq** command. The `--unique` flag compares keys, while the **uniq** command compares entire lines.

1.14.7 tr

The **tr** (translate) command is a simple tool for performing character conversions and deletions in a text stream. A few examples are shown below. See the **tr** man page for details.

We can use it to convert individual characters in a text stream. In this case, it takes two string arguments. Characters in the Nth position in the first string are replaced by characters in the Nth position in the second string:

```
shell-prompt: cat fox.txt
The quick brown fox   jumped over the lazy dog.
shell-prompt: tr 'xl' 'gh' < fox.txt
The quick brown fog   jumped over the hazy dog.
```

There is limited support for character sets enclosed in square brackets [], similar to regular expressions, including predefined sets such as [:lower:] and [:upper:]:

```
shell-prompt: tr '[:lower:]' '[:upper:]' < fox.txt
THE QUICK BROWN FOX   JUMPED OVER THE LAZY DOG.
```

We can use it to "squeeze" repeated characters down to one in a text stream. This is useful for compressing white space:

```
shell-prompt: tr -s ' ' < fox.txt
The quick brown fox jumped over the lazy dog.
```

The **tr** command does not support doing multiple conversions in the same command, but we can use it as a filter:

```
shell-prompt: tr '[:lower:]' '[:upper:]' < fox.txt | tr -s ' '
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
```

There is some overlap between the capabilities of **tr**, **sed**, **awk**, and other tools. Which one you choose for a given task is a matter of convenience.

1.14.8 find

The **find** command is a powerful tool for not only locating path names in a directory tree, but also for taking any desired action when a path name is found.

Unlike popular search utilities in macOS, Windows, and the Unix **locate** command. **find** does not use a previously constructed index of the file system, but searches the file system in its current state. Indexed search utilities very quickly produce results from a recent snapshot of the filesystem, which is rebuilt periodically by a scheduled job. This is much faster than an exhaustive search, but will miss files that were added since the last index build. The **find** command will take longer to search a large directory tree, but also guarantees accurate results.

The basic format of a find command is as follows:

```
shell-prompt: find top-directory search-criteria [optional-action \;]
```

The search-criteria can be any attribute of a file or other path name. To match by name, we use `-name` followed by a globbing pattern, in quotes to prevent the shell from expanding it before passing it to **find**. To search for files owned by a particular user or group, we can use `-user` or `-group`. We can also search for files with certain permissions, a minimum or maximum age, and many other criteria. The man page provides all of these details.

The default action is to print the relative path name of each match. For example, to list all the configuration files under `/etc`, we could use the following:

```
shell-prompt: find /etc -name '*.conf'
```

We can run any Unix command in response to each match using the `-exec` flag followed the command and a ';' or '+'. The ';' must be escaped or quoted to prevent the shell from using it as a command separator and treating everything after it as a new command, separate from the **find** command. The name of the matched path is represented by '{}'.

```
shell-prompt: find /etc -name '*.conf' -exec ls -l '{}' \;
```

With a ';' terminating the command, the command is executed immediately after each match. This may be necessary in some situations, but it entails a great deal of overhead from running the same command many times. Replacing the ';' with a '+' tells **find** to accumulate as many path names as possible and pass them all to one invocation of the command. This means the command could receive thousands of path names as arguments and will be executed far fewer times.

```
shell-prompt: find /etc -name '*.conf' -exec ls -l '{}' +
```

There are also some predefined actions we can use instead of spelling out a `-exec`, such as `-print`, which is the default action, and `-ls`, which is equivalent to `-exec ls -l '{}' +`. The `-print` action is useful for showing path names being processed by another action:

```
shell-prompt: find Data -name '*.bak' -print -exec rm '{}' +
```

Sometimes we may want to execute more than one command for each path matched. Rather than construct a complex and messy `-exec`, we may prefer to write a shell script containing the commands and run the script using `-exec`. Scripting is covered in Chapter 2.

1.14.9 xargs

As stated earlier, most Unix commands that accept a file name as an argument will accept any number of file names. When processing 100 files with the same program, it is usually more efficient to run one process with 100 file name arguments than to run 100 processes with one argument each.

However, there is a limit to how long Unix commands can be. When processing many thousands of files, it may not be possible to run a single command with all of the filenames as arguments. The **xargs** command solves this problem by reading a list of file names from the standard input (which has no limit) and feeding them to another command asarguments, providing as many arguments as possible to each process created.

The arguments processed by **xargs** do not have to be file names, but usually are. The main trick generating the list of files. Suppose we want to change all occurrences of "fox" to "toad" in the files input*.txt in the CWD. Our first thought might be a simple command:

```
shell-prompt: sed -i '' -e 's|fox|toad|g' input*.txt
```

If there are too many files matching "input*.txt", we will get an error such as "Argument list too long". One might think to solve this problem using **xargs** as follows:

```
shell-prompt: ls *.txt | xargs sed -i '' -e 's|fox|toad|g'
```

However, this won't work either, because the shell hits the same argument list limit for the **ls** command as it does for the **sed** command.

The **find** command can come to the rescue:

```
shell-prompt: find . -name '*.txt' | xargs sed -i '' -e 's|fox|toad|g'
```

Since the shell is not trying to expand '*.txt' to an argument list, but instead passing the literal string '*.txt' to **find**, there is no limit on how many file names it can match. The **find** command is sophisticated enough to work around the limits of argument lists.

The **find** command above will send relative path names of every file with a name matching 'input*.txt' in *and under* the CWD. If we don't want to process files in subdirectories of CWD, we can limit the depth of the find command to one directory level:

```
shell-prompt: find . -maxdepth 1 -name '*.txt' \
              | xargs sed -i '' -e 's|fox|toad|g'
```

Note

The **xargs** command places the arguments read from the standard input *after* any arguments included with the command. So the commands run by xargs will have the form

```
sed -i '' -e 's|fox|toad|g' input1.txt input2.txt input3.txt ...
```

Some **xargs** implementations have an option for placing the arguments from the standard input *before* the fixed arguments, but this is still limited. There may be cases where we want the arguments intermingled. The most portable and flexible solution to this is writing a simple script that takes all the arguments from **xargs** last, and constructs the appropriate command with the arguments in the correct order. Scripting is covered in Chapter 2.

Most xargs implementations also support running multiple processes at the same time. This provides a convenient way to utilize multiple cores to parallelize processing. If you have a computer with 16 cores and speeding up your analysis by a factor of nearly 16 is good enough, then this can be a very valuable alternative to using an HPC cluster. If you need access to hundreds of cores to get your work done in a reasonable time, then a cluster is a better option.

```
shell-prompt: find . -name '*.txt' \
              | xargs --max-procs 8 sed -i '' -e 's|fox|toad|g'
```

A value of 0 following --max-procs tells xargs to detect the number of available cores and use all of them.

There is a more sophisticated FOSS program called GNU parallel that can run commands in parallel in a similar way, but with more flexibility. It can be installed via most package managers.

1.14.10 bc

The **bc** (binary calculator) command is an unlimited range and precision calculator with a scripting language very similar to C. When invoked with `-l` or `--mathlib`, it includes numerous additional functions including l(x) (natural log), e(x) (exponential), s(x) (sine), c(x) (cosine), and a(x) (arctangent). There are numerous standard functions available even without `--mathlib`. See the man page for a full list.

By default, **bc** prints the result of each expression evaluated followed by a newline. There is also a **print** statement that does not print a newline. This allows a line of output to be constructed from multiple expressions, the last of which includes a literal "\n".

```
shell-prompt: bc --mathlib
sqrt(2)
1.41421356237309504880
print sqrt(2), "\n"
1.41421356237309504880
l(10)
2.30258509299404568401
x=10
5 * x^2 + 2 * x + 1
521
quit
```

Bc is especially useful for quick computations where extreme range or precision is required, and for checking the results from more traditional languages that lack such range and precision. For example, consider the computation of factorials. N factorial, denoted N!, is the product of all integers from one to N. The factorial function grows so quickly that 21! exceeds the range of a 64-bit unsigned integer, the largest integer value supported by most CPUS and most common languages. The C program and output below demonstrate the limitations of 64-bit integers.

```c
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    unsigned long   c, fact = 1;

    for (c = 1; c <= 22; ++c)
    {
        fact *= c;
        printf("%lu! = %lu\n", c, fact);
    }
    return EX_OK;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
```

```
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768      This does not equal 20! * 21
22! = 17196083355034583040
23! = 8128291617894825984
24! = 10611558092380307456
25! = 7034535277573963776
```

At 21!, an integer overflow occurs. In the limited integer systems used by computers, adding 1 to the largest possible value produces a result of 0. The system is actually circular, which is why 21! above is wrong and 23! is actually smaller than 22!. The limitations of computer number systems are covered in [?].

In constrast, **bc** can compute factorials of any size, limited only by the amount of memory needed to store the value. It is, of course, much slower than C, both because it is an interpreted language and because it performs multiple precision arithmetic, which requires multiple machine instructions for every math operation. However, it is more than fast enough for many purposes and the easiest way to do math that is beyond the capabilities of common languages.

The **bc** script below demonstrates the the superior range of **bc**. The first line (#!/usr/bin/bc -l) tells the Unix shell how to run the script, so we can run it by simply typing its name, such as **./fact.bc**. This will be covered in Chapter 2. For now, create the script using **nano fact.bc** and run it with **bc < fact.bc**.

```
#!/usr/bin/bc -l

fact = 1;
for (c = 1; c <= 100; ++c)
{
    fact *= c;
    print c, "!= ", fact, "\n";
}
quit
```

```
1!= 1
2!= 2
3!= 6
4!= 24
5!= 120
6!= 720
7!= 5040
8!= 40320
9!= 362880
10!= 3628800
11!= 39916800
12!= 479001600
13!= 6227020800
14!= 87178291200
15!= 1307674368000
16!= 20922789888000
17!= 355687428096000
18!= 6402373705728000
19!= 121645100408832000
20!= 2432902008176640000
21!= 51090942171709440000
22!= 1124000727777607680000
23!= 25852016738884976640000
24!= 620448401733239439360000
25!= 15511210043330985984000000

[ Output removed for brevity ]

100!= 93326215443944152681699238856266700490715968264381621468592963\
895217599993229915608941463976156518286253697920827223758251185210910\
```

```
68640000000000000000000000000
```

Someone with a little knowledge of computer number systems might think that we can get around the range problem in general purpose languages like C by using floating point rather than integers. This will not work, however. While a 64-bit floating point number has a much greater range than a 64-bit integer (up to 10^{308} vs 10^{19} for integers), floating point actually has less precision. It sacrifices some precision in order to achieve the greater range. The modified C code and output below show that the double (64-bit floating point) type in C only gets us to 22!, and round-off error corrupts 23! and beyond.

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  c, fact = 1;

    for (c = 1; c <= 25; ++c)
    {
        fact *= c;
        printf("%0.0f! = %0.0f\n", c, fact);
    }
    return EX_OK;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884978212864
24! = 620448401733239409999872
25! = 15511210043330986055303168
```

1.14.11 tar

TBD

1.14.12 gzip, bzip2, xz

TBD xz can be a bottleneck in pipelines when used with default options. Try lowering the compression until it is able to keep up with other processing steps (-4, -3, -2). It will still likely provide better compression than gzip.

1.14.13 zip, unzip

TBD

1.14.14 time

TBD

1.14.15 top

TBD

1.14.16 iostat

TBD

1.14.17 netstat

TBD

1.14.18 iftop

TBD

1.14.19 curl, fetch, wget

TBD

1.14.20 Practice

Instructions

1. Make sure you are using the latest version of this document.
2. Carefully read one section of this document and casually read other material (such as corresponding sections in a textbook, if one exists) if needed.
3. Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
4. Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and demonstrates a lack of interest in learning.
5. Check the answer key to make sure your answer is correct and complete.

   DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE VERY BEST OF YOUR ABILITY. In doing so, you would only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.

Important notes:

- Show all your work. This will improve your understanding and ensure full credit for the homework.
- The practice problems are designed to make you think about the topic, starting from basic concepts and progressing through real problem solving.
- Try to verify your own results. In the working world, no one will be checking your work. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.

---

1. What is a regular expression? Is it the same as a globbing pattern?
2. How can we show lines in analysis.c containing hard-coded floating point constants?
3. How can we speed up grep searches when searching for a fixed string rather than an RE pattern?
4. How can we use extended REs with grep?
5. How can we make the matched pattern visible in the grep output?
6. Describe two major differences between grep and awk.
7. How does awk compare to spreadsheet programs like LibreOffice Calc and MS Excel?
8. The /etc/group file contains colon-separated lines in the form groupname:password:groupid:members. Show an awk command that will print the groupid and members of the group "root".
9. A GFF3 file contains tab-separated lines in the form "seqid source feature-type start end score strand phase attributes". The first attribute for an exon feature is the parent sequence ID. Write an awk script that reports the seqid, start, end, strand, and parent for each feature of type "exon". It should also report the number of exons and the number of genes. To test your script, download Mus_musculus.GRCm39.107.chromosome.1.gff3.gz from ensembl.org and then do the following:

   ```
   gunzip Mus_musculus.GRCm39.107.chromosome.1.gff3.gz
   awk -f your-script.awk Mus_musculus.GRCm39.107.chromosome.1.gff3
   ```

10. Show a cut command roughly equivalent to the following awk command, which processes a tab-separated GFF3 file.

    ```
    awk '{ print $1, $3, $4, $5 }' file.gff3
    ```

11. Show a sed command that replaces all occurrences of "wolf" with "werewolf" in the file halloween-list.txt.
12. Show a command to sort the following data by height. Show a separate command to sort by weight. The data are in params.txt.

```
ID  Height  Weight
1   34      10
2   40      14
3   29      9
4   28      11
```

13. Show a Unix command that replaces the word "fox" with "toad" and converts all lower case letters to upper case in the file fox.txt. Output should be stored in big-toad.txt.
14. Show a Unix command that lists and removes all the files whose names end in '.o' in and under ~/Programs.
15. Why is the xargs command necessary?
16. Show a Unix command that removes all the files with names ending in ".tmp" only in the CWD, assuming that there are too many of them to provide as arguments to one command. The user should not be prompted for each delete. (Check the **rm** man page if needed.)
17. Show a Unix command that processes all the files named 'input*' in the CWD, using as many cores as possible, through a command such as the following:

```
analyze --limit 5 input1 input2
```

18. What is the most portable and flexible way to use xargs when the arguments it provides to the command must precede some of the fixed arguments?
19. What is the major advantage of the **bc** calculator over common programming languages?
20. Show a bc expression that prints the value of the natural number, e.
21. Write a **bc** script that prints the following. Create the script with **nano sqrt.bc** and run it with **bc < sqrt.bc**.

```
sqrt(1) = 1.00000000000000000000
sqrt(2) = 1.41421356237309504880
sqrt(3) = 1.73205080756887729352
sqrt(4) = 2.00000000000000000000
sqrt(5) = 2.23606797749978969640
sqrt(6) = 2.44948974278317809819
sqrt(7) = 2.64575131106459059050
sqrt(8) = 2.82842712474619009760
sqrt(9) = 3.00000000000000000000
sqrt(10) = 3.16227766016837933199
```

1.15 File Transfer

Many users will need to transfer data between other computers and a remote Unix system. For example, users of a shared research computer running Unix will need to transfer input data from their computer to the Unix machine, run the research programs, and finally transfer results back to their computer. There are many software tools available to accomplish this. Some of the more convenient tools are described below.

1.15.1 File Transfers from Unix

sftp (Secure File Transfer Protocol) is often used to remotely log into another machine on the network and transfer files to or from it. Not all remote Unix systems have sftp enabled.

```
shell-prompt: sftp [name@]host
```

```
shell-prompt: sftp joe@unixdev1.ceas.uwm.edu
```

For Unix (including Mac and Cygwin) users, the recommended method for transferring files is the **rsync** command. The rsync command is a simple but intelligent tool that makes it easy to synchronize two directories on the same machine or on different machines across a network. Rsync is free software and part of the base installation of many Unix systems including Mac OS X. On Cygwin, you can easily add the rsync package using the Cygwin Setup utility.

Rsync has two major advantages over other file transfer programs:

- If you have transferred the directory before, and only want to update it, rsync will automatically determine the differences between the two copies and only transfer what is necessary. When conducting research that generates large amounts of data, this can save an enormous amount of time.
- If a transfer fails for any reason (which is more likely for large transfers), rsync's inherent ability to determine the differences between two copies allows it to resume from where it left off. Simply run the exact same rsync command again, and the transfer will resume.

The rsync command can either push (send) files from the local machine to a remote machine, or pull (retrieve) files from a remote machine to the local machine. The command syntax is basically the same in both cases. It's just a matter of how you specify the source and destination for the transfer.

The rsync command has many options, but the most typical usage is to create an exact copy of a directory on a remote system. The general rsync command to push files to another host would be:

```
shell-prompt: rsync -av --delete source-path [username@]hostname:[destination-path]
```

Example 1.4 Pushing data with rsync

The following command synchronizes the directory `Project` from the local machine to ~joeuser/Data/Project on Peregrine:

```
shell-prompt: rsync -av --delete Project joeuser@unixdev1.ceas.uwm.edu:Data
```

The general syntax for pulling files from another host is:

```
shell-prompt: rsync -av --delete [username@]hostname:[source-path] destination-path
```

Example 1.5 Pulling data with rsync

The following command synchronizes the directory ~joeuser/Data/Project on Peregrine to ./Project on the local machine:

```
shell-prompt: rsync -av --delete joeuser@unixdev1.ceas.uwm.edu:Data/project .
```

If you omit "username@" from the source or destination, rsync will try to log into the remote system with your username on the local system.

If you omit destination-path from a push command or source-path from a pull command, rsync will use your home directory on the remote host.

The command-line flags used above have the following meanings:

-a Use *archive* mode. Archive mode copies all subdirectories recursively and preserves as many file attributes as possible, such as ownership, permissions, etc.

-v Verbose copy: Display names of files and directories as they are copied.

--delete Delete files and directories from the destination if they do not exists in the source. Without --delete, rsync will add and replace files in the destination, but never remove anything.

Caution
Note that a trailing "/" on source-path affects where rsync stores the files on the destination system. Without a trailing "/", rsync will create a directory called "source-path" under "destination-path" on the destination host.
With a trailing "/" on source-path, destination-path is assumed to be the directory that will replace source-path on the destination host. This feature is a somewhat cryptic method of allowing you to change the name of the directory during the transfer. However, it is compatible with the basic Unix **cp** command.
Note also that the trailing "/" only affects the command when applied to source-path. A trailing "/" on destination-path has no effect.

The command below creates an identical copy of the directory Data/Model in Model (/home/bacon/Data/Model to be precise) on unixdev1.ceas.uwm.edu. The resulting directory is the same regardless of whether the destination directory existed before the command or not.

```
shell-prompt: rsync -av --delete Model joeuser@unixdev1.ceas.uwm.edu:Data
```

The command below dumps the *contents* of Model directly into Data, and deletes everything else in the Data directory! In other words, it makes the destination directory Data identical to the source directory Model.

```
shell-prompt: rsync -av --delete Model/ joeuser@unixdev1.ceas.uwm.edu:Data
```

To achieve the same effect as the command with no "/", you would need to fully specify the destination path:

```
shell-prompt: rsync -av --delete Model/ joeuser@unixdev1.ceas.uwm.edu:Data/Model
```

Note that if using globbing on the remote system, any globbing patterns must be protected from expansion by the local shell by escaping them or enclosing them in quotes. We want the pattern expanded on the remote system, not the local system:

```
shell-prompt: rsync -av --delete joeuser@unixdev1.ceas.uwm.edu:Data/Study\* .
shell-prompt: rsync -av --delete 'joeuser@unixdev1.ceas.uwm.edu:Data/Study*' .
```

For full details on the rsync command, type

```
shell-prompt: man rsync
```

1.16 Environment Variables

Every Unix process maintains a list of character string variables called the *environment*. When a new process is created, it inherits the environment from the process that created it (its parent process).

Since the shell creates a new process whenever you run an external command, the shell's environment can be used to pass information down to any command that you run. For example, text editors and other programs that manipulate the full terminal screen need to know what type of terminal you are using. Different types of terminals use different magic sequences to move the cursor, clear the screen, scroll, etc. To provide this information, we set the shell's environment variable TERM to the terminal type (usually "xterm"). When you run a command from the shell, it inherits the shell's TERM variable, and therefore knows the correct magic sequences for your terminal.

The **printenv** shows all of the environment variables currently set in your shell process.

```
shell-prompt: printenv
```

Setting environment variables requires a different syntax depending on which shell you are using. Most modern Unix shells are extensions of either Bourne shell (sh) or C shell (csh), so there are only two variations of most shell commands that we need to know for most purposes.

For Bourne shell derivatives, we use the **export** command:

```
shell-prompt: TERM=xterm
shell-prompt: export TERM
```

For C shell derivatives, we use **setenv**:

```
shell-prompt: setenv TERM xterm
```

The `PATH` variable specifies a list of directories containing external Unix commands. When you type a command at the shell prompt, the shell checks the directories listed in `PATH` in order to find the command you typed. For example, when you type the **ls** command, the shell utilizes `PATH` to locate the program in `/bin/ls`.

The directory names within in PATH are separated by colons. A simple value for `PATH` might be `/bin:/usr/bin:/usr/local/bin`. When you type **ls**, the shell first checks for the existence of `/bin/ls`. If it does not exist, the shell then checks for `/usr/bin/ls`, and so on, until it either finds the program or has checked all directories in `PATH`. If the program is not found, the shell issues an error message such as "ls: Command not found".

Environment variables can be set from the shell prompt using the **export** command in Bourne shell and its derivatives (sh, bash, ksh):

```
shell-prompt: export PATH='/bin:/usr/bin:/usr/local/bin'
```

or using **setenv** in C shell and its derivatives (csh, tcsh):

```
shell-prompt: setenv PATH '/bin:/usr/bin:/usr/local/bin'
```

The **env** can be used to alter the environment just for the invocation of one child process, rather than setting it for the current shell process.

Suppose Bob has a script called **rna-trans** that we would like to run in his `~/bin` directory. This script also invokes other scripts in the same directory, so we'll need it in our path while his script runs.

```
shell-prompt: env PATH='/bin:/usr/bin:/usr/local/bin:~bob/bin' rna-trans
```

You can create environment variables with any name and value you like. However, there are some environment variable names that are reserved for specific purposes. A few of the most common ones are listed in Table 1.13.

Name	Purpose
TERM	Terminal type for an interactive shell session
USER	User's login name
HOME	Absolute path of the user's home directory (~)
PATH	List of directories searched for commands
LANG	Character set for the local language
EDITOR	User's preferred interactive text editor

Table 1.13: Reserved Environment Variables

1.16.1 Self-test

1. What are environment variables?
2. Does a Unix process have any environment variables when it starts? If so, where do they come from?
3. How can environment variables be used to communicate information to child processes?

4. Describe one common environment variable that it typically set by the shell and used by processes running under the shell.
5. Show how to set the environment variable TERM to the value "xterm" in
   (a) Bourne shell (sh)
   (b) Korn shell (ksh)
   (c) Bourne again shell (bash)
   (d) C shell (csh)
   (e) T-shell (tcsh)
6. Show a Unix command that runs **ls** with the LSCOLORS environment variable set to "CxFxCxDxBxegedaBaGaCaD". You may not change the LSCOLORS variable for the current shell process.

1.17 Shell Variables

In addition to the environment, shells maintain a similar set of variables for their own use. These variables are not passed down to child processes, and are only used by the shell.

Shell variables can be arbitrary, but each shell also treats certain variable names specially. One common example is the shell variable that stores the shell prompt.

In Bourne-shell derivatives, this variable is called PS1. To set a shell variable in Bourne-shell derivatives, we use a simple assignment. (The export command above actually sets a shell variable called TERM and then exports it to the environment.)

```
shell-prompt: PS1="peregrine: "
```

In C shell derivatives, the variable is called `prompt`, and is set using the **set** command:

```
shell-prompt: set prompt="peregrine: "
```

Note The syntax for **set** is slightly different than for **setenv**. **Set** uses an '=' while **setenv** uses a space.

Shell prompt variables may contain certain special symbols that represent dynamic information they you might want to include in your shell prompt, such as the host name, command counter, current working directory, etc. Consult the documentation for your shell for details.

In all shells, you can view the current shell variables by typing **set** with no arguments:

```
shell-prompt: set
```

1.17.1 Self-test

1. Show how to set the shell prompt to "Peregrine: " in:
   (a) Bourne shell
   (b) C shell
2. How can you view a list of all current shell variables and their values?

1.18 Process Control

Unix systems provide many tools for managing and monitoring processes that are already running.

Note that these tools apply to local Unix processes only. On distributed systems such as clusters and grids, job management is done using networked schedulers such as HTCondor, Grid Engine, or PBS.

It is possible to have multiple processes running under the same shell session. Such processes are considered either *foreground processes* or *background processes*. The foreground process is simply the process that receives the keyboard input. There can be no more than one foreground process under a given shell session, for obvious reasons.

Note that all processes, both foreground and background, can send output to the terminal at the same time, however. It is up to the user to ensure that output is managed properly and not intermixed.

There are three types of tools for process management, described in the following subsections.

1.18.1 External Commands

Unix systems provide a variety of external commands that monitor or manipulate processes based on their process ID (PID). A few of the most common commands are described below.

ps lists the currently running processes.

```
shell-prompt: ps [-a]     # BSD
shell-prompt: ps [-e]     # SYSV
```

ps is one of the rare commands whose options vary across different Unix systems. There are only two standards to which it may conform, however. The BSD version uses `-a` to indicate that all processes (not just your own) should be shown. System 5 (SYSV) **ps** uses `-e` for the same purpose. Run **man ps** on your system to determine which flags should be used.

kill sends a signal to a process (which may kill the process, but could serve other purposes).

```
shell-prompt: kill [-#] pid
```

The pid (process ID) is determined from the output of **ps**.

The signal number is an integer value following a -, such as `-9`. If not provided, the default signal sent is the TERM (terminate) signal.

Some processes ignore the TERM signal. Such processes can be force killed using the KILL (9) signal.

```
shell-prompt: kill -9 2342
```

Run **man signal** to learn about all the signals that can be issued with **kill**.

```
shell-prompt: ps
  PID  TT  STAT      TIME COMMAND
 41167   0  Is     0:00.25 tcsh
 78555   0  S+     0:01.98 ape unix.dbk
shell-prompt: kill 78555
```

The **pkill** command will kill all processes running the program named as the argument. This eliminates the need to find the PID first, and is more convenient for killing multiple processes running the same program.

```
shell-prompt: pkill fdtd
```

1.18.2 Special Key Combinations

Ctrl+c sends a terminate signal to the current foreground process. This usually kills the process immediately, although it is possible that some processes will ignore the signal.

Ctrl+z sends a suspend signal to the current foreground process. The process remains in memory, but does not execute further until it receives a resume signal (usually sent by running **fg**).

Ctrl+s suspends output to the terminal. This does not technically control the process directly, but has the effect of blocking any processes that are sending output, since they will stop running until the terminal begins accepting output again.

Ctrl+q resumes output to the terminal if it has been suspended.

1.18.3 Internal Shell Commands and Symbols

jobs lists the processes running under the current shell, but using the shell's job IDs instead of the system's process IDs.

Caution Shell jobs are ordinary processes running on the local system and should not be confused with cluster and grid jobs, which are managed by networked schedulers.

```
shell-prompt: jobs
```

fg brings a background job into the foreground.

```
shell-prompt: fg [%job-id]
```

There cannot be another job already running in the foreground. If no job ID is provided, and multiple background jobs are running, the shell will choose which background job to bring to the foreground. A job ID should always be provided if more than one background job is running.

bg resumes a job suspended by Ctrl+z in the background.

```
shell-prompt: prog
Ctrl+z
shell-prompt: bg
shell-prompt:
```

An & at the end of any command causes the command to be immediately placed in the background. It can be brought to the foreground using **fg** at any time.

```
shell-prompt: command &
```

nice runs a process at a lower than normal priority.

```
shell-prompt: nice command
```

If (and only if) other processes in the system are competing for CPU time, they will get a bigger share than processes run under **nice**.

time runs a command under the scrutiny of the time command, which keeps track of the process's resource usage.

```
shell-prompt: time command
```

There are both internal and external implementations of the time command. Run **which time** to determine which one your shell is configured to use.

nohup allows you to run a command that will continue after you log out. Naturally, all input and output must be redirected away from the terminal in order for this to work.

Bourne shell and compatible:

```
shell-prompt: nohup ./myprogram < inputfile > outputfile 2>&1
```

C shell and compatible:

```
shell-prompt: nohup ./myprogram < inputfile >& outputfile
```

This is often useful for long-running commands and where network connections are not reliable.

There are also free add-on programs such as GNU screen that allow a session to be resumed if it's disrupted for any reason.

1.18.4 Self-test

1. What is a process?
2. What is the difference between a foreground process and a background process?
3. How many foreground processes can be running at once under a single shell process? Why?
4. How many background processes can be running at once under a single shell process? Why?
5. Show the simplest Unix command that will accomplish each of the following:
   (a) List all processes currently running.
   (b) List processes owned by you.
   (c) Kill the process with ID 7243.
   (d) Kill all processes running the program **netsim**.
   (e) Kill the process with ID 7243 after the first attempt failed.
6. How do you perform each of the following tasks?
   (a) Kill the current foreground process.
   (b) Suspend the current foreground process.
   (c) Resume a suspended process in the foreground.
   (d) Resume a suspended process in the background.
   (e) Start a new process, placing it in the background immediately.
   (f) Suspend terminal output for a process without suspending the process itself.
   (g) Resume suspended terminal output.
   (h) List the currently running jobs as seen by the shell.
   (i) Return job #2 to the foreground.
   (j) Run the program **netsim** at a reduced priority so that other processes will respond faster.
   (k) Run the program **netsim** and report the CPU time used when it finishes.

1.19 Remote Graphics

Most users will not need to run graphical applications on a remote Unix system.. If you know that you will need to use a graphical user interface with your research software, or if you want to use a graphical editor such as gedit or emacs on over the network, read on. Otherwise, you can skip this section for now.

Unix uses a networked graphics interface called the X Window system. It is also sometimes called simply X11 for short. (X11 is the latest major version of the system.) X11 allows programs running on a Unix system to display graphics on the local screen or the screen of another Unix system on the network. The programs are called *clients*, and they display graphical output by sending commands to the *X11 server* on the machine where the output is to be displayed. Hence, your local computer must be running an X11 server in order to display Unix graphics, regardless of whether the client programs are running on your machine or another.

Some versions of OS X had the Unix X11 API included, while others need it installed separately. At the time of this writing, X11 on the latest OS X is provided by the XQuartz project, described at https://support.apple.com/en-us/HT201341. You will need to download and install this free package to enable X11 on your Mac.

1.19.1 Configuration Steps Common to all Operating Systems

Modern Unix systems such as BSD, Linux, and Mac OS X have most of the necessary tools and configuration in place for running remote graphical applications.

However, some additional steps may be necessary on your computer to allow remote systems to access your display. This applies to *all* computers running an X11 server, regardless of operating system. Some additional steps that may be necessary for Cygwin systems are discussed in Section 1.19.2.

If you want to run graphical applications on a remote computer over an ssh connection, you will need for forward your local display to the remote system. This can be done for a single ssh session by providing the `-X` flag:

```
shell-prompt: ssh -X joe@unixdev1.ceas.uwm.edu
```

This causes the **ssh** command to inform the remote system that X11 graphical output should be sent to your local display through the **ssh** connection. (This is called SSH tunneling.)

Caution Allowing remote systems to display graphics on your computer can pose a security risk. For example, a remote user may be able to display a false login window on your computer in order to collect login and password information.

If you want to forward X11 connections to all remote hosts for all users on the local system, you can enable X11 forwarding in your `ssh_config` file (usually found in /etc or /etc/ssh) by adding the following line:

```
ForwardX11 yes
```

Caution Do this only if you are prepared to trust all users of your local system as well as all remote systems to which they might connect.

Some X11 programs require additional protocol features that can pose more security risks to the client system. If you get an error message containing "Invalid MIT-MAGIC-COOKIE" when trying to run a graphical application over an **ssh** connection, try using the `-Y` flag with **ssh** to open a *trusted* connection.

```
shell-prompt: ssh -Y joe@unixdev1.ceas.uwm.edu
```

You can establish trusted connections to *all* hosts by adding the following to your `ssh_config` file:

```
ForwardX11Trusted yes
```

Caution This is generally considered a bad idea, since it states that every host connected to from this computer to should be trusted completely. Since you don't know in advance what hosts people will connect to in the future, this is a huge leap of faith.

If you are using ssh over a slow connection, such as home DSL/cable, and plan to use X11 programs, it can be very helpful to enable compression, which is enabled by the `-C` flag. Packets are then compressed before being sent over the wire and decompressed on the receiving end. This adds more CPU load on both ends, but reduces the amount of data flowing over the network and may significantly improve the responsiveness of a graphical user interface. Run **man ssh** for details.

```
shell-prompt: ssh -YC joe@unixdev1.ceas.uwm.edu
```

1.19.2 Graphical Programs on Windows with Cygwin

It is possible for Unix graphical applications on the remote Unix machine to display on a Windows machine, but this will require installing additional Cygwin packages and performing a few configuration steps on your computer in addition to those discussed in Section 1.19.1.

Installation

You will need to install the x11/xinit and x11/xhost packages using the Cygwin setup utility. This will install an X11 server on your Windows machine.

Configuration

After installing the Cygwin X packages, there are additional configuration steps:

1. Create a working `ssh_config` file by running the following command from a Cygwin shell window:

   ```
   shell-prompt: cp /etc/defaults/etc/ssh_config /etc
   ```

2. Then, using your favorite text editor, update the new `/etc/ssh_config` as described in Section 1.19.1.

3. Add the following line to .bashrc or .bash_profile (in your home directory):

   ```
   export DISPLAY=":0.0"
   ```

   Cygwin uses bash for all users by default. If you are using a different shell, then edit the appropriate start up script instead of .bashrc or .bash_profile.

   This is not necessary when running commands from an xterm window (which is launched from Cygwin-X), but *is* necessary if you want to launch X11 applications from a Cygwin bash terminal which is part of the base Cygwin installation, and not X11-aware.

Start-up

To enable X11 applications to display on your Windows machine, you need to start the X11 server on Windows by clicking Start → All Programs → Cygwin-X → XWin Server. The X server icon will appear in your Windows system tray to indicate that X11 is running. You can launch an xterm terminal emulator from the system tray icon, or use the Cygwin bash terminal, assuming that you have set your DISPLAY variable.

1.20 Where to Learn More

There is a great deal of information available on the web. There are also many length books dedicated to Unix, which can provide more detail than this tutorial.

If you simply want to know what commands are available on your system, list the bin directories!

```
shell-prompt: ls /bin /usr/bin /usr/local/bin | more
```

Chapter 2

Unix Shell Scripting

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 1) and the Unix shell (Section 1.3.3).

2.1 What is a Shell Script?

A shell script is essentially a file containing a sequence of Unix commands. A script is a type of program, but is distinguished from other programs in that it represents programming at a higher level.

While a typical program is largely made of calls to subprograms, a script contains invocations of whole programs.

In other words, a script is a way of automating the execution of multiple separate programs in sequence.

The Unix command-line structure was designed to be convenient for both interactive use and for programming in scripts. Running a Unix command is much like calling a subprogram. The difference is just syntax. A subprogram call in C encloses the arguments in parenthesis and separates them with commas:

```
function_name(arg1,arg2,arg3);
```

A Unix command is basically the same, except that it uses spaces instead of parenthesis and commas:

```
command_name arg1 arg2 arg3
```

2.1.1 Self-test

1. What is a shell script?
2. How are Unix commands similar to and different from subprogram calls in a language like C?

2.2 Scripts vs Programs

It is important to understand the difference between a "script" and a "real program", and which languages are appropriate for each.

Scripts tend to be small (no more than a few hundred or a few thousand lines of code) and do not do any significant computation of their own.

Instead, scripts run "real programs" to do most of the computational work. The job of the script is simply to automate and document the process of running programs.

As a result, scripting languages do not need to be efficient and are generally interpreted rather than compiled. (Interpreted language programs run an order of magnitude or more slower than equivalent compiled programs, unless most of their computation is done by built-in, compiled subprograms.)

Real programs may be quite large and may implement complex computational algorithms. Hence, they need to be fast and as a result are usually written in compiled languages.

If you plan to use exclusively pre-existing programs such as Unix commands and/or add-on application software, and need only automate the execution of these programs, then you need to write a script and should choose a good scripting language.

If you plan to implement your own algorithm(s) that may require a lot of computation, then you need to write a program and should select an appropriate compiled programming language.

2.2.1 Self-test

1. How do scripts differ from programs written in languages like C or Fortran?
2. Why would it not be a good idea to write a matrix multiplication program as a Bourne shell script?

2.3 Why Write Shell Scripts?

2.3.1 Efficiency and Accuracy

Any experienced computer user knows that we often end up running basically the same sequence of commands many times over. Typing the same sequence of commands over and over is a waste of time and highly prone to errors.

All Unix shells share a feature that can help us avoid this repetitive work: They don't care where their input comes from.

It is often said that the Unix shell reads commands from the keyboard and executes them. This is not really true. The shell reads commands from *any input source* and executes them. The keyboard is just one common input source that can be used by the shell. Ordinary files are also very commonly used as shell input.

Recall from Chapter 1 that Unix systems employ device independence, which means that any Unix program that reads from a keyboard can also read the same input from a file or any other input device.

Hence, if we're going to run the same sequence of commands more than once, we don't need to retype the sequence each time. The shell can read the commands from anywhere, not just from the keyboard. We can put those commands into a text file *once* and tell the shell to read the commands from the file, which is much easier than typing them all again.

Rule of Thumb If you might have to do it again, script it.

In theory, Unix commands could also be piped in from another program or read from any other device attached to a Unix system, although in practice, they usually come from the keyboard or a script file.

Self-test

1. Describe two reasons for writing shell scripts.
2. Are there Unix commands that you can run interactively, but not from a shell script? Explain.
3. What feature of Unix makes shell scripts so convenient to implement?
4. What is a good rule of thumb for deciding whether to write a shell script?

2.3.2 Documentation

There is another very good reason for writing shell scripts in addition to saving us a lot of redundant typing:

A shell script is the ultimate documentation of the work we have done on a computer.

By writing a shell script, we record the exact sequence of commands needed to reproduce results, in perfect detail. Hence, the script serves a dual purpose of automating and documenting our processes.

Developing a script has a ratchet effect on your knowledge. Once you add a command to a script, you will never forget how to use it for that task.

Rule of Thumb A Unix user should never find themselves trying to remember how they did something. Script it the first time...

Clear documentation of our work flow is important in order to justify research funding and to be able to reproduce results months or years later.

Note
An important part of documenting code is making the code *self-documenting*. When writing shell scripts, using long options in commands such as **zip --preserve-case** instead of **zip -C** makes the script much easier to read. While -C is less typing and may be preferable when running zip interactively many times, we only have to type `--preserve-case` once when writing the script, so the laziness of using `-C` doesn't pay here.

Imagine that we instead decided to run our sequence of commands manually and document what we did in a word processor. First, we'd be typing everything twice: Once at the shell prompt and again into the document.

The process of typing the same commands each time would be painful enough, but to document it in detail while we do it would be distracting. We'd also have to remember to update the document every time we type a command differently. This is hard to do when we're trying to focus on getting results.

Writing a shell script allows us to stay focused on perfecting the process. Once the script is finished and working perfectly, we have the process perfectly documented. We can and should add comments to the script to make it more readable, but even without comments, the script itself preserves the process in detail.

Many experienced users will *never* run a processing command from the keyboard. Instead, they *only* put commands into a script and run and re-run the script until it's finished.

Self-test

1. Describe another good reason for writing shell scripts.
2. Why is it so important to document the sequence of commands used?

2.3.3 Why Unix Shell Scripts?

There are many scripting languages to choose from, including those used on Unix systems, like Bourne shell, C shell, Perl, Python, etc., as well as some languages confined to other platforms like Visual Basic (Microsoft Windows only) and AppleScript (Apple only).

Note that the Unix-based scripting languages can be used on *any* platform, *including* Microsoft Windows (with Cygwin, for example) and Apple's Mac OS X, which is Unix-compatible out of the box.

Once you learn to write Unix shell scripts, you're prepared to do scripting on any computer, without having to learn another language.

Self-test

1. What are two advantages of writing Unix shell scripts instead of using a scripting language such as Visual Basic or AppleScript?

2.3.4 Self-test

1. Describe three reasons for writing shell scripts instead of running commands from the keyboard.

2.4 Which Shell?

2.4.1 Common Shells

When writing a shell script, there are essentially two scripting languages to choose from: Bourne shell and C shell. These were the first two popular shells for Unix, and all common shells that have come since are compatible with one or the other.

The most popular new shells are Bourne Again shell (bash), which is an extension of Bourne shell, Korn shell (ksh), which is another extension of Bourne shell, Z-shell, a very sophisticated extension of Bourne shell, and T-shell (TENEX C shell, tcsh), which is an extended C shell.

- Bourne shell family
  - Bourne shell (sh)
  - Bourne-again shell (bash)
  - Korn shell (ksh)
  - Z-shell (zsh)
- C shell family
  - C shell (csh)
  - T-shell (tcsh)

Both Bourne shell and C shell have their own pros and cons. C shell syntax is cleaner, more intuitive, and more similar to the C programming language (hence the name C shell). However, C shell lacks some features such as subprograms (although C shell scripts can run other C shell scripts, which is arguably a better approach in many situations).

Bourne shell is used almost universally for Unix system scripts, while C shell is fairly popular in scientific research.

Note Every Unix system has a Bourne shell in `/bin/sh`. Hence, using vanilla Bourne shell (not bash, ksh, or zsh) for scripts maximizes their portability by ensuring that they will run on any Unix system without the need to install any additional shells.

If your script contains only external commands, then it actually won't matter which shell runs it. However, most scripts utilize the shell's internal commands, control structures, and features like redirection and pipes, which differ among shells.

More modern shells such as bash, ksh, and tcsh, are backward-compatible with Bourne shell or C shell, but add additional scripting constructs in addition to convenient interactive features. The details are beyond the scope of this text. For full details, see the documentation for each shell.

2.4.2 Self-test

1. What is one advantage of Bourne shell over C shell?
2. What is one advantage of C shell over Bourne shell?

2.5 Writing and Running Shell Scripts

A shell script is a simple text file and can be written using any Unix text editor. Some discussion of Unix text editors can be found in Section 1.10.4.

Caution Recall from Section 1.9.1 that Windows uses a slightly different text file format than Unix. Hence, editing Unix shell scripts in a Windows editor can be problematic. Users are advised to do all of their editing on a Unix machine rather than write programs and scripts on Windows and transfer them to Unix.

Shell scripts often contain very complex commands that are wider than a typical terminal window. A command can be continued on the next line by typing a backslash (\) immediately before pressing **Enter**. This feature is present in all Unix shells. Of course, it can be used on an interactive CLI as well, but is far more commonly used in scripts to improve readability.

```
printf "%s %s\n" "This command is too long to fit in a single 80-column" \
    "terminal window, so we break it up with a backslash."
```

It's not a bad idea to name the script with a file name extension that matches the shell it uses. This just makes it easier to see which shell each of your script files use. Table 2.1 shows conventional file name extensions for the most common shells. However, if a script is to be installed into the PATH so that it can be used as a regular command, it is usually given a name with no extension. Most users would rather type "cleanup" than "cleanup.bash".

Like all programs, shell scripts should contain comments to explain what the commands in it are doing. In all Unix shells, anything from a '#' character to the end of a line is considered a comment and ignored by the shell.

```
# Print the name of the host running this script
hostname
```

Shell	Extension
Bourne Shell	.sh
C shell	.csh
Bourne Again Shell	.bash
T-shell	.tcsh
Korn Shell	.ksh
Z-shell	.zsh

Table 2.1: Conventional script file name extensions

Practice Break
Using your favorite text editor, enter the following text into a file called `hello.sh`.

1. The first step is to create the file containing your script, using any text editor, such as nano:

   ```
   shell-prompt: nano hello.sh
   ```

   Once in the text editor, add the following text to the file:

   ```
   printf "Hello!\n"
   printf "I am a script running on a computer called `hostname`\n"
   ```

   After typing the above text into the script, save the file and exit the editor. If you are using nano, the menu at the bottom of the screen tells you how to save (write out, Ctrl+o) and exit (Ctrl+x).

2. Once we've written a script, we need a way to run it. A shell script is simply input to a shell program. Like many Unix programs, shells take their input from the standard input by default. We could, therefore, use redirection to make it read the file via standard input:

   ```
   shell-prompt: sh < hello.sh
   ```

Sync-point: Instructor: Make sure everyone in class succeeds at this exercise before moving on.

Since the shell normally reads commands from the standard input, the above command will "trick" **sh** into reading its commands from the file `hello.sh`.

However, Unix shells and other scripting languages provide a more convenient method of indicating what program should interpret them. If we add a special comment, called a *shebang line* to the top of the script file and make the file executable using **chmod**, the script can be executed like a Unix command. We can then simply type its name at the shell prompt, and another shell process will start up and run the commands in the script. If the directory containing such a script is included in $PATH, then the script can be run from any current working directory, just like **ls**, **cp**, etc.

The shebang line consists of the string "#!" followed by the full path name of the command that should be used to execute the script, or the path **/usr/bin/env** followed by the name of the command. For example, both of the following are valid ways to indicate a Bourne shell (sh) script, since `/bin/sh` is the Bourne shell command.

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

```
#!/usr/bin/env sh
```

When you run a script as a command, by simply typing its file name at the Unix command-line, a new shell process is created to interpret the commands in the script. The shebang line specifies which program is invoked for the new shell process that runs the script.

Note The shebang line must begin at the very first character of the script file. There cannot even be blank lines above it or white space to the left of it. The "#!" is an example of a *magic number*. Many files begin with a 16-bit (2-character) code to indicate the type of the file. The "#!" indicates that the file contains some sort of interpreted language program, and the characters that follow will indicate where to find the interpreter.

The **/usr/bin/env** method is used for add-on shells and other interpreters, such as Bourne-again shell (bash), Korn shell (ksh), and Perl (perl). These interpreters may be installed in different directories on different Unix systems. For example, bash is typically found in **/bin/bash** on Linux systems, **/usr/local/bin/bash** on FreeBSD systems, and **/usr/pkg/bin/bash** on NetBSD. The T-shell is found in **/bin/tcsh** on FreeBSD and CentOS Linux and in **/usr/bin/tcsh** on Ubuntu Linux.

In addition, Redhat Enterprise Linux (RHEL) and CentOS users may install a newer version of bash under a different prefix and want to use it to run their shell scripts. RHEL is a special Linux distribution built on an older snapshot of Linux for the sake of

long-term binary compatibility and stability. As such, it runs older versions of bash and other common tools. As of this writing, CentOS 7, the mainstream CentOS version, uses bash 4.2, while pkgsrc, a portable package manager offers bash 5.1.

The **env** command is found in `/usr/bin/env` on virtually all Unix systems. Hence, this provides a method for writing shell scripts that are portable across Unix systems (i.e. they don't need to be modified to run on different Unix systems).

Bourne shell (sh) is present and installed in `/bin` on all Unix-compatible systems, so it's safe to hard-code #!/bin/sh is the shebang line.

C shell (csh) is not included with all systems, but is virtually always in /bin if present, so it is generally safe to use #!/bin/csh as well.

For all other interpreters it's best to use #!/usr/bin/env.

```
#!/bin/sh           (OK and preferred)

#!/bin/csh          (Generally OK)

#!/bin/bash         (Bad idea: Not portable)

#!/usr/bin/perl     (Bad idea: Not portable)

#!/usr/bin/python   (Bad idea: Not portable)

#!/bin/tcsh         (Bad idea: Not portable)

#!/usr/bin/env bash     (This is portable)

#!/usr/bin/env tcsh     (This is portable)

#!/usr/bin/env perl     (This is portable)

#!/usr/bin/env python   (This is portable)
```

Even if your system comes with `/bin/bash` and you don't intend to run the script on any other system, using `/usr/bin/env` is still a good idea, because you or someone else may want to use a newer version of bash that's installed in a different location. The same applies to other scripting languages such as C-shell, Perl, Python, etc.

Example 2.1 A Simple Bash Script

Suppose we want to write a script that is always executed by bash, the Bourne Again Shell. We simply need to add a shebang line indicating the path name of the **bash** executable file.

```
shell-prompt: nano hello.sh
```

Enter the following text in the editor. Then save the file and exit back to the shell prompt.

```
#!/usr/bin/env bash

# A simple command in a shell script
printf "Hello, world!\n"
```

Now, make the file executable and run it:

```
shell-prompt: chmod a+rx hello.sh   # Make the script executable
shell-prompt: ./hello.sh            # Run the script as a command
```

Example 2.2 A Simple T-shell Script

Similarly, we might want to write a script that is always executed by tcsh, the TENEX C Shell. We simply need to add a shebang line indicating the path name of the **tcsh** executable file.

```
shell-prompt: nano hello.tcsh
```

```
#!/usr/bin/env tcsh

# A simple command in a shell script
printf "Hello, world!\n"
```

```
shell-prompt: chmod a+rx hello.tcsh   # Make the script executable
shell-prompt: ./hello.tcsh            # Run the script as a command
```

Note Many of the Unix commands you use regularly may actually be scripts rather than binary programs.

Note

There may be cases where you cannot make a script executable. For example, you may not own it, or the file system may not allow executables, to prevent users from running programs where they shouldn't.
In these cases, we can simply run the script as an argument to an appropriate shell. For example:

```
shell-prompt: sh hello.sh
shell-prompt: bash hello.bash
shell-prompt: csh hello.csh
shell-prompt: tcsh hello.tcsh
shell-prompt: ksh hello.ksh
```

Note also that the shebang line in a script is ignored when you explicitly run a shell this way. The content of the script will be interpreted by the shell that you have manually invoked, regardless of what the shebang line says.

Scripts that you create and intend to use regularly can be placed in your PATH, so that you can run them from anywhere. A common practice among Unix users is to create a directory called `~/bin`, and configure the login environment to that this directory is always in the PATH. Programs and scripts placed in this directory can then be used like any other Unix command, without typing the full path name.

2.5.1 Self-test

1. What tools can be used to write shell scripts?
2. Is it a good idea to write Unix shell scripts under Windows? Why or why not?
3. After creating a new shell script, what must be done in order to make it executable like a Unix command?
4. What is a shebang line?
5. What does the shebang line look like for a Bourne shell script? A Bourne again shell script? Explain the differences.

2.6 Shell Start-up Scripts

Each time you log into a Unix machine or start a new shell (e.g. when you open a new terminal), the shell process runs one or more special scripts called *start up scripts*. Some common start up scripts:

Script	Shells that use it	Executed by
/etc/profile, ~/.profile	Bourne shell family	Login shells only
File named by $ENV (typically .shrc or .shinit)	Bourne shell family	All interactive shells (login and non-login)
~/.bashrc	Bourne-again shell only	All interactive shells (login and non-login)
~/.bash_profile	Bourne-again shell only	Login shells only
~/.kshrc	Korn shell	All interactive shells (login and non-login)
/etc/csh.login, ~/.login	C shell family	Login shells only
/etc/csh.cshrc, ~/.cshrc	C shell family	All shell processes
~/.tcshrc	T-shell	All shell processes

Table 2.2: Shell Start Up Scripts

Note
Non-interactive Bourne-shell family shell processes, such as those used to execute shell scripts, do not execute any start up scripts. Hence, Bourne shell family scripts are not affected by start up scripts.
In contrast, all C shell script processes execute ~/.cshrc if it exists. Hence, C shell family scripts are affected by ~/.cshrc. You can override this in C-shell scripts by invoking the shell with `-f` as follows:

```
#!/bin/csh -f
```

The man page for your shell has all the details about which start up scripts are invoked and when.

Start up scripts are used to configure your PATH and other environment variables, set your shell prompt and other shell features, create aliases, and anything else you want done when you start a new shell.

One of the most common alterations users make to their start up script is editing their PATH to include a directory containing their own programs and scripts. Typically, this directory is named `~/bin`, but you can name it anything you want.

To set up your own ~/bin to store your own scripts and programs, do the following:

1. shell-prompt: mkdir ~/bin
2. Edit your start up script and add ~/bin to the PATH.

   If you're using Bourne-again shell, you can add `~/bin` to your PATH for login shells only by adding the following to your `.bashrc`:

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

   If you're using T-shell, add the following to your `.cshrc` or `.tcshrc`:

   ```
   setenv PATH ${PATH}:~/bin
   ```

   If you are using a different shell, see the documentation for your shell to determine the correct start up script and command syntax.

Caution
Adding ~/bin before (left of) ${PATH} will cause your shell to look in ~/bin before looking in the standard directories such as /bin and /usr/bin. Hence, if a binary or script in ~/bin has the same name as another command, the one in ~/bin will override it. This is considered a security risk, since users could be tricked into running a Trojan-horse **ls** or other common command if care is not taken to protect `~/bin` from modification.
Hence, adding to the tail (right side) of PATH is usually recommended, especially for inexperienced users.

3. Update the PATH in your current shell process by sourcing the start up script, or by logging out and logging back in.

There is no limit to what your start up scripts can do, so you can use your imagination freely and find ways to make your Unix shell environment easier and more powerful.

2.6.1 Self-test

1. What is the purpose of a shell start-up script?
2. What are the limitations on what a start-up script can do compared to a normal script?

2.7 Sourcing Scripts

In some circumstances, we might not want a script to be executed by a separate shell process.

For example, suppose we just made some changes to our .cshrc or .bashrc file that would affect PATH or some other important environment variable.

If we run the start up script by typing **~/.cshrc** or **~/.bashrc**, a new shell process will be started which will execute the commands in the script and then terminate. The shell you are using, which is the parent process, will be unaffected, since parent processes do not inherit environment from their children.

In order to make the "current" shell process run the commands in a script, we must *source* it. This is done using the internal shell command **source** in all shells except Bourne shell, which uses ".". Most Bourne-derived shells support both "." and "source".

Hence, to source `.cshrc`, we would run

```
shell-prompt: source ~/.cshrc
```

To source `.bashrc`, we would run

```
shell-prompt: source ~/.bashrc
```

or

```
. ~/.bashrc
```

To source `.shrc` from a basic Bourne shell, we would have to run

```
. ~/.shrc
```

2.7.1 Self-test

1. What is sourcing?
2. When would we want to source a script?

2.8 Scripting Constructs

Although Unix shells make no distinction between commands entered from the keyboard and those input from a script, there are certain shell features that are meant for scripting and not convenient or useful to use interactively.

Many of these features will be familiar to anyone who has done any computer programming. They include constructs such as comments, conditionals and loops.

The following sections provide a very brief introduction to shell constructs that are used in scripting, but generally not used on the command line.

2.9 Strings

A string constant in a shell script is anything enclosed in single quotes ('this is a string') or double quotes ("this is also a string").

Unlike most programming languages, text in a shell scripts that is not enclosed in quotes and does not begin with a '$' or other special character is also interpreted as a string constant. Hence, all of the following are the same:

```
shell-prompt: ls /etc
shell-prompt: ls "/etc"
shell-prompt: ls '/etc'
```

However, something contains white space (spaces or tabs), then it will be seen as multiple separate strings. The last example below will not work properly, since 'Program' and 'Files' are seen as separate arguments:

```
shell-prompt: cd 'Program Files'
shell-prompt: cd "Program Files"
shell-prompt: cd Program Files
```

Note

Special sequences such as '\n' must be enclosed in quotes or escaped, otherwise the '\' is seen as escaping the 'n'.

```
Hello\n != "Hello\n"
"Hello\n" = 'Hello\n' = Hello\\n
```

2.10 Output

Output commands are only occasionally useful at the interactive command line. We may sometimes use them to take a quick look at a variable such as $PATH.

```
shell-prompt: echo $PATH
```

Output commands are far more useful in scripts, and are used in the same ways as output statements in any programming language.

The **echo** command is commonly used to output something to the terminal screen:

```
shell-prompt: echo 'Hello!'
Hello!
shell-prompt: echo $PATH
/usr/local/bin:/home/bacon/scripts:/home/bacon/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local ↩
    /sbin
```

However, **echo** should be avoided, since it is not portable across different shells and even the same shell on different Unix systems. There are many different implementations of **echo** commands, some internal to the shell and some external programs. Different implementations of **echo** use different command-line flags and special characters to control output formatting.

In addition, the output formatting capabilities of **echo** commands are extremely limited.

The **printf** command supersedes **echo**. It has a rich set of capabilities and is specified in the POSIX.2 standard, so its behavior is the same on all Unix systems.

The **printf** command is an external command, so it is independent of which shell you are using.

The functionality of **printf** closely matches that of the `printf()` function in the standard C library. It recognizes special characters such as '\n' (line feed), '\t' (tab), '\r' (carriage return), etc. and can print numbers in different bases.

```
shell-prompt: printf 'Hello!\n'
Hello!
```

The basic syntax of a printf command is as follows:

```
printf format-string argument [argument ...]
```

The format-string contains literal text and a *format specifier* to match each of the arguments that follows.

Each format specifier begins with a '%' and is followed by a symbol indicating the format in which to print the argument.

Specifier	Output
%s	String
%d	Decimal number
%o	Octal number

Table 2.3: Printf Format Specifiers

The printf command also recognized most of the same special character sequences as the C printf() function:

Sequence	Meaning
\n	Newline (move down to next line)
\r	Carriage Return (go to beginning of current line)
\t	Tab (go to next tab stop)

Table 2.4: Special Character Sequences

```
printf "%s %d %o\n" 10 10 10
```

Output:

```
10 10 12
```

There are many other format specifiers and special character sequences. For complete information, run **man printf**.

To direct printf output to the standard error instead of the standard output, we simply take advantage of device independence and use redirection:

```
        printf 'Hello!\n' >> /dev/stderr
```

Practice Break
Sync-point: Instructor: Make sure everyone in class succeeds at this exercise before moving on.
Write a shell script containing the printf statement above and run it. Write the same script using two different shells, such as Bourne shell and C shell. What is the difference between the two scripts?

2.10.1 Self-test

1. What are the advantages of **printf** over the **echo** command?
2. Does printf work under all shells? Why or why not?

2.11 Shell and Environment Variables

Variables are essential to any programming language, and scripting languages are no exception. Variables are useful for user input, control structures, and for giving short names to commonly used values such as long path names.

Most programming languages distinguish between variables and constants, but in shell scripting, we use variables for both.

Shell processes have access to two separate sets of string variables.

Recall from Section 1.16 that every Unix process has a set of string variables called the *environment*, which are handed down from the parent process in order to communicate important information.

For example, the TERM variable, which identifies the type of terminal a user us using, is used by programs such as top, vi, nano, more, and other programs that need to manipulate the terminal screen (move the cursor, highlight certain characters, etc.) The TERM environment variable is usually set by the shell process so that all of the shell's child processes (those running vi, nano, etc.) will inherit the variable.

Unix shells also keep another set of variables that are not part of the environment. These variables are used only for the shell's purpose and are not handed down to child processes.

There are some special shell variables such as "prompt" and "PS1" (which control the appearance of the shell prompt in C shell and Bourne shell, respectively).

Most shell variables, however, are defined by the user for use in scripts, just like variables in any other programming language.

2.11.1 Assignment Statements

In all Bourne Shell derivatives, a shell variable is created or modified using the same simple syntax:

```
varname=value
```

Caution In Bourne shell and its derivatives, there can be no space around the '='. If there were, the shell would think that 'varname' is a command, and '=' and 'value' are arguments.

```
bash-4.2$ name = Fred
bash: name: command not found
bash-4.2$ name=Fred
bash-4.2$ printf "$name\n"
Fred
```

When assigning a string that contains white space, it must be enclosed in quotes or the white space characters must be escaped:

```
#!/usr/bin/env bash

name=Joe Sixpack    # Error
name="Joe Sixpack"  # OK
name=Joe\ Sixpack   # OK
```

C shell and T-shell use the **set** command for assigning variables.

```
#!/bin/csh

set name="Joe Sixpack"
```

Caution Note that Bourne family shells also have a **set** command, but it has a completely different meaning, so take care not to get confused. The Bourne **set** command is used to set shell command-line options, not variables.

Unlike some languages, shell variables need not be declared before they are assigned a value. Declaring variables is unnecessary, since there is only one data type in shell scripts.

All variables in a shell script are character strings. There are no integers, Booleans, enumerated types, or floating point variables, although there are some facilities for interpreting shell variables as integers, assuming they contain only digits.

If you *must* manipulate real numbers in a shell script, you could accomplish it by piping an expression through **bc**, the Unix arbitrary-precision calculator:

```
printf "scale=5\n243.9 * $variable\n" | bc
```

Such facilities are very inefficient compared to other languages, however, partly because shell languages are interpreted, not compiled, and partly because they must convert each string to a number, perform arithmetic, and convert the results back to a string. Shell scripts are meant to automate sequences of Unix commands and other programs, not perform numerical computations.

In Bourne shell family shells, environment variables are set by first setting a shell variable of the same name and then *exporting* it.

```
TERM=xterm
export TERM
```

Modern Bourne shell derivatives such as bash (Bourne Again Shell) can do it in one line:

```
export TERM=xterm
```

Note Exporting a shell variable permanently tags it as exported. Any future changes to the variable's value will automatically be copied to the environment. This type of linkage between two objects is very rare in programming languages: Usually, modifying one object has no effect on any other.

C shell derivatives use the setenv command to set environment variables:

```
setenv TERM xterm
```

Caution Note that unlike the 'set' command, setenv requires white space, not an '=', between the variable name and the value.

2.11.2 Variable References

To reference a shell variable or an environment variable in a shell script, we must precede its name with a '$'. The '$' tells the shell that the following text is to be interpreted as a variable name rather than a string constant. The variable reference is then *expanded*, i.e. replaced by the value of the variable. This occurs anywhere in a command except inside a string bounded by single quotes or following an escape character (\), as explained in Section 2.9.

These rules are basically the same for all Unix shells.

```
#!/usr/bin/env bash

name="Joe Sixpack"
printf "Hello, name!\n"      # Not a variable reference!
printf "Hello, $name!\n"     # References variable "name"
```

Output:

```
Hello, name!
Hello, Joe Sixpack!
```

Practice Break
Type in and run the following scripts:

```
#!/bin/sh

first_name="Bob"
last_name="Newhart"
printf "%s %s is the man.\n" $first_name $last_name
```

CSH version:

```
#!/bin/csh

set first_name="Bob"
set last_name="Newhart"
printf "%s %s is the man.\n" $first_name $last_name
```

Note
If both a shell variable and an environment variable with the same name exist, a normal variable reference will expand the shell variable.
In Bourne shell derivatives, a shell variable and environment variable of the same name always have the same value, since exporting is the only way to set an environment variable. Hence, it doesn't really matter which one we reference.
In C shell derivatives, a shell variable and environment variable of the same name can have different values. If you want to reference the environment variable rather than the shell variable, you can use the printenv command:

```
Darwin heron bacon ~ 319: set name=Sue
Darwin heron bacon ~ 320: setenv name Bob
Darwin heron bacon ~ 321: echo $name
Sue
Darwin heron bacon ~ 322: printenv name
Bob
```

There are some special C shell variables that are automatically linked to environment counterparts. For example, the shell variable path is always the same as the environment variable PATH. The C shell man page is the ultimate source for a list of these variables.

If a variable reference is immediately followed by a character that could be part of a variable name, we could have a problem:

```
#!/usr/bin/env bash

name="Joe Sixpack"
printf "Hello to all the $names of the world!\n"
```

Instead of printing "Hello to all the Joe Sixpacks of the world", the printf will fail because there is no variable called "names". In Bourne Shell derivatives, non-existent variables are treated as empty strings, so this script will print "Hello to all the of the world!". C shell will complain that the variable "names" does not exist.

We can correct this by delimiting the variable name in curly braces:

```
#!/usr/bin/env bash

name="Joe Sixpack"
printf "Hello to all the ${name}s of the world!\n"
```

This syntax works for all shells.

2.11.3 Using Variables for Code Quality

Another very good use for shell variables is in eliminating redundant string constants from a script:

```
#!/usr/bin/env bash

output_value=`myprog`
printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

If for any reason the relative path `Run2/Output/results.txt` should change, then you'll have to search through the script and make sure that all instances are updated. This is a tedious and error-prone process, which can be avoided by using a variable:

```
#!/usr/bin/env bash

output_value=`myprog`
output_file="Run2/Output/results.txt"
printf "$output_value\n" >> $output_file
more $output_file
cp $output_file latest-results.txt
```

In the second version of the script, if the path name of `results.txt` changes, then only one change must be made to the script.

Avoiding redundancy is one of the primary goals of any good programmer.

In a more general programming language such as C or Fortran, this role would be served by a constant, not a variable. However, shells do not support constants, so we use a variable for this.

In most shells, a variable can be marked read-only in an assignment to prevent accidental subsequent changes. Bourne family shells use the **readonly** command for this, while C shell family shells use **set -r**.

```
#!/bin/sh

readonly output_value=`myprog`
printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

```
#!/bin/csh

set -r output_value=`myprog`
printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

2.11.4 Output Capture

Output from a command can be captured and used as a string in the shell environment by enclosing the command in back-quotes (``). In Bourne-compatible shells, we can use $() in place of back-quotes.

```
#!/bin/sh -e

# Using output capture in a command
printf "Today is %s.\n" `date`
printf "Today is %s.\n" $(date)

# Using a variable.  If using the output more than once, this will
# avoid running the command multiple times.
today=`date`
printf "Today is %s\n" $today
```

2.11.5 Self-test

1. Describe two purposes for shell variables.
2. Are any shell variable names reserved? If so, describe two examples.
3. Show how to assign the value "Roger" to the variable "first_name" in both Bourne shell and C shell.
4. Why can there be no spaces around the '=' in a Bourne shell variable assignment?
5. How can you avoid problems when assigning values that contain white space?
6. Do shell variables need to be declared before they are used? Why or why not?
7. Show how to assign the value "xterm" to the *environment* variable TERM in both Bourne shell and C shell.
8. Why do we need to precede variables names with a '$' when referencing them?
9. How can we output a letter immediately after a variable reference (no spaces between them). For example, show a printf statement that prints the contents of the variable `fruit` immediately followed by the letter 's'.

```
fruit=apple
# Show a printf that will produce the output "I have 10 apples", using
# the variable fruit.
```

10. How can variables be used to enhance code quality? From what kinds of errors does this protect you?
11. How can a variable be made read-only in Bourne shell? In C shell?

2.12 Hard and Soft Quotes

Double quotes are known as *soft quotes*, since shell variable references, history events (!), and command output capture ($() or ``) are all expanded when used inside double quotes.

```
shell-prompt: history
  1003  18:11   ps
  1004  18:11   history

shell-prompt: echo "!hi"
echo "history"
history

shell-prompt: echo "Today is `date`"
Today is Tue Jun 12 18:12:33 CDT 2018

shell-prompt: echo "$TERM"
xterm
```

Single quotes are known as *hard quotes*, since every character inside single quotes is taken literally as part of the string, except for history events. Nothing else inside hard quotes is processed by the shell. If you need a literal ! in a string, it must be escaped.

```
shell-prompt: history
  1003  18:11   ps
  1004  18:11   history
shell-prompt: echo '!hi'
echo 'history'
history
shell-prompt: echo '\!hi'
!hi
shell-prompt: echo 'Today is `date`'
Today is `date`
shell-prompt: echo '$TERM'
$TERM
```

What will each of the following print? (If you're not sure, try it!)

```
#!/usr/bin/env bash

name='Joe Sixpack'
printf "Hi, my name is $name.\n"
```

```
#!/usr/bin/env bash

name='Joe Sixpack'
printf 'Hi, my name is $name.\n'
```

```
#!/usr/bin/env bash

first_name='Joe'
last_name='Sixpack'
name='$first_name $last_name'
printf "Hi, my name is $name.\n"
```

If you need to include a quote character as part of a string, you have two choices:

1. "Escape" it (precede it with a backslash character):

   ```
   printf 'Hi, I\'m Joe Sixpack.\n'
   ```

2. Use the other kind of quotes to enclose the string. A string terminated by double quotes can contain a single quote and vice-versa:

   ```
   printf "Hi, I'm Joe Sixpack.\n"
   printf 'You can use a " in here.\n'
   ```

No special operators are needed to concatenate strings in a shell script. We can simply place multiple strings in any form (variable references, literal text, etc.) next to each other.

```
printf 'Hello ,'$var'.'      # Variable between two hard-quotes strings
printf "Hello, $var."        # Variable between text in a soft-quoted string
```

2.12.1 Self-test

1. What is the difference between soft and hard quotes?
2. Show the output of the following script:

   ```
   #!/bin/sh

   name="Bill Murray"
   printf "$name\n"
   printf '$name\n'
   printf $name\n
   ```

2.13 User Input

In Bourne Shell derivatives, data can be input from the standard input using the **read** command:

```
#!/usr/bin/env bash

printf "Please enter your name: "
read name
printf "Hello, $name!\n"
```

C shell and T-shell use a symbol rather than a command to read input:

```
#!/bin/csh

printf "Please enter your name: "
set name="$<"
printf "Hello, $name!\n"
```

The $< symbol behaves like a variable, which makes it more flexible than the **read** command used by Bourne family shells. It can be used anywhere a regular variable can appear.

```
#!/bin/csh

printf "Enter your name: "
printf "Hi, $<!\n"
```

Note The $< symbol should always be enclosed in soft quotes in case the user enters text containing white space.

Practice Break

Write a shell script that asks the user to enter their first name and last name, stores each in a separate shell variable, and outputs "Hello, first-name last-name".

```
Please enter your first name: Barney
Please enter your last name: Miller
Hello, Barney Miller!
```

2.13.1 Self-test

1. Do the practice break in this section if you haven't already.

2.14 Conditional Execution

Sometimes we need to run a particular command or sequence of commands only if a certain condition is true.

For example, if program B processes the output of program A, we probably won't want to run B at all unless A finished successfully.

2.14.1 Command Exit Status

Conditional execution in Unix shell scripts often utilizes the *exit status* of the most recent command.

All Unix programs return an exit status. It is not possible to write a Unix program that does not return an exit status. Even if the programmer neglects to explicitly return a value, the program will return a default value.

By convention, programs return an exit status of 0 if they determine that they completed their task successfully and a variety of non-zero error codes if they failed. There are some standard error codes defined in the C header file sysexits.h. You can learn about them by running **man sysexits**.

We can check the exit status of the most recent command by examining the shell variable $? in Bourne shell family shells or $status in C shell family shells.

```
bash> ls
myprog.c
bash> echo $?
0
bash> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuwx1] [-D format] [file ...]
bash> echo $?
1
bash>
```

```
tcsh> ls
myprog.c
tcsh> echo $status
0
tcsh> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuwx1] [-D format] [file ...]
tcsh> echo $status
1
tcsh>
```

Practice Break
Run several commands correctly and incorrectly and check the $? or $status variable after each one.

2.14.2 If-then-else Statements

All Unix shells have an if-then-else construct implemented as internal commands. The Bourne shell family of shells all use the same basic syntax. The C shell family of shells also use a common syntax, which is somewhat different from the Bourne shell family.

Bourne Shell Family

The general syntax of a Bourne shell family if statement is shown below. Note that there can be an unlimited number of elifs, but we will use only one for this example.

```
#!/bin/sh

if command1
then
    command
    command
    ...
elif command2
then
    command
    command
    ...
else
    command
    command
    ...
fi
```

Note
The 'if' and the 'then' are actually two separate commands, so they must either be on separate lines as shown above, or separated by an operator such as ';', which can be used instead of a newline to separate Unix commands.

```
cd; ls
if command; then
```

The if command executes command1 and checks the exit status when it completes.

If the exit status of command1 is 0 (indicating success), then all the commands before the elif are executed, and everything after the elif is skipped.

If the exit status is non-zero, then nothing above the elif is executed. The elif command then executes command2 and checks its exit status.

If the exit status of command2 is 0, then the commands between the elif and the else are executed and everything after the else is skipped.

If the exit status of command2 is non-zero, then everything above the else is skipped and everything between the else and the fi is executed.

Note In Bourne shell if statements, an exit status of zero effectively means 'true' and non-zero means 'false', which is the opposite of C and similar languages.

In most programming languages, we use some sort of Boolean expression (usually a comparison, also known as a relation), not a command, as the condition for an if statement.

This is generally true in Bourne shell scripts as well, but the capability is provided in an interesting way. We'll illustrate by showing an example and then explaining how is works.

Suppose we have a shell variable and we want to check whether it contains the string "blue". We could use the following if statement to test:

```
#!/bin/sh

printf "Enter the name of a color: "
read color

if [ "$color" = "blue" ]; then
    printf "You entered blue.\n"
elif [ "$color" = "red" ]; then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
fi
```

The interesting thing about this code is that the square brackets are *not* Bourne shell syntax. As stated above, the Bourne shell if statement simply executes a command and checks the exit status. This is *always* the case.

The '[' in the condition above is actually an external command! In fact, is simply another name for the **test** command. The files /bin/test and /bin/[are actually the same program file:

```
FreeBSD tocino bacon ~ 401: ls -l /bin/test /bin/[
-r-xr-xr-x  2 root  wheel  8516 Apr  9  2012 /bin/[*
-r-xr-xr-x  2 root  wheel  8516 Apr  9  2012 /bin/test*
```

We could have also written the following:

```
if test "$color" = "blue"; then
```

Hence, "$color", =, "blue", and] are all separate arguments to the [command, and must be separated by white space. If the command is invoked as '[', then the last argument must be ']'. If invoked as 'test', then the ']' is not allowed.

The test command can be used to perform comparisons (relational operations) on variables and constants, as well as a wide variety of tests on files. For comparisons, test takes three arguments: the first and third are string values and the second is a relational operator.

```
# Compare a variable to a string constant
test "$name" = 'Bob'
```

```
# Compare the output of a program directly to a string constant
test `myprog` = 42
```

For file tests, test takes two arguments: The first is a flag indicating which test to perform and the second is the path name of the file or directory.

```
# See if output file exists and is readable to the user
# running test
test -r output.txt
```

The exit status of test is 0 (success) if the test is deemed to be true and a non-zero value if it is false.

```
shell-prompt: test 1 = 1
shell-prompt: echo $?
0
shell-prompt: test 1 = 2
shell-prompt: echo $?
1
```

The relational operators supported by **test** are shown in Table 2.5.

Operator	Relation
=	Lexical equality (string comparison)
-eq	Integer equality
!=	Lexical inequality (string comparison)
-ne	Integer inequality
<	Lexical less-than (10 < 9)
-lt	Integer less-than (9 -lt 10)
-le	Integer less-than or equal
>	Lexical greater-than
-gt	Integer greater-than
-ge	Integer greater-than or equal

Table 2.5: Test Command Relational Operators

Caution

Note that some operators, such as < and >, have special meaning to the shell, so they must be escaped or quoted.

```
test 10 > 9      # Redirects output to a file called '9'.
                 # The only argument sent to the test command is '10'.
                 # The test command issues a usage message since it requires
                 # more arguments.
test 10 \> 9     # Compares 10 to 9.
test 10 '>' 9    # Compares 10 to 9.
```

Caution It is a common error to use '==' with the test command, but the correct comparison operator is '='.

Common file tests are shown in Table 2.6. To learn about additional file tests, run "man test".

Flag	Test
-e	Exists
-r	Is readable
-w	Is writable
-x	Is executable
-d	Is a directory
-f	Is a regular file
-L	Is a symbolic link
-s	Exists and is not empty
-z	Exists and is empty

Table 2.6: Test command file operations

Caution

Variable references in a [or test command should usually be enclosed in soft quotes. If the value of the variable contains white space, such as "navy blue", and the variable is not enclosed in quotes, then "navy" and "blue" will be considered two separate arguments to the [command, and the [command will fail. When [sees "navy" as the first argument, it expects to see a relational operator as the second argument, but instead finds "blue", which is invalid.
Furthermore, if there is a chance that a variable used in a comparison is empty, then we must attach a common string to the arguments on both sides of the operator. It can be almost any character, but '0' is popular and easy to read.

```
name=""
if [ "$name" = "Bob" ]; then    # Error, expands to: if [ = Bob; then
if [ 0"$name" = 0"Bob" ]; then  # OK, expands to: if [ 0 = 0Bob ]; then
```

Relational operators are provided by the test command, not by the shell. Hence, to find out the details, we would run "man test" or "man [", not "man sh" or "man bash".

See Section 2.14.1 for information about using the test command.

Practice Break
Run the following commands in sequence and run 'echo $?' after every test or [command under bash and 'echo $status' after every test or [command under tcsh.

```
bash
test 1 = 1
test 1=2
test 1 = 2
[ 1 = 1
[ 1 = 1 ]
[ 1 = '1' ]
[1=1]
[ 2 < 10 ]
[ 2 = 3 ]
[ 2 \< 10 ]
[ 2 '<' 10 ]
[ 2 -lt 10 ]
[ $name = 'Bill' ]
[ 0$name = 0'Bill' ]
name='Bob'
[ $name = 'Bill' ]
[ $name = Bill ]
[ $name = Bob ]
exit
tcsh
[ $name = 'Bill' ]
[ 0$name = 0'Bill' ]
set name='Bob'
[ $name = 'Bill' ]
which [
exit
```

C shell Family

Unlike the Bourne shell family of shells, the C shell family implements its own operators, so there is generally no need for the test or [command (although you can use it in C shell scripts if you really want to).

The C shell if statement requires () around the condition, and the condition is always a Boolean expression, just like in C and similar languages. As in C, and unlike Bourne shell, a value of zero is considered false and non-zero is true.

```
#!/bin/csh -ef

printf "Enter the name of a color: "
set color = "$<"

if ( "$color" == "blue" ) then
    printf "You entered blue.\n"
else if ( "$color" == "red" ) then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
endif
```

The C shell relational operators are shown in Table 2.7.

Note C shell does not directly support string comparisons except for equality and inequality. To see if a string is lexically less-than or greater than another, use the test or [command with <, <=, >, or >=.

Operator	Relation
<	Integer less-than
>	Integer greater-than
<=	Integer less-then or equal
>=	Integer greater-than or equal
==	String equality
!=	String inequality
=~	String matches glob pattern
!~	String does not match glob pattern

Table 2.7: C Shell Relational Operators

Conditions in a C shell if statement do not have to be relations, however. We can check the exit status of a command in a C shell if statement using the { } operator:

```
#!/bin/csh

if ( { command } ) then
    command
    ...
endif
```

The { } essentially inverts the exit status of the command. If command returns 0, the the value of { command } is 1, which means "true" to the if statement. If command returns a non-zero status, then the value of { command } is zero.

C shell if statements also need soft quotes around strings that contain white space. However, unlike the test command, it can handle empty strings, so we don't need to add an arbitrary prefix like '0' if the string may be empty.

```
if [ 0"$name" = 0"Bob" ]; then
```

```
if ( "$name" == "Bob" ) then
```

Practice Break

Type in the following commands in sequence:

```
tcsh
if ( $first_name == Bob ) then
    printf 'Hi, Bob!\n'
endif
set first_name=Bob
if ( $first_name == Bob ) then
    printf 'Hi, Bob!\n'
endif
exit
```

2.14.3 Conditional Operators

The shell's conditional operators allow us to alter the exit status of a command or utilize the exit status of each in a sequence of commands. They include the Boolean operators AND (&&), OR (||), and NOT (!).

```
shell-prompt: command1 && command2
shell-prompt: command1 || command2
shell-prompt: ! command
```

Operator	Meaning	Exit status
! command	NOT	0 if command failed, 1 if it succeeded
command1 && command2	AND	0 if both commands succeeded
command1 || command2	OR	0 if either command succeeded

Table 2.8: Shell Conditional Operators

Note that in the case of the && operator, command2 will not be executed if command 1 fails (exits with non-zero status), since it could not change the exit status. Once any command in an && sequence fails, the exit status of the whole sequence will be 1, so no more commands will be executed.

Likewise in the case of a || operator, once any command succeeds (exits with zero status), the remaining commands will not be executed.

This fact is often used to conditionally execute a command only if another command is successful:

```
pre-processing && main-processing && post-processing
```

When using the test or [commands, multiple tests can be performed using either the shell's conditional operators or the test command's Boolean operators:

```
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]; then
if test 0"$first_name" = 0"Bob" && test 0"$last_name" = 0"Newhart"; then
```

```
if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]; then
if test 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart"; then
```

The latter is probably more efficient, since it only executes a single [command, but efficiency in shell scripts is basically a lost cause, so it's best to aim for readability instead. If you want speed, use a compiled language.

Conditional operators can also be used in a C shell if statement. Parenthesis are recommended around each relation for readability.

```
if ( ("$first_name" == "Bob") && ("$last_name" == "Newhart") ) then
```

Practice Break
Run the following commands in sequence and run 'echo $?' after every command under bash and 'echo $status' after every command under tcsh.

```
bash
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
first_name=Bob
last_name=Newhart
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]
then
    printf 'Hi, Bob!\n'
fi
if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]
then
    printf 'Hi, Bob!\n'
fi
exit
tcsh
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!\n'
endif
set first_name=Bob
set last_name=Nelson
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!\n'
endif
set last_name=Newhart
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!\n'
endif
exit
```

2.14.4 Case and Switch Statements

If you need to compare a single variable to many different values, you could use a long string of elifs or else ifs:

```
#!/bin/sh

printf "Enter a color name: "
read color

if [ "$color" = "red" ] || \
   [ "$color" = "orange" ]; then
    printf "Long wavelength\n"
elif [ "$color" = "yello" ] || \
     [ "$color" = "green" ] || \
     [ "$color" = "blue" ]; then
    printf "Medium wavelength\n"
elif [ "$color" = "indigo" ] || \
     [ "$color" = "violet" ]; then
```

```
    printf "Short wavelength\n"
else
    printf "Invalid color name: $color\n"
fi
```

Like most languages, however, Unix shells offer a cleaner solution.

Bourne shell has the case statement:

```
#!/bin/sh

printf "Enter a color name: "
read color

case $color in
    red|orange)
        printf "Long wavelength\n"
        ;;
    yellow|green|blue)
        printf "Medium wavelength\n"
        ;;
    indigo|violet)
        printf "Short wavelength\n"
        ;;
    *)
        printf "Invalid color name: $color\n"
        ;;
esac
```

C shell has a switch statement that looks almost exactly like the switch statement in C, C++, and Java:

```
#!/bin/csh -ef

printf "Enter a color name: "
set color = "$<"

switch($color)
case    red:
case    orange:
    printf "Long wavelength\n"
    breaksw
case    yellow:
case    green:
case    blue:
    printf "Medium wavelength\n"
    breaksw
case    indigo:
case    violet:
    printf "Short wavelength\n"
    breaksw
default:
    printf "Invalid color name: $color\n"
endsw
```

Note The ;; and breaksw statements cause a jump to the first statement after the entire case or switch. The ;; is required after every value in the case statement. The breaksw is optional in the switch statement. If omitted, the script will simply continue on and execute the statements for the next case value.

2.14.5 Self-test

1. What is an "exit status"? What conventions to Unix programs follow regarding the exit status?
2. How can we find out the exit status of the previous command in Bourne shell? In C shell?
3. Write a Bourne shell script that uses an if statement to run 'ls -l > output.txt' and view the output using 'more' only of the ls command succeeded.
4. Repeat the previous problem using C shell.
5. Write a Bourne shell script that inputs a first name and outputs a different message depending on whether the name is 'Bob'.

```
shell-prompt: ./script
What is your name? Bob
Hey, Bob!
shell-prompt: ./script
What is your name? Bill
Hey, you're not Bob!
```

6. Repeat the previous problem using C shell.
7. Write a Bourne and/or C shell script that inputs a person's age and indicates whether they get free peanuts. Peanuts are provided to senior citizens.

```
shell-prompt: ./script
How old are you? 42
Sorry, no peanuts for you.
shell-prompt: ./script
How old are you? 72
Have some free peanuts, wise sir!
```

8. Why is it necessary to separate the tokens between [and] with white space? What will happen if we don't?
9. What will happen if a value being compared using test or [contains white space? How to we remedy this?
10. What will happen if a value being compared using test or [is an empty string? How to we remedy this?
11. Why do the < and > operators need to be escaped (\<, \>) or quoted when used with the test command?
12. What is the == operator used for with the test command?
13. How do we check the exit status of a command in a C shell if statement?
14. Write a Bourne and/or C shell script that inputs a person's age and indicates whether they get free peanuts. Peanuts are provided to senior citizens and children between the ages of 3 and 12.

```
shell-prompt: ./script
How old are you? 42
Sorry, no peanuts for you.
shell-prompt: ./script
How old are you? 72
Have some free peanuts, wise sir!
```

15. Write a Bourne shell script that uses conditional operators to run 'ls -l > output.txt' and view the output using 'more' only of the ls command succeeded.
16. Write a shell script that checks the output of **uname** using a case or switch statement and reports whether the operating system is supported. Assume supported operating systems include Cygwin, Darwin, FreeBSD, and Linux.

```
shell-prompt: ./script
FreeBSD is supported.
```

```
shell-prompt: ./script
AIX is not supported.
```

2.15 Loops

We often need to run the same command or commands on a group of files or other data.

2.15.1 For and Foreach

Unix shells offer a type of loop that takes an enumerated list of string values, rather than counting through a sequence of numbers. This makes it more flexible for working with sets of files or or arbitrary sets of values.

This type of loop is well suited for use with globbing (file name patterns using wild cards, as discussed in Section 1.7.5):

```
#!/usr/bin/env bash

# Process input-1.txt, input-2.txt, etc.
for file in input-*.txt
do
    ./myprog $file
done
```

```
#!/bin/csh -ef

# Process input-1.txt, input-2.txt, etc.
foreach file (input-*.txt)
    ./myprog $file
end
```

These loops are not limited to using file names, however. We can use them to iterate through any list of string values:

```
#!/bin/sh

for fish in flounder gobie hammerhead manta moray sculpin
do
    printf "%s\n" $fish
done
```

```
#!/usr/bin/env bash

for c in 1 2 3 4 5 6 7 8 9 10
do
    printf "%d\n" $c
done
```

To iterate through a list of integers too long to type out, we can utilize the **seq** command, which takes a starting value, optionally an increment value, and an ending value. We use shell output capture (Section 2.11.4) to represent the output of the **seq** command as a string in the script:

```
#!/bin/sh -e

# Count from 0 to 1000 in increments of 5
for c in $(seq 0 5 1000); do
    printf "%d\n" $c
done
```

```
#!/bin/csh

foreach c (`seq 0 5 1000`)
    printf "%s\n" $c
end
```

The **seq** can also be used to embed integer values in a non-integer list:

```
#!/bin/sh -e

# Process all human chromosomes
for chromosome in $(seq 1 22) X Y; do
    printf "chr%s\n" $chromosome
done
```

Practice Break
Type in and run the fish example above.

Note Note again that the Unix commands, including the shell, don't generally care whether their input comes from a file or a device such as the keyboard. Try running the fish example by typing it directly at the shell prompt as well as by writing a script file. When running it directly, be sure to use the correct shell syntax for the interactive shell you are running.

Example 2.3 Multiple File Downloads

Often we need to download many large files from another site. This process would be tedious to do manually: Start a download, wait for it to finish, start another... There may be special tools provided, but often they are unreliable or difficult to install. In many cases, we may be able to automate the download using a simple script and a file transfer tool such as **curl**, **fetch**, **rsync** or **wget**.

The model scripts below demonstrate how to download a set of files using curl. The local file names will be the same as those on the remote site and if the transfer is interrupted for any reason, we can simply run the script again to resume the download where it left off.

Depending on the tools available on your local machine and the remote server, you may need to substitute another file transfer program for curl.

```
#!/bin/sh -e

# Download genome data from the ACME genome project
site=http://server.with.my.files/directory/with/my/files
for file in frog1 frog2 frog3 toad1 toad2 toad3; do
    printf "Fetching $site/$file.fasta.gz...\n"

    # Use filename from remote site and try to resume interrupted
    # transfers if a partial download already exists
    curl --continue-at - --remote-name $site/$file.fasta.gz
fi
```

```
#!/bin/csh -ef

# Download genome data from the ACME genome project
set site=http://server.with.my.files/directory/with/my/files
foreach file (frog1 frog2 frog3 toad1 toad2 toad3)
    printf "Fetching $site/$file.fasta.gz...\n"

    # Use filename from remote site and try to resume interrupted
    # transfers if a partial download already exists
    curl --continue-at - --remote-name $site/$file.fasta.gz
end
```

2.15.2 While Loops

A for or foreach loop is only convenient for iterating through a fixed set of values. Sometimes we may need to terminate a loop based on inputs that are unknown when the loop begins, or values computed over the course of the loop.

The **while** loop is a more general loop that iterates as long as some condition is true. It uses the same types of expressions as an **if** statement.

The while loop is often used to iterate through long integer sequences:

```
#!/usr/bin/env bash

c=1
while [ $c -le 100 ]
do
    printf "%d\n" $c
    c=$(($c + 1))         # (( )) encloses an integer expression
done
```

Note again that the [above is an external command, as discussed in Section 2.14.1.

```
#!/bin/csh -ef

set c = 1
while ( $c <= 100 )
    printf "%d\n" $c
    @ c = $c + 1         # @ is like set, but indicates an integer expression
end
```

Practice Break
Type in and run the script above.

While loops can also be used to iterate until an input condition is met:

```
#!/bin/sh

continue=''
while [ 0"$continue" != 0'y' ] && [ 0"$continue" != 0'n' ]; do
    printf "Would you like to continue? (y/n) "
    read continue
done
```

```
#!/bin/csh -ef

set continue=''
while ( ("$continue" != 'y') && ("$continue" != 'n') )
    printf "Continue? (y/n) "
    set continue="$<"
end
```

Practice Break
Type in and run the script above.

We may even want a loop to iterate forever. This is often useful when using a computer to collect data at regular intervals:

```
#!/bin/sh

# 'true' is an external command that always returns an exit status of 0
```

```
while true; do
    sample-data     # Read instrument
    sleep 10        # Pause for 10 seconds without using any CPU time
done
```

```
#!/bin/csh -ef

while ( 1 )
    sample-data     # Read instrument
    sleep 10        # Pause for 10 seconds without using any CPU time
end
```

2.15.3 Self-test

1. Write a shell script that prints the square of every number from 1 to 100.
2. Write a shell script that sorts all files with names ending in ".txt" one at a time, removes duplicate entries, and saves the output to `filename.txt.sorted`. The script then merges all the sorted text into a single file called `combined.txt.sorted`. The **sort** can also merge presorted files when used with the `-m` flag.

   The standard Unix **sort** can be used to sort an individual file. The **uniq** command will remove duplicate lines that are adjacent to each other. (Hence, the data should be sorted already.)
3. Do the examples for shell loops above give you any ideas about using multiple computers to speed up processing?

2.16 Generalizing Your Code

All programs and scripts require input in order to be useful.

Inputs commonly include things like scalar parameters to use in equations and the names of files containing more extensive data such as a matrix or a database.

2.16.1 Hard-coding: Failure to Generalize

All too often, inexperienced programmers provide what should be input to a program by hard-coding values and file names into their programs and scripts:

```
#!/bin/csh

# Hard-coded values 1000 and output.txt
calcpi 1000 > output.txt
```

Many programmers will then make another copy of the program or script with different constants or file names in order to do a different run. The problem with this approach should be obvious. It creates a mess of many nearly identical programs or scripts, all of which have to be maintained together. If a bug is found in one of them, then all of them have to be checked and corrected for the same error.

2.16.2 Generalizing with User Input

A better approach is to take these values as input:

```
#!/bin/csh

printf "How many iterations? "
set iterations = "$<"
printf "Output file? "
set output_file = "$<"

calcpi $iterations > $output_file
```

If you don't want to type in the values every time you run the script, you can put them in a separate input file, such as "input-1000.txt" and use redirection:

```
shell-prompt: cat input-1000.txt
1000
output-1000.txt
shell-prompt: calcpi-script < input-1000.txt
```

This way, if you have 50 different inputs to try, you have 50 input files and only one script to maintain instead of 50 scripts.

2.16.3 Generalizing with Command-line Arguments

Another approach is to design the script so that it can take command-line arguments, like most Unix commands. Using command-line arguments is quite simple in most scripting and programming languages.

In all Unix shell scripts, the first argument is denoted by the special variable $1, the second by $2, and so on.

$0 refers to the name of the command as it was invoked.

Bourne Shell Family

In Bourne Shell family shells, we can find out how many command-line arguments were given by examining the special shell variable "$#". This is most often used to verify that the script was invoked with the correct number of arguments.

```
#!/bin/sh

# If invoked incorrectly, tell the user the correct way
if [ $# != 2 ]; then
    printf "Usage: $0 iterations output-file\n"
    exit 1
fi

# Assign to named variables for readability
iterations=$1
output_file="$2"    # File name may contain white space!

calcpi $iterations > "$output_file"
```

```
shell-prompt: calcpi-script
Usage: calcpi-script iterations output-file
shell-prompt: calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494
```

C shell Family

In C shell family shells, we can find out how many command-line arguments were given by examining the special shell variable "$#argv".

```
#!/bin/csh

# If invoked incorrectly, tell the user the correct way
if ( $#argv != 2 ) then
    printf "Usage: $0 iterations output-file\n"
    exit 1
endif

# Assign to named variables for readability
set iterations=$1
set output_file="$2"    # File name may contain white space!

calcpi $iterations > "$output_file"
```

```
shell-prompt: calcpi-script
Usage: calcpi-script iterations output-file
shell-prompt: calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494
```

2.16.4 Self-test

1. Modify the following shell script so that it takes the file name of the dictionary and the sample word as user input instead of hard-coding it. You may use any shell you choose.

   ```
   #!/bin/sh

   if fgrep 'abacus' /usr/share/dict/words; then
       printf 'abacus is a real word.\n'
   else
       printf 'abacus is not a real word.\n'
   fi
   ```

2. Repeat the above exercise, but use command-line arguments instead of user input.

2.17 Scripting an Analysis Pipeline

2.17.1 What's an Analysis Pipeline?

An analysis pipeline is simply a sequence of processing steps.

Since the steps are basically the same for a given type of analysis, we can automate the pipeline using a scripting language for the reasons we discussed at the beginning of this chapter: To save time and avoid mistakes.

A large percentage of scientific research analyses require multiple steps, so pipelines are very common in practice.

2.17.2 Where do Pipelines Come From?

It has been said that for every PhD thesis, there is a pipeline.

There are many preexisting pipelines available for a wide variety of tasks.

Many such pipelines were developed by researchers for a specific project, and then generalized in order to be useful for other projects or other researchers.

Unfortunately, most such pipelines are not well designed or rigorously tested, so they don't work well for analyses that differ significantly from the one for which they were originally designed.

Another problem is that most of them are not well maintained over the long term. Developers set out with good intentions to help other researchers, but once their project is done and they move onto new things, they find that they don't have time to maintain old pipelines anymore. Also, new tools are constantly evolving and old pipelines therefore quickly become obsolete unless they are aggressively updated.

Finally, many pipelines are integrated into a specific system with a graphical user interface (GUI) or web interface, and therefore cannot be used on a generic computer or HPC cluster (unless the entire system is installed and configured, which is often difficult or impossible).

For these reasons, every researcher should know how to develop their own pipelines. Relying on the charity of your competitors for publishing space and grant money will not lead to long-term success.

This is especially true for long-term studies. If you become dependent on a preexisting pipeline early on, and it is not maintained by its developers for the duration of *your* study, then the completion of your study will prove very difficult.

2.17.3 Implementing Your Own Pipeline

A pipeline can be implemented in any programming language.

Since most pipelines involve simply running a series of programs with the appropriate command-line arguments, a Unix shell script is a very suitable choice in most cases.

In some cases, it may be possible to use Unix shell pipes to perform multiple steps at the same time. This will depend on a number of things:

- Do the processing programs use standard input and standard output? If not, then redirecting to and from them with pipes will not be possible.
- What are the resource requirements of each step? Do you have enough memory to run multiple steps at the same time?
- Do you need to save the intermediate files generated by some of the steps? If so, then either don't use a Unix shell pipe, or use the **tee** command to dump output to a file and pipe it to the next command at the same time.

```
shell-prompt: step1 < input1 | tee output1 | step2 > output2
```

2.17.4 An Example Genomics Pipeline

Below is a simple shell script implementation of the AmrPlusPlus pipeline, which, according to their website, is used to "characterize the content and relative abundance of sequences of interest from the DNA of a given sample or set of samples".

People can use this pipeline by uploading their data to the developer's website, or by installing the pipeline to their own Docker container or Galaxy server.

In reality, the analysis is performed by the following command-line tools, which are developed by other parties and freely available:

- Trimmomatic
- BWA
- Samtools
- SNPFinder
- ResistomeAnalyzer
- RarefactionAnalyzer

The role of AmrPlusPlus is to coordinate the operation of these tools. AmrPlusPlus is itself a script.

If you don't want to be dependent on their web service, a Galaxy server, or their Docker containers, or if you would like greater control over and understanding of the analysis pipeline, or if you want to use the newer versions of tools such as samtools, you can easily write your own script to run the above commands.

Also, when developing our own pipeline, we can substitute other tools that perform the same function, such as Cutadapt in place of Trimmomatic, or Bowtie (1 or 2) in place of BWA for alignment.

All of these tools are designed for "short-read" DNA sequences (on the order of 100 base pair per fragment). When we take control of the process rather than rely on someone else's pipeline, we open the possibility of developing an analogous pipeline using a different set of tools for "long-read" sequences (on the order of 1000 base pair per fragment).

For our purposes, we install the above commands via FreeBSD ports and/or pkgsrc (on CentOS and Mac OS X). To facilitate this, I created a metaport that automatically installs all the tools needed by the pipeline (Trimmomatic, BWA, ...) as dependencies. The following will install everything in a few minutes:

```
shell-prompt: cd /usr/ports/wip/amr-cli
shell-prompt: make install
```

Then we just write a Unix shell script to implement the pipeline for our data.

Note that this is a real pipeline used for research at the UWM School of Freshwater Science.

It is not important whether you understand genomics analysis for this example. Simply look at how the script uses loops and other scripting constructs to see how the material you just learned can be used in actual research. I.e., don't worry about what **cutadapt** and **bwa** are doing with the data. Just see how they are run within the pipeline script, using redirection, command line arguments, etc. Also read the comments within the script for a deeper understanding of what the conditionals and loops are doing.

```
#!/bin/sh -e

# Get gene fraction threshold from user
printf "Resistome threshold? "
read threshold

################################################################################
# 1. Enumerate input files

raw_files="SRR*.fastq"

################################################################################
# 2.  Quality control: Remove adapter sequences from raw data

for file in $raw_files; do
    output_file=trimmed-$file
    # If the output file already exists, assume cutadapt was already run
    # successfully.  Remove trimmed-* before running this script to force
    # cutadapt to run again.
    if [ ! -e $output_file ]; then
        cutadapt $file > $output_file
    else
        printf "$raw already processed by cutadapt.\n"
    fi
done

################################################################################
# 3. If sequences are from a host organism, remove host dna

# Index resistance gene database
if [ ! -e megares_database_v1.01.fasta.ann ]; then
    bwa index megares_database_v1.01.fasta
fi
```

```
################################################################################
# 4. Align to target database with bwa mem.

for file in $raw_files; do
    # Output is an aligned sam file.  Replace trimmed- prefix with aligned-
    # and replace .fastq suffix with .sam
    output_file=aligned-${file%.fastq}.sam
    if [ ! -e $output_file ]; then
        printf "\nRunning bwa-mem on $file...\n"
        bwa mem megares_database_v1.01.fasta trimmed-$file > $output_file
    else
        printf "$file already processed by bwa mem\n"
    fi
done

################################################################################
# 5. Resistome analysis.

aligned_files=aligned-*.sam
for file in $aligned_files; do
    if [ ! -e ${file%.sam}group.tsv ]; then
        printf "\nRunning resistome analysis on $file...\n"
        resistome -ref_fp megares_database_v1.01.fasta -sam_fp $file \
            -annot_fp megares_annotations_v1.01.csv \
            -gene_fp ${file%.sam}gene.tsv \
            -group_fp ${file%.sam}group.tsv \
            -class_fp ${file%.sam}class.tsv \
            -mech_fp ${file%.sam}mech.tsv \
            -t $threshold
    else
        printf "$file already processed by resistome.\n"
    fi
done

################################################################################
# 6. Rarefaction analysis?
```

I generally write a companion to every analysis script to remove output files and allow a fresh start for the next attempt:

```
#!/bin/sh -e

rm -f trimmed-* aligned-* aligned-*.tsv megares*.fasta.*
```

2.18 Functions and Calling other Scripts

Most scripts tend to be short, but even a program of 100 lines long can benefit from being broken down and organized into modules.

The Bourne family shells support simple functions for this purpose.

C shell family shells do not support separate functions within a script, but this does not mean that they cannot be modularized. A C shell script can, of course, run other scripts and these separate scripts can serve the purpose of subprograms.

Some would argue that separate scripts are more modular than functions, since a separate script is inherently available to any script that could use it, whereas a function is confined to the script that contains it.

Another advantage of using separate scripts is that they run as a separate process, so they have their own set of shell and environment variables. Hence, they do not have side-effects on the calling script. Bourne shell functions, on the other hand, can modify "global" variables and impact the subsequent behavior of other functions or the main program.

There are some functions that are unlikely to be useful in other scripts, however, and Bourne shell functions are convenient in these cases. Also, it is generally easy to convert a Bourne shell function into a separate script, so there isn't generally much to lose by using a function initially.

2.18.1 Bourne Shell Functions

A Bourne shell function is defined by simply writing a name followed by parenthesis, and a body between curly braces on the lines below:

```
name()
{
    commands
}
```

We call a function the same way we run any other command.

```
#!/bin/sh

line()
{
    printf '------------------------------------------------------------\n'
}

line
```

I we pass arguments to a function, then the variables $1, $2, etc. in the function will be set to the arguments passed to the function. Otherwise, $1, $2, etc. will be the arguments to the main script.

```
#!/bin/sh

print_square()
{
    printf $(($1 * $1))
}

c=1
while [ $c -le 10 ]; do
    printf "%d squared = %d\n" $c `print_square c`
    c=$((c + 1))
done
```

The return statement can be used to return a value to the caller. The return value is received by the caller in $?, just like the exit status of any other command. This is most often used to indicate success or failure of the function.

```
#!/bin/sh

myfunc()
{
    if ! command1; then
        return 1

    if ! command2; then
        return 1

    return 0
}

if ! myfunc; then
    exit 1
fi
```

We can define local variables in a function if we do not want the function to modify a variable outside itself.

```
#!/bin/sh

pause()
{
    local response

    printf "Press return to continue..."
    read response
}

pause
```

2.18.2 C Shell Separate Scripts

Since C shell does not support internal functions, we implement subprograms as separate scripts.

Each script is executed by a separate shell process, so all variables are essentially local to that script.

We can, of course, use the **source** to run another script using the parent shell process as described in Section 2.7. In this case, it will affect the shell and environment variables of the calling script. This is usually what we intend and the very reason for using the **source** command.

When using separate scripts as subprograms, it is especially helpful to place the scripts in a directory that is in your PATH. Most users use a directory such as ~/bin or ~/scripts for this purpose.

2.18.3 Self-test

2.19 Alias

An alternative to functions and separate scripts for very simple things is the **alias** command.

This command creates an alias, or alternate name for another command.

Aliases are supported by both Bourne and C shell families, albeit with a slightly different syntax.

They are most often used to create simple shortcuts for common commands.

In Bourne shell and derivatives, the new alias is followed by an '='. Any command containing white space must be enclosed in quotes, or the white space must be escaped with a \.

```
#!/bin/sh

alias dir='ls -als'

dir
```

C shell family shells use white space instead of an '=' and do not require quotes around commands containing white space.

```
#!/bin/csh

alias dir ls -als

dir
```

An alias can contain multiple commands, but in this case it must be enclosed in quotes, even in C shell.

```
#!/bin/csh

# This will not work:
# alias pause printf "Press return to continue..."; $<
#
# It is the same as:
#
# alias pause printf "Press return to continue..."
# $<

# This works
alias pause 'printf "Press return to continue..."; $<'

pause
```

2.20 Shell Flags and Variables

Unix shells have many command line flags to control their behavior.

One of the most popular shell flags is -e. The -e flag in both Bourne Shell and C shell cause the shell to exit if any command fails. This is almost always a good idea, to avoid wasting time and so that the last output of a script shows any error messages from the failed command.

Flags can be used in the shebang line if the path of the shell is fixed. When using #!/usr/bin/env, we must set the option using a separate command, because the shebang line on some systems treats everything after '#!/usr/bin/env' as a single argument to the env command.

```
#!/bin/sh -e

# The shebang line above is OK
```

```
#!/bin/csh -e

# The shebang line above is OK
```

```
#!/usr/bin/env bash -e

# The shebang line above is invalid on some systems and may cause
# an error such as "bash -e: command not found"
```

We can get around this in Bourne family shells using the set command, which can be used to turn on or off command-line flags within the script. For example, "set -e" in a script causes the shell running the script to terminate if any subsequent commands fail. A "set +e" turns off this behavior.

```
#!/usr/bin/env bash

# Enable exit-on-error
set -e
```

Unfortunately, C shell family shells do not have anything comparable to the Bourne shell set command. Recall that C shell has a set command, but it is use to set shell variables, not command-line flags.

Many features controlled by command-line flags can also be set within a C shell script using special shell variables, but -e is not one of them.

The -x flag is another flag common to both Bourne Shell and C shell. If causes the shell to **echo** commands to the standard output before executing them, which is often useful in debugging a script that it failing at an unknown location.

```
#!/bin/sh -x
```

```
#!/bin/csh -x
```

```
#!/bin/sh

set -x      # Enable command echo
command
command
set +x      # Disable command echo
```

```
#!/bin/csh

set echo    # Enable command echo
command
command
unset echo  # Disable command echo
```

As stated in Section 2.6, Bourne shell family scripts do not source any start up scripts by default. Bourne shells only source files like .shrc, .bashrc, etc. if the shell is interactive, i.e. the standard input is a terminal.

C shell and T shell scripts, on the other hand, will source .cshrc or .tcshrc by default. This behavior can be disabled using the `-f` flag. Disabling this is usually a good idea, since the script may behave differently for different people, depending on what's in their .cshrc.

```
#!/bin/csh -ef
```

There are many other command-line flags and corresponding C shell variables. For more information, run "man sh", "man csh", etc.

2.21 Arrays

Bourne shell does not support arrays, but some commands can process strings containing multiple words separated by white space.

```
#!/bin/sh

names="Barney Bert Billy Bob Brad Brett Brody"
for name in $names; do
    ...
done
```

C shell supports basic arrays. One advantage of this is that we can create lists of strings where some of the elements contain white space.

An array constant is indicated by a list enclosed in parenthesis.

Each array element is identified by an integer subscript.

We can also use a range of subscripts, separated by '-'.

```
#!/bin/csh -ef

set names=("Bob Newhart" "Bob Marley" "Bobcat Goldthwait")
set c=1
while ( $c <= $#names )
    printf "$names[$c]\n"
    @ c++
end

printf "$names[2-3]\n"
```

Caution The foreach command is not designed to work with arrays. It is designed to break a string into white space-separated tokens. Hence, given an array, foreach will view is as one large string and then break it wherever there is white space, which could break individual array elements into multiple pieces.

The $argv variable containing command-line arguments is an array. Hence, the $#argv variable is not special to $argv, but just another example of referencing the number of elements in an array.

2.22 Good and Bad Practices

A very common and very bad practice in shell scripting is checking the wrong information to make decisions. One of the most common ways this bad approach is used involves making assumptions based on the operating system in use.

Take the following code segment, for example:

```
if [ `uname` == 'Linux' ]; then
    compiler='gcc'
    endian='little'
fi
```

Both of the assumptions made about Linux in the code above were taken from real examples!

Setting the compiler to 'gcc' because we're running on Linux is wrong, because Linux can run other compilers such as clang or icc. Compiler selection should be based on the user's wishes or the needs of the program being compiled, not the operating system alone.

Assuming the machine is little-endian is wrong because Linux runs on a variety of CPU types, some of which are big-endian. The user who wrote this code probably assumed that if the computer is running Linux, it must be a PC with an x86 processor, which is not a valid assumption.

There are simple ways to find out the actual endianness of a system, so why would we instead try to infer it from an unrelated fact?? We should instead use something like the open source **endian** program, which runs on any Unix compatible system.

```
if [ `endian` == 'little' ]; then

fi
```

2.23 Here Documents

We often want to output multiple lines of text from a script, for instance to provide detailed instructions to the user. For instance, the output below is a real example from a script that generates random passphrases.

```
========================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed.  Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
========================================================================
```

We could output this text using six printf statements. This would be messy, though, and would require quotes around each line of text.

We could also store it in a separate file and display it with the **cat** command:

```
#!/bin/sh -e

cat instructions.txt
```

This would mean keeping track of multiple files, however.

A "here document", or "heredoc", is another form of redirection that is typically only used in scripts. It essentially redirects the standard input to a portion of the script itself. The general for is as follows:

```
command << end-of-document-marker

end-of-document-marker
```

The end-of-document-marker can be any arbitrary text that you choose. This allows the text to contain literally anything. You simply have to choose a marker that it not in the text you want to display. Common markers are EOM (end of message) or EOF (end of file).

Heredocs can be used with any Unix command that reads from standard input, but are most often used with the **cat** command:

```
#!/bin/sh -e

cat << EOM
================================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed.  Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
================================================================================
EOM
```

Heredocs can also be used to create files from a template that uses shell or environment variables. Any variable references that appear within the text of a heredoc will be expanded. The output of any command reading from a heredoc can, of course, be redirected to a file or other device.

```
#!/bin/csh -ef

# Generate a series of test input files with difference ending values
foreach end_value (10 100 1000 10000)
    foreach tolerance (0.0001 0.0005 0.001)
        cat << EOM > test-input-$end_value-$tolerance.txt
start_value=1
end_value=$end_value
tolerance=$tolerance
EOM
    end
end
```

2.24 Common Unix Tools Used in Scripts

It is often said that most Unix users don't need to write programs. The standard Unix commands contain all the functionality that a typical user needs, so they need only learn how to use the commands and write simple scripts to utilize them.

The sections below introduce some of the popular tools with the sole intention of raising awareness. The details of these tools would fill a separate book by themselves, so we will focus on simple, common examples here.

2.24.1 Grep

The **grep** command, short for General Regular exPressions, is a powerful tool for searching the content of text files.

Regular expressions are a standardized syntax for specifying patterns of text. They are similar to the globbing patterns discussed in Section 1.7.5, but the details are quite different. Also, while globbing patterns are meant to match file names, regular expressions are meant to match strings in any context.

Some of the more common regular expression features are shown in Table 2.9.

Token	Matches
.	Any character
[list]	Any single character in list
[first-last]	Any single character between first and last, in the order they appear in the character set in use. This may be affected by locale settings.
*	Zero or more of the preceding token
+	One or more of the preceding token

Table 2.9: Common Regular Expression Symbols

Note To match any special character, such as '.', or '[', precede it with a '\'.

On BSD systems, a POSIX regular expression reference is available via **man re_format**.

On Linux systems, a similar document is available via **man 7 regex**.

Regular expression pattern matching can be used in any language. At the shell level, patterns are typically matched using the **grep** command.

In short, **grep** searches a text file for patterns specified as arguments and prints matching lines.

```
grep pattern file-spec
```

Note Patterns passed to **grep** should usually be hard-quoted to prevent the shell from interpreting them as globbing patterns or other shell features.

```
# Show lines in Bourne shell scripts containing the string "printf"
grep printf *.sh

# Show lines in C programs containing strings that qualify as variable names
grep '[A-Za-z_][A-Za-z_0-9]*' *.c

# Show lines in C programs containing decimal integers
grep '[0-9]+' *.c

# Show lines in C programs containing real numbers
grep '[0-9]*\.[0-9]+' *.c
```

By default, the **grep** command follows an older standard for traditional regular expressions, in order to maintain backward compatibility in older scripts.

To enable the newer extended regular expressions, use **grep -E** or **egrep**.

To disable the use of regular expressions and treat each pattern as a fixed string, use **grep -F** or **fgrep**. This is sometimes useful for better performance or to eliminate the need for '\' before special characters.

2.24.2 Stream Editors

Stream editors are a class of programs that take input from one stream, often standard input, modify it in some way, and send the output to another stream, often standard output.

The **sed** (Stream EDitor) command is among the most commonly used stream editing programs. The **sed** has a variety of capabilities for performing almost any kind of changes you can imagine. Most often, though, it is used to replace text matching a regular expression with something else. Our introduction here will focus on this feature and we will leave the rest for tutorials dedicated to **sed**.

The basic syntax of a **sed** command for replacing text is as follows:

```
sed -e 's|pattern|replacement|g'
```

The `-e` flag specifies the use of traditional regular expressions. To use the more modern extended regular expressions, use `-E` as with **grep**.

The 's' is the 'substitute' command. Other commands, not discussed here, include 'd' (delete) and 'i' (insert).

The 'l' is a separator. You can use any character as the separator as long as all three separators are the same. This allows any character to appear in the pattern or replacement text. Just use a separator that is not in either. The most popular separators are '|' and '/', since they usually stand out next to typical patterns.

The pattern is a regular expression, just as we would use with **grep**. Again, special characters that we want to match literally must be escaped (preceded by a '\').

The replacement text is not a regular expression, but may contain some special characters specific to **sed**. The most common is '&', which represents the current string matching the pattern. This feature makes it easy to add text to strings matching a pattern, even if they are not the same.

The 'g' means perform a global replacement. If omitted, only the first match on each line is replaced.

```
# Get snooty
sed -e 's|Bob|Robert|g' file.txt > modified-file.txt

# Convert integer constants to long constants in a C program
sed -e 's|[0-9]+|&L|g' prog.c > prog-long.c
```

The **tr** (translate) command is a simpler stream editing tool. It is typically used to replace or delete individual characters from a stream.

```
# Capitalize all occurrences of 'a', 'b', and 'c'
tr 'abc' 'ABC' file.txt > file-caps.txt

# Delete all digits from a file
tr -d '0123456789' file.txt > file-qless.txt
```

2.24.3 Tabular Data Tools

Unix systems provide standard tools for working with tabular data (text data organized in columns).

The **cut** command is a simple tool for removing a portion from each line of a text stream. The user can specify byte, character, or field positions to be removed.

```
# Remove the 3rd and 4th characters from every line
cut -c 3-4 file.txt > chopped-file.txt

# Remove the first column of numbers separated by white space
cut -w -f 1 results.txt > results-without-col1.txt
```

The **awk** command is an extremely sophisticated tool for manipulating tabular data. It is essentially a non-interactive spreadsheet, capable of doing modifications and computations of just about any kind.

Awk includes a scripting language that looks very much like C, with many extensions for easily processing textual data.

Entire books are available on **awk**, so we will focus on just a few basic examples.

Awk is generally invoked in one of two ways. For very simple awk operations (typically 1-line scripts), we can provide the awk script itself as a command-line argument, usually hard-quoted:

```
awk [-F field-separator] 'script' file-spec
```

For more complex, multi-line scripts, it may prove easier to place the awk script in a separate file and refer to it in the command:

```
awk [-F field-separator] -f script.awk file-spec
```

Input is separated into fields by white space by default, but we can specify any field-separator we like using the `-F`. The field separator can also be changed within the awk script by assigning the special variable FS.

Statements within the **awk** script consist of a pattern and an action.

Patterns may be relational expressions comparing a given field (column) to a pattern. In this case, the action will be invoked only on lines matching the pattern.

If pattern is omitted, the action will be performed on every line of input.

The special patterns BEGIN and END are used to perform actions before the first line is processed and after the last line is processed.

The action is essentially a C-like function. If omitted, the default action is to print the entire line matching pattern. (Hence, awk can behave much like grep.)

Example 1: A simple awk command

```
# Print password entries for users with uid >= 1000
shell-prompt: awk -F : '$3 >= 1000 { print $0 }' /etc/passwd
nobody:*:65534:65534:Unprivileged user:/nonexistent:/usr/sbin/nologin
joe:*:4000:4000:Joe User:/home/joe:/bin/tcsh
```

Example 2: A separate awk script

```
# Initialize variables
BEGIN {
    sum1 = sum2 = 0.0;
}

# Add column data to sum for each line
{
    print $1, $2
    sum1 += $1;
    sum2 += $2;
}

# Output sums after all lines are processed
END {
    printf("Sum of column 1 = %f\n", sum1);
    printf("Sum of column 2 = %f\n", sum2);
}
```

```
shell-prompt: cat twocol.txt
4.3     -2.1
5.5     9.0
-7.3    4.6

shell-prompt: awk -f ./sum.awk twocol.txt
4.3 -2.1
5.5 9.0
-7.3 4.6
Sum of column 1 = 2.500000
Sum of column 2 = 11.500000
```

2.24.4 Sort/Uniq

The **sort** command is a highly efficient, general-purpose stream sorting tool. It sorts the input stream line-by-line, optionally prioritizing the sort by one or more columns.

```
shell-prompt: cat names.txt
Kelso Bob
Cox Perry
Dorian John
Turk Christopher
Reid Elliot
Espinosa Carla

# Sort by entire line
shell-prompt: sort names.txt
Cox Perry
Dorian John
Espinosa Carla
Kelso Bob
Reid Elliot
Turk Christopher

# Sort by second column
shell-prompt: sort -k 2 names.txt
Kelso Bob
Espinosa Carla
Turk Christopher
Reid Elliot
Dorian John
Cox Perry

Shell-prompt: cat numbers.txt
45
-12
32
16
7
-12

# Sort sorts lexically by default
Shell-prompt: sort numbers.txt
-12
-12
16
32
45
7

# Sort numerically
Shell-prompt: sort -n numbers.txt
-12
-12
7
16
32
45
```

The **uniq** command eliminates adjacent duplicate lines from the input stream.

```
Shell-prompt: uniq numbers.txt
45
-12
32
16
7
-12
```

```
Shell-prompt: sort numbers.txt | uniq
-12
16
32
45
7
```

2.24.5 Perl, Python, and other Scripting Languages

All of the commands described above are described by the POSIX standard and included with every Unix compatible operating system.

A wide variety of tasks can be accomplished without writing anything more than a shell script utilizing commands like these.

Nevertheless, some Unix users have felt that there is a niche for tools more powerful than shells scripts and standard Unix commands, but more convenient than general-purpose languages like C, Java, etc.

As a result, a new class of scripting languages has evolved that are somewhat more like general-purpose languages. Among the most popular are TCL, Perl, PHP, Python, Ruby, and Lua.

These are interpreted languages, so performance is much slower than a compiled language such as C. However, they are self-contained, using built-in features or library functions instead of relying on external commands such as **awk** and **sed**. As a result, many would argue the they are more suitable for writing sophisticated scripts that would lie somewhere between shell scripts and general programs.

Part I

Systems Management

Chapter 3

Systems Management

3.1 Guiding Principals

Decisions should be based on objective goals, such as

- Improving performance
- Improving reliability (which should also be viewed as part of performance)
- Reducing maintenance cost
- Making all hardware expendable. What end-users ultimately need is access to the programs that do what they need. If a computer they are using to run those programs becomes inaccessible for any reason, it should be easy for them to use another one. Package managers, discussed in Chapter 6, help us achieve this sort of independence. All too often, however, people end up in a panic, unable to get work done, because of a hardware failure. This situation is almost always a symptom of poor systems management.

Apply the KISS principal (Keep It Simple, Stupid) to avoid wasted time and effort on unnecessary complexity.

Unfortunately, many IT professionals are driven by ego or other irrational motives and decisions are based on emotional objectives such as

- Using their favorite tool (solutions looking for problems)
- Favoring the complex solution to make themselves look smart

Top-notch systems managers aim to make everything easily reproducible. All hardware then becomes expendable, because the functionality it provides can be quickly replicated on another machine. This means automating configurations using shell scripts or other tools, and keeping back-ups of important data. Using proprietary tools that may not be around in the future can be a grave mistake. Make sure your automation and backup tools will be readily available as long as you need them.

It's normal to struggle with something the first time you do it. It's incompetent to struggle with it the second time.

Top systems managers also understand how their systems work in detail, so when something does go wrong, they know exactly what to do and can fix it instantly.

Apply the principles of the engineering life cycle, discussed in [?]. Start by throwing out all assumptions about design and implementation of IT solutions, such as which language or operating system will be used. First examine the specification: What does the end-user need to do? Will it be done once, twice, or many times? Then consider ALL viable alternatives from counting on your fingers, to scribbling on paper, to using a supercomputer. Which is the cleanest, simplest, most cost-effective way to enable it?

3.2 Attachment is the Cause of All Suffering

If you're averse to reinstalling your OS from scratch, get over it.

Trying to keep an existing installation running too long is a bad idea for a variety of reasons.

- All hardware fails eventually. Therefore, you should always have your important files backed up in another location and should always be prepared to rebuild your setup on a new disk and restore from backup.
- File systems more than a couple years old can suffer from "bit rot", where the disk is still functional, but some of the bits have faded to the point of being uncertain. For this reason, it's a bad idea to perform OS upgrade after OS upgrade. Instead, the disk should be wiped clean at least once every few years and everything reinstalled from scratch, to "freshen the bits".
- Practice makes perfect. If you avoid doing fresh installs, you will lack the skills needed when it becomes necessary and struggle to recover from hardware failures. If, on the other hand, you do fresh installs frequently, a disk or system failure will not be a big deal to you. Replace the failed hardware and be back in action in an hour or two in most cases.

Chapter 4

Platform Selection

4.1 General Advice

No matter what operating system you use, you are going to have problems.

What you need to decide is what kinds of problems you can live with.

System crashes are the worst kind of problem for scientific computing, where analyses and simulations may takes days, weeks, or even months to run. If a system crash occurs when a job has been running for a month, someone's research may be delayed by a month (unless their software uses checkpointing, allowing it to be resumed from where it left off).

Reliability must be considered as a major factor when assessing the performance of a system. Long-term *throughput* (work completed per unit time) is heavily impacted by systems outages that cause jobs to be restarted.

It doesn't really matter why a system needs to be rebooted. It could be due to system freezes, *panics* (kernel detecting unrecoverable errors), or security updates so critical that they cannot wait. Systems that need to be rebooted frequently for any of these reasons should be considered less reliable.

Uptime, the time a system runs between reboots, should be monitored to determine reliability. The average uptime for popular operating systems varies from days to months.

System crashes are also the worst for IT staff who manage many machines. Suppose you manage 30 machines running an operating system that offers and average up time of a month or two. This means you have to deal with a system crash every day or two on average (unless you reboot machines for other reasons in the interim).

This is exactly the situation I experienced while supporting fMRI research using cutting-edge Linux distributions, such as Redhat (not Redhat enterprise, but the original Redhat, which evolved into Fedora), Mandrake, Caldera, SUSE (again, the original, not SUSE Enterprise).

Some of our Linux workstations would run for months without a problem while others were crashing every week. NFS servers running several different distributions would consistently freeze under heavy load. Systems would freeze for a few minutes at a time while writing DVD-RAMs. These were pristine installations with no invasive modifications. It's not anything we did to the systems, but just the nature of these cutting-edge distributions.

This is a fairly common issue. Some research groups resort to scheduled reboots in order to maximize likely up times from the moment an analysis was started. The HTCondor scheduler has an option to reboot a compute host after a job finishes for similar reasons.

This is in no way a criticism of cutting-edge Linux distributions. They play an important role in the Unix ecosystem, namely as a platform for testing new innovations. We need lots of people using new software systems in order to shake out most of the bugs and make it enterprise-ready, and cutting-edge Linux distributions serve this purpose very well. Many people want to try out the latest new features and don't need a system that can run for months without a reboot. In fact, most of them probably upgrade and reboot their systems every week or so, and as a result, rarely experience a system crash.

However, no operating system is the best at everything, and cutting-edge Linux distributions are not the best at providing stability. Some glitches should be expected from anything on the cutting edge.

For the average user maintaining one or two systems for personal use or development, the stability of a cutting-edge Linux system is generally more than adequate.

For scientists running simulations that take months or IT staff managing many systems, it could be a major nuisance.

One solution is to run an Enterprise Linux distribution, such as Redhat Enterprise, is described in Section 4.3, or SUSE Enterprise.

Another is to run a different Unix variant, such as FreeBSD, described in Section 4.4. This is the route we chose in our fMRI research labs, and it solved almost all of our stability issues. FreeBSD has always been extremely reliable and secure. System crashes are extremely rare. Almost every system crash I've experienced has been traced to a hardware problem or a configuration error on my part. Critical security updates, in my experience, occur less frequently than other systems such as Windows and Linux. If you're looking for a "set and forget" operating system to make your sysadmin duties easy, FreeBSD is a great option.

In addition to choosing an operating system that focuses on reliability, you may want to invest in a UPS and a RAID to protect against power outages and disk failures. If you're really worried, some systems also offer fault-tolerant RAM configurations, using some RAM chips for redundancy, akin to RAIDs.

4.2 Choosing Your Unix the Smart Way

Ultimately, the only thing that matters with respect to which Unix system you use is how well it runs the programs you need.

Many people make the mistake of choosing an operating system based on how "nice" it looks, what their friends (who often are not very computer savvy) are using, or how easy it is to install.

What's really important, though, is what happens *after* the system is up and running. The effort required to maintain it over the course of a couple years is by far the lion's share of the total cost of ownership, so get informed about what that cost will be for your particular needs before deciding.

Each operating system has its own focus, which may be very different from the rest.

The free operating systems include several systems based on BSD (Berkeley Systems Distribution), a free derivative of the original AT&T Unix from the University of California, Berkeley. It is the basis for FreeBSD, Mac OS X, NetBSD, OpenBSD, and a few others.

FreeBSD is the most popular among the free BSD-based systems. FreeBSD is known for its speed, ease of setup, robust network stack, and most of all its unparalleled stability. It is the primary server operating system used by Netflix, and WhatsApp. Netflix alone accounted for more than 1/3 of streaming Internet traffic in North America in 2015. (See http://appleinsider.com/articles/-16/01/20/netflix-boasts-37-share-of-internet-traffic-in-north-america-compared-with-3-for-apples-itunes) FreeBSD is also the basis of advanced file servers such as FreeNAS, Isilon, NAS4Free, NetApp, and Panasas, and network equipment from Juniper Networks, and the open source pfSense firewall.

The FreeBSD ports system makes it trivial to install any of more than 30,000 packages, including most mainstream scientific software. FreeBSD ports can be installed automatically either from a binary package, or from source code if you desire different build options than the packages provide.

Mac OS X is essentially FreeBSD with Apple's proprietary user interface, so OS X users already have a complete Unix system on their Mac. In order to develop programs under Mac OS, you will need to install a compiler. Apple's development system, called Xcode, is available as a free download. The free and open source MacPorts system offers the ability to easily install thousands of software packages directly from the Internet. The MacPorts system is one of the most modern and robust ports systems available for any operating system. There are also other package managers for Mac OS X, such as Fink, Homebrew, and Pkgsrc.

There are many free Linux distributions, as well as commercial versions such as Red Hat Enterprise and SUSE Enterprise. The most popular free distributions for personal use are currently Mint and the Ubuntu line (Ubuntu, Kubuntu, and Xubuntu), which are based on Debian Linux. These systems are known for their ease of installation and maintenance, and cutting-edge new Linux features. Systems based on Debian Linux support the Debian packages system, which offers more than 40,000 packages available for easy installation.

Caution Be careful about judging the popularity of different operating systems based on package counts. The Debian packages collection is somewhat inflated by the fact that they tend to split a single software distribution into several packages. For example, libraries are typically provided by at least three packages, a base package for only the run time components (shared libraries), a -devel package for building your own programs with the library (header files), and a -doc package containing man/info pages, HTML, PDF, etc. Some libraries are further split into single/double precision libs, core and optional components, etc. Many other package systems, such as FreeBSD ports and pkgsrc, tend to provide all library components in a single package.

Gentoo Linux is a Linux system based heavily on ports. The Gentoo system installation process is very selective, and results in a compact, efficient system for each user's particular needs. Gentoo is not as easy to install as other Linux systems, but is a great choice for more experienced Linux users who want to maximize performance and stability. Like FreeBSD and Debian, Gentoo's ports system, known as portage, offers automated installation of nearly 20,000 software packages at the time of this writing.

The NetBSD project is committed to simplicity and portability. For this reason, NetBSD runs on far more hardware configurations than any other operating system.

The OpenBSD is run largely be computer security experts. Core security software such as OpenSSL and OpenSSH are developed by the OpenBSD project and used by most other operating systems.

Redhat Enterprise and SUSE Enterprise Linux are more conservative Linux-based systems designed to provide the stability required in enterprise environments. They are popular in corporate and academic data centers, where they support critical services, often running commercial software applications. They do not include the latest cutting edge Linux features, as doing so would jeopardize the stability they are meant to provide. They are based on older Linux kernels and system tools, which have been well-tested and debugged by users of cutting-edge Linux systems over several years.

4.3 RHEL/CentOS Linux

Redhat Enterprise Linux is a Linux distribution designed for reliability and long-term binary compatibility. Redhat, Inc. took a lot of heat during the 1990s for the inadequate stability of their product. In response, they invented Redhat Enterprise Linux (RHEL) in 2000.

Community Enterprise Linux (CentOS), essentially a free version of RHEL, had its first release in 2004. These systems are created by taking a snapshot of Fedora and spending a lot of time fixing bugs, without upgrading the core tools, which might introduce new bugs. Hence, they run older kernels, compilers, and core libraries like libc.

RHEL, CentOS and their derivatives are used on the vast majority of HPC clusters.

They are also used in data centers around the world to provide all kinds of services needed to keep business, governments, and other organizations running.

One of their major advantages is full support for many commercial scientific software, most of which are supported only on Windows, Mac, and Enterprise Linux.

Enterprise Linux is also more stable than cutting-edge Linux systems. Many Linux users are unaware of this fact, because it is not relevant to them. In my own experience, most Linux systems will provide average up times of a month or two, which is far more than the average computer user needs. Many people will install updates and reboot about once a week anyway, so they will rarely experience a system crash.

One of the disadvantages of Enterprise Linux is that they use older kernels, compilers, standard libraries, and other tools. This makes it difficult to build and run the latest open source software on Enterprise Linux.

The pkgsrc package manager, discussed in Section 6.4.3 can be a big help overcoming this limitation.

4.4 FreeBSD

Stability and performance are the primary goals for the FreeBSD base system.

FreeBSD seems to often be the target of false criticism from people who have little or no experience with it. If someone tells you that FreeBSD is "way behind", "not up to snuff", etc., take the Socratic approach: Ask them to describe some of its shortcomings in detail and watch them demonstrate their lack of knowledge.

In reality, FreeBSD is a very powerful, enterprise-class operating system, used in some of the most demanding environments on the planet. A short list of FreeBSD-based products and services you may be familiar with is below. See https://en.wikipedia.org/-wiki/List_of_products_based_on_FreeBSD for a more complete list.

- Netflix content servers, which alone are responsible for a large portion of all the Internet traffic in North America
- Large cloud services companies such as New York Internet and Webair
- Dell Compellent, FreeNAS, Isilon, NAS4Free, NetApp, and Panasas high-performance storage systems
- Juniper network equipment
- mOnOwall, OPNsense, Nokia IPSO, pfSense firewalls
- Trivago and Whatsapp servers
- CellOS (Playstation 3), and Orbis OS (Playstation 4)

You may hear that FreeBSD is not as cutting-edge as some of the Linux distributions popular for personal use. This may be true from certain esoteric perspectives, but the reality is that only a tiny fraction of programs require cutting-edge features that FreeBSD lacks and FreeBSD is capable of running virtually all the same programs as any Linux distribution, with little or no modification.

A reliable platform on which to run them is far more important in scientific computing and there is no general-use operating system more reliable than FreeBSD.

Enterprise Linux offers comparable reliability, but FreeBSD offers newer compilers and libraries than Enterprise Linux, making it easier to build and run the latest open source software.

The FreeBSD ports collection offers one of the largest available collections of cutting-edge software packages that can be installed in seconds with one simple command. Users can also choose between using the latest packages or packages from a quarterly snapshot of the collection for the sake of stability in their add-on packages as well as the operating system itself. The quarterly snapshot's receive bug fixes, but not upgrades, much like Enterprise Linux distribution.

FreeBSD ports can be easily converted to pkgsrc packages for deployment on Enterprise Linux and other Unix-compatible systems.

FreeBSD has a Linux compatibility system based on CentOS. It can run most closed-source software built on RHEL/CentOS, although complex packages (e.g. Matlab) may be tricky to install. Ultimately, though, FreeBSD is actually more binary-compatible with RHEL than most Linux distributions. It uses tools and libraries straight from the CentOS Yum repository. The RPMs there are easily converted to FreeBSD ports for quick, clean deployment on FreeBSD systems.

Note that the compatibility system is not an emulation layer. There is no performance penalty for running Linux binaries on a FreeBSD system, and in fact some Linux executables may run faster on FreeBSD than they do on Linux. The system consists of a kernel module to support system calls that exist only in Linux, and the necessary run time tools and libraries to support Linux executables. The system only requires a small amount of additional RAM for the kernel module and disk space for Linux tools and libraries.

Hence, if you are running mostly open source and one or two closed-source Linux applications, FreeBSD may be a good platform for you. If you are running primarily complex closed-source Linux applications (Matlab, ANSYS, Abaqus, etc.), you will likely be better off running an Enterprise Linux system.

ZFS is fully-integrated into the FreeBSD kernel, and is becoming the primary file system for FreeBSD. The FreeBSD installer makes it easy to configure and boot from a ZFS RAID.

The UFS2 file system is still fully supported, and a good choice for those who don't want the high memory requirements of ZFS. UFS2 has many advanced features, such as an 8 ZiB file system capacity, soft updates (which ensure file system consistency without the use of a journal), an optional journal for quicker crash recovery, and backgrounded file system checks (which allow the file system to be checked and repaired while in-use, eliminating boot delays even if the journal cannot resolve consistency issues).

There are many other advanced features and tools such as FreeBSD jails (a highly developed container system), bhyve, qemu, VirtualBox, and Xen for virtualization, multiple firewall implementations, network virtualization, and mfiutil for managing LSI MegaRAID controllers, to name a few.

FreeBSD is a great platform for scientific computing in its own right, especially for running the latest open source software. It's also a great sandbox environment for testing software that may later be run on RHEL/CentOS via pkgsrc.

4.5 Running a Desktop Unix System

Most mainstream operating systems today are Unix compatible. Microsoft Windows is the only mainstream operating system that is not Unix compatible, but there are free compatibility systems available for Windows to provide some degree of compatibility and interoperability with Unix.

The de facto standard of Unix compatibility for Windows is Cygwin, http://cygwin.com, which is free and installs in about 10 minutes. There are alternatives to Cygwin, but Cygwin is the easiest to use and offers by far the most features and software packages.

Note None of the Unix compatibility systems for Windows are nearly as fast as a genuine Unix system on the same hardware, but they are fast enough for most purposes. If you want to maximize performance, there are many BSD Unix and Linux systems available for free.

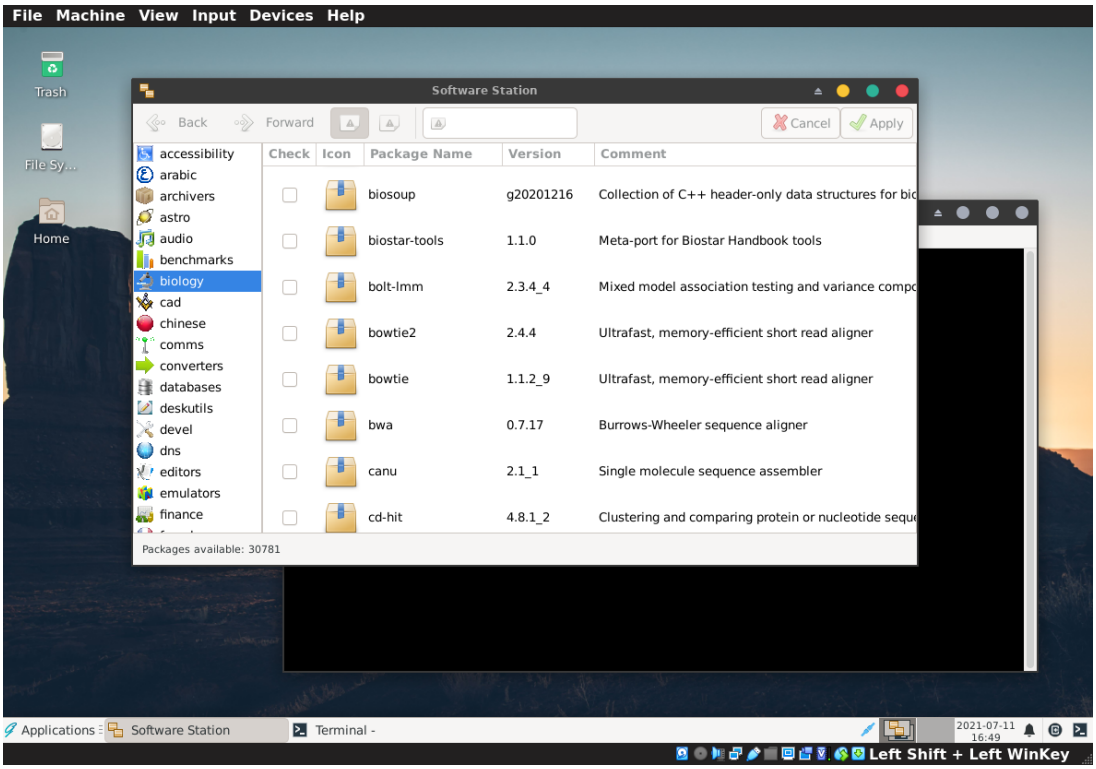
Another option for running Unix programs on a Windows computer is to use a *virtual machine (VM)*. This is discussed in Chapter 7.

Lastly, many Windows programs can be run directly under Unix, without a virtual machine running Windows, if the Unix system is running on x86-based hardware. This is accomplished using WINE, a Windows API emulator. WINE attempts to emulate the entire Windows system, as opposed to virtual machines, which emulate hardware. Emulating Windows is more ambitious, but eliminates the need to install and maintain a separate Windows operating system. Instead, the Windows applications run directly under Unix, with the WINE compatibility layer between them and the Unix system.

While it is possible to create a Unix-like environment under Windows using a system such as Cygwin, such systems have some inherent limitations in their capabilities and performance. Installing a Unix-compatible operating system directly has many benefits, especially for those developing their own code to run on the cluster.

Many professional quality Unix-based operating systems are available free of charge, and with no strings attached. These systems run a wide variety of high-quality free software, as well as many commercial applications. Hence, it is possible for researchers to develop Unix-compatible programs at very low cost that will run both on their personal workstation or laptop, and a cluster or grid.

One of the easiest Unix systems to install and manage is GhostBSD, a free, open source derivative of FreeBSD with a simple graphical installer, "Control Panel", and software manager:



A GhostBSD system running XFCE desktop.

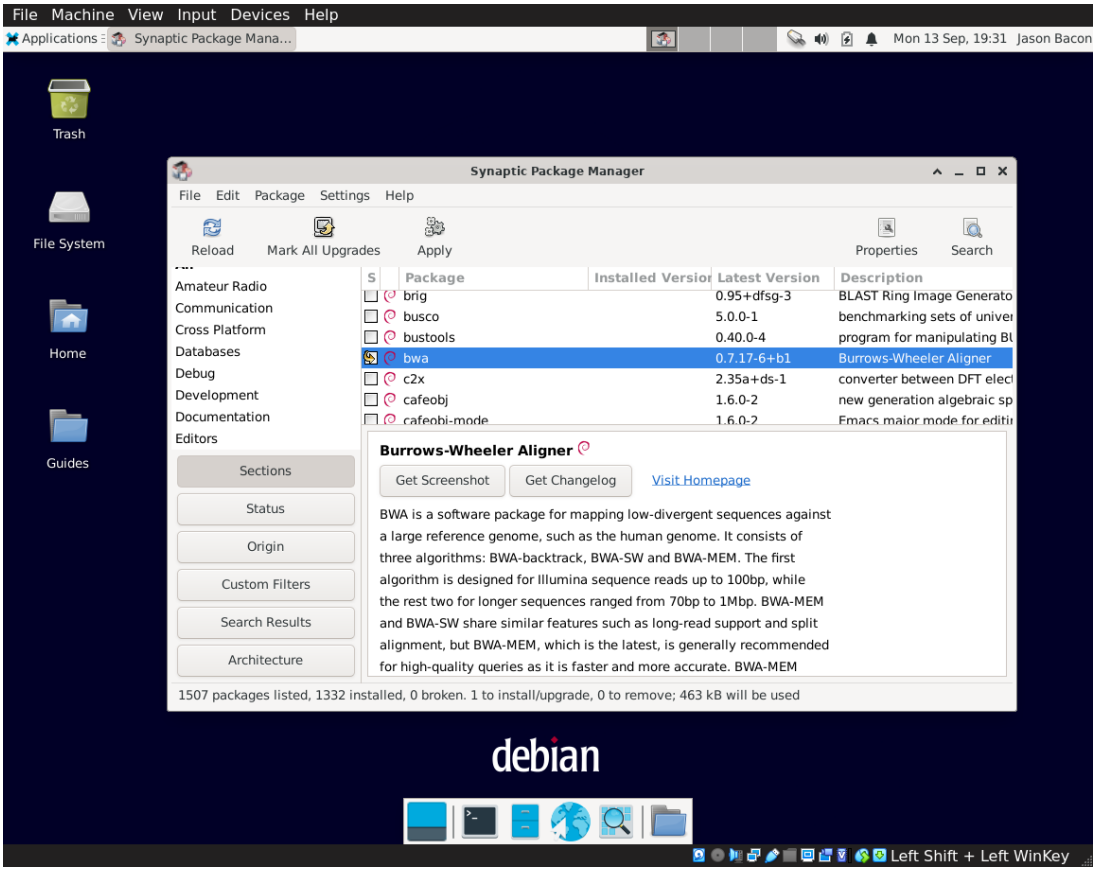
GhostBSD is extremely easy to install and manage, as well as extremely reliable. If you want to try out Unix while encountering as few hurdles as possible, GhostBSD is probably your best bet.

Similar to GhostBSD are the Ubuntu family of Linux systems (Ubuntu, Kubuntu, Xubuntu, Edubuntu, ...). Each of these Linux distributions is built on Debian Linux, with a different desktop environment. (Ubuntu uses Gnome, Kubuntu KDE, Xubuntu XFCE, etc.)

Another alternative for the slightly more computer-savvy is to do a stock FreeBSD installation and then install and run the sysutils/desktop-installer port. This option simply helps you easily configure FreeBSD for use as a desktop system using standard tools provided by the system and FreeBSD ports. The whole process can take as little as 15 minutes on a fast computer with a fast Internet connection. Just run desktop-installer from a terminal and answer the questions.

A FreeBSD system running Lumina desktop.

The Debian system itself has also become relatively easy to install and manage in recent years. It lacks some of the bells and whistles of Ubuntu, but may be a bit faster and more stable as a result.



A Debian system running XFCE desktop.

All of these systems have convenient methods for installing security updates and minor software upgrades.

When it comes time for a serious upgrade of the OS, don't bother with upgrade tools. Back up your important files, reformat the disk, do a fresh install of the newer version, and restore your files.

Many hours are wasted trying to fix systems that have been broken by upgrades or were broken before the upgrade. It would have been faster and easier in many cases to run a backup and do a fresh install. You will need to do fresh installs sometimes anyway, so you might as well become good at it and use it as your primary method.

4.6 Unix File System Comparison

Windows file systems become fragmented over time as file are created and removed. Windows users should therefore run a defragmentation tool periodically to improve disk performance.

Unix file systems, in contrast, do continuous defragmentation, so performance will not degrade significantly over time.

Overwrite performance on some file systems is slower than initial write. Hence, removing files before overwriting them may help program run times.

Most Unix systems offer multiple choices for file systems. Most modern file systems use *journaling*, in which data that is critical to maintaining file system integrity in the event of a system crash is written to the disk immediately instead of waiting in a memory buffer.

To save time, this data is queued to a special area on the disk known called the journal. Writing to a journal is faster than saving the data in it's final location, since it requires fewer disk head movements.

Journaling reduces write performance, since data is first written to a journal and later moved to its final location. This takes more time and more disk head movements than storing data in a memory buffer until it is written to its final location. However,

the performance penalty is marginal if done intelligently. All modern Unix file systems use advanced journaling methods to minimize the performance hit and disk wear.

Popular file systems:

- EXT is the most commonly used file system on Linux systems. EXT3 was the first to including journaling, basically as a feature added to EXT2. EXT2 was notorious for incredibly slow file system checks and repairs. The journaling features added by EXT3 greatly reduced the need for repairs, but EXT3 is not the best performer overall and is also hard on disks due to excessive head movements.

  EXT4 represents a vast improvement over EXT3 due to major redesign of key components. Performance and reliability are solid.

- HFS is the file system used by Mac OS X. Features and performance are generally positive. One potential problem for Unix users is the lack of true case-sensitivity. HFS is *case-preserving*, but not *case-sensitive*. This means that if you create a file named "Tempfile", the "T" will be remembered as a capital. However, it is not distinguished from a lower-case "t", so the file may be referred to as "tempfile". Also, you cannot have two files in the same directory called "Tempfile" and "tempfile", because these two file names are considered the same.

- UFS (Unix File System) evolved from the original Unix system 7 file system and is now used by most BSD systems as well as some commercial systems such as SunOS/Solaris and HP-UX.

  FreeBSD's UFS2 includes a unique feature called *soft updates*, which protects file system integrity in the event of a system crash without using a journal. This allows UFS2 to exhibit better write performance and less disk wear.

- XFS is a file system developed by SGI for it's commercial IRIX operating system during the 1990s, which were popular for high-end graphics. SGI IRIX machines were used to develop and featured in the movie Jurassic Park.

  XFS has been fully integrated into Linux and is now used as an alternative to EXT4 where high performance and very large partitions are desired.

- ZFS is a unique combination of a file system combined with a *volume manager*, developed by Sun Microsystems.

  ZFS is widely regarded as the most advanced file system to date. One of its most unique features is the fact that it does not require partitioning the disk in order to separate file systems. With other file systems, if you want home and /var to be separated and utilize different settings, you must divide the disk into separate partitions. Choosing the optimal size for each partition is almost impossible since we cannot predict the space requirements of the future. With ZFS, you can create multiple file systems, each with its own settings, all of which allocate blocks from the same pool. Thus, you never run out of space in one file system while having unutilized space in others. ZFS also offers advanced software RAID that generally outperforms hardware RAID systems, and many other advanced features such as compression and encryption.

  ZFS has been fully integrated into FreeBSD and is now the default file system for high-end FreeBSD servers as well as the GhostBSD desktop system. ZFS does require a lot of RAM, however, so UFS2 is still a better choice for low-end hardware such as net books and embedded FreeBSD systems.

4.7 Network File System

Network File System, or NFS, is a standard Unix network protocol that allows disk partitions on one Unix to be directly accessed from other computers. In concept, NFS is similar to Apple's AFS and Microsoft's SMB/CIFS.

Access to files across an NFS link is generally somewhat slower than local disk access, due to the overhead of network communication. Speed may be limited either by the local disk performance on the NFS server or by the bandwidth of the network. For example, of an NFS server has a RAID that can deliver 500 megabytes per second locally and a 1 gigabit (~100 megabyte per second) network, then the disk performance seen by NFS clients will be limited by the network to about 100 megabytes per second.

Unix systems also allow other computers to access their disks using non-Unix protocols like AFS and SMB/CIFS if necessary. For example, Samba is an open source implementation of the SMB/CIFS protocol that allows Windows computers to access data on Unix disks.

Chapter 5

System Security

5.1 Securing a new System

- Configure firewall or TCP wrappers to allow incoming traffic from only specific hosts.
- Create ONE account with administrator rights and use it only for system updates and software installations.
- Do not share login accounts. Create SEPARATE accounts for each user, without administrator rights, and use them for all normal work.
- NEVER share your password with ANYONE. PERIOD. NOBODY should ever ask you for your password. Other users have no right to mess with your login account. IT staff with rights to manage a machine do not need your password, so be suspicious if they ask for it.
- Store passwords in KeePassX or a similar encrypted password vault. Use a strong password for each KeePassX database.
- If you set up a computer to allow remote access, use ONLY systems that encrypt ALL traffic. If you are not sure your remote access software encrypts everything, DO NOT ENABLE IT. Talk to a professional about how to securely access the computer remotely before allowing it.

5.2 I've Been Hacked!

If you suspect that your computer has been hacked, unplug it from the network (or disable WiFi), but do not turn it off. Call your local computer security experts, and do not touch the computer until they arrive.

Once a computer has been hacked, that operating system installation is finished. Don't even think about trying to patch your way out of it. The only way to clean a hacked system is by backing up your files, reformatting the hard disk, reinstalling, and changing every password that was ever typed on the computer, whether it was a local password or a password on another computer someone connected to fro the hacked computer.

Antivirus and other antimalware software only detects known malware. If a hacker installs a custom program of their own design, it will not be detected.

There are many sites listing the steps you need to take, but most are incomplete. Below is a fairly comprehensive list.

1. Unplug the computer from the network to cut off the hacker's access immediately.
2. Stop using the computer. Especially, do not use the computer to log into any other computers over the network, as you will likely be giving away your passwords to those machines as you type them.
3. USING A DIFFERENT COMPUTER, immediately change your passwords on every other computer that you have ever connected to from the hacked computer. Every password that has ever been typed on the hacked machine must be changed, as the hacker may have been monitoring all of your keystrokes for a long time before the intrusion was detected. That includes local passwords on the PC as well as passwords entered on the PC to log into remote machines.

4. If you have IT staff trained in computer security, contact them. They may want to do a forensic analysis on the machine to determine who hacked it and how.
5. Back up your data files. Note that they may have been corrupted by the hacker, so check them carefully before relying on them.
6. Do not back up any programs, scripts, installation media, or configuration files. They may be infected with malware and restoring them to the newly installed system will allow the hacker right back in. Antivirus and other antimalware programs do not detect all malware. Don't think for a minute the your computer is clean just because your virus scan didn't find anything. This is foolish wishful thinking that will only cause more problems for you and others around you.
7. Reformat all disks in the computer and reinstall the operating system from trusted install media. (Do not use install media that was stored on the hacked computer!)
8. Do not use any of the same passwords on the new installation. Create new passwords for every user and every application on the computer.
9. Restore your data files from backup.
10. Reinstall all programs from trusted installation media.

Chapter 6

Software Management

6.1 The Stone Age vs. Today

There are many thousands of quality open source applications and libraries available for Unix systems.

Just knowing what exists can be a daunting task. Fortunately, software management systems such as the FreeBSD ports have organized documentation about what is available. You can browse software packages by category on the ports website: http://www.freebsd.org/ports/index.html. Even if you don't use FreeBSD, this software listing is a great resource just for finding out what's available.

Installing various open source packages can also be a daunting task, especially since the developers use many different programming languages and build systems.

Many valuable man-hours are lost to stone-age software management, i.e. manually downloading, unpacking, patching, building, and installing open source software.

Free software isn't very free if it takes 20 hours of someone's time to get it running. An average professional has a total cost to their employer on the order of $50/hour. Hence, 20 hours of their time = $1,000. If 1,000 professionals around the world spend an average of 20 hours installing the same software package, then $1,000,000 worth of highly valuable man-hours have gone to waste. Even worse, many people eventually give up on installing software entirely, so there are no gains to balance this loss.

In most cases, the software could have been installed in seconds using a software management system (SMS) and all that time could have been spent doing something productive.

When choosing a Unix system to run, a good *ports* or *packages* system is an important consideration. A ports or packages system automatically downloads and installs software from the Internet. Such systems also automatically install additional *prerequisite* ports or packages required by the package you requested.

For example, to install Firefox, you would first need to install dozens of libraries and other utilities that Firefox requires in order to run properly. (When you install Firefox on Windows or Max OS X, you are actually installing a bundle of all these packages.)

The ports or packages system will install all of them automatically when you choose to install Firefox. This allows you install software in seconds or minutes that might otherwise take days or weeks for an inexperienced programmer to manually download, patch, and compile.

6.2 Goals

Complete execution well before deadline.

Minimize man-hours.

Maximizing execution speed of every program is a foolish waste of resources.

Focus on big gains, 80/20 rule (Pareto principal). 20% of effort typically yields 80% of gains. Don't waste time or hardware trying to squeeze out marginal gains unless it's really necessary. If it won't mean meeting a deadline that would otherwise be missed, or free up saturated resources, then it's a waste.

6.3 The Computational Science Time Line

The figure below represents the time line of a computational science project.

Development Time	Deployment Time	Learning Time	Run Time
Hours to years	Hours to months (or never)	Hours to weeks	Hours to months

Table 6.1: Computation Time Line

6.3.1 Development Time

Not relevant to most researchers.

Learn software development life cycle, efficient coding and testing techniques.

Understand objective language factors; compiled vs interpreted speed, portability, etc.

6.3.2 Deployment Time

Deployment time virtually eliminated by package managers, described in Section 6.4.

6.3.3 Learning Time

Largely up to end-user.

IT staff can help organize documentation and training.

6.3.4 Run Time

Software efficiency (algorithms, language selection) should always be the first focus. Often software can be made to run many times faster simply by changing the inputs. Is the resolution of your fluid model higher than you really need? Are you analyzing garbage data along with the useful data? Is your algorithm implemented in an interpreted language such as Matlab, Perl, or Python? If so, it might run 100 times faster if rewritten in C, C++, or Fortran. See [?].

System reliability (system crashes cause major setbacks, especially where check pointing is not used). Operating system, (FreeBSD, ENTERPRISE Linux), UPS.

Some scientific analyses take a month or more to run. FSL, single-threaded. Average up time of 1 month is not good enough.

From the researcher's perspective, this may mean restarting simulations or analyses, losing weeks worth of work if check pointing is not possible.

From the sysadmin's perspective, if managing 30 machines with an average up time of 1 month, you average 1 system crash per day.

Some choose scheduled reboots to maximize likelihood of completing jobs. Better to do your homework and find an operating system with longer up times.

Parallelism is expensive in terms of both hardware and learning curve. It should be considered a last resort after attempting to improve software performance.

6.4 Package Managers

6.4.1 Motivation

A package manager is a system for installing, removing, upgrading or downgrading software packages. They ensure that proper versions of dependency software are installed and keep track of all files installed as part of the package.

A *caveman installation* is an installation performed by downloading, patching, building and installing software manually or via a custom script that is not part of a package manager. This is a temporary, isolated solution.

A package added to a package manager is a permanent, global solution.

A package need only be created once, and then allows the software to be easily deployed any number of times on any number of computers worldwide.

1,000 people spending 2 hours each doing cave man (ad hoc) installations = 2000 man-hours = 1 year's salary.

1 person spending 2 hours creating a package + 999 spending 2 seconds typing a package install command = 2.55 man-hours.

There is a significant, but one-time investment in learning to package software. Once learned, creating a package usually takes LESS time than a cave man install.

Have you ever been in a panic because your server went down and you're approaching a deadline to get your analysis or models done? If you deploy the software with a package manager, no problem... Just install it on another machine and carry on. If you've done a caveman install, you might be dead in the water for a while until you can restore the server or duplicate the installation on another.

Some packages managers allow the end-user to build from source with many combinations of options, compilers, alternate libraries (BLAS, Atlas, OpenBLAS). FreeBSD ports, Gentoo Portage, MacPorts, pkgsrc.

This provides the user more flexibility.

Binary packages distributed by any package manager must be portable and thus use only minimal optimizations. We can do optimized builds from source (make.conf, mk.conf or command-line additions) such as -march=native.

Building from source takes longer, but is no more difficult for you. It also provides many advantages, such as:

- The ability to build an optimized installation. Binary (precompiled) packages must be able to run on most CPUs still in use, so they do not take advantage of the latest CPU features (such as AVX2 as of this writing in February 2019). When compiling a package from source, we can add compiler flags such as `-march=native` to optimize for the local CPU. The package produced may not work on older CPUs.
- The ability to install to a different prefix. This can be useful for doing multiple installations with different build options, or installing multiple versions of the same software.
- The ability to easily install software whose license does not allow redistribution in binary form
- The ability to easily test patches.

It also makes it easy to use the package manager to systematically deploy work-in-progress packages that are not yet complete and for which no binary packages have been built.

Most package managers work only on one platform or possible a few closely related platforms. Some work on multiple platforms with some limitations. The pkgsrc package manager is unique in that it provides general purpose package management for any Unix compatible platform.

Table 6.2 provides some information about popular package managers.

A direct comparison of which collection is "biggest" is not really feasible. A raw count of Debian packages will produce a higher number than the others, but this is in part due to the Debian tradition of creating separate packages for documentation (-doc packages) and header files (-dev packages). FreeBSD ports and many other package managers traditionally include documentation and headers in the core package. For example, below are listings of FFTW (fast Fourier transform) packages on Debian and FreeBSD:

Name	Platforms	Build From Source?
Conda	Linux, Mac, Windows	No
dports	Dragonfly BSD (derived from FreeBSD ports)	Yes
Debian Packages	Debian-based (Debian, Ubuntu, etc)	No
FreeBSD Ports	FreeBSD, Dragonfly BSD*	Yes
MacPorts	OS X	Yes
pkgsrc	Any POSIX	Yes
Portage	Gentoo Linux	Yes
Yum/RPM	Redhat Enterprise Linux, CentOS	No

Table 6.2: Package Manager Comparison

Software in the RHEL Yum repository also tends to be older versions than you will find in Debian, FreeBSD, or Gentoo packages.

This is in no way a criticism of Red Hat Enterprise Linux, but simply illustrates that it was designed for a different purpose, namely highly stable enterprise servers running commercial software. The packages in Yum are intended to remain binary compatible with commercial applications for 7 years, not to provide the latest open source applications.

The Pkgsrc software management system can be used to maintain more recent open source software on enterprise Linux systems, separate from the commercial software and support software installed via Yum. Pkgsrc offers over 17,000 packages, most of which are tested on CentOS.

Pkgsrc has the additional benefit of being the only cross-platform SMS. It can currently be used to install most packages on NetBSD, Dragonfly BSD, Linux, Mac OS X, and Interix (a MS Windows Unix layer).

Even if you have to manually build and install your own software, you can probably install most of the the prerequisites from an SMS. Doing so will save you time and avoid conflicting installations of the same package on your system.

Better yet, learn how to create packages for an SMS, so that others don't have to duplicate your effort in the future. Each SMS is a prime example of global collaboration, where each user leverages the work of thousands of others. The best way to support this collaboration is by contributing to it, even if in a very small way.

6.4.2 FreeBSD Ports

The FreeBSD ports system represents one of the largest package collections available, and it runs on a platform offering enterprise stability and near-optimal performance.

29,483 packages as of Feb 2 2018.

Port options allow many possible build option combinations for some ports (R is a good example). Some other package managers would require separate binary packages to provide the same support.

Most core scientific libraries are well-tested and maintained. (BLAS, LAPACK, Eigen, R, Octave, mpich2, openmpi, etc.)

Easy to deploy latest open source software, easy to convert to pkgsrc for deployment on other POSIX systems. Great scientific computing platform and sandbox environment.

Advanced development tools (ports-mgmt category), portlint, stage-qa, poudriere.

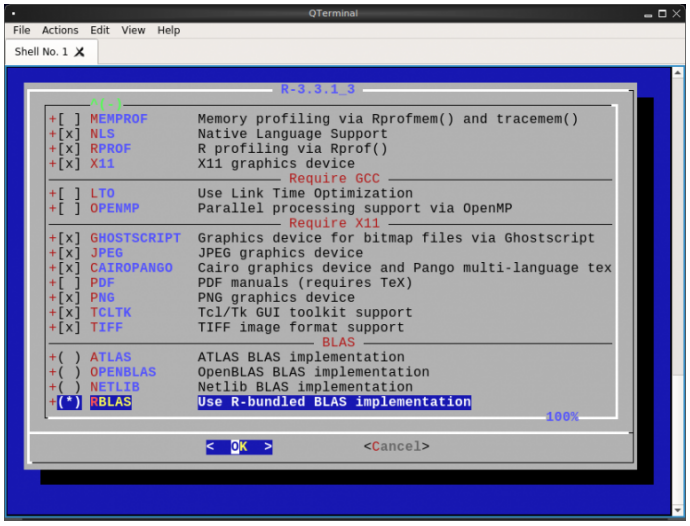
```
DEVELOPER=yes
```

Security checks

```
shell-prompt: pkg install R
```

```
shell-prompt: cd /usr/ports/math/R
shell-prompt: make rmconfig
shell-prompt: make install
```

Port options dialog for R:



Porter's Handbook

Example: http://acadix.biz/hello.php

Install freebsd-ports-wip: https://github.com/outpaddling/freebsd-ports-wip

Add the following to ~root/.porttools:

```
EMAIL="your-email@some.domain"
FULLNAME="Your Full Name"
```

```
shell-prompt: pkg install porttools
shell-prompt: wip-update
shell-prompt: wip-reinstall-port port-dev
shell-prompt: cd /usr/ports/wip
shell-prompt: port create hello

The port directory name given here should usually be all lower-case,
except for ports using perl CPAN and a few other cases.  The PORTNAME
is usally lower-cased as well, but there is not general agreement on this.
It's not that important as "pkg install" is case-insensitive.

shell-prompt: cd hello
shell-prompt: wip-edit

See /usr/ports/Mk/bsd.licenses.db.mk for list of valid licenses.
You can comment out LICENSE_FILE= until after the distfile is downloaded
and unpacked if that's more convenient that figure out it's location
via the web.  It should usually be prefixed with ${WRKSRC}, e.g.

LICENSE_FILE=   ${WRKSRC}/COPYING

shell-prompt: port-check
shell-prompt: port-remake
```

Thorough port testing:

```
shell-prompt: port-poudriere-setup
ZFS pool []: (just hit enter)
Configuration file opens in vi, accept defaults.
shell-prompt: wip-poudriere-test hello all
```

The port-poudriere-setup script will create a basic poudriere setup and a FreeBSD jail for building and testing ports on the underlying architecture and operating system. It also offers the option to create additional jails for older operating systems and lower architectures (i386 if you are running amd64).

The wip-poudriere-test script runs "poudriere testport" on the named port in the wip collection.

Other useful poudriere commands:

```
shell-prompt: poudriere ports -u

Updates the ports tree used by poudriere.  This will obsolete any binary
packages saved from previous builds if the corresponding port is upgraded.
Hence, your next poudriere build may take much longer.

shell-prompt: poudriere bulk wip/hello

This will build a binary package for the named port, which you can deploy
with "pkg add" on other systems.
```

Run "poudriere" or "poudriere <command>" or "man poudriere" for help.

Example 2: https://github.com/cdeanj/snpfinder

```
shell-prompt: cd /usr/ports/wip
shell-prompt: port create snpfinder

USE_GITHUB=yes
GH_ACCOUNT=cdeanj
DISTVERSION=1.0.0

shell-prompt: cd snpfinder
shell-prompt: wip-edit
shell-prompt: port-patch-vi work/snpfinder-1.0.0
shell-prompt: port-check
shell-prompt: port-remake
```

6.4.3 Pkgsrc

Pkgsrc was forked from FreeBSD ports in 1997 by the NetBSD project.

Like everything in the NetBSD project, the primary focus is portability. Pkgsrc aims to support all POSIX environments. Top-tier support for NetBSD, Linux, SmartOS. Strong support for Mac OS X, other BSDs.

Over 18,000 packages as of Feb 2018.

Tools analogous to FreeBSD ports, but often less developed. pkglint, stage-qa, pbulk.

url2pkg, fbsd2pkg

```
PKG_DEVELOPER=yes
```

Pkgsrc Guide (both user and packager documentation)

Example: http://acadix.biz/hello.php

Log into a system using pkgsrc (NetBSD, Linux, Mac, etc.)

Install pkgsrc-wip: https://www.pkgsrc.org/wip/

Install uwm-pkgsrc-wip: https://github.com/outpaddling/uwm-pkgsrc-wip

---

Note The uwm-pkgsrc-wip project is being phased out, but still contains some useful tools.

```
shell-prompt: cd /usr/pkgsrc/uwm-pkgsrc-wip/pkg-dev
shell-prompt: bmake install
```

Install FreeBSD ports and wip on your pkgsrc system: (ports collection is mirrored on Github if you prefer git)

```
shell-prompt: pkgin install subversion
shell-prompt: svn co https://svn.FreeBSD.org/ports/head /usr/ports
shell-prompt: cd /usr/ports
shell-prompt: svn co https://github.com/outpaddling/freebsd-ports-wip.git wip
```

Convert the FreeBSD port to pkgsrc:

```
shell-prompt: cd /usr/pkgsrc/uwm-pkgsrc-wip/fbsd2pkg
shell-prompt: bmake install
shell-prompt: cd ..
shell-prompt: fbsd2pkg /usr/ports/wip/hello your-email-address

You can run the above command repeatedly until the package is done.

shell-prompt: cd hello
shell-prompt: pkg-check
shell-prompt: pkglint -e
shell-prompt: pkglint -Wall
```

Create the package from scratch using url2pkg:

```
shell-prompt: mkdir hello
shell-prompt: cd hello
shell-prompt: url2pkg http://acadix.biz/Ports/distfiles/hello-1.0.tar.xz
```

6.5 What's Wrong with Containers?

Absolutely nothing. Containers are great and play many important roles in computing, especially in development and security.

There are problems with the way some people use them, however. As is often the case, containers have become a solution looking for problems. Many people use them because they think it's "cool" or because it's a path of least resistance.

In scientific computing, containers are often used to isolate badly designed or badly implemented software that is otherwise difficult to install outside a container. For example, software build systems that bundle share libraries can cause conflicts with other versions of the same shared library.

Aside There is no problem that cannot be "solved" by adding another layer of software. This is never a solution, however, and is generally short-sighted.

Isolating such software in a container will get it up and running and work around conflicts, but with some major down sides:

- It eliminates the motivation to clean up the software, contributing to the de-evolution of software.
- It adds overhead to running the software. (Many modern containers advertise their low overhead for this reaason.)

Misusing containers in this creates more disorder and complexity where there is already too little IT talent available to manage things well.

Chapter 7

Running Multiple Operating Systems

You don't necessarily need to maintain a second computer in order to run Unix in addition to Windows. All mainstream Unix operating systems can be installed on a PC alongside Windows on a separate partition, or installed in a virtual machine (VM), such as Oracle VirtualBox, which is also available for free.

VMs are software packages that pretend to be computer hardware. You can install an entire operating system plus the software you need on the VM as if it were a real computer. The OS running under the VM is called the *guest* OS, and the OS running the VM on the real hardware is called the *host*.

Computational code runs at the same speed in the guest operating system as it does in the host. The main limitation imposed on guest operating systems is graphics speed. If you run applications requiring fast 3D rendering, such as video players, they should be run on the host operating system.

There are many VMs available for x86-based PC hardware, including VirtualBox, http://www.virtualbox.org/, which is free and open source, and runs on many different host platforms including FreeBSD, Linux, Mac OS X, Solaris, and Windows.

Running a Unix guest in a VM on Windows or Windows as a guest under Unix will provide a cleaner and more complete Unix experience than can be achieved with a compatibility layer like Cygwin. The main disadvantage of a VM is the additional disk space and memory required for running two operating systems at once. However, given the low cost of today's hardware, this doesn't usually present a problem on modern PCs.

Virtual machines are most often used to run Windows as a guest on a Unix system, to provide access to Windows-only applications to Unix (including Mac) users without maintaining a second computer. This configuration is best supported, and offers the most seamless integration between host and guest. An example is shown in Figure 7.1.

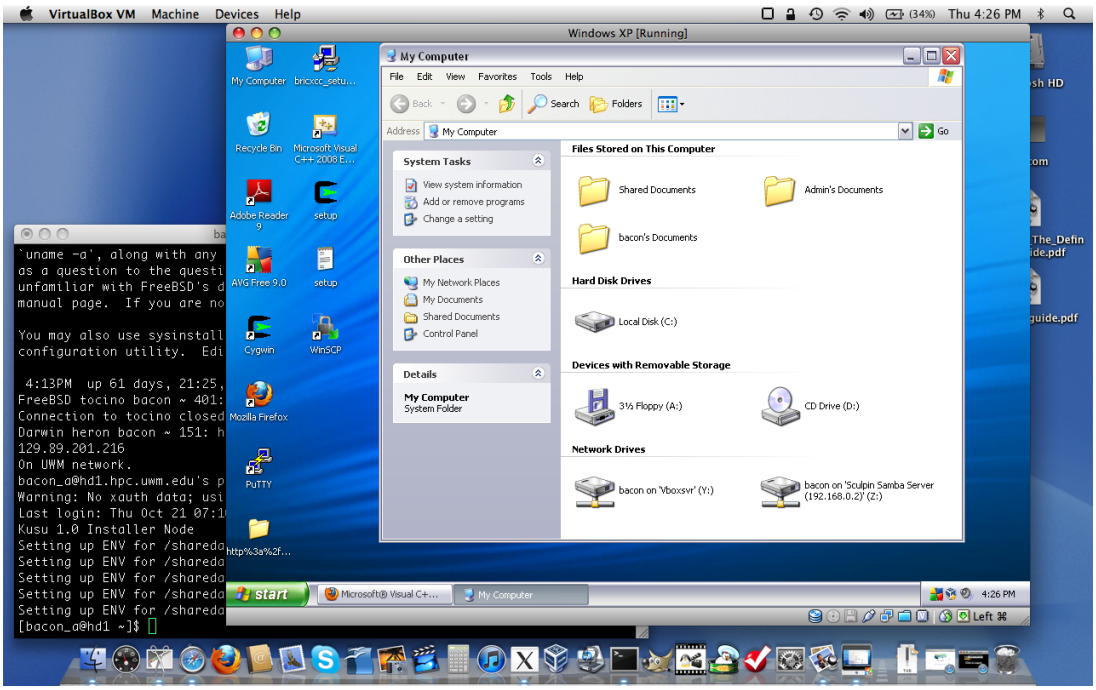


Figure 7.1: Windows as a Guest under VirtualBox on a Mac Host

Another issue is that Windows systems need to be rebooted frequently, often several times per week, to activate security updates. Most Unix systems, on the other hand, can run uninterrupted for months at a time. (FreeBSD systems will typically run for years, if your power source is that stable.) There are far fewer security updates necessary for Unix systems, and most updates can be installed without rebooting. Rebooting a host OS requires rebooting all guests as well, but rebooting a guest OS does not affect the host. Hence, it's best to run the most stable system as the host.

If necessary, it is possible to run Unix as a guest under Windows. FreeBSD and many Linux distributions are fully supported as VirtualBox guest operating systems.

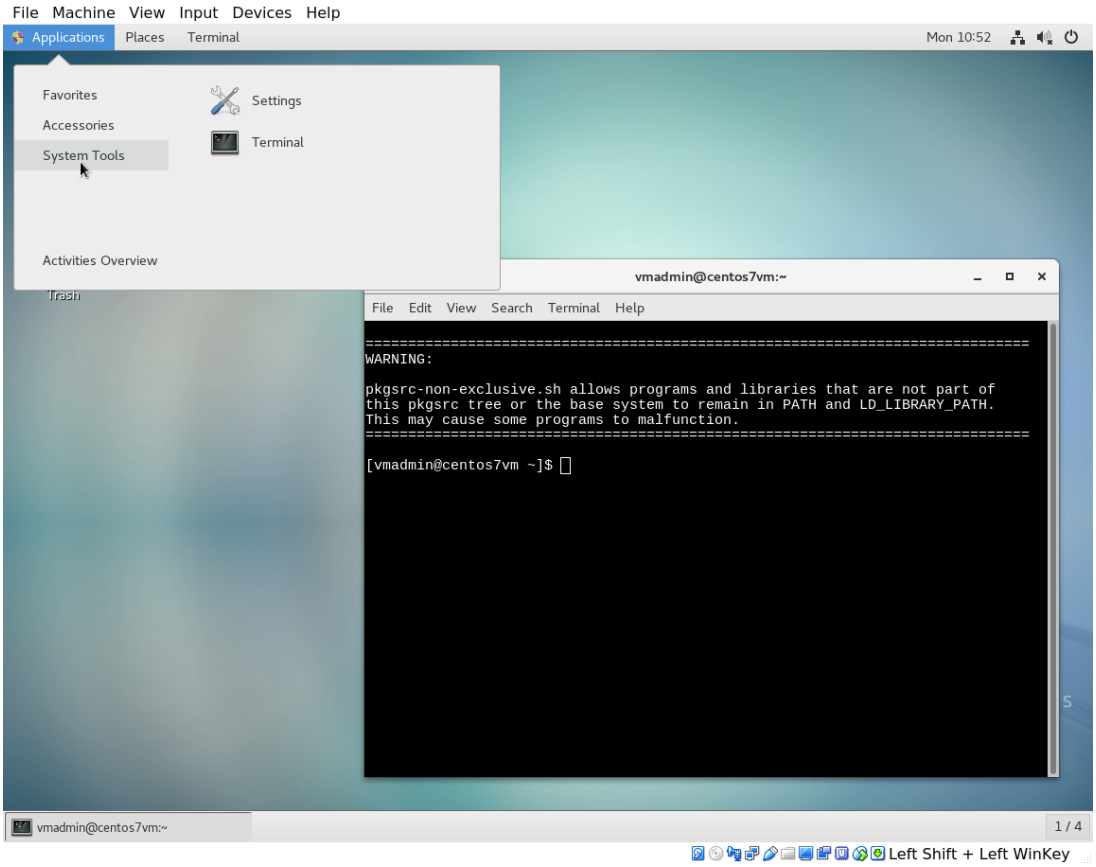


Figure 7.2: CentOS 7 with Gnome Desktop as a Guest under VirtualBox

Figure 7.3: FreeBSD with Lumina Dekstop as a Guest under VirtualBox

Chapter 8

Index