System Administration 1: The Command Line

Getting Started

Welcome to the introductory O'Reilly School of Technology (OST) System Administration Course.

Course Objectives

When you complete this course, you will be able to:

- navigate the filesystem.
- manipulate files and directories.
- change filesystem permissions.
- create and edit text files using vi.
- use many of Bash's powerful built-in features such as pipes and redirects.
- view and interact with processes on the system.

In this course in the system administration series, you'll learn the basics of using a Linux-based system. Topics covered include navigating the filesystem, working with files and directories, file permissions, the vi text editor, the Bash shell, processes, and ways to find help in the open-source community.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

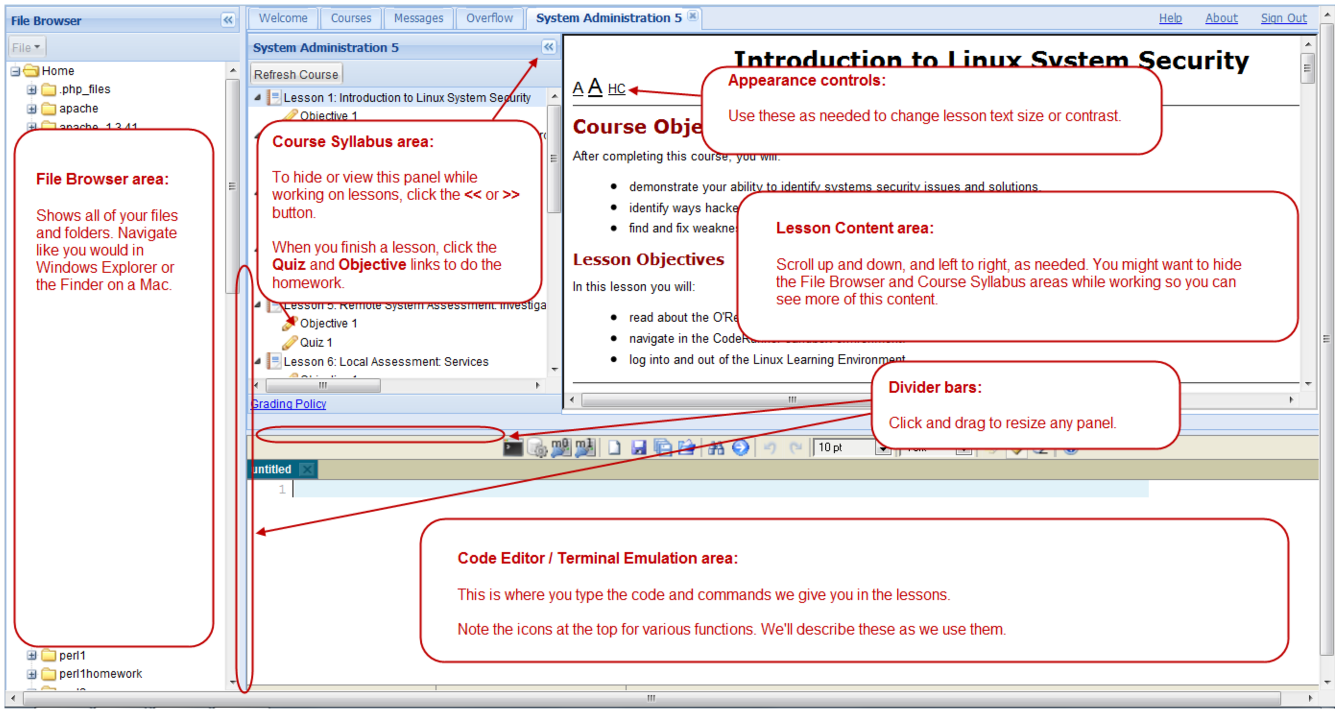
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



File Browser area:

Shows all of your files and folders. Navigate like you would in Windows Explorer or the Finder on a Mac.

Course Syllabus area:

To hide or view this panel while working on lessons, click the **<<** or **>>** button.

When you finish a lesson, click the **Quiz** and **Objective** links to do the homework.

Appearance controls:

Use these as needed to change lesson text size or contrast.

Lesson Content area:

Scroll up and down, and left to right, as needed. You might want to hide the File Browser and Course Syllabus areas while working so you can see more of this content.

Divider bars:

Click and drag to resize any panel.

Code Editor / Terminal Emulation area:

This is where you type the code and commands we give you in the lessons.

Note the icons at the top for various functions. We'll describe these as we use them.

These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

Logging Into the Server

You will use CodeRunner to log into one of our Linux-based servers in order to complete your coursework. To do that, click the **New Terminal** ( ) button in CodeRunner, and enter your O'Reilly School username and password when prompted. This will automatically log you on to the correct server, where you'll see this kind of prompt:

INTERACTIVE SESSION:

```
cold1:~$
```

When you see this prompt, you are logged in and ready to work! After you finish working on your coursework, log out from the machine using the exit command:

INTERACTIVE SESSION:

```
cold1:~$ exit
```

Great! You know to access your Linux prompt. Now, before we start using Linux, I'd like you to know a little about its history and why you've made a smart decision in choosing to learn the language:

A Brief History of Linux

Linux is a Unix clone. Unix was created at Bell Laboratories in the late 1960s. Several variants of Unix are still in use today, including those maintained by IBM and HP. Unfortunately for many users, Unix has always been pretty expensive. In the 1990s, in order to get around those prohibitive costs, programmers created a low-cost alternative to Unix called "MINIX" and made it available to educational institutions. This variant had some licensing problems though, which motivated Linus Torvalds, a student at the University of Helsinki, to write his own Unix clone that would become "Linux," a free alternative to Unix.

One key advantage of Linux over other Unix variants was that it could be run on machines based on Intel x86 processors, which were considerably cheaper and more widely available than systems that ran proprietary Unix variants. It's important to make a distinction between the *kernel* and the *operating system*. The kernel is the piece of code that sits at the heart of the operating system, and is responsible for interfacing between the hardware and the rest of the operating system. The operating system comprises the rest of the software on the system, from utilities and shells, to web and mail server software. Linux is not an operating system. In fact, Linux is a kernel. While some people will refer to an entire operating system as Linux, this isn't technically correct. Instead, we refer to the "Linux-based operating system".

Initially, Linus used a Linux kernel with MINIX components to flesh out the rest of the operating system. Eventually, people began replacing the MINIX components with components from the GNU ("**G**NU's **N**ot **U**nix," a recursive acronym) project. The GNU software licensing remains much less restrictive than Unix; anybody can use it. Additionally, both Linux and the GNU tools are *open source*; anybody can view and/or modify the source code. The low cost and simplified licensing process made Linux and GNU extremely popular in the educational sector, especially with researchers.

Eventually, Linux-based operating systems began finding their way into the server space previously dominated by proprietary Unix variants from companies like IBM, SGI, and HP. Over the next several years Linux's market share grew, while the installed base of Unix systems shrank.

The Current State of Affairs

Today, Linux-based operating systems make up the majority of the world's installed servers. Some estimates place the number of servers running Linux at over 60%; this number continues to climb. In the supercomputing field, as of 2012 Linux is run on a whopping 91% share of the top 500 fastest computers. This is truly amazing for a free kernel and operating system! You might imagine that in order to be free of charge, Linux-based operating systems would be maintained by volunteers in their spare time. While this is true for some software included with many Linux-based operating systems, much of the software is actually maintained by corporations. Companies such as IBM, HP, and RedHat devote considerable resources to developing Linux and contributing software to Linux-based operating systems because it has such a huge server market share. Interestingly, some of these companies continue to maintain and sell their own proprietary Unix variants while also contributing to Linux.

While Linux has enjoyed a huge market share in the server space, it hasn't done quite so well on the desktop. Estimates put Linux's desktop market share at around 1%. This might not sound encouraging, but many workplaces in both the public and private sector are discovering the advantages of running Linux on the desktop. It may be several years before Linux catches on as well in this area, but it has already made a huge impact in one application that lots of people use everday: Google's Android phone. Android's operating system is based on the Linux kernel. In a really short time, Android has become a major player in the phone and tablet operating system market. With this kind of success in the handheld device realm, along with recent market trends to move away from desktop computers to portable computers, it's easy to imagine Linux gaining market share away from Microsoft and Apple fast.

Right now, the future of Linux looks bright. I expect it to continue to dominate the server space as it gains more support from server vendors and the community. Major operations like Google, Facebook, and Amazon have staked their future on Linux, and more and more companies continue to follow this trend. With the ever increasing deployment of Linux servers, more systems administrators with Linux experience will be needed. When you finish this series of courses, you'll be ready with the skills you need to ride the Linux wave!

Moving Forward

Now that you're familiar with the OST way of learning and know how to access our system, you're ready to roll! Make sure you answer all quiz questions and complete any project objectives listed in the syllabus for this lesson. In the next lesson we'll learn about files and directories on Linux. See you there!

File System: Listing Files, Directories, and Navigation

In this lesson, you'll learn how to list files, manage directories, and also get familiar with the layout of the Linux filesystem. Knowing your way around the filesystem will save you lots of time and effort.

Listing Files

Earlier you learned how to log into your own Linux machine. If you aren't logged in already, go ahead and do that now. Once you're logged in, run your first command:

INTERACTIVE SESSION:

```
cold1:~$ ls
 (list of files in your home directory)
cold1:~$
```

The **ls** command is used to list the contents of a directory. You may have gotten some output from the command, but if you didn't, don't worry! Eventually you'll have several files and subdirectories in your directory and the output of **ls** will be much more interesting. As far as utilities on your Linux machine go, **ls** is pretty straightforward; it's also going to be one of the commands you use most often. Now, let's create some subdirectories so you have something to look at with **ls**:

Directories

If you're familiar with Windows or Mac OS, you've worked with folders before. You can put files and other folders inside of folders in order to keep everything organized. We use that same concept in Linux, except instead of calling our containers "folders," we call them *directories*.

Whenever you're logged into a machine, you're "in" a directory somewhere on that machine. This directory is called the *current* or *present working directory*. Unless you specify otherwise, most operations will be performed in or on your present working directory. In order to find out where you are, you can use the command **pwd** (which is short for **P**resent **W**orking **D**irectory). Let's find out where we are now:

INTERACTIVE SESSION:

```
cold1:~$ pwd
/users/username
```

Okay, so we are in **/users/*username***, which means that you are in a directory that's named after your username, and that directory is inside a directory named **users**. The forward slash (**/**) is a *delimiter* between the directory names. Directories that appear to the right of the **/** are inside of the directory that appears to the left of the **/**. This is a special directory, and it's called your *home directory*. There are many other directories in the system, but every time you log in, you will start in this directory.

Making and Removing Directories

In order to make a directory, we use the **mkdir** command. You'll notice that a lot of Linux commands tend to be abbreviated versions of what they do. Okay, let's make a directory named **sysadmin1**:

INTERACTIVE SESSION:

```
cold1:~$ mkdir sysadmin1
cold1:~$ ls
sysadmin1
```

Great! Now let's make another directory named **example**:

INTERACTIVE SESSION:

```
cold1:~$ mkdir example
cold1:~$ ls
example sysadmin1
```

Now you have two *subdirectories* in your home directory: **sysadmin1** and **example**. But suppose you change your mind and want to get rid of your newly created **example** directory. To remove a directory, use **rmdir**:

INTERACTIVE SESSION:

```
cold1:~$ rmdir example
cold1:~$ ls
sysadmin1
```

Now you are back to just having **sysadmin1** in your home directory.

Moving Around the Filesystem

To move from one directory to another, use the **cd** command, which stands for **c**hange **d**irectory. Try the code below to see how **cd** works and how it affects your present working directory:

INTERACTIVE SESSION:

```
cold1:~$ cd sysadmin1
cold1:~/sysadmin1$ pwd
/users/username/sysadmin1
cold1:~/sysadmin1$ ls
cold1:~/sysadmin1$ cd ..
cold1:~$ pwd
/users/username
cold1:~$
```

So, what did we just do there? First, we changed directory into **sysadmin1**. We can tell that we changed to the **sysadmin1** directory by the output of **pwd**. A quick **ls** shows us that we don't have anything in **sysadmin1** yet. But what does **cd ..** do? The **..** is actually a special directory name, representing the *parent directory*. There is a **..** in every directory, and if you **cd** to that directory, it actually moves you up one directory from the current one. Additionally, there is a dot (**.**) directory. The **.** directory corresponds to the present working directory. These directory entries are hidden in a normal **ls** output. You can also **cd** to a specific directory in the filesystem. Notice how your prompt changes depending on what directory you are in. A tilde (**~**) is shorthand that indicates, "your home directory":

INTERACTIVE SESSION:

```
cold1:~$ cd /
cold1:/$ pwd
/
cold1:/$ ls
bin  boot  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  repos  root
sbin  seLinux  srv  sys  tmp  usr  var
cold1:/$ cd /usr/bin
cold1:/usr/bin$ pwd
/usr/bin
cold1:/usr/bin$ cd
cold1:~$ pwd
/home/username
```

Note Running **cd** without specifying an argument like you just did takes you back to your home directory. It's a nice little shortcut to get you home in a hurry!

Now let's put together the commands you've learned so far — **cd**, **mkdir**, **rmdir**, and **ls**. Change directory into **sysadmin1** and create a directory called **test**:

INTERACTIVE SESSION:

```
cold1:~$ cd sysadmin1/
cold1:~/sysadmin1$ mkdir test
cold1:~/sysadmin1$ ls
test
```

Okay, now change directory back to your home directory and remove **sysadmin1**:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ cd
cold1:~$ rmdir sysadmin1
rmdir: failed to remove `sysadmin1': Directory not empty
```

That didn't work so well. The **rmdir** command won't allow you to remove a directory unless it's empty. We'll leave **sysadmin1** where it is for now, and discuss ways of removing non-empty directories later.

The Linux Filesystem Layout

The Linux Filesystem is organized in a tree-like structure, with all other directories residing in one directory called the *root directory* (someone really liked that tree analogy!). This directory is usually indicated with a single forward slash mark (**/**). There are several directories that reside directly within the root directory; you can see them by using **ls** (and note that by using **ls /**, we can avoid changing to the root before listing its contents):

INTERACTIVE SESSION:

```
cold1:~$ ls /
bin boot dev etc home lib lib64 lost+found media mnt opt proc repos root sbin se
linux srv sys tmp usr var
```

That's quite a few directories! Each one serves a purpose in organizing the system files and utilities that make Linux work. One of the most important things you will learn in this course is the role that each of these directories play in your system. Once you know what they do, it's much easier to find what you are looking for when you are trying to accomplish a systems administration task. Here's a list of some of the more commonly accessed directories and their roles:

/bin	/bin contains binaries, or programs, that users have access to. This is where utilities like **ls** are actually located.
/etc	/etc holds the systemwide configuration files. If you want to configure something like the system's hostname or network, this is where you would find it.
/home	/home generally contains all users' home directories. Users keep their personal files inside their home directory.
/lib	/lib is where systemwide libraries are stored. Libraries are collections of functions that other programs use to do their job.
/root	/root is a special home directory reserved for the root user.
/sbin	/sbin contains system binaries, which are generally used to administer the system.

/usr	/usr holds binaries, libraries, documentation, and other odds and ends that are generally not essential to system operation.
/var	/var is used for variable data, such as log files, mail spools and lock files.

Note Not all of the directories in **/** have been outlined in this table. Some of them will pop up in more advanced usage, and we'll address them at that time.

The Instruction Manual

Yes, believe it or not, your machine comes with an instruction manual! Almost all the commands on your system are documented with *manpages* (which is short for "manual pages"). To see the manpage for a command, type **man** *command*. For example, try the manual page for mkdir by typing **man mkdir**. By default on your system, man uses **less** to display the **man** page. This means that the commands you learned for controlling **less** can be used to control **man** output. Most **man** pages adhere to more or less the same format. The **man** pages is the first place you look for information regarding a command on your system; the **man** pages often contain options for the command, as well as real-world usage examples. In fact, of all the commands you will learn in this course, **man** may be the most useful. Before you move on to the next section, spend some time reviewing the man pages for some other commands you already know.

That's it for your first full lesson. You're doing great so far. In the coming lessons we'll be discussing the filesystem and related tools in greater depth. Complete your assignments for this lesson before you move on to the next. See you later!

File System: File Manipulation and Links

In this lesson we 'll learn about some basic file manipulation tools, as well as links.

File Manipulation Tools

There are few commands more useful for handling files and directories than **cp**, **mv**, **touch**, and **rm**.

Copy and Move

The **cp** command copies files and directories. It doesn't remove the originals, it just duplicates them elsewhere, which is useful when you want to edit, but keep the original intact. The general format for the **cp** command is:

OBSERVE:

```
cp source destination
```

Here's an example of **cp** in action:

INTERACTIVE SESSION:

```
cold1:~$ cd sysadmin1/
cold1:~/sysadmin1$ cp /etc/hosts ./
cold1:~/sysadmin1$ ls
blue  green  hosts  red
```

In the **cp** command above, the **/etc/hosts** file is copied to the present working directory. The present working directory is specified by a dot and forward slash (**./**) (if you use that shorthand, you don't have to type out the full path).

Next, you'll copy a directory. To do this, you use the **-r** *flag* in the cp command. Command line flags allow you to specify alternate behavior for the command you are running. Each command has its own set of flags; we'll introduce the more important flags as you use them. The **-r** flag tells **cp** to copy the directory and its contents **r**ecursively. Make a copy of the directory **green** called **yellow**:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ cp -r green yellow
```

This will create a copy of **green** (including all its contents) called **yellow**. Let's take a look at the current contents of **sysadmin1** directory using the **-l** flag to specify that the **l**ong output format should be used. This will provide more information in the **ls** output about files and directories.

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ls -l
total 20
drwxrwxr-x 3 username webusers 4096 Nov 21 11:43 blue
drwxrwxr-x 2 username webusers 4096 Nov 21 11:43 green
-rw-r--r-- 1 username webusers 43   Nov 21 16:29 hosts
drwxrwxr-x 3 username webusers 4096 Nov 21 11:43 red
drwxrwxr-x 2 username webusers 4096 Nov 21 16:29 yellow
```

Note You may have noticed that your ls output is displayed in different colors. This is a feature in ls designed to let you distinguish various file types. It has been turned on by default on your system, but that won't always be the case. Normal files (like text files) will be either white or black, depending on the background color of your terminal. Directories are **blue** and executable files are **green**. (We'll go over what it means for a file to be executable a little later.)

If you are using the cp command to copy a file and you specify a directory as the destination, and that directory exists, the file will be copied into that directory. If the directory does not exist, the copy will fail. If you specify a filename as the destination, the file will be copied to the destination file. For example, **cp /etc/hosts /home/username/my_hosts** will result in the contents of the file on **/etc/hosts** being written into the file **/home/username/my_hosts**.

The **mv** command moves files rather than copying them. This is functionally equivalent to copying a file and then removing the original. Try out the mv command by moving the directory **/yellow** inside of **/green**:

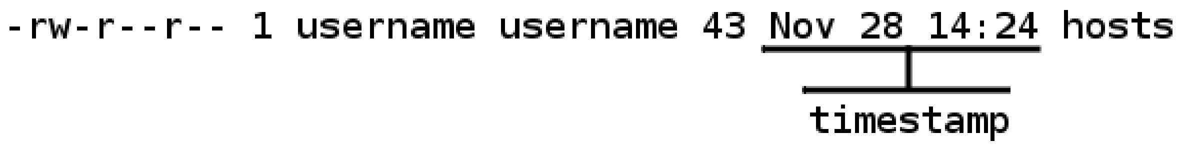
INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ls
blue  green  hosts  red  yellow
cold1:~/sysadmin1$ mv yellow green
cold1:~/sysadmin1$ ls
blue green hosts red
cold1:~/sysadmin1$ ls green
yellow
```

Note If you specify a directory to ls, it will return a listing of the files contained in that directory.

Touch and Remove

The **touch** command updates the time stamp on a specified file. We can find a file's current time stamp by taking a look at the long output from ls:



```
-rw-r--r-- 1 username username 43 Nov 28 14:24 hosts
```

In this example, our file was last edited on November 28th at 14:24 (or 2:24 PM). Now let's see what happens when we **touch** the file:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ touch hosts
cold1:~/sysadmin1$ ls -l hosts
-rw-r--r-- 1 username username 43 Nov 28 14:38 hosts
cold1:~/sysadmin1$ date
Mon Nov 28 14:38:09 CST 2011
```

The time stamp on the hosts file has been updated to 14:38. Okay, that may not be the most fascinating thing in the world, but **touch** does have another more interesting ability. If the file you specify to **touch** does not yet exist, **touch** *creates it for you*. This is handy when you want to create files for testing a script, or you have a piece of software that will not start when a particular file does not exist. Create some files using **touch**:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ touch one two three
cold1:~/sysadmin1$ ls
blue green hosts one red three two
```

Note Many commands let you specify multiple items or arguments on which to take action. This is true for all the commands we've covered so far, except for **cd** (you can't be in more than one directory at the same time in a particular shell). For example, you can copy multiple files to a directory by running **cp file1 file2 file3 /path/to/directory/**. Generally this works when you have multiple sources and only one destination, but not the other way around.

The **rm** command removes files and directories. As with the **cp** command, you must include the **-r** flag to operate on a directory. We don't need our copy of the **hosts** file anymore, so let's delete it:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ rm hosts
cold1:~/sysadmin1$ ls
blue green one red three two
```

Links

In the Linux filesystem, links are used to give a file or directory multiple names. There are two kinds of links: *hard links* and *soft links*. Soft links, also known as *symbolic links* (or *symlinks* for short), are used far more often than hard links in a Linux system. Both kinds of links are created using a tool named **ln**.

Soft Links

Soft links are useful in a variety of situations. For example, say you have a piece of software that requires the (fictional) utility **super_do_stuff**, but you have the functionally equivalent **mega_do_stuff** installed. Other programs or scripts may depend on **mega_do_stuff**, so you can't just use **mv** to rename it to **super_do_stuff**. At the same time, you don't want to use **cp** to copy it to **super_do_stuff**, because you don't want to maintain two copies of the same program. However, you can use a soft link to refer to the file **mega_do_stuff** by the name **super_do_stuff**. This allows you to access the utility using either name, while only having one copy of the program on your hard drive. Let's try creating a soft link, using the **-s** flag to specify that a soft link should be created.

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ln -s ./blue ./indigo
cold1:~/sysadmin1$ ls -l indigo
lrwxrwxrwx 1 username webusers 6 Dec 5 11:26 indigo -> ./blue
```

The long output of the **ls** command tells us to which file our soft link points. So, what happens when we perform an operation that involves our link?

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ cp three indigo
cold1:~/sysadmin1$ ls indigo
three
cold1:~/sysadmin1$ ls blue
three
```

Because **indigo** is a soft link to **blue**, copying a file to **indigo** actually copies the file into **blue**. You can remove the link named **indigo**, and the **three** file will still be in **blue**. If you were to keep **indigo**, but remove **blue**, you'd be left with a *broken link*. A broken link occurs when the file or directory that a soft link points to no longer exists. Try this:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ mkdir dir1
cold1:~/sysadmin1$ ln -s dir1 dir2
cold1:~/sysadmin1$ rmdir dir1
cold1:~/sysadmin1$ cp /etc/hosts ./dir2
cp: not writing through dangling symlink `./dir2'
```

In that example, the directory **dir1** is created and then a link named **dir2** is made to point to **dir1**. Then **dir1** is removed, which results in a broken link. Attempting to copy a file to the broken link results in an error. That's no good. Let's go ahead and remove that broken link:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ rm dir2
cold1:~/sysadmin1$
```

Hard Links

Hard links are fundamentally different from soft links. Rather than acting as a pointer to another file, a hard link points to data on a storage device. When you create a hard link to an existing file, that link points to the same blocks of data to which that the original file pointed. Any changes you make to the data, regardless of the hard link you use to access it, will be reflected across all hard links. Unlike soft links, if you remove the original file, any hard links to that file remain intact and accessible. The data contained in the file only truly gets removed when you remove the last hard link to it.

Note Due to several limitations of hard links, we don't recommend using them. Stick with soft links, they are your friends.

So far, you've got copying, moving, removing, and linking in your bag of tricks. You're making some excellent progress! After you finish your homework, you can move on to the next lesson where we'll discuss how to limit access to your files and directories by other users. See you soon!

File System: Permissions

This lesson is all about file permissions. File permissions keep files safe and secure by allowing only certain people or groups of people to access them.

Access Control

Controlling access to files and directories on a Linux machine is an important aspect not only of system administration, but also Linux in general. While it's the responsibility of the system administrator to keep system files secure, it falls to individual users to secure their own files. Linux controls access to files and directories in three ways: *read*, *write*, and *execute* access. Each of these access types have different meanings depending whether you are referring to a file or a directory.

	file	**directory**
Read (r)	Read the file	List files and directories in the directory
Write (w)	Edit the file	Add files and directories to the directory
Execute (x)	Execute the file as a program	Change into the directory using cd

These three access controls can be specified individually by:

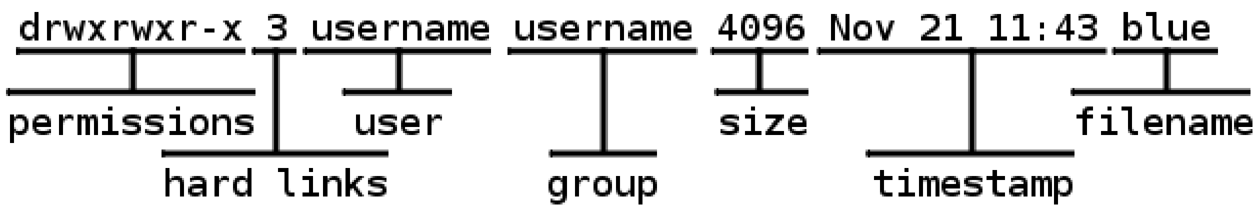
- the user who owns the file or directory.
- a group of users who may need some access to the file or directory.
- every other user on the system who does not fall into either of the first two categories.

Linux groups are organizational units used to allow sets of users with common needs to share a set of permissions on one or more files or directories. This is useful, for example, when there is a file that you want to be able to read from and write to yourself, and allow only some coworkers to have permission to read it (but not modify it). As we did in earlier lessons, we'll be working in **sysadmin1**. Let's use a tool we're already familiar with to list the permissions on your files:

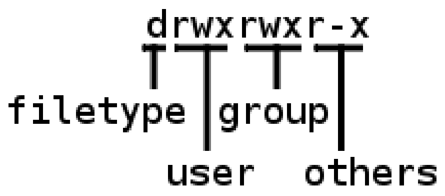
INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ls -l
total 12
drwxr-xr-x 2 username webusers 4096 Jun 26 15:49 blue
drwxr-xr-x 2 username webusers 4096 Jun 26 15:49 green
drwxr-xr-x 2 username webusers 4096 Jun 26 15:49 red
```

Remember, supplying the **-l** flag to **ls** tells it to give you a *long* listing. Let's deconstruct this output to see what we have.



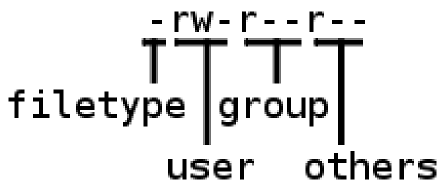
For now, we are primarily interested in the first, third, and fourth columns, which list the permissions, user, and group ownership, respectively. When you list your files, the directories you created are owned by you, and their group ownership is assigned to your group. The first column lists the permissions for the file, as well as the file type. Here's how to decode those letters:



Our permissions tell us that our file is of type **d**irectory, the owner (user) of the file has **r**ead, **w**rite, and e**x**ecute permissions, the group has **r**ead, **w**rite, and e**x**ecute permissions, and all other users have only **r**ead and e**x**ecute permissions. If you see a hyphen (**-**) in the first position, that means the file is just a plain old file, not a directory. If you see a **-** in any other position (in our example, the **w**rite position for the other users), that set of users does not have that permission.

Note Yes, directories are considered files in Linux. In fact, everything is a file in Linux, from hard drives to your ttys.

This is another set of permissions that you are likely to see:



The hyphen (**-**) in the first column tells us that we are looking at a regular file, and our permissions are **r**ead and **w**rite for the user and read-only for the group and others.

Setting Permissions

Now that you can list and understand the permissions on a file, how do you set them? There are three tools that are useful for this task, but for now we will only discuss two of them: **chmod** and **chgrp**. **chmod**, short for **ch**ange **mod**e, operates on the actual r, w, and x permissions of the file. **chmod** has two ways of setting permissions. The first way uses letters as shorthand for both the user class (user, group, and other) and the permissions themselves (rwx). For example, to make it so that our **red** directory is not readable, writable, or executable by anybody in the "others" category in the system, let's run these commands:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ chmod o-rwx red
cold1:~/sysadmin1$ ls -l
total 12
drwxrwxr-x 3 username webusers 4096 Nov 21 11:43 blue
drwxrwxr-x 2 username webusers 4096 Nov 21 11:43 green
drwxrwx--- 3 username webusers 4096 Nov 21 11:43 red
```

See the results in the output? We have removed read, write, and execute permissions from the others for red. Now let's change it back to the way it was before when the others had read and execute permissions on red:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ chmod o+rx red
cold1:~/sysadmin1$ ls -l
total 12
drwxrwxr-x 3 username webusers 4096 Nov 21 11:43 blue
drwxrwxr-x 2 username webusers 4096 Nov 21 11:43 green
drwxrwxr-x 3 username webusers 4096 Nov 21 11:43 red
```

You can build the string that is used to specify the permissions change to **chmod** like this:

1. Modify one or more of these user classes: u, g, and o for user, group, other , respectively.

2. Choose either + or - to add or remove permissions.
3. Select one or more of these permissions to add or remove: r, w, and x.

You can specify multiple modifications in one command by supplying additional modification strings to the **chmod** command, separated by commas, for example: **chmod u+rw,g+rw,o-rwx red**. This first method of changing permission using **chmod** is really handy for doing quick permissions changes, like making a script you just wrote executable so that you can run it; however, it can become more cumbersome to use when you know which specific permissions you want for a file and several modifications must happen to get what you want, or you have to modify multiple files so that they all have the same permissions, but those files start with different permissions.

The second way of using the **chmod** command is to specify exactly which permissions you want your file to have, regardless of its current state. While it will seem more complicated at first, after you get familiar with a few basic rules, it will become second nature to you. Using this method, each permission—r, w, and x—is assigned a value (4, 2, and 1, respectively), then you calculate the permissions for each class by adding these values together. Let's work an example in a directory you already have: blue. Here's what you want to be able to do:

- As the owner, you want read, write, and execute (rwx) access.
- For the group permissions, you just want read and execute (rx) access.
- For everyone else on the system, you don't want to grant any access.

So, for the user field, we have rwx, which means we have 4+2+1=7. Next, for the group field, we have rx: 4+1=5. Finally, the others have no permissions, so we have 0 for that field. When we have all three of these numbers figured out, we just put them together in order: **750**. Let's see what happens when we apply these permissions to blue:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ chmod 750 blue
cold1:~/sysadmin1$ ls -l
total 12
drwxr-x--- 3 username webusers 4096 Nov 21 11:43 blue
drwxrwxr-x 2 username webusers 4096 Nov 21 11:43 green
drwxrwxr-x 3 username webusers 4096 Nov 21 11:43 red
```

Cool, huh?

Setting Group Ownership

So far all of your directories are owned by you and your group. This is fine in most situations, but sometimes you have to share files with people, or make your files readable by a process like a web server, and you don't want those people or processes to be able to modify your files. We've already discussed setting group permissions, but without setting the group ownership of a file or directory, the group permissions field isn't very useful. This is where the **chgrp** command comes in—it lets you change the group ownership of any file or directory that you own to any group of which you are a member. In order to find out which groups have you as a member, do this:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ groups
webusers courses sysadmin1_151 technologies linuxShellUsers students
```

Your list of groups will probably vary from the list in the example above. You can set the group ownership of any of the files or directories that you own to any of these groups. For now, we'll just use the students group. To see what **chgrp** does, modify your **green** directory:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ chgrp sysadmin1_151 green
cold1:~/sysadmin1$ ls -l
total 12
drwxr-x--- 3 username webusers      4096 Nov 21 11:43 blue
drwxrwxr-x 2 username sysadmin1_151 4096 Nov 21 11:43 green
drwxrwxr-x 3 username webusers      4096 Nov 21 11:43 red
```

The group ownership of the directory green is now set to **sysadmin1_151**. Based on the permissions settings for the directory **green**, users who belong to the **sysadmin1_151** group have the ability to list the contents of **green**, write new contents to **green**, and change directory into **green**.

You have some invaluable knowledge for working with Linux based systems now. Correct file permissions are the cornerstone of good system security! Don't forget to hand in your homework for this lesson before moving on to the next lesson where we will discuss an important tool for any systems administrator — the text editor...

Editing Text

In this lesson, you will learn how to use a valuable tool to use and administer Linux systems: the text editor. While there are lots of text editors available, we'll concentrate on the most widely available editor: vi. .

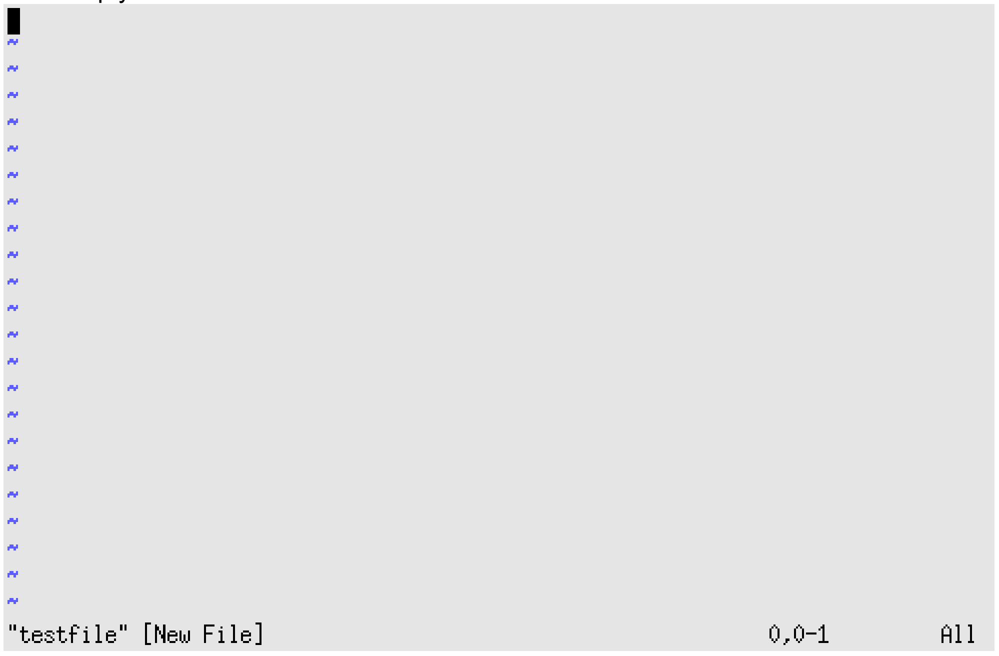
A Tale of Two Modes

vi does not accept command input and text input at the same time. In order to send command input (for tasks like saving a file or searching for text), you must switch to **command** mode; in order to edit text, you must switch to **insert** mode. This might sound strange and maybe a tad inconvenient, but it will begin to make more sense as you use it. To see these modes in action, start **vi**:

INTERACTIVE SESSION:

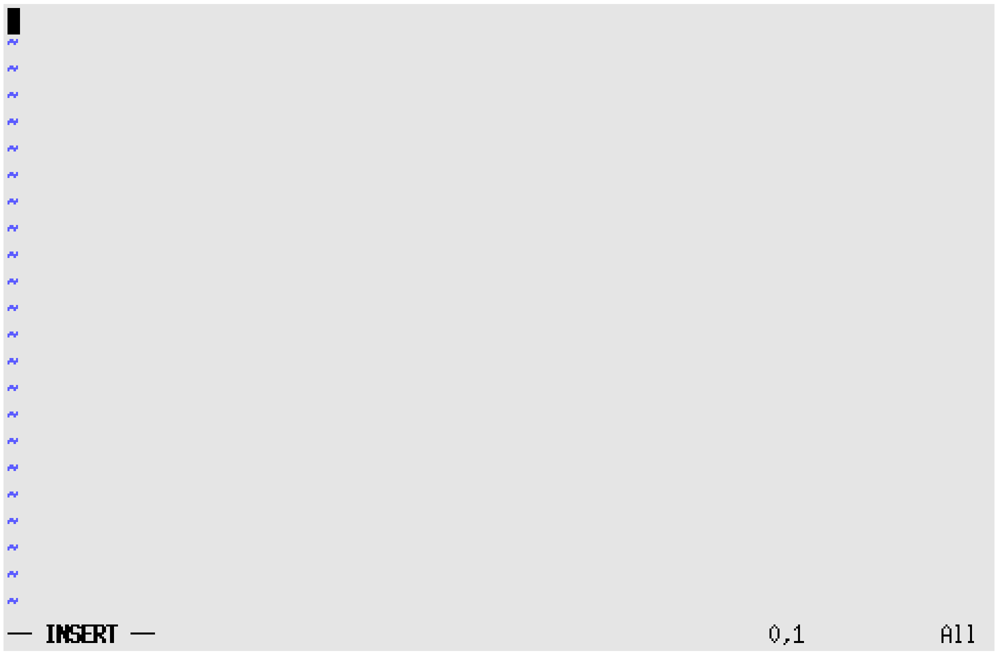
```
cold1:~$ vi testfile
```

vi opens with a new empty file:



```
█
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"testfile" [New File]                                    0,0-1         All
```

By default, you are in the command mode. Let's switch to insert mode and enter some text. Press the **i** key; you'll see this screen:



```
█
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
— INSERT —                                       0,1          All
```

Note When you're in insert mode, it says **-- INSERT --** in the lower-left corner of the window.

Go ahead and enter some text. Create a couple of lines of text by pressing **Enter** to put your cursor on a new line. If you type a line that's longer than your window is wide, the cursor will auto-wrap to the next line in the window, but this does not equate to a new line in the file. We'll discuss that in greater detail later. When you're happy with what you've written and you want to save the file, it's time to switch back to command mode. Press **Esc**; **-- INSERT --** will disappear from the lower-left corner. This indicates that you have entered command mode. Type **:wq** and **:wq** will appear in the lower-left corner. Finally, press **Enter** to **w**rite and **q**uit. Here's an outline of some of these operations:

Esc	Enter command mode.
:wq	Write the file to disk and quit the editor.
:w *filename*	Write the contents of the file to *filename*.
:q	Quit the editor. If you have made changes but have not saved them, you will be prompted to use **:q!** to quit without saving.

Note While there are some commands in command mode that require you to start with a colon (**:**), most editing operations, like those for copying and pasting text, do not require it specifically. We will show the leading colon (**:**) for any commands that require it in this lesson.

There's the command prompt again. Let's check to see if your new file is there:

INTERACTIVE SESSION:

```
cold1:~$ ls -l testfile
-rw-r--r-- 1 username webusers 122 Nov 7 16:07 testfile
```

You have created and edited your first file! In the next section, we'll reopen the file we just created and learn some more functionality.

Navigating in Insert Mode and More Ways to Insert Text

Now you have the power to open and save files in **vi**, but that power doesn't do you much good if you can't move around and edit things. The arrow keys on your keyboard are sufficient when you want to move your cursor around in most environments. It works in both command and insert modes. While this means of getting around is fine for making minor edits that are relatively close to the start of the file or line, sometimes it's handy to have shortcuts that help you move around faster. Here are some shortcuts you can use to get around in command mode:

Keystroke(s)	Description
:*n*	Goes to the *n*th line in the file. For example, **:30** takes you to the 30th line. Use **:0** to go to the beginning of the file.
G	Goes to the last line in the file.
^	Goes to the beginning of the line.
$	Goes to the end of the line.

Note The **^** and **$** are important characters outside **vi** too. Outside of **vi** they generally signify the beginning and end of a line, respectively. These characters will come up again later, in our section about Regular Expressions.

Take a minute to practice moving the cursor around your text file using these shortcuts. Remember, you must be in command mode for these keystrokes to work as intended. Once you're comfortable navigating in your text file, you can explore some alternative ways to add text to your file. You can always just hit **i** in command mode to switch to insert mode and use your arrow keys to move around the file, but sometimes, when you have long lines of text or you want a more direct way to append text to the end of a line, you'll want something faster than **i**. There are common keystrokes we can use to enter insert mode and place your cursor where you want it. Practice using each shortcut key to see how they behave.

Key	Description
i	Inserts text to the left of the current position.
I	Inserts text at the beginning of the current line.
a	Appends text to the right of the current position.
A	Appends text at the end of the current line.
o	Opens a new line below the current line.
O	Opens a new line above the current line.

Note You might have noticed a pattern developing here. *Lowercase* shortcut keys perform a function and *uppercase* shortcut keys perform a modified version of that function. In most of the cases we will cover, you can equate the lowercase function to "do this after my current position" and the uppercase function to "do this before my current position." The **i**, **I**, **a**, and **A** keystrokes are notable exceptions to this rule.

Cut, Copy, Paste and Delete in Vi

Every good text editor should possess the ability to cut, copy, and paste blocks of text in order to save work for the user, and **vi** is no exception. All of these commands must be entered in command mode (keep in mind the previous note about **lower**case versus **UPPER**case functions):

Key	Description
x	Cut (delete) one character after the current position.
X	Cut (delete) one character before the current position.
dw	Delete a word. This means from your current position up to and including any whitespace (one or more spaces or tabs).
dd	Delete an entire line at your current position.
yw	Yank (copy) a word. This means from your current position up to and including a whitespace character.
yy	Yank (copy) the entire line at your current position.
p	Paste text from the buffer after the current location. If the text is a line or multiple lines, the paste will start on a new line after the current line.
P	Paste text from the buffer before the current location. If the text is a line or multiple lines, the paste will start on a new line before the current line.

Note All the "deleted" text from the **x**, **X**, **dw**, and **dd** commands is written to the paste buffer, just as it is for **yw** and **yy** commands. This allows these commands to function as a "cut" in addition to a "delete." Subsequent cuts or deletes overwrite the paste buffer, so you'll have access only to the text that you last cut or deleted.

Repeating Actions

Copying and pasting one line at a time can be useful, but what if you want to copy a whole chunk of text and paste it elsewhere? Or delete ten lines out of your file? Or paste the same line 12 times? Vi has a straightforward solution to these problems. Any time you want to repeat a command *n* times, you prefix the command with *n*. For example, to delete the next 8 lines of a text file, type **8dd**. Or to cut 6 lines of text from one part of a file and move it elsewhere, place your cursor on the first of the 6 lines you want to cut, enter **6dd**, and then move your cursor to where you want it and type **p** to paste it.

Searching for Text

When you're editing a large file and want to find a specific word or phrase in the file, it helps to have a search function. This is another helpful function that **vi** provides. **vi** also allows you to search for a string, and then replace that string with something else. Searches can be run from command mode. This chart outlines the basic search functionality in vi:

Key	Description
/	Initiates a search for a string. Typing **/cheese** will search for the string "cheese" after your current position in the file.
?	Initiates a backwards search for a string. Use this if you want to find a string before your current position in the file.
n	Repeats your last search to find the next instance of a search string.

The search/replace operation is a little different. By default, the search/replace command searches only the line where your cursor currently sits. The simplest search/replace operation can be carried out by typing **:s/***search_string***/***replace_string***/** in command mode. This operation finds the first instance of *search_string* in your current line and replaces it with *replace_string*. This is useful if you have only one instance of a string in your line that you want to replace. But if you wanted to, say, change all instances of "Linux" to "Unix" on a given line, you'd have to modify the statement slightly. The new statement would look like **:s/Linux/Unix/g**. That extra **g** at the end indicates that the search operation is to be executed globally, or across the whole line. Finally, if you want to execute your search/replace over every line in file, use **:%s/Linux/Unix/g**. The **%** before the **s** tells vi to search all lines.

vi Cheatsheet

It will probably take a while to get used to all the functions that vi has to offer, so for your convenience, we've provided a link to a vi cheatsheet.

Alright then, you're really starting to get the hang of this stuff. Good for you! Finish up the homework and see you in the next lesson!

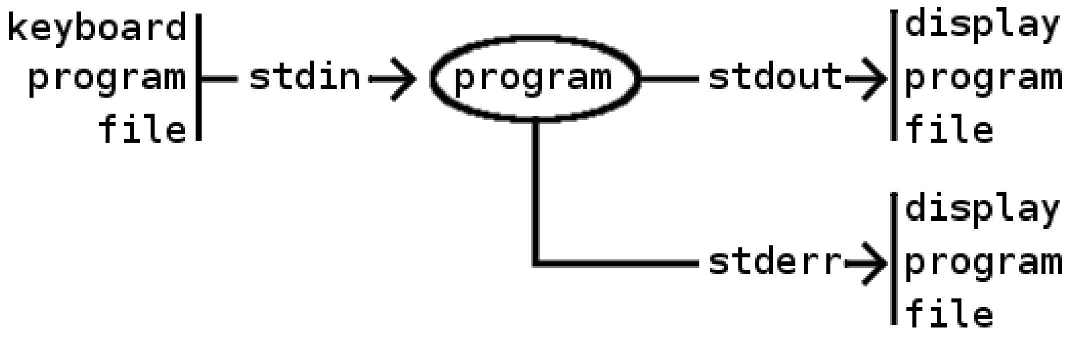
Bash: Input and Output

In this lesson we introduce Bash and some of its features. Bash is a *shell*, which is a utility; its primary purpose is to receive user input, perform actions based on that input, and then return output. So far, everything you have typed into your console has been handled by Bash. Bash stands for the **B**ourne-**A**gain **Sh**ell. It was written as a replacement for the Bourne shell (which explains how we get "Bourne-Again"). Bash is the standard shell on most Linux distributions.

The Standard Streams

In order to understand what Bash does for us, we must first talk about *Standard Input*, *Standard Output*, and *Standard Error*. Generally, these are referred to as *stdin*, *stdout*, and *stderr*. These three *standard streams* are used to communicate with programs. Stdin is the stream that provides data to the program and stdout is where data is output from the program. Stderr is another output stream, but it is usually reserved for programs to output error messages.

Bash uses its standard streams to communicate with the user, taking input from the user, performing actions, and then reporting results back to the user. When some program or utility is run from within a shell, the shell handles the standard streams of that program, giving the opportunity to redirect input and output in useful ways. The diagram below illustrates how data flows in and out of a program. Notice that stdin can take data from the keyboard, another program, or a file and stdout and stderr can output data to the display, another program, or a file.



Each of these streams can be configured to input and output to and from different sources, independently, with the exception that stdout and stderr cannot be redirected to two different programs. The default behavior, as you've seen, is for stdin to come from the keyboard and for stdout and stderr to go to the display.

Redirects and Pipes

In a shell, a *redirect* is an operator that takes the output of a command and sends it to a file, or reads the contents of a file into the input of a command. There are two redirect operators, **<** and **>**. For now, we'll focus on the **>** operator, which redirects output to a file. To see this in action, we'll send the output of a command to a file, and then look at that file using the **cat** command:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ echo "this is my string" > output.txt
cold1:~/sysadmin1$ cat output.txt
this is my string
cold1:~/sysadmin1$
```

Note **Cat** is a handy little program. It reads the contents of the specified file(s) and prints them to the screen.

The default behavior, as you've seen, is for stdin to come from the keyboard and for stdout and stderr to go to the display. Referring back to the above example where we redirected the output of the **echo** command to a file named **output.txt**, we can describe the behavior as: "The stdout stream for the **echo** command was redirected to the file **output.txt**." Now try redirecting the stdout stream from **ls -l** to **output.txt** and use **cat** to view the results:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ls -l > output.txt
cold1:~/sysadmin1$ cat output.txt
total 20
drwxr-x--- 2 username  webusers      4096 Jun 26 15:49 blue
drwxr-xr-x 2 username  sysadmin1_151 4096 Jul 24 16:20 green
lrwxrwxrwx 1 username  webusers         6 Jul 24 16:19 indigo -> ./blue
lrwxrwxrwx 1 username  webusers         7 Jul 24 16:20 lime -> ./green
drwxr-xr-x 2 username  sysadmin1_151 4096 Jul 24 16:23 permissions
drwxr-xr-x 2 username  webusers      4096 Jun 26 15:49 red
drwxr-xr-x 2 username  webusers      4096 Jul 24 16:19 yellow
```

What happened to the original contents ("this is my string")? The **>** operator overwrites the contents of the file to which it is redirecting, if that file exists. Sometimes this behavior is exactly what you want, but other times it's helpful to keep what you have and add more output to the end of the file. In order to append output to a file rather than overwriting the file, use the **>>** operator:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ echo "line one" > append.txt
cold1:~/sysadmin1$ cat append.txt
line one
cold1:~/sysadmin1$ echo "line two" >> append.txt
cold1:~/sysadmin1$ cat append.txt
line one
line two
cold1:~/sysadmin1$
```

While redirects are handy, pipes are even more useful. The figure above showed that input can come from some program via stdin and output can go to another program via stdout. The operator that is used to send data from one program to another is called the *pipe*, and it is symbolized by the vertical bar,**|**, located directly below the backspace key, on the same key as the "\" character, on most keyboards. Let's use the pipe to do something interesting.

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ls | grep txt
append.txt
output.txt
cold1:~/sysadmin1$
```

The stdout stream of **ls** is piped to the stdin stream of **grep**. The grep command then takes that data and searches though it for any string containing the substring "txt." In this case, both "output.txt" and "append.txt" match, and they are printed to the screen. Not only does grep search for strings supplied by its stdin stream, it can also search for **strings** in **files** when run as:

OBSERVE:

```
grep string filename
```

To demonstrate pipe a little better, run the **history** command and pipe its stdout stream to **grep**, searching for "echo." You'll see a list of commands you entered that contain the string "echo." This is certainly more convenient—and more precise—than visually scanning the whole output of the history command to find the instances in which you ran echo. There are many commands that can do useful things with the stdout stream of another program, from sorting and counting to much more complex tasks. You can even construct a chain of commands to achieve a particular goal using pipe. For example, suppose you wanted to see all the files in your home directory that contain both the strings "txt" and "out" in their filenames? You could do this:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ ls | grep txt | grep out
output.txt
cold1:~/sysadmin1$
```

Regular Expressions

grep stands for **gl**obal **r**egular **e**xpression **p**rint. A *regular expression* is a way of defining a pattern.

Within a grep search string, there are several special characters and flags you can use to narrow a search. For example, you can use the **-i** flag with the **grep** command to search for words without being case sensitive. That way you'd find occurrences of both **name** and **Name**. Another way to do this is to use brackets.

Observe the following:

```
cold1:~/sysadmin1$  grep [nN]ame /etc/services
```

This will find either **n** or **N** followed by **ame**. The brackets are used to give a list of possibilities. You can either type each character to be matched or you can give a list. For example, **[a-z]a[a-z]e** matches any lowercase letter (a through z), followed by an **a**, then any lowercase letter, and finally an **e**.

Here are some examples of other lists you can use:

[aeiouAEIOU]		matches any lower or upper case vowel.
[^aeiouAEIOU]		matches a non-vowel character. (The **^** means NOT when inside a bracket.)
[0-9]		matches any single digit.
[^0-9]		matches any character that is not a digit.
[a-z]		matches any lowercase letter.
[a-zA-Z]		matches any lower OR upper case letter.
[a-zA-Z0-9]		matches any digit or letter.

.

You can match any character by using a period.

For example, **n..e** will match the letters **n** and **e** with any two characters between them. It will therefore match **name**, **nine**, **nZWe**, **n3be**, and **n/(e**.

*

You can use an asterisk (*) to find zero or more occurrences of a character.

For example, **[nN]e*d** will match **Ned**, **need**, **nd**, and **neeeeed**. If you want to match one or more blank spaces, you must use quotes around the regular expression.

"a space" will find the word "a," followed by a space, followed by the word "space."

"a *lot" will match zero or more blank spaces, which will account for the common misspelling of "a lot" as "alot." Keep in mind, however, that it's testing each line for a match separately. So if the letter "a" is at the end of one line and the word "lot" is at the beginning of the next line, no match will be found.

^ and $

You can specify to match a pattern when it is at the beginning or end of a line. If **^** is used, it will match the word when it's located at the beginning of a line, so **^the** will match any line that starts with **the**.

If **$** is used, it will match the word when it's located at the end of a line, so **the$** will match any line that ends with **the**.

You can use any combination of these characters to match just about anything you want.

"^ *[A-Z]" will match any line that starts with zero or more spaces followed by an uppercase letter. It could be useful for finding the first line of a paragraph.

The most difficult thing about regular expressions is deciding which one is best to use. They are an extremely useful tool for programmers, so be sure to experiment with them a bit.

Job Control

The bash shell includes *job control* functionality that allows you to control the way that programs run. But how does this relate to input and output from your shell? Normally when you run a program, it executes in the *foreground*. That is, its input comes from stdin and its output goes to stdout. Sometimes it's necessary to have a program running, but you don't need or want to give it input or see its output. In fact, you may want to be able to continue to interact with your shell specifically while the program is running, in order to do other tasks. When that's the case, you can have the shell run the program in the *background*. To do this, add an ampersand (**&**) at the end of your command line. Let's try it with the **cat** command, which, when run without any arguments, will take input from stdin and print that input back to stdout:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ cat &
[1] 794
cold1:~/sysadmin1$
```

And you are back at your prompt. However, **cat** is still lurking in the background. In order to see the jobs you have executing in your shell, run the **jobs** command:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ jobs
[1]+  Stopped                 cat
```

You may notice that the number in brackets is the same number that appeared in brackets when you initially launched cat in the background. This is intentional. The number you see in brackets is the *job number*. You can use this number to refer to the job in the future to do things like bring its operation back to the foreground. Let's go ahead and do that:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ fg %1
cat
hello world
hello world
```

The **fg** command brings the specified job to the foreground. The **%** character prefixing the job number lets the shell know that we are talking about a job number, or *jobspec*. When a job is returned to the foreground, the first line you see is the command line that initiated the job. Everything else after that is normal output from that command. In this example, we demonstrate the behavior of the command we have brought into the foreground, "cat." When cat is invoked without providing a file name, it reads data from the standard input stream (in this case, the keyboard). When the **Enter** key is pressed, it prints all of the text it has received on the standard input stream to its standard output stream (in this case, the screen). Here, you typed "hello world" into the terminal, and when you pressed the **Enter** key, cat printed it back to the screen. If we want to return this job to the background, we must first *suspend* it to regain access to our terminal. Do this by pressing **Ctrl+z** (that is, hold down the **Ctrl** key and press **z**). You will see something like this:

INTERACTIVE SESSION:

```
^Z
[1]+  Stopped                 cat
cold1:~/sysadmin1$
```

From here, you must run **bg** *jobspec* to "background" the job. There is one other imporant set of keystrokes you can use for interacting with jobs in the foreground. First, bring your cat job back to the foreground. You may find that if you try to quit cat, there's not much you can do. (That's the thing about a cat, isn't it?) Typing "exit" or "quit" or "Why won't you just die!?" will not help you. In fact, as if it were mocking you, cat prints everything right back. The only way you can quit cat from your terminal is to issue the keystrokes **Ctrl+c**. As with **Ctrl+z** earlier, this means holding down the **Ctrl** key and pressing **c**. This key sequence sends a signal to cat that causes it to exit, returning you back to your shell.

You are now well on your way to being a Bash master! Redirects and Pipes are extremely useful tools for every sysadmin. Redirecting the output of a command to a file for later viewing is really handy, but the real star here is the pipe, which allows you to create powerful chains of commands to achieve complex tasks. Also, remember that control key sequence to kill a job (**Ctrl+c**). You'll find yourself using it frequently, because it allows you to end a job that may not be behaving the way you want it to!

Coming up next, we have more essential Bash features that will have you using your shell like a pro! But before you move on, remember to do your assignments and hand them in to your mentor!

Bash: Expansion

This lesson covers a powerful function of Bash called *expansion*. There are many sub-types of expansion available in Bash and we'll introduce a few in this lesson.

Environment Variables and Aliases

Certain aspects of your shell's behavior are determined by *environment variables*. The collection of all environment variables that are set within your shell is referred to as your *environment*. You can see your environment by running the **env** command. Sometimes it's useful to see the whole environment, but more often you'll want to see the contents of a specific variable. For that task, use the **echo** command. One of the most important variables in your environment is the *PATH* variable. The PATH variable determines where the shell will look for the program or *binary* that corresponds to a command you have typed. For example, when you type **ls**, the shell is actually executing the binary located at "/bin/ls," because the "/bin" directory is in your PATH. Let's use the echo command to take a look at your path:

INTERACTIVE SESSION:

```
cold1:~$ echo $PATH
/users/username/.gemhome/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sb
in
```

The directories are separated by colons (:). There are several directories *containing the word* **bin** in the output; the actual **/bin** directory where **ls** can be found is highlighted in **red**.

A few of the more important environment variables are listed in this table, along with their functions:

Variable Name	Description
PATH	A list of directories in which your shell searches for binaries.
PWD	Your present working directory (set by the shell).
USER	The username of the person who started the shell.
HOME	Your home directory.
PS1	Defines what your prompt looks like.
VISUAL	Defines what text editor is opened when a utility needs one to edit a file.

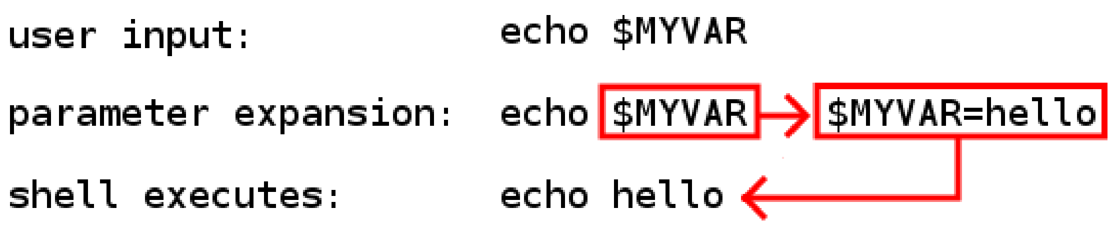
You can also define your own variables. User-defined variables are useful for tasks where you may have to use the same command or set of arguments over and over and you want to have a way of storing them for future use. Not only does this save typing, but it can help reduce the number of typos in your code. In this next example, we assign a value to a variable called MYVAR:

INTERACTIVE SESSION:

```
cold1:~$ MYVAR="hello"
cold1:~$ echo $MYVAR
hello
```

Note You only need to use quotation marks if your variable contains spaces. For example, you don't need to use them for **hello**, but you would need to put quotation marks around **hello world**. We'll talk more about using quotation marksin your code, or "quoting," soon.

While this might appear to be a rather simple operation at first glance, there is something more subtle going on here. You probably noticed that when you set the variable, you used "MYVAR," but when echoing the variable back, you used "$MYVAR." That "$" is necessary for referencing the contents of the variable "MYVAR" because of something called *parameter expansion*. What this means in practice is that the shell substitutes the contents of a variable wherever it sees a variable name prefixed with a "$." This happens before the shell executes the command line.



We've seen that the general format for setting a variable is **VARIABLE=contents**. It's important to understand how variable assignments interact with parameter expansion. As we saw in the example above, parameter expansion occurs before the shell executes the given command line. This means that when one or more variables are supplied on the right side of a variable assignment, those variables are expanded and their contents are substituted on the command line first, then the command line is executed. Let's look at an example of that behavior:

INTERACTIVE SESSION:

```
cold1:~$ REDS="cherry fireengine brick"
cold1:~$ GREENS="grass olive lime"
cold1:~$ BLUES="aqua sky powder"
cold1:~$ COLORS="$REDS $GREENS $BLUES"
cold1:~$ echo $COLORS
cherry fireengine brick grass olive lime aqua sky powder
```

COLORS="$REDS $GREENS $BLUES" below expands to COLORS="cherry fireengine brick grass olive lime aqua sky powder" before being executed by the shell. As you can see, the contents of COLORS ends up matching our expectations based on this expansion. Variable assignments can include a mixture of text and variables, not just one or the other. You can demonstrate this behavior:

INTERACTIVE SESSION:

```
cold1:~$ NAME1="Steve"
cold1:~$ NAME2="Kerry"
cold1:~$ THANKS="Thanks for the help $NAME1 and $NAME2!"
-bash: !": event not found
```

Hmm. That didn't work out so well, did it? It looks like we've fallen into one of Bash's quoting traps. The Bash shell uses the exclamation point (**!**) character for a different kind of expansion (which we'll explore later), so we have to treat it with some care. In order to understand how expansion is affected by quotation, let's go over what Bash does with a command line at execution time. Bash takes the command line and splits it up into distinct words, then those words are interpreted by the shell to determine what action should be taken. Quotation marks allow you to group together multiple words into one logical word for bash to interpret. Let's demonstrate the difference between an unquoted and a quoted string in a variable assignment:

INTERACTIVE SESSION:

```
cold1:~$ VARIABLE=some stuff
-bash: stuff: command not found
cold1:~$ VARIABLE="some stuff"
```

In the first command seen here, Bash assigns the word "some" to the variable "VARIABLE" and then tries to execute the command "stuff" (which doesn't exist). This is not what we wanted to do. Instead, we want to assign the phrase "some stuff" to the variable "VARIABLE," which is what the second command accomplishes using quoting. In this case, VARIABLE='some stuff' would be equivalent, but this is not always the case. Here is a case in which using single and double quotes makes a huge difference:

INTERACTIVE SESSION:

```
cold1:~$ VARIABLE="text"
cold1:~$ echo "VARIABLE contains $VARIABLE"
VARIABLE contains text
cold1:~$ echo 'VARIABLE contains $VARIABLE'
VARIABLE contains $VARIABLE
```

This demonstrates a simple rule of quoting in Bash. If a group of words is surrounded by double quotes, that group is subject to expansion by Bash. If a group is surrounded by single quotes, Bash will *not* expand it. With this in mind, let's return to the previous example, this time omitting the exclamation point (**!**):

INTERACTIVE SESSION:

```
cold1:~$ NAME1="Steve"
cold1:~$ NAME2="Kerry"
cold1:~$ THANKS="Thanks for the help $NAME1 and $NAME2"
cold1:~$ echo $THANKS
Thanks for the help Steve and Kerry
```

We must use double quotes in order to have $NAME1 and $NAME2 expanded by the shell, but how do we get that pesky exclamation point to show up at the end of the sentence? Again, we must consider how Bash breaks its command line up into words. Bash uses whitespace characters (spaces and tabs) to separate words from each other. Therefore, by putting one element ("Thanks for the help $NAME1 and $NAME2") right next to a second element ('!'), we can form one word:

INTERACTIVE SESSION:

```
cold1:~$ THANKS="Thanks for the help $NAME1 and $NAME2"'!'
cold1:~$ echo $THANKS
Thanks for the help Steve and Kerry!
```

Excellent! The single quotes around the **!** are necessary to keep Bash from expanding it. We can use the information about quoting and expansion in order to perform a common task: adding text to a variable that already exists and contains text. Try this:

INTERACTIVE SESSION:

```
cold1:~$ MYVAR="hello"
cold1:~$ echo $MYVAR
hello
cold1:~$ MYVAR="$MYVAR world"
cold1:~$ echo $MYVAR
hello world
```

Note You should generally use all uppercase letters to specify your variable names. It's not specifically required, but it's fairly standard practice to do so.

WARNING Bash variables are *case sensitive*! Bash would interpret VAR, Var, and var as three unique variables, so it's a good idea to to use all uppercase letters for your variables for consistency.

Variables you define using the **VARIABLE=contents** method do not become part of your environment. When you start a new shell or run a shell script, your environment gets copied to that shell or script, making the contents of all of the environmental variables available to them. In order to make a user-defined variable part of your environment, and therefore capable of being passed on to another shell or script run from your current shell, you must use the **export** command. Our next example demonstrates this concept by setting a variable and launching a new shell, first without exporting, and then with exporting. The prompt for the new shell is highlighted in **red**:

INTERACTIVE SESSION:

```
cold1:~$ SOMEVAR="some contents"
cold1:~$ echo $SOMEVAR
some contents
cold1:~$ bash
cold1:~$ echo $SOMEVAR

cold1:~$ exit
cold1:~$ export SOMEVAR
cold1:~$ bash
cold1:~$ echo $SOMEVAR
some contents
cold1:~$ exit
```

Aliases are similar to shell variables. An alias allows you to give a command or even a whole command line a more convenient name. You define and view aliases using the **alias** command. As you'll see in the next example, there are several aliases already defined for you. Not all Linux systems will do this. First we'll list existing aliases, then we'll add one, and then we'll list the aliases again to see our new alias (highlighted in **red**):

INTERACTIVE SESSION:

```
cold1:~$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
cold1:~$ alias mkdp="mkdir -p"
cold1:~$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mkdp='mkdir -p'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

Now, when you type **mkdp *some_directory***, it will be interpreted by the shell as **mkdir -p *some_directory***.

Dotfiles

When bash starts up, it goes through a procedure in which it constructs your initial environment. Part of this process is done to read configuration files in order to set things like variables and aliases to the user's preferences. These files are called *dotfiles* because they all have a period as the first character in their name. These files are usually hidden because of the leading period. To see them, you must use **ls -a**. The **-a** flag tells **ls** to list **a**ll files, even the hidden ones:

INTERACTIVE SESSION:

```
cold1:~$ ls -a
.  ..  .bash_history  .bash_logout  .bash_profile  .handin  .vim  .viminfo  sysadmin1
```

The two files you'll most likely encounter are ".bash_login" and ".bashrc." Each file is read by bash under different circumstances. If the shell is started as a *login* shell (that is, a shell that starts as a result of you logging into the system), bash will read the ".bash_login" file. Or, if you are already logged into the system and you start a new shell (called an *interactive* shell), bash will read ".bashrc." Use your text editor to open a file called ".bashrc" to edit in your home directory. At first it will be empty. Add the code below to the file and write it out:

CODE TO TYPE:

```
PS1="my_new_prompt-> "
```

To see what effect this has on bash, start a new **interactive** bash shell from your current bash shell:

INTERACTIVE SESSION:

```
cold1:~$ bash
my_new_prompt-> pwd
/users/username
my_new_prompt-> exit
cold1:~$
```

Now try starting a new shell on cold by clicking the New Terminal button in code runner. This will be **login** shell:

INTERACTIVE SESSION:

```
cold1:~$
```

Your new interactive shell used the .bashrc file you created, but the new login shell did not! If you would like to make sure that your login and interactive shells are consistent, you can add some lines to your .bash_login that will instruct bash to read your .bashrc as well. Open your .bash_login file and add this (be sure to replace "*username*" with your username):

CODE TO TYPE:

```
if [ -f /users/username/.bashrc ]; then
 . /users/username/.bashrc
fi
```

This tells bash that if the file **/users/username/.bashrc** exists, to read or *source* it. Once you have done this, you can add more variable definitions and aliases to your .bashrc, one per line. This helps to ensure that you have a consistent environment, regardless of whether the shell is a login shell or interactive shell. (If you leave the PS1 variable assignment in your ".bashrc", the next time you log in, your prompt will say "my_new_prompt->", so you probably want to remove that line.)

You're doing great! There were some big concepts in this lesson, the most important of which was shell expansions. As we move forward, we'll introduce even more ways to use shell expansion to save time and effort. For now, pratice using your new tools on your homework, and we'll see you in the next lesson!

Bash: More Expansions

In this lesson we'll dig deeper into shell expansion and introduce several new types of expansions and related topics.

Brace Expansion

Sometimes you'll need to perform one or more actions on a set of items that have similar names. For example, you might want to create a set of directories, one for each letter in the alphabet. You could type 26 **mkdir** commands, but bash provides a mechanism called *brace expansion* that allows you to achieve this goal using just one command:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ mkdir -p letters/{a..z}
cold1:~/sysadmin1$ ls letters
a b c d e f g h i j k l m n o p q r s t u v w x y z
cold1:~/sysadmin1$
```

Note The **-p** flag used with the "mkdir" command requests that the parent directory or directories of the directory we are trying to make (in this case, "letters") be made first, if they don't exist.

This works for both characters and numbers. You can also specify an increment value for a range statement, in the form **{start..end..increment}**:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ mkdir -p numbers/{0..10..2}
cold1:~/sysadmin1$ ls numbers
0  10  2  4  6  8
```

Brace expansion isn't just limited to sequences of letters or numbers; you can also specify multiple individual values separated by commas:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ mkdir -p images/bi{g,gger,ggest}
cold1:~/sysadmin1$ ls images/
big bigger biggest
cold1:~/sysadmin1$
```

Statements can be combined and nested to achieve complex goals. For example, you could create a set of 16 directories with names that follow the pattern of "a1, a2, ... d3, d4" by doing **mkdir {a..d}{1..4}**. Much like parameter expansion, brace expansion is performed before any commands are run. In fact, brace expansion is the first expansion performed by the shell, even before parameter expansion.

Pattern Matching

Similar to brace expansion, bash's *pattern matching* function allows you to work with multiple items at one time. Pattern matching contains a feature that is not available in brace expansion: *wildcard characters*. These characters let you match individual characters, entire strings that may vary widely, or even characters or strings that are unknown. To see this in action, we'll generate some files to experiment on using brace expansion:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ mkdir pattern
cold1:~/sysadmin1$ touch pattern/{a..e}{9..12}
cold1:~/sysadmin1$ ls pattern
a10  a11  a12  a9  b10  b11  b12  b9  c10  c11  c12  c9  d10  d11  d12  d9  e10  e11  e
12  e9
cold1:~/sysadmin1$
```

Now we have 20 files we can pattern-match against. Let's start off with the most basic wildcard character, the asterisk *****. This character matches everything (and nothing!). Specifically, it matches any character or string of characters, including the NULL character. Let's see what happens when we use *****:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ cd pattern
cold1:~/sysadmin1/pattern$ ls *
a10 a11 a12 a9 b10 b11 b12 b9 c10 c11 c12 c9 d10 d11 d12 d9 e10 e11 e12 e9
```

Yep, it matched every last file. That could be useful in some cases, but look what we can do by using it in conjunction with some defined characters:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1/pattern$ ls b*
b10 b11 b12 b9
```

Now we've done something more practical. By using **b***, we are saying "Match every file that has 'b' as its first character." The ***** can appear anywhere in the pattern you are matching against. Try out **ls *9** and see what happens. A good real-world application of this is moving all of the files of a certain type from one directory to another by doing something like **mv ~/Downloads/*.pdf ~/Documents/pdfs/**.

Note What's this tilde (**~**) business anyway? It's actually yet another kind of expansion called *tilde expansion*. When the first character of a word is a tilde, Bash expands the tilde to one of two things. In a situation where the word starts "~/something", the tilde is expanded to your home directory. If the word starts "~somebody/something", the "~somebody" is expanded to the home directory of the user whose name is "somebody".

To match any single character, use the wildcard character, the question mark (**?**). This is handy when the ***** character may match more items than intended. For example, if you want to match "bana" and "baca" but not "banana", you could specify "ba?a." To see this, let's list all the files with names that start with an "a" followed by a two-digit number:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1/pattern$ ls a??
a10 a11 a12
```

By specifying **a??**, we indicate that we want exactly two characters after the **a**. We can also supply a specific range of characters to match against. This is done using a set of square brackets, **[ ]**. We can apply this to find all the files that start with either "a" or "d":

INTERACTIVE SESSION:

```
cold1:~/sysadmin1/pattern$ ls [ad]*
a10 a11 a12 a9 d10 d11 d12 d9
```

You can also specify a range of characters, just as you can in brace expansion, but to specify a range, you use a hyphen (**-**) rather than two periods (**..**). To test this out, try **ls [a-c]9**. Finally, you can specify characters *not* to match against using a **!** or **^** as the first character inside the bracket set. Here's an example that lists all files that start with a character between a and c, that have a number that is 10 or greater, and do not include 11:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1/pattern$ ls [a-c]1[^1]
a10 a12 b10 b12 c10 c12
```

If that looks a little complicated, break it down into its individual pieces. The first set of brackets specifies that the first character must be in the range of a-c. The second character must be the digit 1. The last bracket set specifies that the third character (the second digit), must not be equal to 1.

Command History and Tab Completion

Up to this point, each time you wanted to run a command you had to type it in. But often you'll find you have to repeat the same command multiple times, perhaps making small changes each time. Typing the whole command in every time would be pretty tedious; fortunately, bash provides a solution. The *command history* feature in bash remembers every command you execute in a particular shell; there are multiple ways that you can recall and execute individual commands. The simplest way to navigate your command history is to use the up and down arrow keys on your keyboard. Give it a try by running **ls** and then pressing the up arrow. At your prompt, you see **ls** on your command line again. You can press **Enter** to run it—or you can modify it and then press **Enter** to run it, for example with **-l**. Each time you press the up arrow, you go back in your history one more command, until you get to the first command you ran when you started your shell. If you have moved backwards in your command history, hitting the down arrow will move you forward until you are back at a blank prompt where you can enter a new command.

You can also get a list of all the commands in your history by running the **history** command (to save room, we omit some of the output in this example, as indicated by the "..."):

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ history
1 pwd
2 ls
3 mkdir red
...
295 rm -rf free bird
296 ls
297 history
cold1:~/sysadmin1$
```

Your output will probably look different, and will include all entries, beginning at 1. As you can see, each command you have run is associated with a number, and that number can be used with *history expansion* in order to rerun that command. In order to run the *n*th command from your history, you would enter **!*n*** (This is the special use for the exclamation point (!) that we hinted at earlier). Give this a try by rerunning one of the commands in your command history, however be careful to select a command such as "echo" that won't cause any changes to your system. Keep in mind, when re-executing a command, that command is subject to the state of your environment and other defined variables at the time that the command is run, not at the time that the command was originally run. Consider this:

INTERACTIVE SESSION:

```
cold1:~$ date
Tue Jun 12 13:28:31 CDT 2007
cold1:~$ PRESIDENT="Bush"
cold1:~$ echo "The President is $PRESIDENT"
The President is Bush
cold1:~$ date
Tue Jun 12 13:29:01 CDT 2012
cold1:~$ PRESIDENT="Obama"
cold1:~$ history
    1  date
    2  PRESIDENT="Bush"
    3  echo "The President is $PRESIDENT"
    4  date
    5  PRESIDENT="Obama"
    6  history
cold1:~$ !3
echo "The President is $PRESIDENT"
The President is Obama
```

The value of PRESIDENT when command 3 was originally run was "Bush." However, times change, and by the second time command 3 is executed using history expansion, the value of PRESIDENT has changed to "Obama." While this might seem harmless for a command like **echo**, imagine how having different values for PWD might affect a command such as **rm** when the path to the specified files is relative to your PWD!

Tab completion is another useful feature of bash. You can press the **Tab** key to complete commands and file names without fully typing them. Try it. At the command prompt type **ls gr** and then press **Tab**. Now you'll see **ls green/** at the prompt. The same thing works for commands. Type **tou** and then press **Tab** and you'll get **touch**. On short commands and file names the advantages don't seem to be that great, but there are situations when it's useful. Say you want to edit a config file with the name "SuperLong_ConfigFile_Version1.0". Using tab completion to fill in the whole file name after typing only **Super** is a real time saver. If you try to use tab completion and there are multiple matches based on what you've already typed, the shell will display a list of possible completions and a prompt with what you have already typed:

INTERACTIVE SESSION:

```
cold1:~$ mk(Press Tab twice)
mkdict            mke2fs            mkfs              mkfs.ext2         mkfs.ext4
   mkhomedir_helper  mknod             mkswap
mkdir             mkfifo            mkfs.cramfs       mkfs.ext3         mkfs.ext4dev
   mklost+found      mksock            mktemp
cold1:~$ mk
```

Nice, huh? But let's step back and consider what's actually going on behind the scenes when you press the **Tab** key. Remember the PWD and PATH variables? When you hit the **Tab** key after having typed something in, Bash takes what it has so far and attempts to match that string against files in your PWD and PATH. If it finds only one match, it substitutes the matched file name for the string you have typed so far. If it finds multiple matches, it gives you choices. But that's not all it can do! Try this:

INTERACTIVE SESSION:

```
cold1:~$ FOOBAR="baz"
cold1:~$ echo $FOO(Press Tab)
```

Wow! Bash will also complete variable names for you! How neat is that? Just imagine all the typing you'll save with these cool tricks!

Well that's it for Bash for now. You've learned quite a bit about Bash over the last few lessons! With a little practice you'll be working on the command line with an amazing amount of efficiency. Now go use all of this new knowledge and ace this lesson's homework! We'll see you in the next one!

Useful Utilities

In this lesson, we will cover several utilities that will help you to use and administer a Linux system.

Viewing Data

You will often find that you want to view a file without editing it, or view some command's standard output stream in a way that is searchable or scrollable. For these tasks, the utilities **more** and **less** do the trick quite nicely.

Less is More?

The **less** utility is based on the **more** utility, but **less** has more functionality than **more**! Both read data from a file or their standard input stream; but **more** only allows you to scroll from the beginning to the end of the file. **Less**, on the other hand, allows you to scroll both forward and backward. Additionally, while both provide search functionality, **less**'s search feature not only finds, but also highlights matching strings, something that **more** does not do. Generally, the only reason you would ever want or need to use **more** is on a system that doesn't have **less** installed (which is unlikely).

Let's take a look at how **less** works, first with a file, then using its standard input stream:

INTERACTIVE SESSION:

```
cold1:~$ less /etc/services
```

Our example shows the contents of the file **/etc/services**, which is a list of services and the corresponding port numbers upon which they listen. There are multiple methods you can use to scroll through the file. The most direct is to use your arrow keys, which will scroll the file, one line at a time. For scrolling larger chunks, use your PgUp and PgDn keys. If you prefer using keyboard shortcuts rather than arrow keys, press **Enter** to scroll down one line, **y** to scroll up one line, the **space** bar will scroll down a whole page, and **b** will scroll back a whole page. **G** will take you to the bottom of the file, and **g** will take you to the top. You may recall that **G** is also used in **vi**'s command mode to go to the bottom of the file. The similarities in the command sets extend beyond that.

The method used for searching a file in **less** is identical to the method used in **vi**. While the file is open in **less**, type a **/** followed by the text you want to find; for example, **/rje**. Less will scroll to the first instance of that string and highlight it, as well as any other occurrences of that string. To repeat a search, press **n**. Backwards searching is also supported, and as in vi, the question mark (**?**) character is used. To repeat a search in the backwards direction, use **N**. Additionally, you can move directly to a line in a file by typing **:***n*, where *n* is the line number. To quit viewing data in less, type **q**.

Now let's explore reading text using **less**'s standard input stream. Earlier, you learned about pipes, which allow the standard output stream of a command to be sent to the standard input stream of another command, rather than printing it on the screen. We can use that to redirect the output of an **ls -l** (which would normally return far too many entries to fit on one screen) into **less** so we can scroll through it and even search it.

INTERACTIVE SESSION:

```
cold1:~$ ls -l /etc | less
```

You will see the output of **ls -l /etc** on your screen, but instead of having the whole thing scroll by at once on your terminal, you can scroll up and down (and search) at your leisure. The same keystrokes that you used in the previous examples can be used here.

Finding Things

Sometimes you need to find a file or directories based on something like its name or the date it was last modified. You can accomplish this task with the (surprisingly named) utility, **find**. Find has an extensive list of features allowing you to search for files based on a multitude of parameters. We will cover only a few in this lesson, but take a look at find's **man** page to see the many methods you can use to search for files.

First we'll search for files based on their names. You may recall that in the previous lesson we created a directory named "pattern" and we filled it with several files. We'll search for files in /home/username/pattern that start with a:

INTERACTIVE SESSION:

```
cold1:~$ find /users/username/sysadmin1/pattern -name "a*"
/users/username/sysadmin1/pattern/a12
/users/username/sysadmin1/pattern/a9
/users/username/sysadmin1/pattern/a11
/users/username/sysadmin1/pattern/a10
```

The general format of **find** is:

OBSERVE:

```
find options /searchpath expression1 expression2 .. expressionN
```

In the last example, we specified no **options** to find, our **path** was **/home/username/pattern** and we specified only one expression: **-name "a*"**. You can specify multiple expressions to find, which allows you to narrow your search focus by specifying auxilary conditions that a file must meet in order to be considered a match. **find** can use a file's access time, modification time, user, group and several other characteristics for comparison. You want to work on a few examples now, so go ahead and create several files in a subdirectory within your home directory and call it "look":

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ mkdir look
cold1:~/sysadmin1$ touch look/file1
cold1:~/sysadmin1$ touch -d "3 days ago" look/file2
cold1:~/sysadmin1$ touch -d "5 days ago" look/file3
cold1:~/sysadmin1$ touch -d "7 days ago" look/file4
cold1:~/sysadmin1$ touch -d "9 days ago" look/file5
cold1:~/sysadmin1$ ls -l look
total 0
-rw-rw-r-- 1 username webusers 0 Jan  5 11:04 file1
-rw-rw-r-- 1 username webusers 0 Jan  2 11:04 file2
-rw-rw-r-- 1 username webusers 0 Dec 31 11:04 file3
-rw-rw-r-- 1 username webusers 0 Dec 29 11:05 file4
-rw-rw-r-- 1 username webusers 0 Dec 27 11:05 file5
```

Note Remember our old friend **touch**? It was introduced as a way to update the timestamp on a file, but initially we used it to create empty files. Here you can see the power **touch** has over timestamps. The **-d** flag instructs **touch** to interpret the quoted string as a date to use for the timestamp. Consult touch's man page for more information about how to specify dates.

So now we have some files with varying timestamps we can search through. It's possible to find files that were modified an exact number of days ago by specifying **-mtime *n***, where *n* is the number of days,; but that's not usually all that useful. More often you'll want to find files that are were created before or after than a certain date. A great real-world application of this is to locate files that are older than a certain number of days and then move them elsewhere for storage. In order to specify that you want files that were created before or after *n* days, you would use **+*n*** or **-*n*** respectively. Let's see this in action:

INTERACTIVE SESSION:

```
cold1:~/sysadmin1$ find ./look -mtime +4
./look/file4
./look/file5
./look/file3
```

Your list will be different if you do this example a day or more after you originally touched the files in ~/look, but you can verify that it's correct by viewing the output of **ls -l ~/look**. You might notice that the files have printed out in a strange order. There is a helpful little tool that can solve this problem for us called **sort**. Try using a pipe to send the standard output stream of **find** to **sort** and see what happens. Pretty cool, right? When your search returns dozens of results it can be quite helpful. For example, you could pipe the output of **find** to **sort**, and then pipe that output to **less** and have a sorted, scrollable list of files to browse through.

So now you know about **less** (and its companion **more**), **man**, **find**, and **sort**. You've got quite a well-stocked toolbox now—the next lesson will give you a tool that can enhance all of your existing utilities.

Go ahead and do your homework, and we'll see you in the next lesson!

Processes

In this lesson you will learn about *processes*, what they are, how to view them, and how to interract with them.

What is a Process?

In the computing world, a process is a stream of commands that are being executed by the host's CPU. When you run **ls** or **vi**, that's a process. Your shell is just a process running on your machine. Any program, utility, or command you run is a process, or part of another process. To view the processes that are currently running on your system, use the **ps** command. Let's go ahead and run it to see what we can find out:

INTERACTIVE SESSION:

```
cold1:~$ ps
  PID TTY          TIME CMD
  606 tty0     00:00:00 bash
 1284 tty0     00:00:00 ps
```

Just two processes for the whole system? If you don't think that sounds reasonable, you're right. Without any options specified, the ps command just displays the processes running in the current terminal owned by the same user who ran it. Usually you will want more information than that, so we'll get right into methods for displaying processes with ps. The most common set of flags you will give to ps is **aux**. The **a** means show processes from all users, the **u** makes the output more user friendly, and the **x** flag tells ps to show processes with no associated tty in addition to ones that do have a tty.

INTERACTIVE SESSION:

```
cold1:~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  19112  1448 ?        Ss   Jan18   0:01 /sbin/init
root         2  0.0  0.0      0     0 ?        S    Jan18   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Jan18   0:13 [ksoftirqd/0]
root         4  0.0  0.0      0     0 ?        S    Jan18   0:00 [events/0]
... lots of output...
```

Note Before you go any further, look at the **man** page for **ps** to find out why the flags were specified as **aux** rather than **-aux**.

Your output should look a lot like this, though it may be cut off on the right side. If you want to see the whole thing, you can always pipe the output stream to **less** and use it to scroll both vertically and horizontally. This brings us to an important concept in processes: the *pid*, or *process id number*. As with the uid and gid, the pid is a unique number given to each process. It is assigned by the system when the process is started and is used to refer to the process for future operations, such as changing its state or killing it. In our ps output above, you can find the pid in the second column.

Ps has another common set of options that give you slightly different information, **-ef**. Unlike aux, it *does* have a leading dash. Run **ps -ef**:

INTERACTIVE SESSION:

```
cold1:~$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Jan18 ?        00:00:01 /sbin/init
root         2     0  0 Jan18 ?        00:00:00 [kthreadd]
root         3     2  0 Jan18 ?        00:00:23 [ksoftirqd/0]
root         4     2  0 Jan18 ?        00:00:00 [events/0]
root         5     2  0 Jan18 ?        00:00:00 [khelper]
root         8     2  0 Jan18 ?        00:00:00 [async/mgr]
root        44     2  0 Jan18 ?        00:00:00 [sync_supers]
root        46     2  0 Jan18 ?        00:00:00 [bdi-default]
root        48     2  0 Jan18 ?        00:00:00 [kblockd/0]
root        66     2  0 Jan18 ?        00:00:00 [kswapd0]
root        67     2  0 Jan18 ?        00:00:00 [ksmd]
root        68     2  0 Jan18 ?        00:00:00 [aio/0]
root       185     2  0 Jan18 ?        00:00:00 [jbd2/ubda-8]
root       186     2  0 Jan18 ?        00:00:00 [ext4-dio-unwrit]
root       262     1  0 Jan18 ?        00:00:00 /sbin/udevd -d
root       371   262  0 Jan18 ?        00:00:00 /sbin/udevd -d
root       518     1  0 Jan18 ?        00:00:00 /sbin/rsyslogd -c 4
root       585     1  0 Jan18 ttyS0    00:00:00 /bin/awk -f /usr/sbin/handin.awk /dev/t
tyS0
root       589     1  0 Jan18 ?        00:00:00 login -- username
root       593     1  0 Jan18 tty2     00:00:00 /sbin/mingetty --noclear /dev/tty2
root       595     1  0 Jan18 tty3     00:00:00 /sbin/mingetty --noclear /dev/tty3
username   606   589  0 Jan18 tty0     00:00:00 -bash
root      1497     1  0 Jan19 tty1     00:00:00 /sbin/mingetty --noclear /dev/tty1
root      1830     2  0 10:36 ?        00:00:00 [flush-98:0]
username  1835   606  0 10:36 tty0     00:00:00 ps -ef
```

The output in the above example has been greatly shortened for demonstration purposes. The output is similar to that of **aux**, but we now have another important piece of information, the *ppid* or *parent process id number*. The ppid tells us the pid of the process that spawned (started) the listed process. A process that is spawned by another process is called the *child* of that process. We can see the relationship between parent and child processes firsthand in this output. In our example, the **ps -ef** that we ran has the pid **1835** and its ppid is **606**. Looking through the list of processes, we see that the process with the pid of **606** is **bash**, our shell. We ran the ps from our shell, so ps is a child process of our shell process. Let's keep going up the family tree. The ppid for our bash shell is **589**, which corresponds to a **login** process. When you log into the system, the login process is responsible for starting your shell for you. If we go one step further, we see the ppid for login is **1**. If you think that any process with the pid 1 must be important, you're right. In fact, the same process is always #1. This is the **init** process, which is responsible for starting everything else on the system.

Interacting with Processes

On occasion you'll encounter a process that's misbehaving and you have to end it. This task can be accomplished with the frighteningly named **kill**. Kill sends signals to processes, and while that signal is almost always meant to end the process, occasionally non-fatal signals can be sent to processes to have them execute alternative actions. In this lesson though, we'll concentrate on terminating processes using kill.

Before we can move forward, we need a test subject to experiment on. Luckily we have the perfect do-nothing utility in our arsenal, **sleep**. While sleep definitely has its uses (mostly in scripting), it literally does nothing. You tell sleep how long to run, and it sits doing nothing for that amount of time, then exits. To test **kill**, we can start a sleep process with a long sleep time and kill it before it reaches that time.

First, log into a second shell on cold by clicking the New Terminal button. In one shell, type **sleep 3600**. Uninterrupted, this will run for 3600 seconds (an hour) and then terminate. Next, switch to your other shell and use **ps** to find the pid of the sleep you just started. Don't forget that you have a tool you can use to help find the correct process. With the correct pid in hand, you can terminate your sleep process by typing **kill *pid*** (substituting the pid number you found for *pid*, of course). When you switch back to the console you ran **sleep** in, you should see something like this:

OBSERVE:

```
cold1:~$ sleep 3600
Terminated
cold1:~$
```

You have just killed your first process. You may find a time when a process just refuses to die with a simple **kill *pid***. When that happens, you can give a specific signal to kill to send to the process. The most powerful of all kill signals is 9 or KILL. Try the sleep example again, but this time instead of typing **kill *pid***, type **kill -9 *pid***. You should then see these last words from your sleep process:

OBSERVE:

```
cold1:~$ sleep 3600
Killed
cold1:~$
```

Another Way to View and Interact with Processes

Top is another tool you can use for viewing and killing processes. Unlike **ps**, **top** is interactive and shows a continuously updating list of processes, as well as some useful information about system resource usage. By default, **top** sorts processes by cpu usage, but you can sort them based on any of a number of other metrics. Go ahead and run top:

INTERACTIVE SESSION:

```
cold1:~$ top
top - 10:30:13 up 1 day, 19:13,  2 users,  load average: 0.00, 0.00, 0.00
Tasks:  29 total,   1 running,  28 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1015076k total,   111136k used,   903940k free,     7288k buffers
Swap:        0k total,        0k used,        0k free,    82708k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 19112 1444 1200 S  0.0  0.1   0:00.01 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:22.46 ksoftirqd/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.00 events/0
    5 root      20   0     0    0    0 S  0.0  0.0   0:00.00 khelper
    8 root      20   0     0    0    0 S  0.0  0.0   0:00.00 async/mgr
   44 root      20   0     0    0    0 S  0.0  0.0   0:00.00 sync_supers
   46 root      20   0     0    0    0 S  0.0  0.0   0:00.00 bdi-default
   48 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kblockd/0
   66 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kswapd0
   67 root      25   5     0    0    0 S  0.0  0.0   0:00.00 ksmd
   68 root      20   0     0    0    0 S  0.0  0.0   0:00.00 aio/0
  185 root      20   0     0    0    0 S  0.0  0.0   0:00.00 jbd2/ubda-8
  186 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ext4-dio-unwrit
  262 root      16  -4 10572  732  424 S  0.0  0.1   0:00.71 udevd
  489 root      18  -2 10568  756  448 S  0.0  0.1   0:00.00 udevd
```

Your output will be different but similar to this. Every three seconds, the display will update. Interaction with **top** happens via keystrokes. First, quit top by press **q**. We're not finished with top though, so start it back up and we'll cover a few other important keystrokes. Once you are back into top, press **F**. You'll see a list of fields by which you can sort your process. Try **x**, which sorts processes by command name. You then again see the list of processes. Next, press **R** to reverse the sort order of the processes. Finally, you can use **k** to kill a process.

Go ahead and practice this new stuff in your homework. See you in the next lesson!

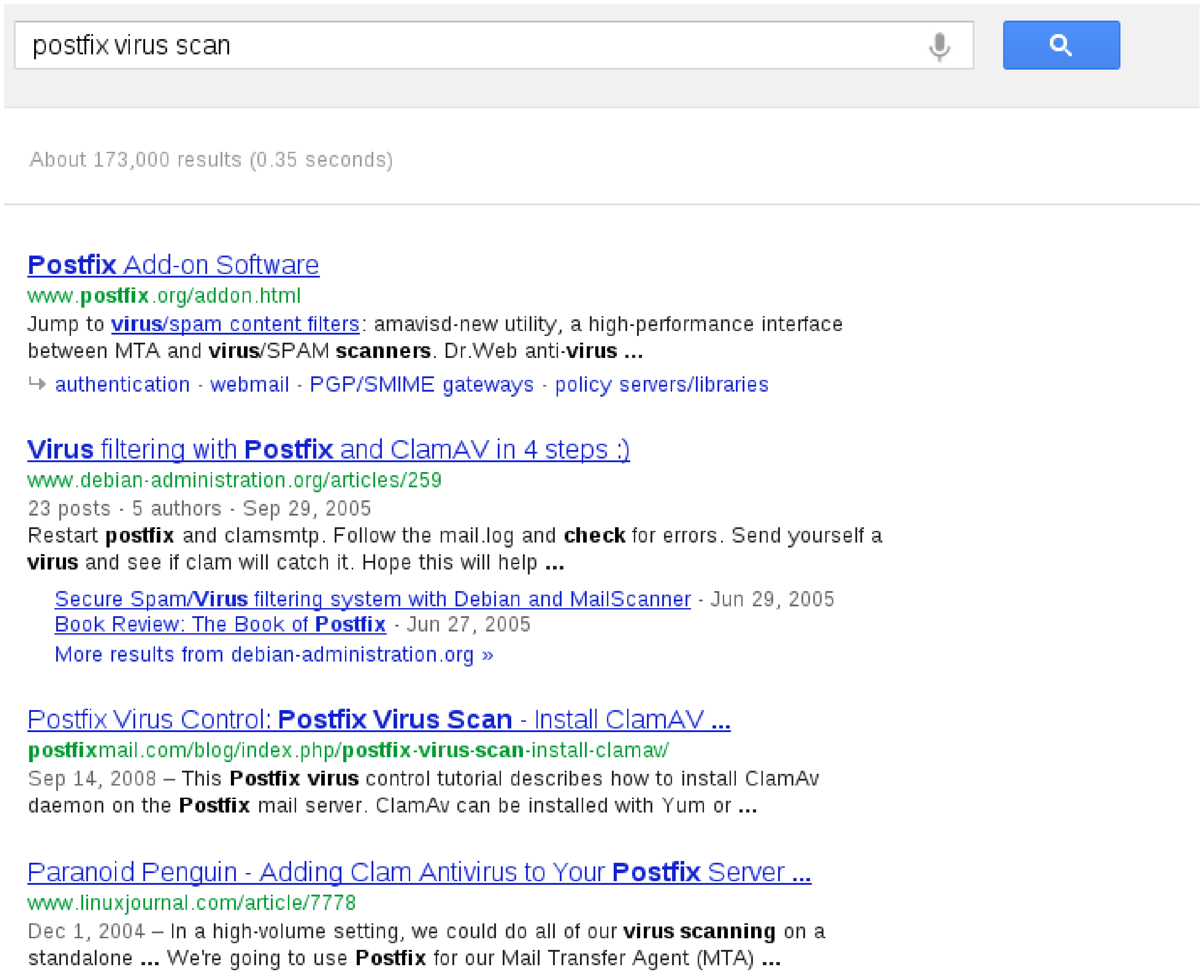
Finding Help

In this lesson, we'll discuss ways to find solutions to problems you encounter with your Linux system. You'll also earn how to find help and report software bugs to developers.

Extending Your Knowledge

Good systems administrators don't have all the answers, they just know how to find them. It would be impossible to know everything there is to know about administering Linux systems without consulting any external sources. Fortunately, there is a huge community of Linux developers, administrators, and users out there who help each other. There are formal documentation projects like The Linux Documentation Project, as well as dozens of forums and hundreds of blogs where people post problems and solutions to common (and not so common) issues. A search engine such as Google or Bing can be your most powerful systems administration tool.

Chances are that someone has already run into your problem, solved it, and posted the solution on the internet. The trick is to find it. You'll get a better sense of this process in our next example. Your task is to find out how to set up automatic virus scanning for incoming mail using the Postfix mail server software. You could search the internet for something like, "how do I configure postfix to scan incoming mail for viruses," but that search contains a few extraneous words that will return untargeted results. Instead, distill your search query to its most basic level. A better query would be "postfix virus scan incoming mail" or even "postfix virus scan." These concise queries may not provide you with the correct answer instantly, but they will lead you in the right direction. Consider these search results from google for "postfix virus scan":


postfix virus scan

About 173,000 results (0.35 seconds)

Postfix Add-on Software
www.**postfix**.org/addon.html
Jump to **virus**/spam content filters: amavisd-new utility, a high-performance interface between MTA and **virus**/SPAM **scanners**. Dr.Web anti-**virus** ...
↳ authentication - webmail - PGP/SMIME gateways - policy servers/libraries

Virus filtering with **Postfix** and ClamAV in 4 steps :)
www.debian-administration.org/articles/259
23 posts - 5 authors - Sep 29, 2005
Restart **postfix** and clamsmtp. Follow the mail.log and **check** for errors. Send yourself a **virus** and see if clam will catch it. Hope this will help ...
Secure Spam/**Virus** filtering system with Debian and MailScanner - Jun 29, 2005
Book Review: The Book of **Postfix** - Jun 27, 2005
More results from debian-administration.org »

Postfix Virus Control: **Postfix Virus Scan** - Install ClamAV ...
postfixmail.com/blog/index.php/**postfix**-**virus**-**scan**-install-clamaw/
Sep 14, 2008 – This **Postfix virus** control tutorial describes how to install ClamAv daemon on the **Postfix** mail server. ClamAv can be installed with Yum or ...

Paranoid Penguin - Adding Clam Antivirus to Your **Postfix** Server ...
www.linuxjournal.com/article/7778
Dec 1, 2004 – In a high-volume setting, we could do all of our **virus scanning** on a standalone ... We're going to use **Postfix** for our Mail Transfer Agent (MTA) ...

Three of the first four results specifically mention "Clam" or "ClamAV." If none of those pages offer a good solution for how to set this software up, at least you've learned that "ClamAV" was a popular antivirus solution for postfix. You can incorporate this information to further refine your search, "postfix clamav howto," for example.

Asking For Help and Reporting Bugs

Despite your obvious dedication, search as you might, there will still be times when you cannot find the answer. That's when you'll want to ask your question in a public forum or mailing list. When using this method, the most important consideration you have is the audience to which you are posing your question. For example, you may want to find out the best way to perform a certain task using a Bash shell script. Posting this question to the Bash developers mailing list will not likely result in an answer that can help you . However, posting to a website like stack overflow will effectively pose your question to a large group of programmers experienced with Bash shell scripting. Conversely, posting a question about how Bash handles shell expansions internally to the Bash developers mailing list will probably result in a wealth of information. Still other times, you'll need to interact with software developers—for instance, when a bug arises.

Unfortunately, there's not a single piece of perfect software out there. All software is susceptible to bugs, and Linux is no exception. If you aren't a programmer, you'll need the developers of the software to fix bugs for you. Reporting bugs in a way that benefits you as well as the community is critical. So how does one go about reporting a bug? There are several ways to inform developers of a bug (mailing list, project website, or bug tracking portal); always use the most clear and concise format available. Take a look at this potential issue:

OBSERVE:

```
cold1:~$ spiffy_util -f spifcfg.conf -F -h 192.168.0.10 -p 31337 -x
Starting up spiffy util version 3.6.11 ... Ok!
Listening on 192.168.0.10 on port 31337.
Received TCP connection request from 192.168.0.55!
192.168.0.55 requested file "/users/dbassett/somefile.txt"...
Begin transmitting data...
Transmission finished, closing connection.
Received TCP connection request from 192.168.0.55!
192.168.0.55 requested file "~/somefile.txt"...
ILLEGAL FILE NAME! spiffy util exiting :-(
error code 639
cold1:~$
```

Using the fictitious utility **spiffy_util**, a request for the file "**/users/dbassett/somefile.txt**" succeeds, while a request for that **same file using tilde expansion** fails with an "ILLEGAL FILE NAME" error. This behavior is probably evidence of a bug in the software, because it should be able to handle tilde expansion in a better way (by either returning the correct file, or reporting the error to the user without crashing). The first step you take when reporting a bug is to try to reproduce the behavior of the bug, and determine a set of actions that will reliably cause the bug to occur. In the above example, it would be a good idea to attempt to retrieve several different files, attempting to use tilde notation in each case. Not all causes of bugs will be so readily apparent. For example, though highly unlikely, the error might be a result of filenames that are exactly 14 characters long, or filenames that contain an "f," and not a result of the tilde. Once you have determined a series of steps that will reliably cause a failure, gather information about your environment. That is, which operating system you are using, the version of the software in question you're running, and sometimes even the version of the kernel on your system. If the software uses a configuration file, it's a good idea to include that as well.

In our example, the version of spiffy_util is listed right in the output of the command: version 3.6.11. There are a couple of ways to determine the operating system version. If you're running a Redhat-based Linux distribution (Redhat Enterprise Linux, Fedora Core, CentOS), you can do this:

INTERACTIVE SESSION:

```
cold1:~$ cat /etc/redhat-release
CentOS release 6.2 (Final)
```

Or on a Debian-based machine (Debian, Ubuntu, Mint):

OBSERVE:

```
debhost:~$ cat /etc/debian_version
6.0.3
```

Your kernel version can be determined using the same command on all platforms; the important bit is highlighted in

red:

INTERACTIVE SESSION:

```
cold1:~$ uname -a
Linux cold1.useractive.com 2.6.32-220.17.1.el6.x86_64 #1 SMP Wed May 16 00:01:37 BST 20
12 x86_64 x86_64 x86_64 GNU/Linux
```

When you have this information, you can begin crafting your bug report. Regardless of which channel you use to report the bug, be clear, provide only the necessary information, and be polite! Here is an example of a *bad* bug report:

OBSERVE:

```
Spiffy_util sucks!  I tried to get a file (~/somefile.txt) and it died on me!  You guys
 should fix this right away.
What a ridiculous fault.
```

This bug report gives almost no information, and is quite rude as well! A lot of developers will outright ignore this kind of bug report. Now let's look at a better example:

OBSERVE:

```
This morning I was using spiffy_util to retrieve some files and I noticed that when I t
ry to specify a filename using a
tilde, the utility crashes. I'm using spiffy_util version 3.6.11. If you could take a l
ook at this, I'd really appreciate it!
```

(I'm sure you are a polite and awesome person who will be an asset to the Linux community, but I just wanted to put this out there. You'd be surprised at how often some cranky-pants poster actually submits stuff like that bad report!) The second report is much better than the first, but it still lacks some important information. You'll probably get a response from a developer, but it will be loaded with requests for more information. Finally, let's see what a really excellent bug report would look like:

OBSERVE:

```
Spiffy_util version: 3.6.11
Operating System: CentOS 6.2
Kernel version: 2.6.32-220.17.1.el6.x86_64

I was using spiffy_util to retrieve some files and I noticed that when I tried to speci
fy a filename using a
tilde, the utility crashes with the following error message:

192.168.0.55 requested file "~/somefile.txt"...
ILLEGAL FILE NAME! spiffy util exiting :-(
error code 639

I tried to get several different files from mutiple home directories (~/a_different_fil
e.txt, ~someuser/somefile.txt) using
tildes, and they all caused the same results. The contents of my spifcfg.conf are as fo
llows:

spif_protocol_version=3
use_protocol=tcp
max_concurrent_connections=5
auto_disconnect=yes
idle_timeout=30s
```

WARNING While it's a good idea to send your configuration file along with the bug report, you should remove any personal information that may be in it!

Keep in mind, as long as you are giving information that's related to the issue, you generally can't give too much. Also, you don't have to replicate that sample bug report exactly in order to have success. It's just a general format for a good bug report. Just keep in mind that the most important factors in writing a solid bug report are to supply sufficient information and to be polite and professional.

Congratulations! You've finished the first System Administration course—and in stellar form, if I do say so myself. You now have the basic tools for managing files and processes on a Linux system. We hope you'll join us for the next course where we introduce the super user and discuss networking topics!

If you keep all of this information in mind, you're bound to be a great member of the open source community! Don't forget to turn in your homework assignment for this lesson and we'll see you next time!