An introduction to Linux environment

Please note the manual is also available at http://linuxcommand.org/lc3_learning_the_shell.php

Note: The terminal in Linux is similar to command prompt in windows PC where we type commands to perform any specific task.

The manual is divided into two section with different lessons:

Part 1 – Shell program, terminal and commands

Lessons:

1. What is "the Shell"?
2. Navigation
3. Looking Around
4. A Guided Tour
5. Manipulating files
6. Working with Commands
7. I/O Redirection
8. Expansion
9. Permissions
10. Job Control

Part 2 – Creating Shell script (i.e, file containing pre-loaded series of commands to execute task

Lessons:

11. Writing Your First Script and Getting It to Work
12. Editing the Scripts You Already Have
13. Here Scripts
14. Variables
15. Command Substitution and Constants
16. Shell Functions
17. Some Real Work
18. Flow Control - Part 1
19. Stay Out of Trouble
20. Keyboard Input and Arithmetic
21. Flow Control - Part 2
22. Positional Parameters
23. Flow Control - Part3
24. Errors and Signals and Traps (Oh My!) - Part 1
25. Errors and Signals and Traps (Oh My!) - Part 2

Suggested manual book:*The Linux Command Line, A complete Introduction*, by William Shotts available on major sites, eg. Amazon or online at https://www.linuxzasve.com/preuzimanje/TLCL-09.12.pdf

Learning the Shell

Why Bother?

Why do you need to learn the command line anyway? Well, let me tell you a story. A few years ago we had a problem where I used to work. There was a shared drive on one of our file servers that kept getting full. I won't mention that this legacy operating system did not support user quotas; that's another story. But the server kept getting full and it stopped people from working. One of our software engineers spent the better part of a day writing a C++ program that would look through all the user's directories and add up the space they were using and make a listing of the results. Since I was forced to use the legacy OS while I was on the job, I installed a Linux-like command line environment for it. When I heard about the problem, I realized I could do all the work this engineer had done with this single line:

```
du -s * | sort -nr > $HOME/user_space_report.txt
```

Graphical user interfaces (GUIs) are helpful for many tasks, but they are not good for all tasks. I have long felt that most computers today are not powered by electricity. They instead seem to be powered by the "pumping" motion of the mouse! Computers were supposed to free us from manual labor, but how many times have you performed some task you felt sure the computer should be able to do but you ended up doing the work yourself by tediously working the mouse? Pointing and clicking, pointing and clicking.

I once heard an author say that when you are a child you use a computer by looking at the pictures. When you grow up, you learn to read and write. Welcome to Computer Literacy 101. Now let's get to work.

Contents

What is "the Shell"?

Simply put, the shell is a program that takes commands from the keyboard and gives them to the operating system to perform. In the old days, it was the only user interface available on a Unix-like system such as Linux. Nowadays, we have *graphical user interfaces (GUIs)* in addition to *command line interfaces (CLIs)* such as the shell.

On most Linux systems a program called **bash** (which stands for Bourne Again SHell, an enhanced version of the original Unix shell program, **sh**, written by Steve Bourne) acts as the shell program. Besides **bash**, there are other shell programs that can be installed in a Linux system. These include: **ksh**, **tcsh** and **zsh**.

What's a "Terminal?"

It's a program called a *terminal emulator*. This is a program that opens a window and lets you interact with the shell. There are a bunch of different terminal emulators you can use. Most Linux distributions supply several, such as: **gnome-terminal**, **konsole**, **xterm**, **rxvt**, **kvt**, **nxterm**, and **eterm**.

Starting a Terminal

Your window manager probably has a way to launch a terminal from the menu. Look through the list of programs to see if anything looks like a terminal emulator. If you are a KDE user, the terminal program is called "konsole," in Gnome it's called "gnome-terminal." You can start up as many of these as you want and play with them. While there are a number of different terminal emulators, they all do the same thing. They give you access to a shell session. You will probably develop a preference for one, based on the different bells and whistles each one provides.

Testing the Keyboard

OK, let's try some typing. Bring up a terminal window. You should see a *shell prompt* that contains your user name and the name of the machine followed by a dollar sign. Something like this:

```
[me@linuxbox me]$
```

Excellent! Now type some nonsense characters and press the enter key.

```
[me@linuxbox me]$ kdkjflajfks
```

If all went well, you should have gotten an error message complaining that it cannot understand you:

```
[me@linuxbox me]$ kdkjflajfks
bash: kdkjflajfks: command not found
```

Wonderful! Now press the up-arrow key. Watch how our previous command "kdkjflajfks" returns. Yes, we have *command history*. Press the down-arrow and we get the blank line again.

Recall the "kdkjflajfks" command using the up-arrow key if needed. Now, try the left and right-arrow keys. You can position the text cursor anywhere in the command line. This allows you to easily correct mistakes.

You're not logged in as root, are you?

If the last character of your shell prompt is # rather than $, you are operating as the *superuser*. This means that you have administrative privileges. This can be potentially dangerous, since you are able to delete or overwrite any file on the system. Unless you absolutely need administrative privileges, do not operate as the superuser.

Using the Mouse

Even though the shell is a command line interface, the mouse is still handy.

Besides using the mouse to scroll the contents of the terminal window, you can copy text with the mouse. Drag your mouse over some text (for example, "kdkjflajfks" right here on the browser window) while holding down the left button. The text should highlight. Release the left button and move your mouse pointer to the terminal window and press the middle mouse button (alternately, you can press both the left and right buttons at the same time if you are working on a touch pad). The text you highlighted in the browser window should be copied into the command line.

A few words about focus...

When you installed your Linux system and its window manager (most likely Gnome or KDE), it was configured to behave in some ways like that legacy operating system.

In particular, it probably has its *focus policy* set to "click to focus." This means that in order for a window to gain focus (become active) you have to click in the window. This is contrary to traditional X Window behavior. You should consider setting the focus policy to "focus follows mouse". You may find it strange at first that windows don't raise to the front when they get focus (you have to click on the window to do that), but you will enjoy being able to work on more than

one window at once without having the active window obscuring the the other. Try it and give it a fair trial; I think you will like it. You can find this setting in the configuration tools for your window manager.

Navigation

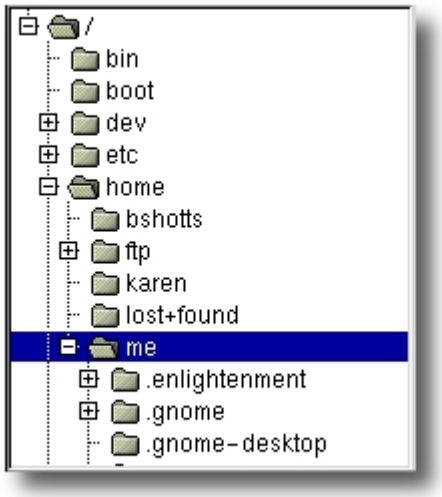
In this lesson, I will introduce your first three commands: **pwd** (print working directory), **cd** (change directory), and **ls** (list files and directories).

If you have not worked with a command line interface before, you will need to pay close attention to this lesson, since the concepts will take some getting used to.

File System Organization

Like that legacy operating system, the files on a Linux system are arranged in what is called a *hierarchical directory structure*. This means that they are organized in a tree-like pattern of *directories* (called folders in other systems), which may contain files and other directories. The first directory in the file system is called the *root directory*. The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on.

Most graphical environments today include a file manager program to view and manipulate the contents of the file system. Often you will see the file system represented like this:



One important difference between the legacy operating system and Unix-like operating systems such as Linux is that Linux does not employ the concept of drive letters. While drive letters split the file system into a series of different trees (one for each drive), Linux always has a single tree. Different storage devices may contain different branches of the tree, but there is always a single tree.

pwd

Since a command line interface cannot provide graphic pictures of the file system structure, it must have a different way of representing it. Think of the file system tree as a maze, and you are standing in it. At any given moment, you are located in a single directory. Inside that directory, you can see its

files and the pathway to its *parent directory* and the pathways to the subdirectories of the directory in which you are standing.

The directory you are standing in is called the *working directory*. To find the name of the working directory, use the **pwd** command.

```
[me@linuxbox me]$ pwd
/home/me
```

When you first log on to a Linux system, the working directory is set to your *home directory*. This is where you put your files. On most systems, your home directory will be called /home/your_user_name, but it can be anything according to the whims of the system administrator.

To list the files in the working directory, use the **ls** command.

```
[me@linuxbox me]$ ls
Desktop     Xrootenv.0    linuxcmd
GNUstep     bin           nedit.rpm
GUILG00.GZ  hitni123.jpg  nsmail
```

I will come back to **ls** in the next lesson. There are a lot of fun things you can do with it, but I have to talk about pathnames and directories a bit first.

cd

To change your working directory (where you are standing in the maze) you use the **cd** command. To do this, type **cd** followed by the *pathname* of the desired working directory. A pathname is the route you take along the branches of the tree to get to the directory you want. Pathnames can be specified in one of two different ways; *absolute pathnames* or *relative pathnames*. Let's look with absolute pathnames first.

An absolute pathname begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed. For example, there is a directory on your system in which most programs are installed. The pathname of the directory is /usr/bin. This means from the root directory (represented by the leading slash in the pathname) there is a directory called "usr" which contains a directory called "bin".

Let's try this out:

```
[me@linuxbox me]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
[                    lwp-request
2to3                 lwp-rget
```

```
2to3-2.6              lxterm
a2p                   lz
aalib-config          lzcat
aconnect              lzma
acpi_fakekey          lzmadec
acpi_listen           lzmainfo
add-apt-repository    m17n-db
addpart               magnifier

and many more...
```

Now we can see that we have changed the current working directory to /usr/bin and that it is full of files. Notice how your prompt has changed? As a convenience, it is usually set up to display the name of the working directory.

Where an absolute pathname starts from the root directory and leads to its destination, a relative pathname starts from the working directory. To do this, it uses a couple of special notations to represent relative positions in the file system tree. These special notations are "." (dot) and ".." (dot dot).

The "." notation refers to the working directory itself and the ".." notation refers to the working directory's parent directory. Here is how it works. Let's change the working directory to /usr/bin again:

```
[me@linuxbox me]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

O.K., now let's say that we wanted to change the working directory to the parent of /usr/bin which is /usr. We could do that two different ways. First, with an absolute pathname:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

Or, with a relative pathname:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Two different methods with identical results. Which one should you use? The one that requires the least typing!

Likewise, we can change the working directory from /usr to /usr/bin in two different ways. First using an absolute pathname:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Or, with a relative pathname:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Now, there is something important that I must point out here. In almost all cases, you can omit the "./". It is implied. Typing:

```
[me@linuxbox usr]$ cd bin
```

would do the same thing. In general, if you do not specify a pathname to something, the working directory will be assumed. There is one important exception to this, but we won't get to that for a while.

A Few Shortcuts

If you type **cd** followed by nothing, **cd** will change the working directory to your home directory.

A related shortcut is to type **cd *~user_name***. In this case, **cd** will change the working directory to the home directory of the specified user.

Typing **cd -** changes the working directory to the previous one.

Important facts about file names

1. File names that begin with a period character are hidden. This only means that **ls** will not list them unless you say **ls -a**. When your account was created, several hidden files were placed in your home directory to configure things for your account. Later on we will take a closer look at some of these files to see how you can customize your *environment*. In addition, some applications will place their configuration and settings files in your home directory as hidden files.

2. File names in Linux, like Unix, are case sensitive. The file names "File1" and "file1" refer to different files.

3. Linux has no concept of a "file extension" like legacy operating systems. You may name files any way you like. However, while Linux itself does not care about file extensions, many application programs do.

4. Though Linux supports long file names which may contain embedded spaces and punctuation characters, limit the punctuation characters to period, dash, and underscore. **Most importantly, do not embed spaces in file names.** If you want to represent spaces between words in a file name, use underscore characters. You will thank yourself later.

Looking Around

Now that you know how to move from working directory to working directory, we're going to take a tour of your Linux system and, along the way, learn some things about what makes it tick. But before we begin, I have to teach you some tools that will come in handy during our adventure. These are:

- **ls** (list files and directories)
- **less** (view text files)
- **file** (classify a file's contents)

ls

The **ls** command is used to list the contents of a directory. It is probably the most commonly used Linux command. It can be used in a number of different ways. Here are some examples:

Examples of the ls command

Command	Result
`ls`	List the files in the working directory
`ls /bin`	List the files in the /bin directory (or any other directory you care to specify)
`ls -l`	List the files in the working directory in long format
`ls -l /etc /bin`	List the files in the /bin directory and the /etc directory in long format
`ls -la ..`	List all files (even ones with names beginning with a period character, which are normally hidden) in the parent of the working directory in long format

These examples also point out an important concept about commands. Most commands operate like this:

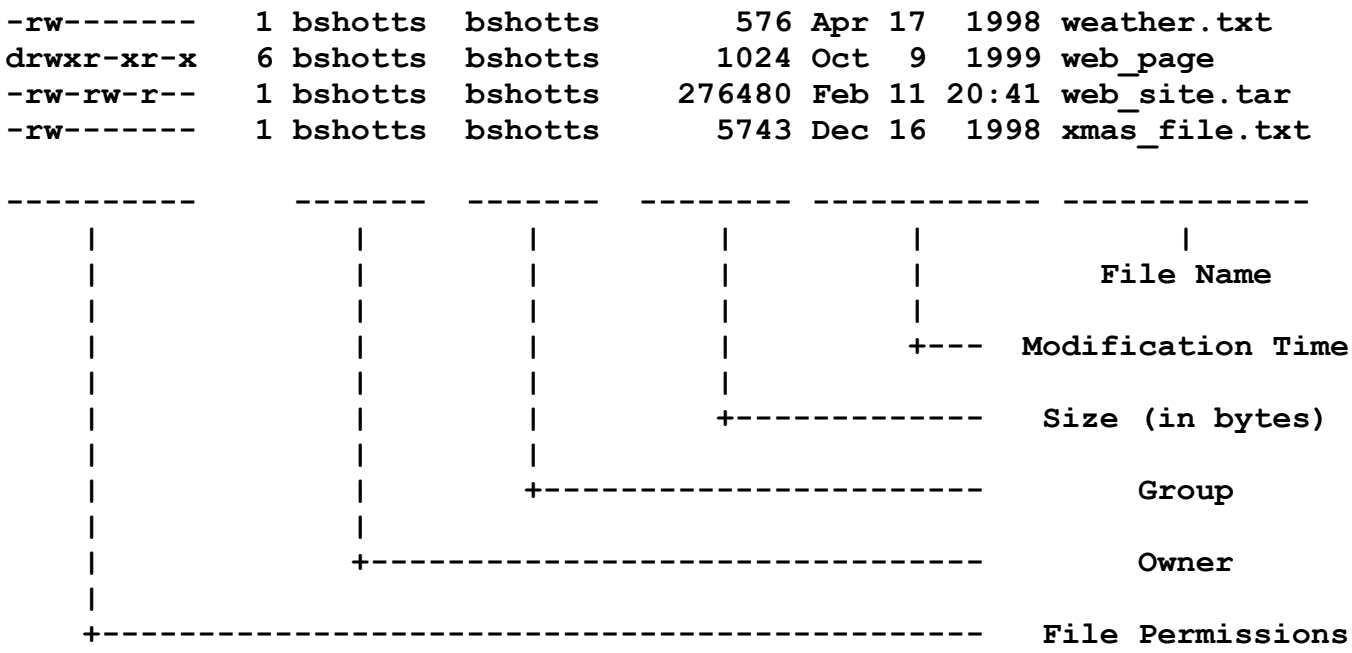
```
command -options arguments
```

where *command* is the name of the command, *-options* is one or more adjustments to the command's behavior, and *arguments* is one or more "things" upon which the command operates.

In the case of **ls**, we see that **ls** is the name of the command, and that it can have one or more options, such as **-a** and **-l**, and it can operate on one or more files or directories.

A Closer Look at Long Format

If you use the **-l** option with **ls**, you will get a file listing that contains a wealth of information about the files being listed. Here's an example:



File Name
The name of the file or directory.

Modification Time
The last time the file was modified. If the last modification occurred more than six months in the past, the date and year are displayed. Otherwise, the time of day is shown.

Size
The size of the file in bytes.

Group
The name of the group that has file permissions in addition to the file's owner.

Owner
The name of the user who owns the file.

File Permissions
A representation of the file's access permissions. The first character is the type of file. A "-" indicates a regular (ordinary) file. A "d" indicates a directory. The second set of three characters represent the read, write, and execution rights of the file's owner. The next three represent the rights of the file's group, and the final three represent the rights granted to everybody else. I'll discuss this in more detail in a later lesson.

less

less is a program that lets you view text files. This is very handy since many of the files used to control and configure Linux are human readable.

What is "text"?

There are many ways to represent information on a computer. All methods involve defining a relationship between the information and some numbers that will be used to represent it. Computers, after all, only understand numbers and all data is converted to numeric representation.

Some of these representation systems are very complex (such as compressed multimedia files), while others are rather simple. One of the earliest and simplest is called *ASCII text*. ASCII (pronounced "As-Key") is short for American Standard Code for Information Interchange. This is a simple encoding scheme that was first used on Teletype machines to map keyboard characters to numbers.

Text is a simple one-to-one mapping of characters to numbers. It is very compact. Fifty characters of text translates to fifty bytes of data. Throughout a Linux system, many files are stored in text format and there are many Linux tools that work with text files. Even the legacy operating systems recognize the importance of this format. The well-known NOTEPAD.EXE program is an editor for plain ASCII text files.

The **less** program is invoked by simply typing:

```
less text_file
```

This will display the file.

Controlling less

Once started, **less** will display the text file one page at a time. You may use the Page Up and Page Down keys to move through the text file. To exit **less**, type "q". Here are some commands that **less** will accept:

Keyboard commands for the less program

Command	Action
Page Up or b	Scroll back one page
Page Down or space	Scroll forward one page

G	Go to the end of the text file
1G	Go to the beginning of the text file
/*characters*	Search forward in the text file for an occurrence of the specified *characters*
n	Repeat the previous search
h	Display a complete list less commands and options
q	Quit

file

As you wander around your Linux system, it is helpful to determine what kind of data a file contains before you try to view it. This is where the **`file`** command comes in. **`file`** will examine a file and tell you what kind of file it is.

To use the **`file`** program, just type:

```
file name_of_file
```

The **`file`** program can recognize most types of files, such as:

Various kinds of files

File Type	Description	Viewable as text?
ASCII text	The name says it all	yes
Bourne-Again shell script text	A **`bash`** script	yes
ELF 32-bit LSB core	A core dump file (a program will create this	no

file	when it crashes)	
ELF 32-bit LSB executable	An executable binary program	no
ELF 32-bit LSB shared object	A shared library	no
GNU tar archive	A tape archive file. A common way of storing groups of files.	no, use **`tar tvf`** to view listing.
gzip compressed data	An archive compressed with **`gzip`**	no
HTML document text	A web page	yes
JPEG image data	A compressed JPEG image	no
PostScript document text	A PostScript file	yes
RPM	A Red Hat Package Manager archive	no, use **`rpm -q`** to examine contents.
Zip archive data	An archive compressed with **`zip`**	no

While it may seem that most files cannot be viewed as text, you will be surprised how many can. This is especially true of the important configuration files. You will also notice during our adventure that many features of the operating system are controlled by shell scripts. In Linux, there are no secrets!

A Guided Tour

It's time to take our tour. The table below lists some interesting places to explore. This is by no means a complete list, but it should prove to be an interesting adventure. For each of the directories listed below, do the following:

- **cd** into each directory.
- Use **ls** to list the contents of the directory.
- If you see an interesting file, use the **file** command to determine its contents.
- For text files, use **less** to view them.

Interesting directories and their contents

Directory	Description
/	The root directory where the file system begins. In most cases the root directory only contains subdirectories.
/boot	This is where the Linux kernel and boot loader files are kept. The kernel is a file called **vmlinuz**.
/etc	The **/etc** directory contains the configuration files for the system. All of the files in **/etc** should be text files. Points of interest: **/etc/passwd** The **passwd** file contains the essential information for each user. It is here that users are defined. **/etc/fstab** The **fstab** file contains a table of devices that get mounted when your system boots. This file defines your disk drives. **/etc/hosts** This file lists the network host names and IP addresses that are intrinsically known to the system. **/etc/init.d** This directory contains the scripts that start various system services typically at boot time.
/bin, /usr/bin	These two directories contain most of the programs for the system. The **/bin** directory has the essential programs that the system requires to operate, while **/usr/bin** contains applications for the system's users.
/sbin, /usr/sbin	The **sbin** directories contain programs for system administration, mostly for use by the superuser.
/usr	The **/usr** directory contains a variety of things that support user

	applications. Some highlights: **/usr/share/X11** Support files for the X Window system **/usr/share/dict** Dictionaries for the spelling checker. Bet you didn't know that Linux had a spelling checker. See **look** and **aspell**. **/usr/share/doc** Various documentation files in a variety of formats. **/usr/share/man** The man pages are kept here. **/usr/src** Source code files. If you installed the kernel source code package, you will find the entire Linux kernel source code here.
/usr/local	**/usr/local** and its subdirectories are used for the installation of software and other files for use on the local machine. What this really means is that software that is not part of the official distribution (which usually goes in **/usr/bin**) goes here. When you find interesting programs to install on your system, they should be installed in one of the **/usr/local** directories. Most often, the directory of choice is **/usr/local/bin**.
/var	The **/var** directory contains files that change as the system is running. This includes: **/var/log** Directory that contains log files. These are updated as the system runs. You should view the files in this directory from time to time, to monitor the health of your system. **/var/spool** This directory is used to hold files that are queued for some process, such as mail messages and print jobs. When a user's mail first arrives on the local system (assuming you have local mail), the messages are first stored in **/var/spool/mail**
/lib	The shared libraries (similar to DLLs in that other operating system) are kept here.
/home	**/home** is where users keep their personal work. In general, this is the only place users are allowed to write files. This keeps things nice and clean :-)
/root	This is the superuser's home directory.
/tmp	**/tmp** is a directory in which programs can write their temporary files.
/dev	The **/dev** directory is a special directory, since it does not really contain files in the usual sense. Rather, it contains devices that are available to the system. In Linux (like Unix), devices are treated like files. You can

	read and write devices as though they were files. For example **/dev/fd0** is the first floppy disk drive, **/dev/sda** (**/dev/hda** on older systems) is the first hard drive. All the devices that the kernel understands are represented here.
/proc	The **/proc** directory is also special. This directory does not contain files. In fact, this directory does not really exist at all. It is entirely virtual. The **/proc** directory contains little peep holes into the kernel itself. There are a group of numbered entries in this directory that correspond to all the processes running on the system. In addition, there are a number of named entries that permit access to the current configuration of the system. Many of these entries can be viewed. Try viewing **/proc/cpuinfo**. This entry will tell you what the kernel thinks of your CPU.
/media,/mnt	Finally, we come to **/media**, a normal directory which is used in a special way. The **/media** directory is used for *mount points*. As we learned in the second lesson, the different physical storage devices (like hard disk drives) are attached to the file system tree in various places. This process of attaching a device to the tree is called *mounting*. For a device to be available, it must first be mounted. When your system boots, it reads a list of mounting instructions in the file **/etc/fstab**, which describes which device is mounted at which mount point in the directory tree. This takes care of the hard drives, but you may also have devices that are considered temporary, such as CD-ROMs, thumb drives, and floppy disks. Since these are removable, they do not stay mounted all the time. The **/media** directory is used by the automatic device mounting mechanisms found in modern desktop oriented Linux distributions. On systems that require manual mounting of removable devices, the **/mnt** directory provides a convenient place for mounting these temporary devices. You will often see the directories **/mnt/floppy** and **/mnt/cdrom**. To see what devices and mount points are used, type **mount**.

A weird kind of file...

During your tour, you probably noticed a strange kind of directory entry, particularly in the **/boot** and **/lib** directories. When listed with **ls -l**, you would have seen something like this:

```
lrwxrwxrwx     25 Jul  3 16:42 System.map -> /boot/System.map-2.0.36-3
-rw-r--r-- 105911 Oct 13  1998 System.map-2.0.36-0.7
-rw-r--r-- 105935 Dec 29  1998 System.map-2.0.36-3
-rw-r--r-- 181986 Dec 11  1999 initrd-2.0.36-0.7.img
-rw-r--r-- 182001 Dec 11  1999 initrd-2.0.36.img
lrwxrwxrwx     26 Jul  3 16:42 module-info -> /boot/module-info-2.0.36-3
-rw-r--r--  11773 Oct 13  1998 module-info-2.0.36-0.7
-rw-r--r--  11773 Dec 29  1998 module-info-2.0.36-3
lrwxrwxrwx     16 Dec 11  1999 vmlinuz -> vmlinuz-2.0.36-3
-rw-r--r-- 454325 Oct 13  1998 vmlinuz-2.0.36-0.7
```

```
-rw-r--r-- 454434 Dec 29  1998 vmlinuz-2.0.36-3
```

Notice the files, `System.map, module-info` and `vmlinuz`. See the strange notation after the file names?

These three files are called *symbolic links*. Symbolic links are a special type of file that points to another file. With symbolic links, it is possible for a single file to have multiple names. Here's how it works: Whenever the system is given a file name that is a symbolic link, it transparently maps it to the file it is pointing to.

Just what is this good for? This is a very handy feature. Let's consider the directory listing above (which is the `/boot` directory of an old Red Hat 5.2 system). This system has had multiple versions of the Linux kernel installed. We can see this from the files `vmlinuz-2.0.36-0.7` and `vmlinuz-2.0.36-3`. These file names suggest that both version 2.0.36-0.7 and 2.0.36-3 are installed. Because the file names contain the version it is easy to see the differences in the directory listing. However, this would be confusing to programs that rely on a fixed name for the kernel file. These programs might expect the kernel to simply be called `"vmlinuz"`. Here is where the beauty of the symbolic link comes in. By creating a symbolic link called `vmlinuz` that points to `vmlinuz-2.0.36-3`, we have solved the problem.

To create symbolic links, use the **`ln`** command.

Manipulating Files

This lesson will introduce you to the following commands:

- **cp** - copy files and directories
- **mv** - move or rename files and directories
- **rm** - remove files and directories
- **mkdir** - create directories

These four commands are among the most frequently used Linux commands. They are the basic commands for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, you can drag and drop a file from one directory to another, cut and paste files, delete files, etc. So why use these old command line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. For example, how would you copy all the HTML files from one directory to another, but only copy files that did not exist in the destination directory or were newer than the versions in the destination directory? Pretty hard with with a file manager. Pretty easy with the command line:

```
[me@linuxbox me]$ cp -u *.html destination
```

Wildcards

Before I begin with our commands, I want to talk about a shell feature that makes these commands so powerful. Since the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are called *wildcards*. Wildcards allow you to select filenames based on patterns of characters. The table below lists the wildcards and what they select:

Summary of wildcards and their meanings

Wildcard	Meaning
*	Matches any characters
?	Matches any single character

[*characters*]	Matches any character that is a member of the set *characters*. The set of characters may also be expressed as a *POSIX character class* such as one of the following: POSIX Character Classes **[:alnum:]** Alphanumeric characters **[:alpha:]** Alphabetic characters **[:digit:]** Numerals **[:upper:]** Uppercase alphabetic characters **[:lower:]** Lowercase alphabetic characters
[!*characters*]	Matches any character that is not a member of the set *characters*

Using wildcards, it is possible to construct very sophisticated selection criteria for filenames. Here are some examples of patterns and what they match:

Examples of wildcard matching

Pattern	Matches
`*`	All filenames
`g*`	All filenames that begin with the character "g"
`b*.txt`	All filenames that begin with the character "b" and end with the characters ".txt"
`Data???`	Any filename that begins with the characters "Data" followed by exactly 3 more characters
`[abc]*`	Any filename that begins with "a" or "b" or "c" followed by any other characters
`[[:upper:]]*`	Any filename that begins with an uppercase letter. This is an example of a character class.

BACKUP.[[:digit:]][[:digit:]]	Another example of character classes. This pattern matches any filename that begins with the characters "BACKUP." followed by exactly two numerals.
***[![:lower:]]**	Any filename that does not end with a lowercase letter.

You can use wildcards with any command that accepts filename arguments.

cp

The **cp** program copies files and directories. In its simplest form, it copies a single file:

```
[me@linuxbox me]$ cp file1 file2
```

It can also be used to copy multiple files (and/or directories) to a different directory:

```
[me@linuxbox me]$ cp file... directory
```

A note on notation: ... signifies that an item can be repeated one or more times.

Other useful examples of **cp** and its options include:

Examples of the cp command

Command	Results
cp *file1 file2*	Copies the contents of *file1* into *file2*. If *file2* does not exist, it is created; **otherwise, *file2* is silently overwritten with the contents of *file1*.**
cp -i *file1 file2*	Like above however, since the "-i" (interactive) option is specified, if *file2* exists, the user is prompted before it is overwritten with the contents of *file1*.
cp *file1 dir1*	Copy the contents of *file1* (into a file named *file1*) inside of directory *dir1*.

cp -R *dir1 dir2*	Copy the contents of the directory *dir1*. If directory *dir2* does not exist, it is created. Otherwise, it creates a directory named *dir1* within directory *dir2*.

mv

The **mv** command moves or renames files and directories depending on how it is used. It will either move one or more files to a different directory, or it will rename a file or directory. To rename a file, it is used like this:

```
[me@linuxbox me]$ mv filename1 filename2
```

To move files (and/or directories) to a different directory:

```
[me@linuxbox me]$ mv file... directory
```

Examples of **mv** and its options include:

Examples of the mv command

Command	Results
mv *file1 file2*	If *file2* does not exist, then *file1* is renamed *file2*. **If *file2* exists, its contents are silently replaced with the contents of *file1*.**
mv -i *file1 file2*	Like above however, since the "-i" (interactive) option is specified, if *file2* exists, the user is prompted before it is overwritten with the contents of *file1*.
mv *file1 file2 file3 dir1*	The files *file1, file2, file3* are moved to directory *dir1*. If *dir1* does not exist, **mv** will exit with an error.
mv *dir1 dir2*	If *dir2* does not exist, then *dir1* is renamed *dir2*. If *dir2* exists, the directory *dir1* is moved within directory *dir2*.

rm

The **rm** command removes (deletes) files and directories.

```
[me@linuxbox me]$ rm file...
```

It can also be used to delete directories:

```
[me@linuxbox me]$ rm -r directory...
```

Examples of **rm** and its options include:

Examples of the rm command

Command	Results
rm *file1 file2*	Delete *file1* and *file2*.
rm -i *file1 file2*	Like above however, since the "-i" (interactive) option is specified, the user is prompted before each file is deleted.
rm -r *dir1 dir2*	Directories *dir1* and *dir2* are deleted along with all of their contents.

Be careful with rm!

Linux does not have an undelete command. Once you delete something with **rm**, it's gone. You can inflict terrific damage on your system with **rm** if you are not careful, particularly with wildcards.

Before you use rm with wildcards, try this helpful trick: construct your command using **ls** instead. By doing this, you can see the effect of your wildcards before you delete files. After you have tested your command with **ls**, recall the command with the up-arrow key and then substitute **rm** for **ls** in the command.

mkdir

The **mkdir** command is used to create directories. To use it, you simply type:

```
[me@linuxbox me]$ mkdir directory...
```

Using Commands with Wildcards

Since the commands we have covered here accept multiple file and directories names as arguments, you can use wildcards to specify them. Here are a few examples:

Command examples using wildcards

Command	Results
cp *.txt text_files	Copy all files in the current working directory with names ending with the characters ".txt" to an existing directory named *text_files*.
mv my_dir ../*.bak my_new_dir	Move the subdirectory *my_dir* and all the files ending in ".bak" in the current working directory's parent directory to an existing directory named *my_new_dir*.
rm *~	Delete all files in the current working directory that end with the character "~". Some applications create backup files using this naming scheme. Using this command will clean them out of a directory.

Working with Commands

Up until now you have seen a number of commands and their mysterious options and arguments. In this lesson, we will try to remove some of that mystery. This lesson will introduce the following commands.

- **type** - Display information about command type
- **which** - Locate a command
- **help** - Display reference page for shell builtin
- **man** - Display an on-line command reference

What are "Commands?"

Commands can be one of 4 different kinds:

1. **An executable program** like all those files we saw in /usr/bin. Within this category, programs can be *compiled binaries* such as programs written in C and C++, or programs written in *scripting languages* such as the shell, Perl, Python, Ruby, etc.
2. **A command built into the shell itself.** bash provides a number of commands internally called *shell builtins*. The **cd** command, for example, is a shell builtin.
3. **A shell function.** These are miniature shell scripts incorporated into the *environment*. We will cover configuring the environment and writing shell functions in later lessons, but for now, just be aware that they exist.
4. **An alias.** Commands that you can define yourselves, built from other commands. This will be covered in a later lesson.

Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of ways to find out.

type

The **type** command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
type command
```

where “command” is the name of the command you want to examine. Here are some examples:

```
[me@linuxbox me]$ type type
type is a shell builtin

[me@linuxbox me]$ type ls
ls is aliased to `ls --color=tty'

[me@linuxbox me]$ type cp
cp is /bin/cp
```

Here we see the results for three different commands. Notice that the one for ls (taken from a Fedora system) and how the ls command is actually an alias for the ls command with the “-- color=tty” option added. Now we know why the output from ls is displayed in color!

which

Sometimes there is more than one version of an executable program installed on a system. While this is not very common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the **which** command is used:

```
[me@linuxbox me]$ which ls
/bin/ls
```

which only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs.

Getting Command Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

help

bash has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. Optionally, you may add the -m option to change the format of the output. For example:

```
[me@linuxbox me]$ help -m cd

NAME
    cd - Change the shell working directory.

SYNOPSIS
    cd [-L|-P] [dir]

DESCRIPTION
    Change the shell working directory.

    Change the current directory to DIR.  The default DIR is the value of the
    HOME shell variable.

    The variable CDPATH defines the search path for the directory containing
    DIR.  Alternative directory names in CDPATH are separated by a colon (:).
    A null directory name is the same as the current directory.  If DIR begins
    with a slash (/), then CDPATH is not used.

    If the directory is not found, and the shell option `cdable_vars' is set,
    the word is assumed to be  a variable name.  If that variable has a value,
    its value is used for DIR.

    Options:
```

```
          -L      force symbolic links to be followed
          -P      use the physical directory structure without following symbolic
          links

      The default is to follow symbolic links, as if `-L' were specified.

      Exit Status:
      Returns 0 if the directory is changed; non-zero otherwise.

  SEE ALSO
      bash(1)

  IMPLEMENTATION
      GNU bash, version 4.1.5(1)-release (i486-pc-linux-gnu)
      Copyright (C) 2009 Free Software Foundation, Inc.
```

A note on notation: When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the **cd** command above:

```
cd [-L|-P] [dir]
```

This notation says that the command **cd** may be followed optionally by either a “-L” or a “-P” and further, optionally followed by the argument “dir”.

--help

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options. For example:

```
[me@linuxbox me]$ mkdir --help

Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

   -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options
too.
   -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
   -p, --parents     no error if existing, make parent directories as
                     needed
   -v, --verbose     print a message for each created directory
   --help            display this help and exit
   --version         output version information and exit
```

Some programs don't support the “--help” option, but try it anyway. Often it results in an error message that will reveal similar usage information.

man

Most executable programs intended for command line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called **man** is used to view them. It is used like this:

```
man program
```

where “program” is the name of the command to view. Man pages vary somewhat in format but generally contain a title, a synopsis of the command's syntax, a description of the command's purpose, and a listing and description of each of the command's options. Man pages, however, do not usually include examples, and are intended as a reference, not a tutorial. As an example, let's try viewing the man pagefor the **ls** command:

```
[me@linuxbox me]$ man ls
```

On most Linux systems, **man** uses **less** to display the manual page, so all of the familiar **less** commands work while displaying the page.

README and Other Documentation Files

Many software packages installed on your system have documentation files residing in the /usr/share/doc directory. Most of these are stored in plain text format and can be viewed with **less**. Some of the files are in HTML format and can be viewed with your web browser. You may encounter some files ending with a “.gz” extension. This indicates that they have been compressed with the **gzip** compression program. The gzip package includes a special version of **less** called **zless** that will display the contents of gzip-compressed text files.

I/O Redirection

In this lesson, we will explore a powerful feature used by many command line programs called *input/output redirection*. As we have seen, many commands such as **ls** print their output on the display. This does not have to be the case, however. By using some special notations we can *redirect* the output of many commands to files, devices, and even to the input of other commands.

Standard Output

Most command line programs that display their results do so by sending their results to a facility called *standard output*. By default, standard output directs its contents to the display. To redirect standard output to a file, the ">" character is used like this:

```
[me@linuxbox me]$ ls > file_list.txt
```

In this example, the **ls** command is executed and the results are written in a file named file_list.txt. Since the output of **ls** was redirected to the file, no results appear on the display.

Each time the command above is repeated, file_list.txt is overwritten from the beginning with the output of the command **ls**. If you want the new results to be *appended* to the file instead, use ">>" like this:

```
[me@linuxbox me]$ ls >> file_list.txt
```

When the results are appended, the new results are added to the end of the file, thus making the file longer each time the command is repeated. If the file does not exist when you attempt to append the redirected output, the file will be created.

Standard Input

Many commands can accept input from a facility called *standard input*. By default, standard input gets its contents from the keyboard, but like standard output, it can be redirected. To redirect standard input from a file instead of the keyboard, the "<" character is used like this:

```
[me@linuxbox me]$ sort < file_list.txt
```

In the example above, we used the **sort** command to process the contents of file_list.txt. The results are output on the display since the standard output was not redirected. We could redirect standard output to another file like this:

```
[me@linuxbox me]$ sort < file_list.txt > sorted_file_list.txt
```

As you can see, a command can have both its input and output redirected. Be aware that the order of the redirection does not matter. The only requirement is that the redirection operators (the "<" and ">") must appear after the other options and arguments in the command.

Pipelines

The most useful and powerful thing you can do with I/O redirection is to connect multiple commands together with what are called *pipelines*. With pipelines, the standard output of one command is fed into the standard input of another. Here is my absolute favorite:

```
[me@linuxbox me]$ ls -l | less
```

In this example, the output of the **ls** command is fed into **less**. By using this **"| less"** trick, you can make any command have scrolling output. I use this technique all the time.

By connecting commands together, you can acomplish amazing feats. Here are some examples you'll want to try:

Examples of commands used together with pipelines

Command	What it does
ls -lt | head	Displays the 10 newest files in the current directory.
du | sort -nr	Displays a list of directories and how much space they consume, sorted from the largest to the smallest.
find . -type f -print | wc -l	Displays the total number of files in the current working directory and all of its subdirectories.

Filters

One kind of program frequently used in pipelines is called *filters*. Filters take standard input and perform an operation upon it and send the results to standard output. In this way, they can be combined to process information in powerful ways. Here are some of the common programs that can act as filters:

Common filter commands

Program	What it does
sort	Sorts standard input then outputs the sorted result on standard output.
uniq	Given a sorted stream of data from standard input, it removes duplicate lines of data (i.e., it makes sure that every line is unique).
grep	Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters.
fmt	Reads text from standard input, then outputs formatted text on standard output.
pr	Takes text input from standard input and splits the data into pages with page breaks, headers and footers in preparation for printing.
head	Outputs the first few lines of its input. Useful for getting the header of a file.
tail	Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file.
tr	Translates characters. Can be used to perform tasks such as upper/lowercase conversions or changing line termination characters from one type to another (for example, converting DOS text files into Unix style text files).
sed	Stream editor. Can perform more sophisticated text translations than **tr**.
awk	An entire programming language designed for constructing filters. Extremely powerful.

Performing tasks with pipelines

1. **Printing from the command line.** Linux provides a program called **lpr** that accepts standard input and sends it to the printer. It is often used with pipes and filters. Here are a couple of examples:

   ```
   cat poorly_formatted_report.txt | fmt | pr | lpr

   cat unsorted_list_with_dupes.txt | sort | uniq | pr | lpr
   ```

   In the first example, we use **cat** to read the file and output it to standard output, which is piped into the standard input of **fmt. fmt** formats the text into neat paragraphs and outputs it to standard output, which is piped into the standard input of **pr. pr** splits the text neatly into pages and outputs it to standard output, which is piped into the standard input of **lpr. lpr** takes its standard input and sends it to the printer.

   The second example starts with an unsorted list of data with duplicate entries. First, **cat** sends the list into **sort** which sorts it and feeds it into **uniq** which removes any duplicates. Next **pr** and **lpr** are used to paginate and print the list.

2. **Viewing the contents of tar files** Often you will see software distributed as a *gzipped tar file*. This is a traditional Unix style tape archive file (created with **tar**) that has been compressed with **gzip**. You can recognize these files by their traditional file extensions, ".tar.gz" or ".tgz". You can use the following command to view the directory of such a file on a Linux system:

   ```
   tar tzvf name_of_file.tar.gz | less
   ```

Expansion

Each time you type a command line and press the enter key, bash performs several processes upon the text before it carries out your command. We have seen a couple of cases of how a simple character sequence, for example “*”, can have a lot of meaning to the shell. The process that makes this happen is called *expansion*. With expansion, you type something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the **echo** command. **echo** is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

```
[me@linuxbox me]$ echo this is a test
this is a test
```

That's pretty straightforward. Any argument passed to **echo** gets displayed. Let's try another example:

```
[me@linuxbox me]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

So what just happened? Why didn't **echo** print “*”? As you recall from our work with wildcards, the “*” character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the “*” into something else (in this instance, the names of the files in the current working directory) before the **echo** command is executed. When the enter key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the **echo** command never saw the “*”, only its expanded result. Knowing this, we can see that **echo** behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called *pathname expansion*. If we try some of the techniques that we employed in our earlier lessons, we will see that they are really expansions. Given a home directory that looks like this:

```
[me@linuxbox me]$ ls

Desktop
ls-output.txt
Documents Music
Pictures
Public
Templates
Videos
```

we could carry out the following expansions:

```
[me@linuxbox me]$ echo D*
Desktop Documents
```

and:

```
[me@linuxbox me]$ echo *s
Documents Pictures Templates Videos
```

or even:

```
[me@linuxbox me]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

and looking beyond our home directory:

```
[me@linuxbox me]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Tilde Expansion

As you may recall from our introduction to the **cd** command, the tilde character ("~") has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user, or if no user is named, the home directory of the current user:

```
[me@linuxbox me]$ echo ~
/home/me
```

If user "foo" has an account, then:

```
[me@linuxbox me]$ echo ~foo
/home/foo
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allow us to use the shell prompt as a calculator:

```
[me@linuxbox me]$ echo $((2 + 2))
4
```

Arithmetic expansion uses the form:

```
$((expression))
```

where expression is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic expansion only supports integers (whole numbers, no decimals), but can perform quite a number of different operations.

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply five squared by three:

```
[me@linuxbox me]$ echo $(($((5**2)) * 3))
75
```

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the example above and get the same result using a single expansion instead of two:

```
[me@linuxbox me]$ echo $(((5**2) * 3))
75
```

Here is an example using the division and remainder operators. Notice the effect of integer division:

```
[me@linuxbox me]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox me]$ echo with $((5%2)) left over.
with 1 left over.
```

Brace Expansion

Perhaps the strangest expansion is called *brace expansion*. With it, you can create multiple text strings from a pattern containing braces. Here's an example:

```
[me@linuxbox me]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a *preamble* and a trailing portion called a *postscript*. The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
[me@linuxbox me]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

A range of letters in reverse order:

```
[me@linuxbox me]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested:

```
[me@linuxbox me]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

So what is this good for? The most common application is to make lists of files or directories to be created. For example, if you were a photographer and had a large collection of images you wanted to organize into years and months, the first thing you might do is create a series of directories named in numeric “Year-Month” format. This way, the directory names will sort in chronological order. You could type out a complete list of directories, but that's a lot of work and it's error-prone too. Instead, you could do this:

```
[me@linuxbox me]$ mkdir Photos
[me@linuxbox me]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-0{1..9} {2007..2009}-{10..12}
[me@linuxbox Photos]$ ls

2007-01 2007-07 2008-01 2008-07 2009-01 2009-07
2007-02 2007-08 2008-02 2008-08 2009-02 2009-08
2007-03 2007-09 2008-03 2008-09 2009-03 2009-09
2007-04 2007-10 2008-04 2008-10 2009-04 2009-10
2007-05 2007-11 2008-05 2008-11 2009-05 2009-11
2007-06 2007-12 2008-06 2008-12 2009-06 2009-12
```

Pretty slick!

Parameter Expansion

We're only going to touch briefly on *parameter expansion* in this lesson, but we'll be covering it more later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called *variables*, are available for your examination. For example, the variable named “USER” contains your user name. To invoke parameter expansion and reveal the contents of USER you would do this:

```
[me@linuxbox me]$ echo $USER
me
```

To see a list of available variables, try this:

```
[me@linuxbox me]$ printenv | less
```

You may have noticed that with other types of expansion, if you mistype a pattern, the expansion will not take place and the echo command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable, the expansion will still take place, but will result in an empty string:

```
[me@linuxbox me]$ echo $SUER
[me@linuxbox ~]$
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion:

```
[me@linuxbox me]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

One of my favorites goes something like this:

```
[me@linuxbox me]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Here we passed the results of which **cp** as an argument to the **ls** command, thereby getting the listing of of the **cp** program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output shown):

```
[me@linuxbox me]$ file $(ls /usr/bin/* | grep bin/zip)

/usr/bin/bunzip2:
/usr/bin/zip:      ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipcloak: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipgrep:  POSIX shell script text executable
/usr/bin/zipinfo:  ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipnote:  ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
/usr/bin/zipsplit: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
```

In this example, the results of the pipeline became the argument list of the file command. There is an alternate syntax for command substitution in older shell programs which is also supported in **bash**. It uses back-quotes instead of the dollar sign and parentheses:

```
[me@linuxbox me]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take for example:

```
[me@linuxbox me]$ echo this is a     test
this is a test
```

or:

```
[me@linuxbox me]$ [me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

In the first example, word-splitting by the shell removed extra whitespace from the echo command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of “$1” because it was an undefined variable. The shell provides a mechanism called quoting to selectively suppress unwanted expansions.

Double Quotes

The first type of quoting we will look at is double quotes. If you place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are “$”, “\” (backslash), and “`” (back- quote). This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say you were the unfortunate victim of a file called two words.txt. If you tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument:

```
[me@linuxbox me]$ ls -l two words.txt

ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, you can stop the word-splitting and get the desired result; further, you can even repair the damage:

```
[me@linuxbox me]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2008-02-20 13:03 two words.txt
[me@linuxbox me]$ mv "two words.txt" two_words.txt
```

There! Now we don't have to keep typing those pesky double quotes. Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

```
[me@linuxbox me]$ echo "$USER $((2+2)) $(cal)"

me 4
February 2008
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word-splitting appears to remove extra spaces in our text:

```
[me@linuxbox me]$ echo this is a     test
this is a test
```

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as delimiters between words. This means that unquoted spaces, tabs, and newlines are not

considered to be part of the text. They only serve as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

```
[me@linuxbox me]$ echo "this is a     test"
this is a     test
```

word-splitting is suppressed and the embedded spaces are not treated as delimiters, rather they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument. The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
[me@linuxbox me]$ echo $(cal)
February 2008 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox me]$ echo "$(cal)"

February 2008
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing thirty-eight arguments. In the second, a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

If you need to suppress all expansions, you use single quotes. Here is a comparison of unquoted, double quotes, and single quotes:

```
[me@linuxbox me]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox me]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox me]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As you can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

Escaping Characters

Sometimes you only want to quote a single character. To do this, you can precede a character with a backslash, which in this context is called the *escape character*. Often this is done inside double quotes to selectively prevent an expansion:

```
[me@linuxbox me]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include “$”, “!”, “&”, “ “, and others. To include a special character in a filename you can to this:

```
[me@linuxbox me]$ mv bad\&filename good_filename
```

To allow a backslash character to appear, escape it by typing “\\”. Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

More Backslash Tricks

If you look at the **man** pages for any program written by the GNU project, you will notice that in addition to command line options consisting of a dash and a single letter, there are also long option names that begin with two dashes. For example, the following are equivalent:

```
ls -r
ls --reverse
```

Why do they support both? The short form is for lazy typists on the command line and the long form is mostly for scripts though some options may only be long form. I sometimes use obscure options, and I find the long form useful if I have to review a script again months after I wrote it. Seeing the long form helps me understand what the option does, saving me a trip to the **man** page. A little more typing now, a lot less work later. Laziness is maintained.

As you might suspect, using the long form options can make a single command line very long. To combat this problem, you can use a backslash to get the shell to ignore a newline character like this:

```
ls -l \
   --reverse \
   --human-readable \
   --full-time
```

Using the backslash in this way allows us to embed newlines in our command. Note that for this trick to work, the newline must be typed immediately after the backslash. If you put a space after the backslash, the space

will be ignored, not the newline. Backslashes are also used to insert special characters into our text. These are called *backslash escape characters*. Here are the common ones:

Escape Character	Name	Possible Uses
\n	newline	Adding blank lines to text
\t	tab	Inserting horizontal tabs to text
\a	alert	Makes your terminal beep
\\	backslash	Inserts a backslash
\f	formfeed	Sending this to your printer ejects the page

The use of the backslash escape characters is very common. This idea first appeared in the C programming language. Today, the shell, C++, perl, python, awk, tcl, and many other programming languages use this concept. Using the **echo** command with the -e option will allow us to demonstrate:

```
[me@linuxbox me]$ echo -e "Inserting several blank lines\n\n\n"
Inserting several blank lines



[me@linuxbox me]$ echo -e "Words\tseparated\tby\thorizontal\ttabs."
Words separated   by  horizontal  tabs
[me@linuxbox me]$ echo -e "\aMy computer went \"beep\"."
My computer went "beep".
[me@linuxbox me]$ echo -e "DEL C:\\WIN2K\\LEGACY_OS.EXE"
DEL C:\WIN2K\LEGACY_OS.EXE
```

Permissions

The Unix-like operating systems, such as Linux differ from other computing systems in that they are not only *multitasking* but also *multi-user*.

What exactly does this mean? It means that more than one user can be operating the computer at the same time. While your computer only has one keyboard and monitor, it can still be used by more than one user. For example, if your computer is attached to a network, or the Internet, remote users can log in via **ssh** (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the output displayed on a remote computer. The X Window system supports this.

The multi-user capability of Unix-like systems is a feature that is deeply ingrained into the design of the operating system. If you remember the environment in which Unix was created, this makes perfect sense. Years ago before computers were "personal," they were large, expensive, and centralized. A typical university computer system consisted of a large mainframe computer located in some building on campus and *terminals* were located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

In order to make this practical, a method had to be devised to protect the users from each other. After all, you could not allow the actions of one user to crash the computer, nor could you allow one user to interfere with the files belonging to another user.

This lesson will cover the following commands:

- **chmod** - modify file access rights
- **su** - temporarily become the superuser
- **sudo** - temporarily become the superuser
- **chown** - change file ownership
- **chgrp** - change a file's group ownership

File Permissions

On a Linux system, each file and directory is assigned access rights for the owner of the file, the members of a group of related users, and everybody else. Rights can be assigned to read a file, to write a file, and to execute a file (i.e., run the file as a program).

To see the permission settings for a file, we can use the **ls** command. As an example, we will look at the **bash** program which is located in the **/bin** directory:

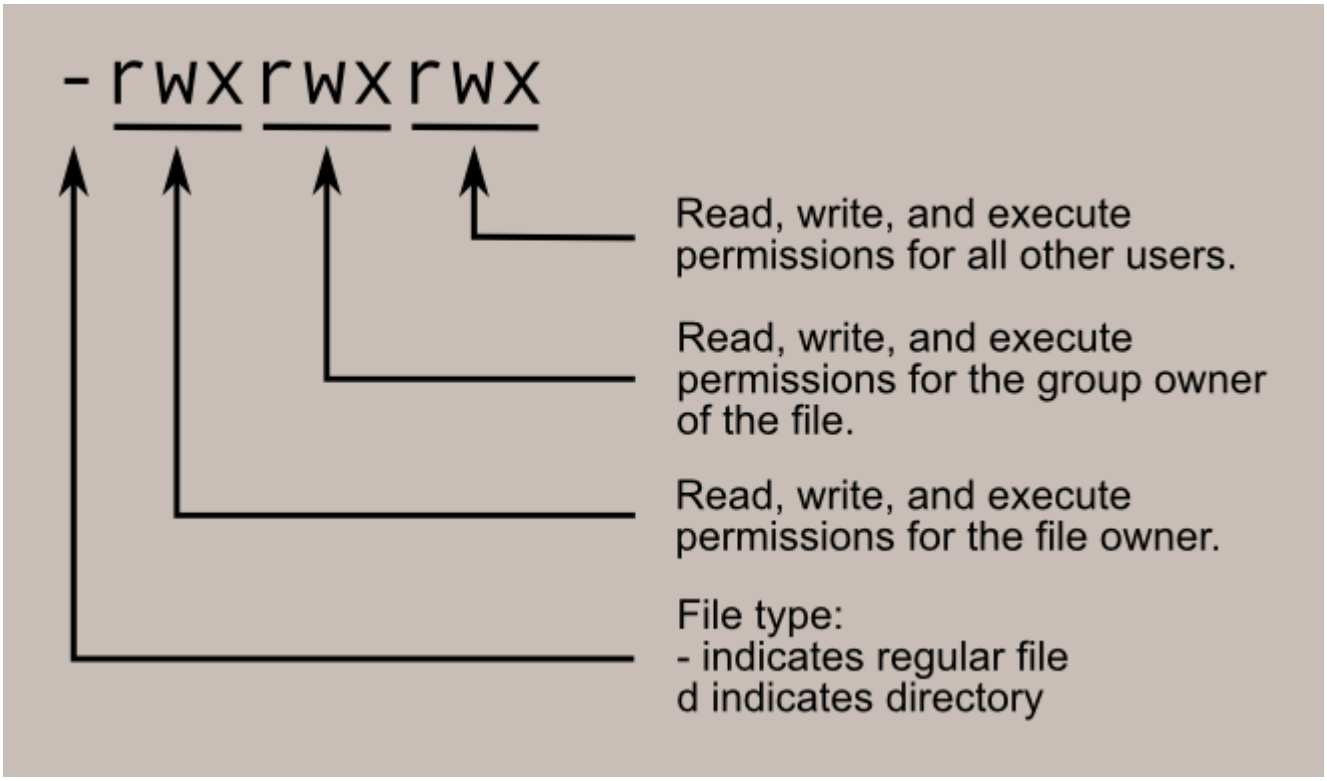
```
[me@linuxbox me]$ ls -l /bin/bash

-rwxr-xr-x 1 root root  316848 Feb 27  2000 /bin/bash
```

Here we can see:

- The file "/bin/bash" is owned by user "root"
- The superuser has the right to read, write, and execute this file
- The file is owned by the group "root"
- Members of the group "root" can also read and execute this file
- Everybody else can read and execute this file

In the diagram below, we see how the first portion of the listing is interpreted. It consists of a character indicating the file type, followed by three sets of three characters that convey the reading, writing and execution permission for the owner, group, and everybody else.



chmod

The **chmod** command is used to change the permissions of a file or directory. To use it, you specify the desired permission settings and the file or files that you wish to modify. There are two ways to specify the permissions. In this lesson we will focus on one of these, called the *octal notation* method.

It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them). Here's how it works:

```
rwx rwx rwx = 111 111 111
rw- rw- rw- = 110 110 110
rwx --- --- = 111 000 000

and so on...
```

```
rwx = 111 in binary = 7
rw- = 110 in binary = 6
r-x = 101 in binary = 5
r-- = 100 in binary = 4
```

Now, if you represent each of the three sets of permissions (owner, group, and other) as a single digit, you have a pretty convenient way of expressing the possible permissions settings. For example, if we wanted to set `some_file` to have read and write permission for the owner, but wanted to keep the file private from others, we would:

```
[me@linuxbox me]$ chmod 600 some_file
```

Here is a table of numbers that covers all the common settings. The ones beginning with "7" are used with programs (since they enable execution) and the rest are for other kinds of files.

Value	Meaning
777	**(rwxrwxrwx)** No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.
755	**(rwxr-xr-x)** The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
700	**(rwx------)** The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
666	**(rw-rw-rw-)** All users may read and write the file.
644	**(rw-r--r--)** The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.
600	**(rw-------)** The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

Directory Permissions

The **chmod** command can also be used to control the access permissions for directories. Again, we can use the octal notation to set permissions, but the meaning of the r, w, and x attributes is different:

- **r** - Allows the contents of the directory to be listed if the x attribute is also set.
- **w** - Allows files within the directory to be created, deleted, or renamed if the x attribute is also set.
- **x** - Allows a directory to be entered (i.e. **cd dir**).

Here are some useful settings for directories:

Value	Meaning
777	**(rwxrwxrwx)** No restrictions on permissions. Anybody may list files, create new files in the directory and delete files in the directory. Generally not a good setting.
755	**(rwxr-xr-x)** The directory owner has full access. All others may list the directory, but cannot create files nor delete them. This setting is common for directories that you wish to share with other users.
700	**(rwx------)** The directory owner has full access. Nobody else has any rights. This setting is useful for directories that only the owner may use and must be kept private from others.

Becoming the Superuser for a Short While

It is often necessary to become the superuser to perform important system administration tasks, but as you have been warned, you should not stay logged in as the superuser. In most distributions, there is a program that can give you temporary access to the superuser's privileges. This program is called **su** (short for substitute user) and can be used in those cases when you need to be the superuser for a small number of tasks. To become the superuser, simply type the **su** command. You will be prompted for the superuser's password:

```
[me@linuxbox me]$ su
Password:
[root@linuxbox me]#
```

After executing the **su** command, you have a new shell session as the superuser. To exit the superuser session, type **exit** and you will return to your previous session.

In some distributions, most notably Ubuntu, an alternate method is used. Rather than using **su**, these systems employ the **sudo** command instead. With **sudo**, one or more users are granted superuser privileges on an as needed basis. To execute a command as the superuser, the desired command is simply preceeded with the **sudo** command. After the command is entered, the user is prompted for the user's password rather than the superuser's:

```
[me@linuxbox me]$ sudo some_command
Password:
[me@linuxbox me]$
```

Changing File Ownership

You can change the owner of a file by using the **chown** command. Here's an example: Suppose I wanted to change the owner of `some_file` from "me" to "you". I could:

```
[me@linuxbox me]$ su
Password:
[root@linuxbox me]# chown you some_file
[root@linuxbox me]# exit
[me@linuxbox me]$
```

Notice that in order to change the owner of a file, you must be the superuser. To do this, our example employed the **su** command, then we executed **chown**, and finally we typed **exit** to return to our previous session.

chown works the same way on directories as it does on files.

Changing Group Ownership

The group ownership of a file or directory may be changed with **chgrp**. This command is used like this:

```
[me@linuxbox me]$ chgrp new_group some_file
```

In the example above, we changed the group ownership of `some_file` from its previous group to "new_group". You must be the owner of the file or directory to perform a **chgrp**.

Job Control

In the previous lesson, we looked at some of the implications of Linux being a multi-user operating system. In this lesson, we will examine the multitasking nature of Linux, and how this is manipulated with the command line interface.

As with any multitasking operating system, Linux executes multiple, simultaneous processes. Well, they appear simultaneous, anyway. Actually, a single processor computer can only execute one process at a time but the Linux kernel manages to give each process its turn at the processor and each appears to be running at the same time.

There are several commands that can be used to control processes. They are:

- **ps** - list the processes running on the system
- **kill** - send a signal to one or more processes (usually to "kill" a process)
- **jobs** - an alternate way of listing your own processes
- **bg** - put a process in the background
- **fg** - put a process in the forground

A Practical Example

While it may seem that this subject is rather obscure, it can be very practical for the average user who mostly works with the graphical user interface. You might not know this, but most (if not all) of the graphical programs can be launched from the command line. Here's an example: there is a small program supplied with the X Window system called **xload** which displays a graph representing system load. You can excute this program by typing the following:

```
[me@linuxbox me]$ xload
```

Notice that the small **xload** window appears and begins to display the system load graph. Notice also that your prompt did not reappear after the program launched. The shell is waiting for the program to finish before control returns to you. If you close the **xload** window, the **xload** program terminates and the prompt returns.

Putting a Program into the Background

Now, in order to make life a little easier, we are going to launch the **xload** program again, but this time we will put it in the background so that the prompt will return. To do this, we execute xload like this:

```
[me@linuxbox me]$ xload &
[1] 1223

[me@linuxbox me]$
```

In this case, the prompt returned because the process was put in the background.

Now imagine that you forgot to use the "&" symbol to put the program into the background. There is still hope. You can type Ctrl-z and the process will be suspended. The process still exists, but is idle. To resume the process in the background, type the **bg** command (short for background). Here is an example:

```
[me@linuxbox me]$ xload
[2]+ Stopped xload

[me@linuxbox me]$ bg
[2]+ xload &
```

Listing Your Processes

Now that we have a process in the background, it would be helpful to display a list of the processes we have launched. To do this, we can use either the **jobs** command or the more powerful **ps** command.

```
[me@linuxbox me]$ jobs
[1]+ Running xload &

[me@linuxbox me]$ ps
PID TTY TIME CMD
1211 pts/4 00:00:00 bash
1246 pts/4 00:00:00 xload
1247 pts/4 00:00:00 ps

[me@linuxbox me]$
```

Killing a Process

Suppose that you have a program that becomes unresponsive; how do you get rid of it? You use the **kill** command, of course. Let's try this out on xload. First, you need to identify the process you want to kill. You can use either **jobs** or **ps**, to do this. If you use **jobs** you will get back a job number. With **ps**, you are given a process id (PID). We will do it both ways:

```
[me@linuxbox me]$ xload &
[1] 1292

[me@linuxbox me]$ jobs
[1]+ Running xload &

[me@linuxbox me]$ kill %1

[me@linuxbox me]$ xload &
[2] 1293
[1] Terminated xload

[me@linuxbox me]$ ps
PID TTY TIME CMD
1280 pts/5 00:00:00 bash
1293 pts/5 00:00:00 xload
1294 pts/5 00:00:00 ps

[me@linuxbox me]$ kill 1293
[2]+ Terminated xload

[me@linuxbox me]$
```

A Little More About kill

While the **kill** command is used to "kill" processes, its real purpose is to send *signals* to processes. Most of the time the signal is intended to tell the process to go away, but there is more to it than that. Programs (if they are properly written) listen for signals from the operating system and respond to them, most often to allow some graceful method of terminating. For example, a text editor might listen for any signal that indicates that the user is logging off, or that the computer is shutting down. When it receives this signal, it saves the work in progress before it exits. The **kill** command can send a variety of signals to processes. Typing:

```
kill -l
```

will give you a list of the signals it supports. Most are rather obscure, but several are useful to know:

Signal #	Name	Description
1	SIGHUP	Hang up signal. Programs can listen for this signal and act upon it. This signal is sent to processes running in a terminal when you close the terminal.
2	SIGINT	Interrupt signal. This signal is given to processes to interrupt them.

		Programs can process this signal and act upon it. You can also issue this signal directly by typing Ctrl-c in the terminal window where the program is running.
15	**SIGTERM**	Termination signal. This signal is given to processes to terminate them. Again, programs can process this signal and act upon it. This is the default signal sent by the **kill** command if no signal is specified.
9	**SIGKILL**	Kill signal. This signal causes the immediate termination of the process by the Linux kernel. Programs cannot listen for this signal.

Now let's suppose that you have a program that is hopelessly hung and you want to get rid of it. Here's what you do:

1. Use the **ps** command to get the process id (PID) of the process you want to terminate.
2. Issue a **kill** command for that PID.
3. If the process refuses to terminate (i.e., it is ignoring the signal), send increasingly harsh signals until it does terminate.

```
[me@linuxbox me]$ ps x | grep bad_program
PID TTY STAT TIME COMMAND
2931 pts/5 SN 0:00 bad_program

[me@linuxbox me]$ kill -SIGTERM 2931

[me@linuxbox me]$ kill -SIGKILL 2931
```

In the example above I used the **ps** command with the x option to list all of my processes (even those not launched from the current terminal). In addition, I piped the output of the **ps** command into **grep** to list only list the program I was interested in. Next, I used **kill** to issue a SIGTERM signal to the troublesome program. In actual practice, it is more common to do it in the following way since the default signal sent by **kill** is SIGTERM and **kill** can also use the signal number instead of the signal name:

```
[me@linuxbox me]$ kill 2931
```

Then, if the process does not terminate, force it with the SIGKILL signal:

```
[me@linuxbox me]$ kill -9 2931
```

That's It!

This concludes the "Learning the shell" series of lessons. In the next series, "Writing shell scripts," we will look at how to automate tasks with the shell.

Writing Shell Scripts

Here is Where the Fun Begins

With the thousands of commands available for the command line user, how can you remember them all? The answer is, you don't. The real power of the computer is its ability to do the work for you. To get it to do that, we use the power of the shell to automate things. We write *shell scripts*.

What are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

Scripts unlock the power of your Linux machine. So let's have some fun!

Contents

Writing Your First Script and Getting It to Work

To successfully write a shell script, you have to do three things:

1. Write a script
2. Give the shell permission to execute it
3. Put it somewhere the shell can find it

Writing a Script

A shell script is a file that contains ASCII text. To create a shell script, you use a *text editor*. A text editor is a program, like a word processor, that reads and writes ASCII text files. There are many, many text editors available for your Linux system, both for the command line environment and the GUI environment. Here is a list of some common ones:

Name	Description	Interface
vi, **vim**	The granddaddy of Unix text editors, **vi**, is infamous for its difficult, non-intuitive command structure. On the bright side, **vi** is powerful, lightweight, and fast. Learning **vi** is a Unix rite of passage, since it is universally available on Unix-like systems. On most Linux distributions, an enhanced version of the traditional **vi** editor called **vim** is used.	command line
Emacs	The true giant in the world of text editors is Emacs by Richard Stallman. Emacs contains (or can be made to contain) every feature ever conceived for a text editor. It should be noted that **vi** and Emacs fans fight bitter religious wars over which is better.	command line
nano	**nano** is a free clone of the text editor supplied with the **pine** email program. **nano** is very easy to use but is very short on features. I recommend **nano** for first-time users who need a command line editor.	command line
gedit	**gedit** is the editor supplied with the Gnome desktop environment.	graphical

kwrite	**kwrite** is the "advanced editor" supplied with KDE. It has syntax highlighting, a helpful feature for programmers and script writers.	graphical

Now, fire up your text editor and type in your first script as follows:

```
#!/bin/bash
# My first script

echo "Hello World!"
```

The clever among you will have figured out how to copy and paste the text into your text editor ;-)

If you have ever opened a book on programming, you would immediately recognize this as the traditional "Hello World" program. Save your file with some descriptive name. How about `hello_world`?

The first line of the script is important. This is a special clue, called a *shebang*, given to the shell indicating what program is used to interpret the script. In this case, it is `/bin/bash`. Other scripting languages such as `Perl, awk, tcl, Tk,` and `python` also use this mechanism.

The second line is a *comment*. Everything that appears after a "#" symbol is ignored by **bash**. As your scripts become bigger and more complicated, comments become vital. They are used by programmers to explain what is going on so that others can figure it out. The last line is the **echo** command. This command simply prints its arguments on the display.

Setting Permissions

The next thing we have to do is give the shell permission to execute your script. This is done with the **chmod** command as follows:

```
[me@linuxbox me]$ chmod 755 hello_world
```

The "755" will give you read, write, and execute permission. Everybody else will get only read and execute permission. If you want your script to be private (i.e., only you can read and execute), use "700" instead.

Putting It in Your Path

At this point, your script will run. Try this:

```
[me@linuxbox me]$ ./hello_world
```

You should see "Hello World!" displayed. If you do not, see what directory you really saved your script in, go there and try again.

Before we go any further, I have to stop and talk a while about paths. When you type in the name of a command, the system does not search the entire computer to find where the program is located. That would take a long time. You have noticed that you don't usually have to specify a complete path name to the program you want to run, the shell just seems to know.

Well, you are right. The shell does know. Here's how: the shell maintains a list of directories where executable files (programs) are kept, and only searches the directories in that list. If it does not find the program after searching each directory in the list, it will issue the famous `command not found` error message.

This list of directories is called your *path*. You can view the list of directories with the following command:

```
[me@linuxbox me]$ echo $PATH
```

This will return a colon separated list of directories that will be searched if a specific path name is not given when a command is attempted. In our first attempt to execute your new script, we specified a pathname ("./") to the file.

You can add directories to your path with the following command, where *directory* is the name of the directory you want to add:

```
[me@linuxbox me]$ export PATH=$PATH:directory
```

A better way would be to edit your `.bash_profile` or `.profile` file (depending on your distribution) to include the above command. That way, it would be done automatically every time you log in.

Most Linux distributions encourage a practice in which each user has a specific directory for the programs he/she personally uses. This directory is called `bin` and is a subdirectory of your home directory. If you do not already have one, create it with the following command:

```
[me@linuxbox me]$ mkdir bin
```

Move your script into your new `bin` directory and you're all set. Now you just have to type:

```
[me@linuxbox me]$ hello_world
```

and your script will run. On some distributions, most notably Ubuntu, you will need to open a new terminal session before your newly created `bin` directory will be recognised.

Editing the Scripts You Already Have

Before we get to writing new scripts, I want to point out that you have some scripts of your own already. These scripts were put into your home directory when your account was created, and are used to configure the behavior of your sessions on the computer. You can edit these scripts to change things.

In this lesson, we will look at a couple of these scripts and learn a few important new concepts about the shell.

During your session, the system is holding a number of facts about the world in its memory. This information is called the *environment*. The environment contains such things as your path, your user name, the name of the file where your mail is delivered, and much more. You can see a complete list of what is in your environment with the **`set`** command.

Two types of commands are often contained in the environment. They are *aliases* and *shell functions*.

How is the Environment Established?

When you log on to the system, the bash program starts, and reads a series of configuration scripts called *startup files*. These define the default environment shared by all users. This is followed by more startup files in your home directory that define your personal environment. The exact sequence depends on the type of shell session being started. There are two kinds: a *login shell session* and a *non-login shell session*. A login shell session is one in which we are prompted for our user name and password; when we start a virtual console session, for example. A non-login shell session typically occurs when we launch a terminal session in the GUI.

Login shells read one or more startup files as shown below:

File	Contents
`/etc/profile`	A global configuration script that applies to all users.
`~/.bash_profile`	A user's personal startup file. Can be used to extend or override settings in the global configuration script.
`~/.bash_login`	If `~/.bash_profile` is not found, bash attempts to read this script.
`~/.profile`	If neither `~/.bash_profile` nor `~/.bash_login` is found, bash attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

Non-login shell sessions read the following startup files:

File	Contents
/etc/bash.bashrc	A global configuration script that applies to all users.
~/.bashrc	A user's personal startup file. Can be used to extend or override settings in the global configuration script.

In addition to reading the startup files above, non-login shells also inherit the environment from their parent process, usually a login shell.

Take a look at your system and see which of these startup files you have. Remember— since most of the file names listed above start with a period (meaning that they are hidden), you will need to use the “-a” option when using ls.

The `~/.bashrc` file is probably the most important startup file from the ordinary user’s point of view, since it is almost always read. Non-login shells read it by default and most startup files for login shells are written in such a way as to read the `~/.bashrc` file as well.

If we take a look inside a typical `.bash_profile` (this one taken from a CentOS 4 system), it looks something like this:

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
```

Lines that begin with a “#” are comments and are not read by the shell. These are there for human readability. The first interesting thing occurs on the fourth line, with the following code:

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

This is called an *if compound command*, which we will cover fully in a later lesson, but for now I will translate:

If the file "~/.bashrc" exists, then read the "~/.bashrc" file.

We can see that this bit of code is how a login shell gets the contents of `.bashrc`. The next thing in our startup file does is set set PATH variable to add the `~/bin` directory to the path.

Lastly, we have:

```
export PATH
```

The **export** command tells the shell to make the contents of PATH available to child processes of this shell.

Aliases

An alias is an easy way to create a new command which acts as an abbreviation for a longer one. It has the following syntax:

```
alias name=value
```

where *name* is the name of the new command and *value* is the text to be executed whenever *name* is entered on the command line.

Let's create an alias called "l" and make it an abbreviation for the command "ls -l". Make sure you are in your home directory. Using your favorite text editor, open the file `.bashrc` and add this line to the end of the file:

```
alias l='ls -l'
```

By adding the **alias** command to the file, we have created a new command called "l" which will perform "ls -l". To try out your new command, close your terminal session and start a new one. This will reload the `.bashrc` file. Using this technique, you can create any number of custom commands for yourself. Here is another one for you to try:

```
alias today='date +"%A, %B %-d, %Y"'
```

This alias creates a new command called "today" that will display today's date with nice formatting.

By the way, the **alias** command is just another shell builtin. You can create your aliases directly at the command prompt; however they will only remain in effect during your current shell session. For example:

```
[me@linuxbox me]$ alias l='ls -l'
```

Shell Functions

Aliases are good for very simple commands, but if you want to create something more complex, you should try *shell functions* . Shell functions can be thought of as "scripts within scripts" or little sub-scripts. Let's try one. Open `.bashrc` with your text editor again and replace the alias for "today" with the following:

```
today() {
    echo -n "Today's date is: "
    date +"%A, %B %-d, %Y"
```

```
}
```

Believe it or not, **()** is a shell builtin too, and as with **alias**, you can enter shell functions directly at the command prompt.

```
[me@linuxbox me]$ today() {
> echo -n "Today's date is: "
> date +"%A, %B %-d, %Y"
> }
[me@linuxbox me]$
```

However, again like **alias**, shell functions defined directly on the command line only last as long as the current shell session.

Here Scripts

Beginning with this lesson, we will construct a useful application. This application will produce an HTML document that contains information about your system. I spent a lot of time thinking about how to teach shell programming, and the approach I have chosen is very different from most others that I have seen. Most favor a systematic treatment of shell features, and often presume experience with other programming languages. Although I do not assume that you already know how to program, I realize that many people today know how to write HTML, so our program will produce a web page. As we construct our script, we will discover step by step the tools needed to solve the problem at hand.

Writing an HTML File with a Script

As you may know, a well formed HTML file contains the following content:

```
<html>
<head>
    <title>
    The title of your page
    </title>
</head>

<body>
    Your page content goes here.
</body>
</html>
```

Now, with what we already know, we could write a script to produce the above content:

```
#!/bin/bash

# sysinfo_page - A script to produce an html file

echo "<html>"
echo "<head>"
echo "  <title>"
echo "  The title of your page"
echo "  </title>"
echo "</head>"
echo ""
```

```
echo "<body>"
echo "  Your page content goes here."
echo "</body>"
echo "</html>"
```

This script can be used as follows:

```
[me@linuxbox me]$ sysinfo_page > sysinfo_page.html
```

It has been said that the greatest programmers are also the laziest. They write programs to save themselves work. Likewise, when clever programmers write programs, they try to save themselves typing.

The first improvement to this script will be to replace the repeated use of the `echo` command with a single instance by using quotation more efficiently:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

echo "<html>
 <head>
   <title>
   The title of your page
   </title>
 </head>

 <body>
   Your page content goes here.
 </body>
 </html>"
```

Using quotation, it is possible to embed carriage returns in our text and have the `echo` command's argument span multiple lines.

While this is certainly an improvement, it does have a limitation. Since many types of markup used in html incorporate quotation marks themselves, it makes using a quoted string a little awkward. A quoted string can be used but each embedded quotation mark will need to be escaped with a backslash character.

In order to avoid the additional typing, we need to look for a better way to produce our text. Fortunately, the shell provides one. It's called a *here script*.

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat << _EOF_
<html>
<head>
    <title>
    The title of your page
    </title>
</head>

<body>
    Your page content goes here.
</body>
</html>
_EOF_
```

A here script (also sometimes called a here document) is an additional form of I/O redirection. It provides a way to include content that will be given to the standard input of a command. In the case of the script above, the standard input of the `cat` command was given a stream of text from our script.

A here script is constructed like this:

```
command << token
content to be used as command's standard input
token
```

token can be any string of characters. I use "_EOF_" (EOF is short for "End Of File") because it is traditional, but you can use anything, as long as it does not conflict with a bash reserved word. The token that ends the here script must exactly match the one that starts it, or else the remainder of your script will be interpreted as more standard input to the command.

There is one additional trick that can be used with a here script. Often you will want to indent the content portion of the here script to improve the readability of your script. You can do this if you change the script as follows:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
```

```
        <title>
        The title of your page
        </title>
    </head>

    <body>
        Your page content goes here.
    </body>
    </html>
_EOF_
```

Changing the the "<<" to "<<-" causes bash to ignore the leading tabs (but not spaces) in the here script. The output from the cat command will not contain any of the leading tab characters.

O.k., let's make our page. We will edit our page to get it to say something:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
        <title>
        My System Information
        </title>
    </head>

    <body>
    <h1>My System Information</h1>
    </body>
    </html>
_EOF_
```

In our next lesson, we will make our script produce real information about the system.

Variables

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
        <title>
        My System Information
        </title>
    </head>

    <body>
    <h1>My System Information</h1>
    </body>
    </html>
_EOF_
```

Now that we have our script working, let's improve it. First off, we'll make some changes because we want to be lazy. In the script above, we see that the phrase "My System Information" is repeated. This is wasted typing (and extra work!) so we improve it like this:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="My System Information"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title
        </title>
    </head>

    <body>
    <h1>$title</h1>
```

```
        </body>
        </html>
    _EOF_
```

As you can see, we added a line to the beginning of the script and replaced the two occurrences of the phrase "My System Information" with `$title`.

Variables

What we have done is to introduce a very fundamental idea that appears in almost every programming language, *variables*. Variables are areas of memory that can be used to store information and are referred to by a name. In the case of our script, we created a variable called "title" and placed the phrase "My System Information" into memory. Inside the here script that contains our HTML, we use "$title" to tell the shell to perform *parameter expansion* and replace the name of the variable with the variable's contents.

Whenever the shell sees a word that begins with a "$", it tries to find out what was assigned to the variable and substitutes it.

How to Create a Variable

To create a variable, put a line in your script that contains the name of the variable followed immediately by an equal sign ("="). No spaces are allowed. After the equal sign, assign the information you wish to store.

Where Do Variable Names Come From?

You make them up. That's right; you get to choose the names for your variables. There are a few rules.

1. Names must start with a letter.
2. A name must not contain embedded spaces. Use underscores instead.
3. You cannot use punctuation marks.

How Does This Increase Our Laziness?

The addition of the `title` variable made our life easier in two ways. First, it reduced the amount of typing we had to do. Second and more importantly, it made our script easier to maintain.

As you write more and more scripts (or do any other kind of programming), you will learn that programs are rarely ever finished. They are modified and improved by their creators and others. After all, that's what open source development is all about. Let's say that you wanted to change the phrase "My System Information" to "Linuxbox System Information." In the previous version of the script, you would have had to change this in two locations. In the new version with the `title` variable, you only have to change it in one place. Since our script is so small, this might seem like a trivial matter, but as scripts get larger and more complicated, it becomes very important.

Environment Variables

When you start your shell session, some variables are already set by the startup file we looked at earlier. To see all the variables that are in your environment, use the **printenv** command. One variable in your environment contains the host name for your system. We will add this variable to our script like so:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title $HOSTNAME
        </title>
    </head>

    <body>
    <h1>$title $HOSTNAME</h1>
    </body>
    </html>
_EOF_
```

Now our script will always include the name of the machine on which we are running. Note that, by convention, environment variables names are uppercase.

Command Substitution and Constants

In the previous lesson, we learned how to create variables and perform expansions with them. In this lesson, we will extend this idea to show how we can substitute the results from a command.

When we last left our script, it could create an HTML page that contained a few simple lines of text, including the host name of the machine which we obtained from the environment variable HOSTNAME. Now, we will add a time stamp to the page to indicate when it was last updated, along with the user that did it.

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title $HOSTNAME
        </title>
    </head>

    <body>
    <h1>$title $HOSTNAME</h1>
    <p>Updated on $(date +"%x %r %Z") by $USER</p>
    </body>
    </html>
_EOF_
```

As you can see, we employed another environment variable, `USER`, to get the user name. In addition, we used this strange looking thing:

```
$(date +"%x %r %Z")
```

The characters "$( )" tell the shell, "substitute the results of the enclosed command." In our script, we want the shell to insert the results of the command `date +"%x %r %Z"` which expresses the current date and time. The **date** command has many features and formatting options. To look at them all, try this:

```
[me@linuxbox me]$ date --help | less
```

Be aware that there is an older, alternate syntax for "$(command)" that uses the backtick character " ` ". This older form is compatible with the original Bourne shell (sh). I tend not to use the older form since I am teaching modern `bash` here, not `sh`, and besides, I think backticks are ugly. The bash shell fully supports scripts written for sh, so the following forms are equivalent:

```
$(command)
`command`
```

Assigning a Command's Result to a Variable

You can also assign the results of a command to a variable:

```
right_now=$(date +"%x %r %Z")
```

You can even nest the variables (place one inside another), like this:

```
right_now=$(date +"%x %r %Z")
time_stamp="Updated on $right_now by $USER"
```

Constants

As the name variable suggests, the content of a variable is subject to change. This means that it is expected that during the execution of your script, a variable may have its content modified by something you do.

On the other hand, there may be values that, once set, should never be changed. These are called *constants*. I bring this up because it is a common idea in programming. Most programming languages have special facilities to support values that are not allowed to change. Bash also has these facilities but, to be honest, I never see it used. Instead, if a value is intended to be a constant, it is given an uppercase name to remind the programmer that it should be considered a constant even if it's not being enforced.

Environment variables are usually considered constants since they are rarely changed. Like constants, environment variables are given uppercase names by convention. In the scripts that follow, I will use this convention - uppercase names for constants and lowercase names for variables.

So with everything we know, our program looks like this:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

cat <<- _EOF_
```

```
        <html>
        <head>
            <title>
            $title
            </title>
        </head>

        <body>
        <h1>$title</h1>
        <p>$TIME_STAMP</p>
        </body>
        </html>
    _EOF_
```

Shell Functions

As programs get longer and more complex, they become more difficult to design, code, and maintain. As with any large endeavor, it is often useful to break a single, large task into a series of smaller tasks.

In this lesson, we will begin to break our single monolithic script into a number of separate functions.

To get familiar with this idea, let's consider the description of an everyday task -- going to the market to buy food. Imagine that we were going to describe the task to a man from Mars.

Our first top-level description might look like this:

1. Leave house
2. Drive to market
3. Park car
4. Enter market
5. Purchase food
6. Drive home
7. Park car
8. Enter house

This description covers the overall process of going to the market; however a man from Mars will probably require additional detail. For example, the "Park car" sub task could be described as follows:

1. Find parking space
2. Drive car into space
3. Turn off motor
4. Set parking brake
5. Exit car
6. Lock car

Of course the task "Turn off motor" has a number of steps such as "turn off ignition" and "remove key from ignition switch," and so on.

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called *top-down design*. This technique allows you to break large complex tasks into many small, simple tasks.

As our script continues to grow, we will use top down design to help us plan and code our script.

If we look at our script's top-level tasks, we find the following list:

1. Open page
2. Open head section
3. Write title

4. Close head section
5. Open body section
6. Write title
7. Write time stamp
8. Close body section
9. Close page

All of these tasks are implemented, but we want to add more. Let's insert some additional tasks after task 7:

7. Write time stamp
8. Write system release info
9. Write up-time
10. Write drive space
11. Write home space
12. Close body section
13. Close page

It would be great if there were commands that performed these additional tasks. If there were, we could use command substitution to place them in our script like so:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

While there are no commands that do exactly what we need, we can create them using *shell functions*.

As we learned in lesson 2, shell functions act as "little programs within programs" and allow us to follow top-down design principles. To add the shell functions to our script, we change it so:

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{

}


show_uptime()
{

}


drive_space()
{

}


home_space()
{

}

##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
```

```
        $(system_info)
        $(show_uptime)
        $(drive_space)
        $(home_space)
    </body>
    </html>
_EOF_
```

A couple of important points about functions: First, they must appear before you attempt to use them. Second, the function body (the portions of the function between the { and } characters) must contain at least one valid command. As written, the script will not execute without error, because the function bodies are empty. The simple way to fix this is to place a **return** statement in each function body. After you do this, our script will execute successfully again.

Keep Your Scripts Working

When you are developing a program, it is is often a good practice to add a small amount of code, run the script, add some more code, run the script, and so on. This way, if you introduce a mistake into your code, it will be easier to find and correct.

As you add functions to your script, you can also use a technique called *stubbing* to help watch the logic of your script develop. Stubbing works like this: imagine that we are going to create a function called "system_info" but we haven't figured out all of the details of its code yet. Rather than hold up the development of the script until we are finished with system_info, we just add an **echo** command like this:

```
system_info()
{
    # Temporary function stub
    echo "function system_info"
}
```

This way, our script will still execute successfully, even though we do not yet have a finished system_info function. We will later replace the temporary stubbing code with the complete working version.

The reason we use an **echo** command is so we get some feedback from the script to indicate that the functions are being executed.

Let's go ahead and write stubs for our new functions and keep the script working.

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML file
```

```
    ##### Constants

    TITLE="System Information for $HOSTNAME"
    RIGHT_NOW=$(date +"%x %r %Z")
    TIME_STAMP="Updated on $RIGHT_NOW by $USER"

    ##### Functions

    system_info()
    {
        # Temporary function stub
        echo "function system_info"
    }


    show_uptime()
    {
        # Temporary function stub
        echo "function show_uptime"
    }


    drive_space()
    {
        # Temporary function stub
        echo "function drive_space"
    }


    home_space()
    {
        # Temporary function stub
        echo "function home_space"
    }


    ##### Main

    cat <<- _EOF_
      <html>
      <head>
          <title>$TITLE</title>
      </head>

      <body>
          <h1>$TITLE</h1>
          <p>$TIME_STAMP</p>
          $(system_info)
          $(show_uptime)
          $(drive_space)
          $(home_space)
      </body>
      </html>
```

```
_EOF_
```

Some Real Work

In this lesson, we will develop some of our shell functions and get our script to produce some useful information.

show_uptime

The show_uptime function will display the output of the **uptime** command. The uptime command outputs several interesting facts about the system, including the length of time the system has been "up" (running) since its last re-boot, the number of users and recent system load.

```
[me@linuxbox me]$ uptime
9:15pm up 2 days, 2:32, 2 users, load average: 0.00, 0.00, 0.00
```

To get the output of the uptime command into our HTML page, we will code our shell function like this, replacing our temporary stubbing code with the finished version:

```
show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"
}
```

As you can see, this function outputs a stream of text containing a mixture of HTML tags and command output. When the command substitution takes place in the main body of the our program, the output from our function becomes part of the here script.

drive_space

The drive_space function will use the **df** command to provide a summary of the space used by all of the mounted file systems.

```
[me@linuxbox me]$ df
```

```
Filesystem   1k-blocks      Used Available Use% Mounted on

/dev/hda2       509992    225772    279080  45% /
/dev/hda1        23324      1796     21288   8% /boot
/dev/hda3     15739176   1748176  13832360  12% /home
/dev/hda5      3123888   3039584     52820  99% /usr
```

In terms of structure, the drive_space function is very similar to the show_uptime function:

```
drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"
}
```

home_space

The home_space function will display the amount of space each user is using in his/her home directory. It will display this as a list, sorted in descending order by the amount of space used.

```
home_space()
{
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    echo "Bytes Directory"
    du -s /home/* | sort -nr
    echo "</pre>"
}
```

Note that in order for this function to successfully execute, the script must be run by the superuser, since the **du** command requires superuser privileges to examine the contents of the /home directory.

system_info

We're not ready to finish the system_info function yet. In the meantime, we will improve the stubbing code so it produces valid HTML:

```
system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"
}
```

Flow Control - Part 1

In this lesson, we will look at how to add intelligence to our scripts. So far, our project script has only consisted of a sequence of commands that starts at the first line and continues line by line until it reaches the end. Most programs do much more than this. They make decisions and perform different actions depending on conditions.

The shell provides several commands that we can use to control the flow of execution in our program. In this lesson, we will look at the following:

- **if**
- **test**
- **exit**

if

The first command we will look at is **if**. The **if** command is fairly simple on the surface; it makes a decision based on the *exit status* of a command. The **if** command's syntax looks like this:

The if statement has the following syntax:

```
if commands; then
commands
[elif commands; then
commands...]
[else
commands]
fi
```

where *commands* is a list of commands. This is a little confusing at first glance. But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.

Exit Status

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
```

```
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

In this example, we execute the **ls** command twice. The first time, the command executes successfully. If we display the value of the parameter **$?**, we see that it is zero. We execute the ls command a second time, producing an error and examine the parameter **$?** again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled “Exit Status,” describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The **true** command always executes successfully and the **false** command always executes unsuccessfully:

```
[me@linuxbox~]$ true
[me@linuxbox~]$ echo $?
0
[me@linuxbox~]$ false
[me@linuxbox~]$ echo $?
1
```

We can use these commands to see how the **if** statement works. What the **if** statement really does is evaluate the success or failure of commands:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

The command `echo "It's true."` is executed when the command following **if** executes successfully, and is not executed when the command following **if** does not execute successfully.

test

The **test** command is used most often with the **if** command to perform true/false decisions. The command is unusual in that it has two different syntactic forms:

```
# First form

test expression

# Second form
```

```
[ expression ]
```

The **test** command works simply. If the given expression is true, **test** exits with a status of zero; otherwise it exits with a status of 1.

The neat feature of **test** is the variety of expressions you can create. Here is an example:

```
if [ -f .bash_profile ]; then
    echo "You have a .bash_profile. Things are fine."
else
    echo "Yikes! You have no .bash_profile!"
fi
```

In this example, we use the expression " **-f .bash_profile** ". This expression asks, "Is .bash_profile a file?" If the expression is true, then **test** exits with a zero (indicating true) and the **if** command executes the command(s) following the word **then**. If the expression is false, then **test** exits with a status of one and the **if** command executes the command(s) following the word **else**.

Here is a partial list of the conditions that **test** can evaluate. Since **test** is a shell builtin, use "**help test**" to see a complete list.

Expression	**Description**
-d *file*	True if *file* is a directory.
-e *file*	True if *file* exists.
-f *file*	True if *file* exists and is a regular file.
-L *file*	True if *file* is a symbolic link.
-r *file*	True if *file* is a file readable by you.

-w *file*	True if *file* is a file writable by you.
-x *file*	True if *file* is a file executable by you.
file1 -nt *file2*	True if *file1* is newer than (according to modification time) *file2*
file1 -ot *file2*	True if *file1* is older than *file2*
-z *string*	True if *string* is empty.
-n *string*	True if *string* is not empty.
string1 = *string2*	True if *string1* equals *string2.*
string1 != *string2*	True if *string1* does not equal *string2.*

Before we go on, I want to explain the rest of the example above, since it also reveals more important ideas.

In the first line of the script, we see the **`if`** command followed by the **`test`** command, followed by a semicolon, and finally the word **`then`**. I chose to use the **`[ expression ]`** form of the **`test`** command since most people think it's easier to read. Notice that the spaces required between the "**`[`**" and the beginning of the expression are required. Likewise, the space between the end of the expression and the trailing "**`]`**".

The semicolon is a command separator. Using it allows you to put more than one command on a line. For example:

```
[me@linuxbox me]$ clear; ls
```

will clear the screen and execute the ls command.

I use the semicolon as I did to allow me to put the word **`then`** on the same line as the **`if`** command, because I think it is easier to read that way.

On the second line, there is our old friend **echo**. The only thing of note on this line is the indentation. Again for the benefit of readability, it is traditional to indent all blocks of conditional code; that is, any code that will only be executed if certain conditions are met. The shell does not require this; it is done to make the code easier to read.

In other words, we could write the following and get the same results:

```
# Alternate form

if [ -f .bash_profile ]
then
    echo "You have a .bash_profile. Things are fine."
else
    echo "Yikes! You have no .bash_profile!"
fi

# Another alternate form

if [ -f .bash_profile ]
then echo "You have a .bash_profile. Things are fine."
else echo "Yikes! You have no .bash_profile!"
fi
```

exit

In order to be good script writers, we must set the exit status when our scripts finish. To do this, use the **exit** command. The **exit** command causes the script to terminate immediately and set the exit status to whatever value is given as an argument. For example:

```
exit 0
```

exits your script and sets the exit status to 0 (success), whereas

```
exit 1
```

exits your script and sets the exit status to 1 (failure).

Testing for Root

When we last left our script, we required that it be run with superuser privileges. This is because the **home_space** function needs to examine the size of each user's home directory, and only the superuser is allowed to do that.

But what happens if a regular user runs our script? It produces a lot of ugly error messages. What if we could put something in the script to stop it if a regular user attempts to run it?

The **id** command can tell us who the current user is. When executed with the "-u" option, it prints the numeric user id of the current user.

```
[me@linuxbox me]$ id -u
501
[me@linuxbox me]$ su
Password:
[root@linuxbox me]# id -u
0
```

If the superuser executes **id -u**, the command will output "0." This fact can be the basis of our test:

```
if [ $(id -u) = "0" ]; then
    echo "superuser"
fi
```

In this example, if the output of the command **id -u** is equal to the string "0", then print the string "superuser."

While this code will detect if the user is the superuser, it does not really solve the problem yet. We want to stop the script if the user is not the superuser, so we will code it like so:

```
if [ $(id -u) != "0" ]; then
    echo "You must be the superuser to run this script" >&2
    exit 1
fi
```

With this code, if the output of the **id -u** command is not equal to "0", then the script prints a descriptive error message, exits, and sets the exit status to 1, indicating to the operating system that the script executed unsuccessfully.

Notice the ">&2" at the end of the **echo** command. This is another form of I/O direction. You will often notice this in routines that display error messages. If this redirection were not done, the error message would go to standard output. With this redirection, the message is sent to standard error.

Since we are executing our script and redirecting its standard output to a file, we want the error messages separated from the normal output.

We could put this routine near the beginning of our script so it has a chance to detect a possible error before things get under way, but in order to run this script as an ordinary user, we will use the same idea and modify the `home_space` function to test for proper privileges instead, like so:

```
function home_space
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
            du -s /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space
```

This way, if an ordinary user runs the script, the troublesome code will be passed over, rather than executed and the problem will be solved.

Stay Out of Trouble

Now that our scripts are getting a little more complicated, I want to point out some common mistakes that you might run into. To do this, create the following script called **trouble.bash**. Be sure to enter it exactly as written.

```
#!/bin/bash

number=1

if [ $number = "1" ]; then
    echo "Number equals 1"
else
    echo "Number does not equal 1"
fi
```

When you run this script, it should output the line "Number equals 1" because, well, **number** equals 1. If you don't get the expected output, check your typing; you made a mistake.

Empty Variables

Edit the script to change line 3 from:

```
number=1
```

to:

```
number=
```

and run the script again. This time you should get the following:

```
[me@linuxbox me]$ ./trouble.bash
/trouble.bash: [: =: unary operator expected.
Number does not equal 1
```

As you can see, **bash** displayed an error message when we ran the script. You probably think that by removing the "1" on line 3 it created a syntax error on line 3, but it didn't. Let's look at the error message again:

```
./trouble.bash: [: =: unary operator expected
```

We can see that **./trouble.bash** is reporting the error and the error has to do with "**[**". Remember that "**[**" is an abbreviation for the **test** shell builtin. From this we can determine that the error is occurring on line 5 not line 3.

First, let me say there is nothing wrong with line 3. **number=** is perfectly good syntax. You will sometimes want to set a variable's value to nothing. You can confirm the validity of this by trying it on the command line:

```
[me@linuxbox me]$ number=
[me@linuxbox me]$
```

See, no error message. So what's wrong with line 5? It worked before.

To understand this error, we have to see what the shell sees. Remember that the shell spends a lot of its life expanding text. In line 5, the shell expands the value of **number** where it sees **$number**. In our first try (when **number=1**), the shell substituted 1 for **$number** like so:

```
if [ 1 = "1" ]; then
```

However, when we set number to nothing (**number=**), the shell saw this after the expansion:

```
if [ = "1" ]; then
```

which is an error. It also explains the rest of the error message we received. The "**=**" is a binary operator; that is, it expects two items to operate upon - one on each side. What the shell is trying to

tell us is that there is only one item and there should be a unary operator (like "**!**") that only operates on a single item.

To fix this problem, change line 5 to read:

```
if [ "$number" = "1" ]; then
```

Now when the shell performs the expansion it will see:

```
if [ "" = "1" ]; then
```

which correctly expresses our intent.

This brings up an important thing to remember when you are writing your scripts. Consider what happens if a variable is set to equal nothing.

Missing Quotes

Edit line 6 to remove the trailing quote from the end of the line:

```
    echo "Number equals 1
```

and run the script again. You should get this:

```
[me@linuxbox me]$ ./trouble.bash
./trouble.bash: line 8: unexpected EOF while looking for matching "
./trouble.bash: line 10 syntax error: unexpected end of file
```

Here we have another case of a mistake in one line causing a problem later in the script. What happens is the shell keeps looking for the closing quotation mark to tell it where the end of the string is, but runs into the end of the file before it finds it.

These errors can be a real pain to find in a long script. This is one reason you should test your scripts frequently when you are writing them so there is less new code to test. I also find that text editors with syntax highlighting make these kinds of bugs easier to find.

Isolating Problems

Finding bugs in your programs can sometimes be very difficult and frustrating. Here are a couple of techniques that you will find useful:

Isolate blocks of code by "commenting them out." This trick involves putting comment characters at the beginning of lines of code to stop the shell from reading them. Frequently, you will do this to a block of code to see if a particular problem goes away. By doing this, you can isolate which part of a program is causing (or not causing) a problem.

For example, when we were looking for our missing quotation we could have done this:

```
#!/bin/bash

number=1

if [ $number = "1" ]; then
    echo "Number equals 1
#else
#   echo "Number does not equal 1"
fi
```

By commenting out the **else** clause and running the script, we could show that the problem was not in the **else** clause even though the error message suggested that it was.

Use echo commands to verify your assumptions. As you gain experience tracking down bugs, you will discover that bugs are often not where you first expect to find them. A common problem will be that you will make a false assumption about the performance of your program. You will see a problem develop at a certain point in your program and assume that the problem is there. This is often incorrect, as we have seen. To combat this, you should place **echo** commands in your code while you are debugging, to produce messages that confirm the program is doing what is expected. There are two kinds of messages that you should insert.

The first type simply announces that you have reached a certain point in the program. We saw this in our earlier discussion on stubbing. It is useful to know that program flow is happening the way we expect.

The second type displays the value of a variable (or variables) used in a calculation or test. You will often find that a portion of your program will fail because something that you assumed was correct earlier in your program is, in fact, incorrect and is causing your program to fail later on.

Watching Your Script Run

It is possible to have **bash** show you what it is doing when you run your script. To do this, add a "**-x**" to the first line of your script, like this:

```
#!/bin/bash -x
```

Now, when you run your script, bash will display each line (with expansions performed) as it executes it. This technique is called *tracing*. Here is what it looks like:

```
[me@linuxbox me]$ ./trouble.bash
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number equals 1'
Number equals 1
```

Alternately, you can use the **set** command within your script to turn tracing on and off. Use **set -x** to turn tracing on and **set +x** to turn tracing off. For example.:

```
#!/bin/bash

number=1

set -x
if [ $number = "1" ]; then
    echo "Number equals 1"
else
    echo "Number does not equal 1"
fi
set +x
```

Keyboard Input and Arithmetic

Up to now, our scripts have not been interactive. That is, they did not require any input from the user. In this lesson, we will see how your scripts can ask questions, and get and use responses.

read

To get input from the keyboard, you use the **read** command. The **read** command takes input from the keyboard and assigns it to a variable. Here is an example:

```
#!/bin/bash

echo -n "Enter some text > "
read text
echo "You entered: $text"
```

As you can see, we displayed a prompt on line 3. Note that "-n" given to the **echo** command causes it to keep the cursor on the same line; i.e., it does not output a linefeed at the end of the prompt.

Next, we invoke the **read** command with "text" as its argument. What this does is wait for the user to type something followed by a carriage return (the Enter key) and then assign whatever was typed to the variable text.

Here is the script in action:

```
[me@linuxbox me]$ read_demo.bash
Enter some text > this is some text
You entered: this is some text
```

If you don't give the **read** command the name of a variable to assign its input, it will use the environment variable REPLY.

The **read** command also takes some command line options. The two most interesting ones are -t and -s. The -t option followed by a number of seconds provides an automatic timeout for the **read** command. This means that the **read** command will give up after the specified number of seconds if no response has been received from the user. This option could be used in the case of a script that

must continue (perhaps resorting to a default response) even if the user does not answer the prompts. Here is the `-t` option in action:

```
#!/bin/bash

echo -n "Hurry up and type something! > "
if read -t 3 response; then
    echo "Great, you made it in time!"
else
    echo "Sorry, you are too slow!"
fi
```

The `-s` option causes the user's typing not to be displayed. This is useful when you are asking the user to type in a password or other confidential information.

Arithmetic

Since we are working on a computer, it is natural to expect that it can perform some simple arithmetic. The shell provides features for *integer arithmetic*.

What's an integer? That means whole numbers like 1, 2, 458, -2859. It does not mean fractional numbers like 0.5, .333, or 3.1415. If you must deal with fractional numbers, there is a separate program called **bc** which provides an arbitrary precision calculator language. It can be used in shell scripts, but is beyond the scope of this tutorial.

Let's say you want to use the command line as a primitive calculator. You can do it like this:

```
[me@linuxbox me]$ echo $((2+2))
```

As you can see, when you surround an arithmetic expression with the double parentheses, the shell will perform arithmetic expansion.

Notice that whitespace is ignored:

```
[me@linuxbox me]$ echo $((2+2))
4
[me@linuxbox me]$ echo $(( 2+2 ))
4
[me@linuxbox me]$ echo $(( 2 + 2 ))
4
```

The shell can perform a variety of common (and not so common) arithmetic operations. Here is an example:

```
#!/bin/bash

first_num=0
second_num=0

echo -n "Enter the first number --> "
read first_num
echo -n "Enter the second number -> "
read second_num

echo "first number + second number = $((first_num + second_num))"
echo "first number - second number = $((first_num - second_num))"
echo "first number * second number = $((first_num * second_num))"
echo "first number / second number = $((first_num / second_num))"
echo "first number % second number = $((first_num % second_num))"
echo "first number raised to the"
echo "power of the second number   = $((first_num ** second_num))"
```

Notice how the leading "`$`" is not needed to reference variables inside the arithmetic expression such as "`first_num + second_num`".

Try this program out and watch how it handles division (remember, this is integer division) and how it handles large numbers. Numbers that get too large *overflow* like the odometer in a car when you exceed the number of miles it was designed to count. It starts over but first it goes through all the negative numbers because of how integers are represented in memory. Division by zero (which is mathematically invalid) does cause an error.

I'm sure that you recognize the first four operations as addition, subtraction, multiplication and division, but that the fifth one may be unfamiliar. The "`%`" symbol represents remainder (also known as *modulo*). This operation performs division but instead of returning a quotient like division, it returns the remainder. While this might not seem very useful, it does, in fact, provide great utility when writing programs. For example, when a remainder operation returns zero, it indicates that the first number is an exact multiple of the second. This can be very handy:

```
#!/bin/bash

number=0

echo -n "Enter a number > "
read number

echo "Number is $number"
if [ $((number % 2)) -eq 0 ]; then
    echo "Number is even"
else
```

```
    echo "Number is odd"
fi
```

Or, in this program that formats an arbitrary number of seconds into hours and minutes:

```
#!/bin/bash

seconds=0

echo -n "Enter number of seconds > "
read seconds

hours=$((seconds / 3600))
seconds=$((seconds % 3600))
minutes=$((seconds / 60))
seconds=$((seconds % 60))

echo "$hours hour(s) $minutes minute(s) $seconds second(s)"
```

Flow Control - Part 2

Hold on to your hats. This lesson is going to be a big one!

More Branching

In the previous lesson on flow control we learned about the **if** command and how it is used to alter program flow based on a command's exit status. In programming terms, this type of program flow is called *branching* because it is like traversing a tree. You come to a fork in the tree and the evaluation of a condition determines which branch you take.

There is a second and more complex kind of branching called a *case*. A case is multiple-choice branch. Unlike the simple branch, where you take one of two possible paths, a case supports several possible outcomes based on the evaluation of a value.

You can construct this type of branch with multiple **if** statements. In the example below, we evaluate some input from the user:

```
#!/bin/bash

echo -n "Enter a number between 1 and 3 inclusive > "
read character
if [ "$character" = "1" ]; then
    echo "You entered one."
elif [ "$character" = "2" ]; then
    echo "You entered two."
elif [ "$character" = "3" ]; then
    echo "You entered three."
else
    echo "You did not enter a number between 1 and 3."
fi
```

Not very pretty.

Fortunately, the shell provides a more elegant solution to this problem. It provides a built-in command called **case**, which can be used to construct an equivalent program:

```
#!/bin/bash

echo -n "Enter a number between 1 and 3 inclusive > "
read character
case $character in
```

```
    1 ) echo "You entered one."
        ;;
    2 ) echo "You entered two."
        ;;
    3 ) echo "You entered three."
        ;;
    * ) echo "You did not enter a number between 1 and 3."
esac
```

The **case** command has the following form:

```
case word in
    patterns ) commands ;;
esac
```

case selectively executes statements if word matches a pattern. You can have any number of patterns and statements. Patterns can be literal text or wildcards. You can have multiple patterns separated by the "|" character. Here is a more advanced example to show what I mean:

```
#!/bin/bash

echo -n "Type a digit or a letter > "
read character
case $character in
                                # Check for letters
    [[:lower:]] | [[:upper:]] ) echo "You typed the letter $character"
                                ;;

                                # Check for digits
    [0-9] )                     echo "You typed the digit $character"
                                ;;

                                # Check for anything else
    * )                         echo "You did not type a letter or a digit"
esac
```

Notice the special pattern "*". This pattern will match anything, so it is used to catch cases that did not match previous patterns. Inclusion of this pattern at the end is wise, as it can be used to detect invalid input.

Loops

The final type of program flow control we will discuss is called *looping*. Looping is repeatedly executing a section of your program based on the exit status of a command. The shell provides three commands for

looping: **while**, **until** and **for**. We are going to cover **while** and **until** in this lesson and **for** in a upcoming lesson.

The **while** command causes a block of code to be executed over and over, as long as the exit status of a specified command is true. Here is a simple example of a program that counts from zero to nine:

```
#!/bin/bash

number=0
while [ "$number" -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

On line 3, we create a variable called `number` and initialize its value to 0. Next, we start the **while** loop. As you can see, we have specified a command that tests the value of `number`. In our example, we test to see if `number` has a value less than 10.

Notice the word `do` on line 4 and the word `done` on line 7. These enclose the block of code that will be repeated as long as the exit status remains zero.

In most cases, the block of code that repeats must do something that will eventually change the exit status, otherwise you will have what is called an *endless loop*; that is, a loop that never ends.

In the example, the repeating block of code outputs the value of `number` (the **echo** command on line 5) and increments `number` by one on line 6. Each time the block of code is completed, the test command's exit status is evaluated again. After the tenth iteration of the loop, `number` has been incremented ten times and the **test** command will terminate with a non-zero exit status. At that point, the program flow resumes with the statement following the word `done`. Since `done` is the last line of our example, the program ends.

The **until** command works exactly the same way, except the block of code is repeated as long as the specified command's exit status is false. In the example below, notice how the expression given to the **test** command has been changed from the **while** example to achieve the same result:

```
#!/bin/bash

number=0
until [ "$number" -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Building a Menu

One common way of presenting a user interface for a text based program is by using a *menu*. A menu is a list of choices from which the user can pick.

In the example below, we use our new knowledge of loops and cases to build a simple menu driven application:

```
#!/bin/bash

selection=
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - Display free disk space
    2 - Display free memory

    0 - exit program
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ;;
        2 ) free ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"
    esac
done
```

The purpose of the **until** loop in this program is to re-display the menu each time a selection has been completed. The loop will continue until selection is equal to "0," the "exit" choice. Notice how we defend against entries from the user that are not valid choices.

To make this program better looking when it runs, we can enhance it by adding a function that asks the user to press the Enter key after each selection has been completed, and clears the screen before the menu is displayed again. Here is the enhanced example:

```
#!/bin/bash

press_enter()
{
    echo -en "\nPress Enter to continue"
    read
    clear
}

selection=
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - display free disk space
    2 - display free memory

    0 - exit program
```

```
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ; press_enter ;;
        2 ) free ; press_enter ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"; press_enter
    esac
done
```

When your computer hangs...

We have all had the experience of an application *hanging*. Hanging is when a program suddenly seems to stop and become unresponsive. While you might think that the program has stopped, in most cases, the program is still running but its program logic is stuck in an endless loop.

Imagine this situation: you have an external device attached to your computer, such as a USB disk drive but you forgot to turn it on. You try and use the device but the application hangs instead. When this happens, you could picture the following dialog going on between the application and the interface for the device:

```
Application:    Are you ready?
Interface:  Device not ready.

Application:    Are you ready?
Interface:  Device not ready.

Application:    Are you ready?
Interface:  Device not ready.

Application:    Are you ready?
Interface:  Device not ready.
```

and so on, forever.

Well-written software tries to avoid this situation by instituting a *timeout*. This means that the loop is also counting the number of attempts or calculating the amount of time it has waited for something to happen. If the number of tries or the amount of time allowed is exceeded, the loop exits and the program generates an error and exits.

Positional Parameters

When we last left our script, it looked something like this:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"

}   # end of system_info


show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"

}   # end of show_uptime


drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"

}   # end of drive_space


home_space()
```

```
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space



##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>
  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

We have most things working, but there are several more features I want to add:

1. I want to specify the name of the output file on the command line, as well as set a default output file name if no name is specified.

2. I want to offer an interactive mode that will prompt for a file name and warn the user if the file exists and prompt the user to overwrite it.

3. Naturally, we want to have a help option that will display a usage message.

All of these features involve using command line options and arguments. To handle options on the command line, we use a facility in the shell called *positional parameters*. Positional parameters are a series of special variables (`$0` through `$9`) that contain the contents of the command line.

Let's imagine the following command line:

```
[me@linuxbox me]$ some_program word1 word2 word3
```

If `some_program` were a bash shell script, we could read each item on the command line because the positional parameters contain the following:

- $0 would contain "some_program"
- $1 would contain "word1"
- $2 would contain "word2"
- $3 would contain "word3"

Here is a script you can use to try this out:

```
#!/bin/bash

echo "Positional Parameters"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

Detecting Command Line Arguments

Often, you will want to check to see if you have arguments on which to act. There are a couple of ways to do this. First, you could simply check to see if `$1` contains anything like so:

```
#!/bin/bash

if [ "$1" != "" ]; then
    echo "Positional parameter 1 contains something"
else
    echo "Positional parameter 1 is empty"
fi
```

Second, the shell maintains a variable called `$#` that contains the number of items on the command line in addition to the name of the command (`$0`).

```
#!/bin/bash

if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
else
    echo "Your command line contains no arguments"
```

```
fi
```

Command Line Options

As we discussed before, many programs, particularly ones from the GNU Project, support both short and long command line options. For example, to display a help message for many of these programs, you may use either the "`-h`" option or the longer "`--help`" option. Long option names are typically preceded by a double dash. We will adopt this convention for our scripts.

Here is the code we will use to process our command line:

```
interactive=
filename=~/sysinfo_page.html

while [ "$1" != "" ]; do
    case $1 in
        -f | --file )           shift
                                filename=$1
                                ;;
        -i | --interactive )    interactive=1
                                ;;
        -h | --help )           usage
                                exit
                                ;;
        * )                     usage
                                exit 1
    esac
    shift
done
```

This code is a little tricky, so bear with me as I attempt to explain it.

The first two lines are pretty easy. We set the variable `interactive` to be empty. This will indicate that the interactive mode has not been requested. Then we set the variable `filename` to contain a default file name. If nothing else is specified on the command line, this file name will be used.

After these two variables are set, we have default settings, in case the user does not specify any options.

Next, we construct a **`while`** loop that will cycle through all the items on the command line and process each one with **`case`**. The **`case`** will detect each possible option and process it accordingly.

Now the tricky part. How does that loop work? It relies on the magic of **`shift`**.

`shift` is a shell builtin that operates on the positional parameters. Each time you invoke **`shift`**, it "shifts" all the positional parameters down by one. `$2` becomes `$1`, `$3` becomes `$2`, `$4` becomes

`$3`, and so on. Try this:

```
#!/bin/bash

echo "You start with $# positional parameters"

# Loop until all parameters are used up
while [ "$1" != "" ]; do
    echo "Parameter 1 equals $1"
    echo "You now have $# positional parameters"

    # Shift all the parameters down by one
    shift

done
```

Getting an Option's Argument

Our "`-f`" option requires a valid file name as an argument. We use **`shift`** again to get the next item from the command line and assign it to `filename`. Later we will have to check the content of `filename` to make sure it is valid.

Integrating the Command Line Processor into the Script

We will have to move a few things around and add a usage function to get this new routine integrated into our script. We'll also add some test code to verify that the command line processor is working correctly. Our script now looks like this:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"

}   # end of system_info
```

```
show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"

}   # end of show_uptime


drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"

}   # end of drive_space


home_space()
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space


write_page()
{
    cat <<- _EOF_
    <html>
        <head>
        <title>$TITLE</title>
        </head>
        <body>
        <h1>$TITLE</h1>
        <p>$TIME_STAMP</p>
        $(system_info)
        $(show_uptime)
        $(drive_space)
        $(home_space)
        </body>
    </html>
_EOF_
```

```
}

usage()
{
    echo "usage: sysinfo_page [[[-f file ] [-i]] | [-h]]"
}


##### Main

interactive=
filename=~/sysinfo_page.html

while [ "$1" != "" ]; do
    case $1 in
        -f | --file )           shift
                                filename=$1
                                ;;
        -i | --interactive )    interactive=1
                                ;;
        -h | --help )           usage
                                exit
                                ;;
        * )                     usage
                                exit 1
    esac
    shift
done


# Test code to verify command line processing

if [ "$interactive" = "1" ]; then
        echo "interactive is on"
else
        echo "interactive is off"
fi
echo "output file = $filename"


# Write page (comment out until testing is complete)

# write_page > $filename
```

Adding Interactive Mode

The interactive mode is implemented with the following code:

```
if [ "$interactive" = "1" ]; then

    response=

    echo -n "Enter name of output file [$filename] > "
    read response
    if [ -n "$response" ]; then
        filename=$response
    fi

    if [ -f $filename ]; then
        echo -n "Output file exists. Overwrite? (y/n) > "
        read response
        if [ "$response" != "y" ]; then
            echo "Exiting program."
            exit 1
        fi
    fi
fi
```

First, we check if the interactive mode is on, otherwise we don't have anything to do. Next, we ask the user for the file name. Notice the way the prompt is worded:

```
echo -n "Enter name of output file [$filename] > "
```

We display the current value of `filename` since, the way this routine is coded, if the user just presses the enter key, the default value of `filename` will be used. This is accomplished in the next two lines where the value of `response` is checked. If `response` is not empty, then `filename` is assigned the value of `response`. Otherwise, `filename` is left unchanged, preserving its default value.

After we have the name of the output file, we check if it already exists. If it does, we prompt the user. If the user response is not "y," we give up and exit, otherwise we can proceed.

Flow Control - Part 3

Now that you have learned about positional parameters, it is time to cover the remaining flow control statement, **for**. Like **while** and **until**, **for** is used to construct loops. **for** works like this:

```
for variable in words; do
    commands
done
```

In essence, **for** assigns a word from the list of words to the specified variable, executes the commands, and repeats this over and over until all the words have been used up. Here is an example:

```
#!/bin/bash

for i in word1 word2 word3; do
    echo $i
done
```

In this example, the variable `i` is assigned the string "`word1`", then the statement `echo $i` is executed, then the variable `i` is assigned the string "`word2`", and the statement `echo $i` is executed, and so on, until all the words in the list of words have been assigned.

The interesting thing about **for** is the many ways you can construct the list of words. All kinds of expansions can be used. In the next example, we will construct the list of words using command substitution:

```
#!/bin/bash

count=0
for i in $(cat ~/.bash_profile); do
    count=$((count + 1))
    echo "Word $count ($i) contains $(echo -n $i | wc -c) characters"
done
```

Here we take the file `.bash_profile` and count the number of words in the file and the number of characters in each word.

So what's this got to do with positional parameters? Well, one of the features of **`for`** is that it can use the positional parameters as the list of words:

```
#!/bin/bash

for i in "$@"; do
    echo $i
done
```

The shell variable `"$@"` contains the list of command line arguments. This technique is often used to process a list of files on the command line. Here is a another example:

```
#!/bin/bash

for filename in "$@"; do
    result=
    if [ -f "$filename" ]; then
        result="$filename is a regular file"
    else
        if [ -d "$filename" ]; then
            result="$filename is a directory"
        fi
    fi
    if [ -w "$filename" ]; then
        result="$result and it is writable"
    else
        result="$result and it is not writable"
    fi
    echo "$result"
done
```

Try this script. Give it a list of files or a wildcard like "`*`" to see it work.

Here is another example script. This one compares the files in two directories and lists which files in the first directory are missing from the second.

```
#!/bin/bash

# cmp_dir - program to compare two directories

# Check for required arguments
if [ $# -ne 2 ]; then
    echo "usage: $0 directory_1 directory_2" 1>&2
```

```
    exit 1
fi

# Make sure both arguments are directories
if [ ! -d $1 ]; then
    echo "$1 is not a directory!" 1>&2
    exit 1
fi

if [ ! -d $2 ]; then
    echo "$2 is not a directory!" 1>&2
    exit 1
fi

# Process each file in directory_1, comparing it to directory_2
missing=0
for filename in $1/*; do
    fn=$(basename "$filename")
    if [ -f "$filename" ]; then
        if [ ! -f "$2/$fn" ]; then
            echo "$fn is missing from $2"
            missing=$((missing + 1))
        fi
    fi
done
echo "$missing files missing"
```

Now on to the real work. We are going to improve the `home_space` function in our script to output more information. You will recall that our previous version looked like this:

```
home_space()
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    echo "Bytes Directory"
        du -s /home/* | sort -nr
    echo "</pre>"
    fi

}   # end of home_space
```

Here is the new version:

```
home_space()
{
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    format="%8s%10s%10s   %-s\n"
    printf "$format" "Dirs" "Files" "Blocks" "Directory"
    printf "$format" "----" "-----" "------" "---------"
    if [ $(id -u) = "0" ]; then
        dir_list="/home/*"
    else
        dir_list=$HOME
    fi
    for home_dir in $dir_list; do
        total_dirs=$(find $home_dir -type d | wc -l)
        total_files=$(find $home_dir -type f | wc -l)
        total_blocks=$(du -s $home_dir)
        printf "$format" $total_dirs $total_files $total_blocks
    done
    echo "</pre>"

}   # end of home_space
```

This improved version introduces a new command **printf**, which is used to produce formatted output according to the contents of a *format string*. **printf** comes from the C programming language and has been implemented in many other programming languages including C++, Perl, awk, java, PHP, and of course, bash. You can read more about **printf** format strings at:

- GNU Awk User's Guide - Control Letters
- GNU Awk User's Guide - Format Modifiers

We also introduce the **find** command. **find** is used to search for files or directories that meet specific criteria. In the `home_space` function, we use **find** to list the directories and regular files in each home directory. Using the **wc** command, we count the number of files and directories found.

The really interesting thing about `home_space` is how we deal with the problem of superuser access. You will notice that we test for the superuser with **id** and, according to the outcome of the test, we assign different strings to the variable `dir_list`, which becomes the list of words for the **for** loop that follows. This way, if an ordinary user runs the script, only his/her home directory will be listed.

Another function that can use a **for** loop is our unfinished `system_info` function. We can build it like this:

```
system_info()
{
    # Find any release files in /etc

    if ls /etc/*release 1>/dev/null 2>&1; then
        echo "<h2>System release info</h2>"
```

```
            echo "<pre>"
            for i in /etc/*release; do

                # Since we can't be sure of the
                # length of the file, only
                # display the first line.

                head -n 1 $i
            done
            uname -orp
            echo "</pre>"
        fi

    }   # end of system_info
```

In this function, we first determine if there are any release files to process. The release files contain the name of the vendor and the version of the distribution. They are located in the `/etc` directory. To detect them, we perform an **`ls`** command and throw away all of its output. We are only interested in the exit status. It will be true if any files are found.

Next, we output the HTML for this section of the page, since we now know that there are release files to process. To process the files, we start a **`for`** loop to act on each one. Inside the loop, we use the **`head`** command to return the first line of each file.

Finally, we use the **`uname`** command with the "o", "r", and "p" options to obtain some additional information from the system.

Errors and Signals and Traps (Oh My!) - Part 1

In this lesson, we're going to look at handling errors during the execution of your scripts.

The difference between a good program and a poor one is often measured in terms of the program's *robustness*. That is, the program's ability to handle situations in which something goes wrong.

Exit Status

As you recall from previous lessons, every well-written program returns an exit status when it finishes. If a program finishes successfully, the exit status will be zero. If the exit status is anything other than zero, then the program failed in some way.

It is very important to check the exit status of programs you call in your scripts. It is also important that your scripts return a meaningful exit status when they finish. I once had a Unix system administrator who wrote a script for a production system containing the following 2 lines of code:

```
# Example of a really bad idea

cd $some_directory
rm *
```

Why is this such a bad way of doing it? It's not, if nothing goes wrong. The two lines change the working directory to the name contained in `$some_directory` and delete the files in that directory. That's the intended behavior. But what happens if the directory named in `$some_directory` doesn't exist? In that case, the `cd` command will fail and the script executes the `rm` command on the current working directory. Not the intended behavior!

By the way, my hapless system administrator's script suffered this very failure and it destroyed a large portion of an important production system. Don't let this happen to you!

The problem with the script was that it did not check the exit status of the `cd` command before proceeding with the `rm` command.

Checking the Exit Status

There are several ways you can get and respond to the exit status of a program. First, you can examine the contents of the `$?` environment variable. `$?` will contain the exit status of the last command executed. You can see this work with the following:

```
[me] $ true; echo $?
0
[me] $ false; echo $?
1
```

The `true` and `false` commands are programs that do nothing except return an exit status of zero and one, respectively. Using them, we can see how the `$?` environment variable contains the exit status of the previous program.

So to check the exit status, we could write the script this way:

```
# Check the exit status

cd $some_directory
if [ "$?" = "0" ]; then
        rm *
else
        echo "Cannot change directory!" 1>&2
        exit 1
fi
```

In this version, we examine the exit status of the `cd` command and if it's not zero, we print an error message on standard error and terminate the script with an exit status of 1.

While this is a working solution to the problem, there are more clever methods that will save us some typing. The next approach we can try is to use the `if` statement directly, since it evaluates the exit status of commands it is given.

Using `if`, we could write it this way:

```
# A better way

if cd $some_directory; then
        rm *
else
        echo "Could not change directory! Aborting." 1>&2
        exit 1
fi
```

Here we check to see if the `cd` command is successful. Only then does `rm` get executed; otherwise an error message is output and the program exits with a code of 1, indicating that an error has occurred.

An Error Exit Function

Since we will be checking for errors often in our programs, it makes sense to write a function that will display error messages. This will save more typing and promote laziness.

```
# An error exit function

error_exit()
{
        echo "$1" 1>&2
        exit 1
}

# Using error_exit

if cd $some_directory; then
        rm *
else
        error_exit "Cannot change directory!  Aborting."
fi
```

AND and OR Lists

Finally, we can further simplify our script by using the AND and OR control operators. To explain how they work, I will quote from the **bash** man page:

"The control operators && and || denote AND lists and OR lists, respectively. An AND list has the form

```
command1 && command2
```

`command2` is executed if, *and only if*, `command1` returns an exit status of zero.

An OR list has the form

```
command1 || command2
```

`command2` is executed if, and only if, `command1` returns a non-zero exit status. The exit status of AND and OR lists is the exit status of the last command executed in the list."

Again, we can use the `true` and `false` commands to see this work:

```
[me] $ true || echo "echo executed"
[me] $ false || echo "echo executed"
echo executed
[me] $ true && echo "echo executed"
echo executed
[me] $ false && echo "echo executed"
[me] $
```

Using this technique, we can write an even simpler version:

```
# Simplest of all

cd $some_directory || error_exit "Cannot change directory! Aborting"
rm *
```

If an exit is not required in case of error, then you can even do this:

```
# Another way to do it if exiting is not desired

cd $some_directory && rm *
```

I want to point out that even with the defense against errors we have introduced in our example for the use of `cd`, this code is still vulnerable to a common programming error, namely, what happens if the name of the variable containing the name of the directory is misspelled? In that case, the shell will interpret the variable as empty and the `cd` succeed, but it will change directories to the user's home directory, so beware!

Improving the Error Exit Function

There are a number of improvements that we can make to the `error_exit` function. I like to include the name of the program in the error message to make clear where the error is coming from. This becomes more important as your programs get more complex and you start having scripts launching other scripts, etc. Also, note the inclusion of the `LINENO` environment variable which will help you identify the exact line within your script where the error occurred.

```
#!/bin/bash

# A slicker error handling routine

# I put a variable in my scripts named PROGNAME which
# holds the name of the program being run.  You can get this
# value from the first item on the command line ($0).

PROGNAME=$(basename $0)

error_exit()
{

#       ------------------------------------------------------------
#       Function for exit due to fatal program error
#               Accepts 1 argument:
#                       string containing descriptive error message
#       ------------------------------------------------------------
```

```
        echo "${PROGNAME}: ${1:-"Unknown Error"}" 1>&2
        exit 1
}

# Example call of the error_exit function.  Note the inclusion
# of the LINENO environment variable.  It contains the current
# line number.

echo "Example of error with line number and message"
error_exit "$LINENO: An error has occurred."
```

The use of the curly braces within the `error_exit` function is an example of *parameter expansion*. You can surround a variable name with curly braces (as with `${PROGNAME}`) if you need to be sure it is separated from surrounding text. Some people just put them around every variable out of habit. That usage is simply a style thing. The second use, `${1:-"Unknown Error"}` means that if parameter 1 (`$1`) is undefined, substitute the string "Unknown Error" in its place. Using parameter expansion, it is possible to perform a number of useful string manipulations. You can read more about parameter expansion in the **bash** man page under the topic "EXPANSIONS".

Errors and Signals and Traps (Oh, My!) - Part 2

Errors are not the only way that a script can terminate unexpectedly. You also have to be concerned with signals. Consider the following program:

```
#!/bin/bash

echo "this script will endlessly loop until you stop it"
while true; do
        : # Do nothing
done
```

After you launch this script it will appear to hang. Actually, like most programs that appear to hang, it is really stuck inside a loop. In this case, it is waiting for the `true` command to return a non-zero exit status, which it never does. Once started, the script will continue until bash receives a signal that will stop it. You can send such a signal by typing Ctrl-c which is the signal called SIGINT (short for SIGnal INTerrupt).

Cleaning Up After Yourself

Okay, so a signal can come along and make your script terminate. Why does it matter? Well, in many cases it doesn't matter and you can ignore signals, but in some cases it will matter.

Let's take a look at another script:

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

pr $1 > $TEMP_FILE

echo -n "Print file? [y/n]: "
read
if [ "$REPLY" = "y" ]; then
        lpr $TEMP_FILE
fi
```

This script processes a text file specified on the command line with the **pr** command and stores the result in a temporary file. Next, it asks the user if they want to print the file. If the user types "y", then the temporary file is passed to the **lpr** program for printing (you may substitute `less` for `lpr` if you don't actually have a printer attached to your system.)

Now, I admit this script has a lot of design problems. While it needs a file name passed on the command line, it doesn't check that it got one, and it doesn't check that the file actually exists. But the problem I want to focus on here is the fact that when the script terminates, it leaves behind the temporary file.

Good practice would dictate that we delete the temporary file `$TEMP_FILE` when the script terminates. This is easily accomplished by adding the following to the end of the script:

```
rm $TEMP_FILE
```

This would seem to solve the problem, but what happens if the user types ctrl-c when the "Print file? [y/n]:" prompt appears? The script will terminate at the `read` command and the `rm` command is never executed. Clearly, we need a way to respond to signals such as SIGINT when the Ctrl-c key is typed.

Fortunately, bash provides a method to perform commands if and when signals are received.

trap

The `trap` command allows you to execute a command when a signal is received by your script. It works like this:

```
trap arg signals
```

"signals" is a list of signals to intercept and "arg" is a command to execute when one of the signals is received. For our printing script, we might handle the signal problem this way:

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM

pr $1 > $TEMP_FILE

echo -n "Print file? [y/n]: "
read
if [ "$REPLY" = "y" ]; then
```

```
        lpr $TEMP_FILE
fi
rm $TEMP_FILE
```

Here we have added a `trap` command that will execute "`rm $TEMP_FILE`" if any of the listed signals is received. The three signals listed are the most common ones that you will encounter, but there are many more that can be specified. For a complete list, type "`trap -l`". In addition to listing the signals by name, you may alternately specify them by number.

Signal 9 from Outer Space

There is one signal that you cannot trap: SIGKILL or signal 9. The kernel immediately terminates any process sent this signal and no signal handling is performed. Since it will always terminate a program that is stuck, hung, or otherwise screwed up, it is tempting to think that it's the easy way out when you have to get something to stop and go away. Often you will see references to the following command which sends the SIGKILL signal:

```
kill -9
```

However, despite its apparent ease, you must remember that when you send this signal, no processing is done by the application. Often this is OK, but with many programs it's not. In particular, many complex programs (and some not-so-complex) create *lock files* to prevent multiple copies of the program from running at the same time. When a program that uses a lock file is sent a SIGKILL, it doesn't get the chance to remove the lock file when it terminates. The presence of the lock file will prevent the program from restarting until the lock file is manually removed.

Be warned. Use SIGKILL as a last resort.

A clean_up Function

While the trap command has solved the problem, we can see that it has some limitations. Most importantly, it will only accept a single string containing the command to be performed when the signal is received. You could get clever and use ";" and put multiple commands in the string to get more complex behavior, but frankly, it's ugly. A better way would be to create a function that is called when you want to perform any actions at the end of your script. In my scripts, I call this function `clean_up`.

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

clean_up() {
```

```
        # Perform program exit housekeeping
        rm $TEMP_FILE
        exit
}

trap clean_up SIGHUP SIGINT SIGTERM

pr $1 > $TEMP_FILE

echo -n "Print file? [y/n]: "
read
if [ "$REPLY" = "y" ]; then
        lpr $TEMP_FILE
fi
clean_up
```

The use of a clean up function is a good idea for your error handling routines too. After all, when your program terminates (for whatever reason), you should clean up after yourself. Here is finished version of our program with improved error and signal handling:

```
#!/bin/bash

# Program to print a text file with headers and footers

# Usage: printfile file

# Create a temporary file name that gives preference
# to the user's local tmp directory and has a name
# that is resistant to "temp race attacks"

if [ -d "~/tmp" ]; then
        TEMP_DIR=~/tmp
else
        TEMP_DIR=/tmp
fi
TEMP_FILE=$TEMP_DIR/printfile.$$.$RANDOM
PROGNAME=$(basename $0)

usage() {

        # Display usage message on standard error
        echo "Usage: $PROGNAME file" 1>&2
}

clean_up() {

        # Perform program exit housekeeping
        # Optionally accepts an exit status
        rm -f $TEMP_FILE
        exit $1
```

```
}

error_exit() {

        # Display error message and exit
        echo "${PROGNAME}: ${1:-"Unknown Error"}" 1>&2
        clean_up 1
}

trap clean_up SIGHUP SIGINT SIGTERM

if [ $# != "1" ]; then
        usage
        error_exit "one file to print must be specified"
fi
if [ ! -f "$1" ]; then
        error_exit "file $1 cannot be read"
fi

pr $1 > $TEMP_FILE || error_exit "cannot format file"

echo -n "Print file? [y/n]: "
read
if [ "$REPLY" = "y" ]; then
        lpr $TEMP_FILE || error_exit "cannot print file"
fi
clean_up
```

Creating Safe Temporary Files

In the program above, there a number of steps taken to help secure the temporary file used by this script. It is a Unix tradition to use a directory called `/tmp` to place temporary files used by programs. Everyone may write files into this directory. This naturally leads to some security concerns. If possible, avoid writing files in the `/tmp` directory. The preferred technique is to write them in a local directory such as `~/tmp` (a tmp subdirectory in the user's home directory.) If you must write files in `/tmp`, you must take steps to make sure the file names are not predictable. Predictable file names allow an attacker to create symbolic links to other files that the attacker wants you to overwrite.

A good file name will help you figure out what wrote the file, but will not be entirely predictable. In the script above, the following line of code created the temporary file `$TEMP_FILE`:

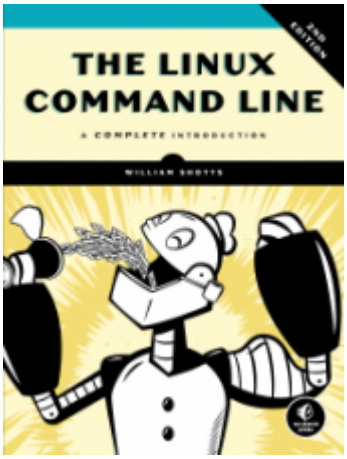
```
TEMP_FILE=$TEMP_DIR/printfile.$$.$RANDOM
```

The `$TEMP_DIR` variable contains either `/tmp` or `~/tmp` depending on the availability of the directory. It is common practice to embed the name of the program into the file name. We have done that with the string "printfile". Next, we use the `$$` shell variable to embed the process id (pid) of the program. This further helps identify what process is responsible for the file. Surprisingly, the process id alone is not unpredictable enough to make the file safe, so we add the `$RANDOM` shell variable to

append a random number to the file name. With this technique, we create a file name that is both easily identifiable and unpredictable.

There You Have It

This concludes the LinuxCommand.org tutorials. I sincerely hope you found them both useful and enjoyable. If you did, continue your command line adventure by downloading my book.



The Linux Command Line

A Book By William Shotts

Fifth Internet Edition Available Now!

Designed for the new command line user, this 555-page volume covers the same material as LinuxCommand.org but in much greater detail. In addition to the basics of command line use and shell scripting, *The Linux Command Line* includes chapters on many common programs used on the command line, as well as more advanced topics.

- **Released under a Creative Commons license , this book is available for free download in PDF format.** Download it here .
- **The book is available in printed form, published by No Starch Press .** Copies may be purchased wherever fine books are sold. No Starch Press also offers electronic formats for popular e-readers.

  In addition to English, the printed book is also available in the following languages:

  - Korean
  - Serbian
  - Chinese (Traditional)
  - Chinese (Simplified)
  - Russian

- **Find *The Linux Command Line* at your local library .**
- **Want to translate the Creative Commons version of *The Linux Command Line*?** Please see the Community Translation Policy for details.

  The following community translations are now available:

  - **Arabic** Translation by Abd Allatif Eymsh
  - **Spanish (in progress).** Translation by Adriano Sánchez
  - **Portuguese (in progress).** Translation by Matheus Sales
  - **Italian (in progress).** Translation by Francesco Cargiuli
  - **Romanian (in progress).** Translation by Vlad Paval

What Readers Are Saying About *The Linux Command Line*!

"I have been using Linux for a little over a year. Read as much as I could including Rute, and many others. I have learned more in the first 93 pages of your book than any others!!!!"

"What a great help your book has been for me! I was looking for a book like yours, but I never was able to find one that really walks the reader through the Linux command line in a general sort of way. Do you know of any others? If I recall, there are a lot of books dedicated to shell scripting, but none that are geared to regular command-line use."

"Fantastic book!! I have recently switched from the legacy OS to Linux and I have been trying to find a distro that mimics this legacy OS in fear of using the terminal. I ran across this book and opened the terminal for the first time. This book teaches you everything you need to know about the shell and does it with ease. It starts by giving you a solid foundation and builds from there. Its simplicity and informative structure is ideal for all new beginners switching to Linux. I am now always on the terminal even for the simplest things. Not only does this build love, but I also get to practice my shell scripting. Great book!!"

"I've been enjoying reading the book and have learned a lot from every chapter. Your writing is very clear, and I've enjoyed following along with your examples. I've skimmed a couple of Linux books before, but never had enough time to really get into them. From those experiences however, I think your book is much clearer and approachable for beginners."

"WOW! What a great book. It will clearly provide a stepping stone for many people who want to wean themselves off of their GUI habit, or perhaps just "Make the difficult possible."

"I think one of the greatest assets of the book that most others about Linux don't have is the conversational tone. I feel like in almost every chapter, there are parts where you lead the reader to ask questions of the material that aren't necessarily covered."

Read more reviews at **No Starch Press** and **Amazon** .
