Part Workbook 2. Filesystem Basics
# Table of Contents

1. Filesystem Navigation ................................................................................. 5
   Discussion ......................................................................................................... 5
   The Filesystem Inverted Tree ................................................................. 5
   Exploring the Directory Tree using Nautilus ........................................... 6
   The Current Working Directory (cwd) ..................................................... 9
   Where Am I? The pwd command ............................................................... 9
   Moving Around - The cd command ......................................................... 10
   Absolute and Relative References ......................................................... 11

Examples ........................................................................................................... 12
   Example 1. Discovering your home directory ......................................... 12
   Example 2. Climbing up and down the tree. ............................................ 12

Online Exercises .............................................................................................. 13
   Online Exercise 1. Exploring the filesystem .......................................... 13
      Specification ......................................................................................... 13
      Deliverables ......................................................................................... 13
      Possible Solution .................................................................................. 13

Questions ........................................................................................................... 13

Navigating the Filesystem ................................................................................ 13

2. Important Directories .................................................................................. 15
   Discussion ......................................................................................................... 15
   Standard Linux Directory Scheme ....................................................... 15
   The User's Home Directory .................................................................. 15
   The /tmp Temporary Directory ........................................................... 16
   The /etc Configuration Directory ........................................................... 16
   The /bin and /usr/bin Command Directories ......................................... 16
   The /bin and /usr/sbin Command Directories ....................................... 17
   The /var “Variable” Directory ............................................................... 17
   root vs. /root vs. / (the filesystem root) .................................................. 17

Examples ........................................................................................................... 17
   Example 1. The /tmp Directory ............................................................ 17
   Example 2. Looking for commands in /bin and /usr/bin ..................... 18

Online Exercises .............................................................................................. 18
   Online Exercise 1. Exploring Important Directories ............................... 18
      Setup ................................................................................................. 19
      Specification ....................................................................................... 19
      Deliverables ......................................................................................... 19
      Possible Solution .................................................................................. 19

Questions ........................................................................................................... 19

3. Managing Files ............................................................................................ 22
   Discussion ......................................................................................................... 22
   Redirection ................................................................................................. 22
   Copy files with cp .................................................................................... 23
      Examples: ........................................................................................... 23
   Move/rename files with mv ................................................................. 24
      Examples: ........................................................................................... 24
   Remove (delete) files with rm ............................................................. 25
      Examples: ........................................................................................... 25
   Clobbering files (oops!) ........................................................................... 26

Examples ........................................................................................................... 27
   Example 1. Moving/Archiving a Simple Website .................................. 27
   Example 2. Moving a Simple Website ................................................... 28
Example 1. Identifying Documents with file ................................................. 60
Example 2. Determining the Correct File Type with file ................................................. 60
Example 3. Skimming Files with head ............................................................. 61
Example 4. Combining Multiple Files into a Single File with head ................................................. 61
Example 5. Monitoring Multiple Files with tail ...................................................... 62
Online Exercises ............................................................................................ 63
  Online Exercise 1. Browsing Text Files .............................................................. 63
    Specification ........................................................................................................... 63
    Deliverables ........................................................................................................... 64
    Cleaning Up ........................................................................................................... 64
Questions .................................................................................................................. 64
7. Editing Files ........................................................................................................ 67
  Discussion ................................................................................................................ 67
    Text Editor vs. Word Processor .............................................................................. 67
    Text Editors in Linux ............................................................................................ 67
      nano ...................................................................................................................... 67
      gedit ...................................................................................................................... 68
Online Exercises ................................................................................................. 69
  Online Exercise 1. Text Editors .............................................................................. 69
    Specification ........................................................................................................... 69
    Deliverables ........................................................................................................... 70
Chapter 1. Filesystem Navigation

**Key Concepts**

- The Linux filesystem is an "inverted tree" of directories and files, with a root directory called "/".
- Every process has a current working directory, often called it's "cwd".
- The `pwd` command reports the `bash` shell's cwd.
- The `cd` command changes the `bash` shell's cwd.
- Files can be referenced through either absolute or relative references.

**Discussion**

**The Filesystem Inverted Tree**

The concept of a filesystem organized into directories is common to many computer systems. An individual file is given a name (*filename*), and filenames get organized into a *directory* (called a *folder* in some operating systems). But directories are themselves a kind of file, so they, too, can get collected into other directories. This process can continue, level after level, to create a highly structured environment.

For example, a user alice might have several songs stored in files with names like `song1.midi` and `song2.midi`. These songs might be collected into a directory named `songs`. User alice might also have some photographs stored in the files `picture1.png` and `picture2.png`. These she might have placed into a directory called `photos`. These two directories, `songs` and `photos` might be organized into a directory called `media`. This new directory might be only one of several in a `website` directory.

One way to visualize this is as a branching tree structure. The `website` directory may contain the subdirectories `html`, `forms` and `media`. Together with the directories and files named above, this arrangement might be diagrammed as shown in Figure 1-1.

**Figure 1.1. Sample Directory Tree**

```
  -- html
  |
website -- -- forms
  |
  |
  |
  |
  -- songs --
  |
  |
-- song1.midi

|

|

|

|

-- song2.midi

|

|

-- media --

|

|

-- picture1.png

|

|

-- photos--

|

-- picture2.png
```

It is the branching appearance of this diagram that gives the notion of a *directory tree* its name. This diagram is most often drawn with the branches hanging down, and with the root of the tree (the directory `website` in this case) at the top, and so is referred to as an *inverted tree* directory structure.
Linux uses a single directory tree for its entire collection of files and directories, all branching from a single root named "/" (read as “slash”) and called the root directory. This directory tree can be visualized, in part, as shown in Figure 1-2.

Figure 1.2. Directory Tree Starting at /

```
-- bin...
/ -- -- etc...
  |    -- mystuff...
  |         -- alice --
  |          -- website...
  -- home --
  |      -- docs...
  |          -- elvis --
  |                   -- graphics...
```

The ellipses in the diagram are meant to suggest that there are many files and directories not shown here. You can see, for example, where the website directory tree drawn earlier could be attached to extend the diagram. The important feature of the diagram at this point is the tree structure growing out of /, not the specific contents.

When naming a file or directory, start from the root of the Linux filesystem tree and list all of the directory branches out to the file or directory you want, separating each part with a slash (/). This is known as the item's fully qualified name, or FQN for short. For example, the FQN of the website directory above would be /home/alice/website. User alice's song1.midi file would be identified as /home/alice/website/songs/song1.midi. The string of directory names leading up to the filename is the path to the file. This type of naming helps guarantee each file and directory has a unique fully qualified name. In actual practice, special shortcuts and default assumptions are used to simplify the typing of filenames.

Exploring the Directory Tree using Nautilus

In the Red Hat Enterprise Linux X graphical environment, users can use Nautilus as a sophisticated tool for navigating the filesystem. In the Red Hat Enterprise Linux desktop, a Nautilus window can be opened by (left) double clicking on the home icon found in the upper left-hand corner.

Figure 1.3. Nautilus Window
As an alternative, you can use the Nautilus browser (which might be more familiar to people familiar with previous releases of Red Hat Enterprise Linux) by selecting the Applications : System Tools : File Browser menu item.

**Figure 1.4. Nautilus's Browser View**

![Nautilus's Browser View](image)

If you would prefer to use the browser by default, you may choose "Always open in Browser Windows" from the Behavior page of Nautilus's Preferences, found under the Edit menu.

**Figure 1.5. Configuring Nautilus to use Browser View by Default**

![Configuring Nautilus to use Browser View by Default](image)

By default, the Nautilus browser's sidepane includes a "hotlist" of commonly used directories. The entire directory tree can be browsed, however, by choosing "tree" from within the side pane's popup menu.
Figure 1.6. Choosing the Nautilus tree sidepane

The Nautilus sidepane should now show list of directories (folders), arranged as two trees, one rooted at the root directory (labeled “File System”), and the other rooted at your home directory. A directory can be expanded by (left) clicking on the triangle icon next to the folder icon. When the folder icon, or folder name, is clicked, the contents of the folder will be displayed in Nautilus's main pane.

Figure 1.7. Exploring the directory tree with Nautilus

Spend some time exploring the filesystem with Nautilus, and viewing the contents of various directories. Try viewing the contents of the file /etc/sysconfig/network-scripts/ifcfg-lo, by first selecting the directory /etc/sysconfig/network-scripts in the sidepane, and then (left) double clicking on the ifcfg-lo icon in the main pane.

Note that when you select a particular directory for viewing, the Location is represented as buttons above the main pane. Typing CTRL+L converts the buttons into a text entry, from which you can navigate by just typing a directory path. Notice the use of TAB completion, as well.
Now that you have spent some time exploring the concept of a directory tree, and how absolute references to directories and files are constructed, we will spend some time talking about Linux filesystem related concepts.

The Current Working Directory (cwd)

Each Linux process (i.e., program, command) is assigned a default directory when it runs. This default directory is used to complete the FQN for any file whose name is not specified as an FQN to start with. This has two effects. First, it allows the process to refer to files more simply, without using tiresomely long names. Second, it allows the process to operate more flexibly since its actions depend in part on the default directory. Change the default directory, and the action of the process is changed. This default directory is referred to as the current working directory, or cwd, for the process.

One common approach to take with directories is to think of a directory as a place rather than a thing. From this point of view, then, the cwd for a process can be thought of as the process’s current location. If a process changes its cwd, then it has moved from one directory to another. This view of a directory as a place is so common in the Linux world that the vocabulary of place is quite pervasive in the language. For instance, users talk about navigating the filesystem as they move from directory to directory.

A process is first assigned a cwd by the parent process which starts it. However, a process is not locked into a particular cwd the whole time it runs. When needed, a process can change its cwd to another directory before continuing its work.

Where Am I? The pwd command

One of the processes found running on a Linux system is the command shell. If you login to the system through a virtual terminal, or start a terminal program in X, you see a command prompt where you can enter commands for the system to act on. This command prompt is produced by the command shell, the process which is responsible for reading and interpreting your commands. The default command shell for Red Hat Enterprise Linux systems is the bash (Bourne-again shell) command shell.

Like any other process, the bash shell keeps track of its cwd while it runs. This current working directory affects the commands you type at the command prompt, and becomes their cwd as they run. In essence,
then, the cwd of the **bash** shell is *your* cwd, and can be thought of as where you are within the system. Obviously, keeping track of this cwd is important. Fortunately, there is help. First, the command prompt itself shows the last directory of the path to the cwd. For example, user alice, while working in her website directory, might see a command prompt like this:

```
[alice@station website]$  
```

This prompt reminds her that she is logged in with username "alice" on the computer "station," and is in the directory website. But there can be other website directories elsewhere on the system. The complete (or absolute) path to the current working directory can be displayed with the **pwd** (**p**rint **w**orking **d**irectory) command.

Usage:

```
pwd  
```

User alice wants to verify she is in the correct directory. She would use **pwd**.

```
[alice@station student]$ pwd  
/home/alice/website  
```

### Moving Around - The cd command

As was noted earlier, processes are able to change their cwd as needed. This includes the **bash** command shell, which provides the **cd** (**c**hange **d**irectory) command for changing the current directory from the command prompt.

Usage:

```
cd [DIRECTORY]  
```

If not specified, DIRECTORY defaults to the user's home directory.

Consider the following sequence of commands:

```
[alice@station website]$ pwd  
/home/alice/website  
[alice@station website]$ cd /home  
[alice@station home]$ pwd  
/home  
[alice@station home]$ cd /home/alice/website/songs  
[alice@station songs]$ pwd  
/home/alice/website/songs  
```

Notice how the result of the **pwd** command and the last part of the command prompt both change to reflect the new cwd after each **cd** command.

Because navigating through directories is so important, special abbreviations are available for referencing certain directories:

#### Table 1.1. Special Directory Names

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>The current working directory</td>
</tr>
<tr>
<td>..</td>
<td>The parent directory</td>
</tr>
<tr>
<td>~</td>
<td>The user's home directory</td>
</tr>
<tr>
<td>-</td>
<td>The previous working directory</td>
</tr>
</tbody>
</table>
The entries in this table require a little explanation. First, all but the last of these symbolic names are recognized by the bash shell in most contexts, not just in the cd command. Second, the directory tree is often described using a parent/child analogy. If dir1 contains dir2, then dir2 is said to be a child directory of dir1, and dir1 is the parent directory of dir2. Thus, the .. directory is one level closer to the root of the tree than is the cwd. Next, each user account is assigned a home directory, usually a subdirectory of /home that matches their username (more on this in the next chapter). The ~ directory represents that directory. Finally, the dash (-) is a special cd command option that refers to the previous working directory, making it possible to switch back-and-forth between directories easily.

Again, consider the following sequence of commands:

```
[alice@station songs]$ pwd
/home/alice/website/songs
[alice@station songs]$ cd ~
[alice@station alice]$ pwd
/home/alice
[alice@station alice]$ cd -
[alice@station songs]$ pwd
/home/alice/website/songs
[alice@station songs]$ cd..
[alice@station website]$ pwd
/home/alice/website
[alice@station website]$ cd
[alice@station alice]$ pwd
/home/alice
```

The last set of commands in the example above shows that, by default, the cd command uses the ~ directory if no directory is given.

### Absolute and Relative References

This section describes two ways of identifying the location of a file. In this section, as in almost every other place in these lessons, it is important to remember that in Linux a directory is a kind of file, so all the things said about naming files apply to directories as well as ordinary data files.

To identify a file, enough information has to be provided to locate the file within the filesystem. This location can be given in two ways: as an absolute reference (or absolute pathname), or as a relative reference.

Absolute references start with a slash (/) and give the FQN of the file. That is, the absolute reference names each branch of the filesystem directory tree, starting from /, that must be traversed to reach the file. Regardless of where you are in the filesystem (that is, regardless of the value of the cwd), an absolute reference unambiguously identifies a specific resource. Several examples of absolute references have already been discussed in this lesson.

A relative reference does not describe a path to the file starting from /, but rather describes a path starting from the current directory. For example, if the cwd is /home/alice, then a relative reference to song1.midi might be website/songs/song1.midi. This is a relative reference because it does not start with /. This reference only names the directories that must be traversed starting from /home/alice, not starting from /. In order to be valid, a relative reference must start by naming a directory (or file) in the cwd.

All directories in Linux contain two special entries, the . and .. directories, that represent the current and parent directories, respectively. Thus, in the discussion earlier about the cd command, the example cd .. was really just a use of a relative reference.

Table 1-2 contains some additional examples of relative references. Each of these is a reference to the file with FQN /home/alice/sample.txt. Some of these examples are intentionally “inefficient.”
Table 1.2. Sample Relative References to /home/alice/sample.txt

<table>
<thead>
<tr>
<th>cwd</th>
<th>Relative Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/alice</td>
<td>sample.txt or ./sample.txt</td>
</tr>
<tr>
<td>/home/alice/website/songs</td>
<td>../../sample.txt</td>
</tr>
<tr>
<td>/home/elvis/docs</td>
<td>../../alice/sample.txt</td>
</tr>
<tr>
<td>/home</td>
<td>alice/sample.txt</td>
</tr>
<tr>
<td>/home</td>
<td>../home/alice/website/../../sample.txt</td>
</tr>
</tbody>
</table>

Yes, that last example is pretty silly, but nevertheless is still perfectly valid.

Examples

Discovering your home directory

Hogan has just logged in and is unaware that, by default, his home directory is assigned as his cwd. Further, he is unaware that his home directory is /home/hogan. He sets out to discover what his home directory is. He also wants to see if, in fact, his default cwd is his home directory.

```
[hogan@station hogan]$ cd ~
[hogan@station hogan]$ pwd
/home/hogan
[hogan@station hogan]$ cd -
/home/hogan
```

1. Hogan knows the cwd ends in hogan (from the command prompt) but he doesn't know if this is home. He uses the cd command with the ~ directory symbol to change to his home directory.
2. Next, Hogan uses the pwd to display his home directory's FQN. He now knows /home/hogan is home, but is that where he started from?
3. To see where he started from, Hogan again uses the cd command, this time to go back to his previous working directory.
4. The cd - command displays the cwd after making the change (why is this reasonable behavior?). Hogan now knows that his system logs him in with his home directory set as his initial working directory.

Climbing up and down the tree.

Alice needs to edit some of the html files in her website. After she logs in, she will need to change to the appropriate subdirectory under her home directory. When she is done, she wants to change to the / etc to examine a configuration file. Finally, she wants to change back to her home directory to begin working on another task.

```
[alice@station alice]$ cd website/html
[alice@station html]$ ...
... (edits the html files) ...
[alice@station html]$ cd /etc
[alice@station etc]$ ...
... (examines the configuration file) ...
[alice@station etc]$ cd
[alice@station alice]$ ...
... (continues with other tasks) ...
```
Alice uses a relative path to change to the /home/alice/website/html directory because it is shorter than the FQN, but either would have worked.

After finishing her work on the html files, Alice uses an absolute reference to change directories to /etc. In this case, the absolute path is much shorter and easier to understand than the equivalent relative reference, ../../../../etc.

Note that the command prompt does not contain a / before the etc directory name. The default bash prompt displays only the final component of the cwd.

Done with her work in /etc, Alice returns to her home directory using the default behavior of the cd command.

Online Exercises

Exploring the filesystem

Lab Exercise

Objective: Demonstrate the use of cd.

Estimated Time: 10 mins.

Specification

This lab will have you set the current working directory of four simultaneously running bash shells. If you are using the X graphical environment, you may simply open four terminal windows. If you are using virtual consoles, you may use any four of the six provided virtual consoles.

Open four terminals with bash shells (as described above), using your primary account. Using the cd command, set the current working directories of the shells to the following four directories:

1. ~/..
2. /tmp
3. /etc/sysconfig
4. /usr/share/gnome

Deliverables

1. Four concurrently running bash shells, with each shell's current working directory set to one of the four directories listed above.

Possible Solution

The following command would set a shell's current working directory to the first directory listed above.

[student@station student]$ cd ~/..

Questions

Navigating the Filesystem

All these questions assume you are logged in as user student on a typical classroom system.
1. Which of the following is a relative directory reference?
   a. /home/student
   b. /etc
   c. ..
   d. ~

2. Which of the following could have been displayed by `pwd`?
   a. home/student
   b. /etc
   c. ..
   d. ~

3. Following the command `cd ~`, which is the most likely result from `pwd`?
   a. /home/student
   b. /etc
   c. ..
   d. ~

Your screen shows the following:

```
[student@station html]$ cd ../bin
[student@station bin]$ pwd
/home/student/bin
[student@station bin]$ cd -
```

4. What would be the output of the command `pwd`?
   a. /home/student
   b. /home/student/html
   c. /home/html
   d. ~

5. Which of the following pairs of commands always produce the same effect?
   a. `cd ..` and `cd -`
   b. `cd /` and `cd root`
   c. `cwd` and `pwd`
   d. `cd ~` and `cd`
Chapter 2. Important Directories

Key Concepts

- Every user account is assigned a home directory.
- The /tmp directory is used for global scratch space.
- The /bin and /usr/bin directories contain commonly executed files.
- The /etc directory contains system configuration files.
- Root's home directory, /root, should not be confused with the filesystem root directory, /.

Discussion

Standard Linux Directory Scheme

Linux can be used to support many different types of computer systems: servers, development workstations, personal desktop systems, etc. In order to standardize the filesystem's directory structure across this varied range of systems, most Linux systems employ a common naming and utilization scheme that makes the systems much easier to use and maintain. Using the same type of tree diagram as in the last chapter, part of the first layer of the directory tree might look something like this:

Figure 2.1. Partial Directory Tree Starting at /

```
-- bin...
   |-- etc...
   `-- home...
|   `-- root...
/   `-- sbin...
     |-- tmp...
     `-- var...
```

This chapter presents a brief (sometimes very brief) description of the role played by each of these directories.

The User's Home Directory

Each user of a Linux system is assigned a special directory, called their home directory, that represents their "private" space on the system. Typically, this is a subdirectory under the /home directory whose name matches the user's login username. (Examples of this, such as /home/alice or /home/hogan were seen in the last chapter.) The one significant exception to this is the superuser (or root user) whose home directory is usually /root. For any user, the tilde character (~) represents the FQN of the user's home directory when used as the first character of a file or directory reference.

The most obvious purpose for a user's home directory is as private data space, a place where they can store files away from the files of other users. Normally, users are free to create subdirectories under their home directory and organize their data as they see fit (subject to any quota restrictions that might
be in place). Two different users can even give files or directories the same name without conflict, since each is stored in their own space. For example, both Alice and Hogan can have public_html subdirectories of their own, /home/alice/public_html and /home/hogan/public_html respectively. Keeping each user's space separate from the others makes security much easier for the users, as well.

Another important aspect of a user's home directory is that it provides a place to store user-specific configuration files. When Blondie logs in, for example, she may need a different environment than does Prince. Blondie may prefer different screen colors, different command shortcuts, perhaps even a completely different desktop environment than Prince. Local, user-specific configuration files make this possible. Many of these configuration files are created automatically in a default form when a user's account is created, or on first use of a particular system resource.

Users normally are given relatively complete control over their own home directories, while access to the rest of the system is restricted. Alice, for example, most likely cannot modify or delete any of the configuration files in the /etc directory, even though she may be able to read them. And most likely she cannot read, modify or delete any of the files in the directories of other users of the system.

When a user first logs in to the system, they are placed "in" their home directory; that is, the system sets their initial working directory as their home directory.

The /tmp Temporary Directory

In addition to their personal space in their home directories, users are given access to shared "scratch" space in the /tmp directory. A program to do data compression might store partial results in /tmp, only placing the final results in the user's home directory when the work was complete. Linux systems often implement quotas on user accounts to prevent any one user from consuming an unfair proportion of the available space. The /tmp directory gives all users access to additional space to meet short-term needs without charging the space against their quota. This is especially important since users may not always be aware of how much extra space a service needs, and some services (such as X) cannot run at all if there is no temporary storage space available. This "global" scratch space is available to all processes on the system as well as all users. Files placed in this directory are deleted automatically by the system after a few days.

The /etc Configuration Directory

One of the unique characteristics of a Linux system is its flexibility. Virtually every aspect of the system can be configured in some fashion by editing a configuration file. These configuration files are normally placed in /etc or a subdirectory of /etc. For example, the sendmail program, often used for running a mail server, uses configuration files placed in /etc/mail. System startup scripts are found in /etc/rc.d, while network configuration files are found in /etc/sysconfig. Obviously, ordinary users cannot modify the files in /etc (or even read them, in some cases), but system administrators spend a lot of their time working with files stored here.

The /bin and /usr/bin Command Directories

Most system commands are stored as binary files in a machine-readable format. Commands appropriate for use by ordinary users are usually placed in the /bin or /usr/bin binary directories. Core utilities like ls, cd, cp, mv and the vi text editor, without which the system would not be usable, go in /bin. Supplementary utilities like compilers, your web browser and office suite go in /usr/bin, which can be made available to other systems over the network. Think of /bin and /usr/bin as unprivileged command directories, since no special privileges are required to use the commands found in them.

Think of /bin and /usr/bin as unprivileged command directories, since no special privileges are required to use the commands found in them.
Important Directories

The /sbin and /usr/sbin Command Directories

Just as /bin and /usr/bin store command files for ordinary users, so also /sbin and /usr/sbin store command files for use by the superuser, root. These include commands for attaching and removing hardware, for starting and stopping the system, and for performing system maintenance. These privileged commands also are stored in two separate directories, for the same reasons as for /bin and /usr/bin.

The /var "Variable" Directory

The collections of commands and configuration files found in places like /bin or /etc change very little, if at all, from day to day. These files tend to be very stable. Some files, however, change frequently. These include incoming/outgoing email, system logs, news group collections, web sites, ftp file archives and the like. These variable content files and directories are usually collected in the /var directory. Placing such files here makes it easier to allocate space for them, and also makes it easier to protect the more stable files found elsewhere on the system.

root vs. /root vs. / (the filesystem root)

It is an unfortunate accident of history that the term root plays such an important and yet confusing role in Linux. The word "root" is the username of the superuser, that is, the user with supreme authority over the system. It is also the name of that user's home directory, /root. And it is also the term used for the base (top?) of the filesystem's directory tree, the / directory. Normally, the meaning of the term can be gleaned from context, but a phrase such as "root directory" can be ambiguous. Try to anticipate and avoid such confusion in your own communication, and seek clarification if you encounter a use of the word that cannot be resolved from context.

Examples

The /tmp Directory

Alice just learned that some processes automatically use /tmp for scratch space, and she wishes to see if anything she has done used that space. She also wants to test to see if she really can create files there.

To create a file, Alice will use the command touch.

[alice@station alice]$ cd /tmp
[alice@station tmp]$ ls
orbit-alice ssh-XXDg4ke3
[alice@station tmp]$ ls -l
total 8
drwx------  2 alice alice          4096 Mar 16 08:04 orbit-alice
drwx------  2 alice alice          4096 Mar 16 07:07 ssh-XXDg4ke3
[alice@station tmp]$ touch newfile
[alice@station tmp]$ ls -l
total 8
-rw-rw-r--  1 alice alice          0 Mar 16 14:14 newfile
drwx------  2 alice alice          4096 Mar 16 08:04 orbit-alice
drwx------  2 alice alice          4096 Mar 16 07:07 ssh-XXDg4ke3

After changing to /tmp, Alice uses ls and ls -l to view the contents of the /tmp directory. While there is a lot about the long listing produced by ls -l she doesn't know, she sees her username enough to convince her that, as she had been told, some of the programs she runs had left files in /tmp. In fact, both orbit-alice and ssh-XXDg4ke3 were placed there when she started her desktop environment.
Alice uses the `touch` command to create a new, empty file named `newfile` in `/tmp`. Like many Linux commands, when `touch` succeeds, it succeeds silently (with no visual feedback on the screen). Alice verifies that `touch` worked with another `ls -l`.

### Looking for commands in `/bin` and `/usr/bin`

Hogan wants to see if some of the common commands he uses are in either of the binary command directories `/bin` or `/usr/bin`. He decides to test `cp`, `mozilla`, `cd` and `fdisk`.

Hogan could use `ls` to look for these commands, but he wants to try a new command instead, `which`.

Simple usage:

```
which PROGRAM
```

Displays the FQN of the `PROGRAM` command file that will be used if the user runs the command.

```
[hogan@station hogan]$ which cp
/bin/cp
[hogan@station hogan]$ which mozilla
/usr/bin/mozilla
[hogan@station hogan]$ which cd
/usr/bin/which: no cd in (/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/hogan/bin:)
[hogan@station hogan]$ which fdisk
/usr/bin/which: no fdisk in (/usr/local/bin:/usr/bin:/usr/X11R6/bin:/home/hogan/bin:)
[hogan@station hogan]$ ls /sbin
... ether-wake idconfig portmap usbmodules
extendfs lilo poweroff vboxd
fdisk logdump pppoe vgcfgbackup
... 
```

- Hogan sees that `cp` is found in the `/bin` directory, which seems right to him since files in `/bin` should be files that ordinary users would want that should always be present on a system.
- Hogan finds `mozilla` (his web browser of choice) in `/usr/bin`. This also seems right since files in `/usr/bin` should be files an ordinary user might want that might not always be available.
- Hogan is surprised at the results for `cd`, which seems to say there is no `cd` command on this system. He was expecting to see a response of `/bin/cd` (Why?) What Hogan does not know is that `cd` is an example of an `internal` command, one that’s provided by the `bash` shell itself. On the other hand, if he checked, he would find `bash` in `/bin`.
- Hogan is also surprised at the results for `fdisk`. He knows this is an important but dangerous command used to partition disk drives, and should only be available to the superuser, so he was expecting a response of `/sbin`. He can even see `fdisk` in `/sbin` when he looks using `ls`. Upon closer inspection, he discovers that `which` didn’t even look for `fdisk` in `/sbin`. Later, Hogan will learn about the `command path` and discover why this makes perfect sense.

### Online Exercises

#### Exploring Important Directories

**Lab Exercise**

**Objective:** Explore some of the characteristics of the important directories discussed in this chapter.

**Estimated Time:** 15 mins.
Setup

In this exercise you will need to use the `touch` command to create files. This command was demonstrated in the examples for this chapter. You will need to use the `which` command to locate command files. This command was demonstrated in the examples as well.

Specification

1. Open a new terminal session or virtual console (this is necessary for the online grading mechanism to evaluate your work).

   You want to test whether or not you can use `touch` to create files in various directories. You expect this to work in your home directory and in `/tmp`, but not in other directories. Use the command `touch newfile.username` (where `username` is replaced by your username) in each of the following directories:

   a. ~
   b. /tmp
   c. /bin
   d. /root

   Clearly, you should expect some of your efforts to fail. Return to your home directory when you are done.

2. Using `which`, determine the FQN of the binary command file for each of the following: `pwd`, `ls` and `whoami`. Run each of these commands in its "raw" form by using the FQN at the command prompt. Finally, `exit` the terminal session or virtual console.

   When you have finished, exit your shell, so that your command history is saved to the file `~/.bash_history`.

Deliverables

1. 1. Two files called `~/newfile.username` and `/tmp/newfile.username`, where `username` is your primary account name.

2. A `~/.bash_history` file that contains a record of your "raw" commands.

Possible Solution

The following sequence of commands demonstrates the use of the `which` and `touch` commands, and running a command using its absolute reference.

```
[student@station student]$ touch /tmp/newfile.student
[student@station student]$ which pwd
/bin/pwd
[student@station student]$ /bin/pwd
/home/student
```

Questions

1. Which of these is most likely not someone's home directory?
Important Directories

1. **Important Directories**
   
   a. /home/student
   
   b. /root
   
   c. /
   
   d. None of these - all are likely to be home directories

2. The file named.conf is a system configuration file. This file belongs in
   
   a. /tmp
   
   b. /etc
   
   c. /bin
   
   d. /sbin

3. The file e2fsck is a privileged command that must always be available to the system. This file would be found in
   
   a. /tmp
   
   b. /etc
   
   c. /bin
   
   d. /sbin

4. The /root directory is noteworthy because
   
   a. It is the root of the Linux filesystem
   
   b. It is the superuser's home directory
   
   c. It can always be abbreviated as ~
   
   d. Its contents cannot be read by any user.

5. Which of the following commands could not be used to create a file in /tmp?
   
   a. touch /newfile
   
   b. touch /tmp/newfile
   
   c. touch ../newfile
   
   d. touch ../tmp/newfile

6. T/F: /bin and /usr/bin contain the same files.
   
   a. True
   
   b. False

7. T/F: Only the root user can read files in etc.
   
   a. True
8. T/F: Files for a website might often be found in \var.  
   a. True 
   b. False 

9. T/F: Configuration files that customize each user's account are placed in \etc\users.d.  
   a. True 
   b. False 

10. T/F: X uses /tmp.  
    a. True 
    b. False
Chapter 3. Managing Files

Key Concepts

- Files can be easily created (or appended to) using shell redirection.
- The `cp` command copies files.
- The `mv` command moves files.
- The `rm` command removes files.
- Files can be "clobbered" as a result of redirection, moving, or copying.

Discussion

Redirection

Many commands produce “visible” output. Normally, output is displayed on the screen. Notice, for example, how the output from `pwd` is displayed for Julius:

```
[julius@station julius]$ pwd
/home/julius
[julius@station julius]$
```

Linux likes to think of everything as a file. In the example above, `pwd` sent the output to the *standard output* file, or STDOUt, which is by default the screen for users logged into the system.

One of the features of the `bash` command shell is that output which would normally be directed at the screen through STDOUt can, instead, be *redirected* to some other file. This is done by using the special redirection symbol, >, as in this example:

```
[julius@station julius]$ ls
[julius@station julius]$ pwd > results.txt
[julius@station julius]$ ls
results.txt
```

In this example, the usual results of the command were redirected to the file `results.txt`, creating the file in the process. A helpful command that can let us verify what has happened is the `cat` (concatenate) command.

Usage:

```
cat [OPTIONS] [FILE...]  
```

Concatenate FILE(s) to standard output.

The `cat` command copies each of the files listed to standard output. If more than one file is listed, this effectively concatenates the files. As for any other command, this output will display on the screen by default, but could also be redirected to a file. We will use it to display a file by naming a single file and not redirecting the results.

```
[julius@station julius]$ pwd > results.txt
[julius@station julius]$ ls
results.txt
[julius@station julius]$ cat results.txt
/home/julius
```
Managing Files

This time, Julius captured the output of `pwd` by redirecting the results to `results.txt`, and then displayed `results.txt` with `cat`.

If the file already exists, redirection will delete and re-create the file empty, then capture the new output. If, however, a double arrow (>>) is used, the new output is appended to the file. If >> is used and the target file does not exist, it is created, just as if > had been used. The > symbol is an example of a *shell meta-character*, that is, a symbol with special meaning which the bash shell sees and interprets before the rest of the command is acted upon. There are many of these as we shall learn in coming lessons, and all require careful attention.

The simplest command of all that produces visible output is `echo`. The `echo` command takes whatever text is typed as part of the command and echoes it to STDOUT (usually the display). This simple command, together with redirection, can be used to create text files.

**Usage:**

`echo [OPTIONS] [STRING...]`

Echoes STRING(s) to standard output.

**Example 1:**

```
[julius@station julius]$ echo Hello > greetings.txt
[julius@station julius]$ echo How are you >> greetings.txt
[julius@station julius]$ cat greetings.txt
Hello
How are you
```

**Example 2:**

```
[julius@station julius]$ ls
[julius@station julius]$ pwd > results.txt
[julius@station julius]$ cat results.txt
/home/julius
[julius@station julius]$ ls >> results.txt
[julius@station julius]$ cat results.txt
/home/julius/results.txt
```

**Copy files with cp**

Duplicate copies of files can be created with the `cp` (copy) command.

**Usage:**

`cp [OPTIONS] {SOURCE} {TARGET}`

`cp [OPTIONS] {SOURCE...} {DIRECTORY}`

In the first form, a copy of the SOURCE file is made as TARGET. In the second form, one or more files can be copied at one time to a directory. A copy of SOURCE, ..., is made in DIRECTORY and named DIRECTORY/SOURCE, .... With the appropriate options (not discussed here - try `man cp`) whole subdirectory trees can be copied at once.

**Examples:**

1. Make a copy of mysong.midi and name the copy backup.midi:

   `cp mysong.midi backup.midi`

2. Make a copy of mynovel.txt in /tmp:

   `cp mynovel.txt /tmp`
Managing Files

3. Copy the files songs.tar and novels.tgz into the directory /tmp:

   `cp` songs.tar novels.tgz /tmp

   The resulting files will be named /tmp/songs.tar and /tmp/novels.tgz.

4. Make a copy of `webpage.html` from your home directory to the current directory:

   `cp` ~/webpage.html .

   The resulting file will be named ./webpage.html.

Move/rename files with `mv`

Files can be moved from one directory to another, or from one name to another (renamed), with the `mv` (move) command.

Usage:

`mv` [OPTION...] {SOURCE} {TARGET}

`mv` [OPTION...] {SOURCE...} {DIRECTORY}

In the first form, the SOURCE file is renamed as TARGET. In the second form, one or more files can be moved at one time to a directory. The files SOURCE, ..., are moved to DIRECTORY and named DIRECTORY/SOURCE, .... SOURCE can be a directory, in which case the directory is moved/renamed.

The `mv` command is particularly interesting because buried in the fabric of how it works is a critical fact about Linux filesystems: Linux treats the name of a file as being distinctly separate from the contents of the file. Even though the command `mv` stems from the word "move," `mv` rarely actually moves data around. Instead, the filesystem merely records a name change. If the name changes from /somedir/somefile to /somedir/newname, we see this as "renaming" the file. If the name changes from /somedir/somefile to /newdir/somename, we see this as "moving" the file. If the name changes from /somedir/somefile to /newdir/newname, we see this as a double change, both moving and renaming. But to Linux, all of these are the same thing, a change in the FQN for the file.

Examples:

1. Rename `mysong.midi` as `backup.midi`:

   `mv` mysong.midi backup.midi

2. Move `mynovel.txt` to `/tmp`:

   `mv` mynovel.txt /tmp

   The resulting file will be named /tmp/mynovel.txt.

3. Move both `songs.tar` and `novels.tgz` to `/tmp`:

   `mv` songs.tar novels.tgz /tmp

   The resulting files will be named /tmp/songs.tar and /tmp/novels.tgz.
4. Move webpage.html from your home directory to the current working directory:

   `mv ~/webpage.html .`

   The resulting file will be named ./webpage.html.

5. Rename the html subdirectory of the current working directory to public_html:

   `mv html public_html`

   This renames ./html as ./public_html, assuming that ./public_html does not already exist - see the next example.

6. Your cwd is your home directory. You have two subdirectories, ~/images and ~/html. Move the images directory to the html directory:

   `mv images html`

   The resulting directory will be named ~/html/images. Note the similarity between this example and the previous one. The critical difference is that, in this case, the target directory ~/html already existed, so `mv` moved the source directory inside (underneath) the target. In the previous example, the target directory did not already exist, so `mv` renamed the source directory to the new name.

### Remove (delete) files with rm

Files can be removed (erased, deleted) with the **rm** (remove) command.

Usage:

   `rm [OPTIONS] {FILE...}`

   Removes the FILE(s) from the filesystem. Technically, it *unlinks* the FILE(s), a distinction which will be made more clear later. With the appropriate options (not discussed here - try **man rm**) whole subdirectory trees can be removed at once.

   **Note**

   **rm** cannot remove a directory unless special command options are used. There is a separate command, **rmdir**, for this purpose.

   **Warning**

   Yes, the **rm** command *can* remove entire directory trees at once. In the hands of the superuser it can delete the *entire* contents of the filesystem in a single stroke -- not usually what was intended.

   **Warning**

   The documentation for **rm** includes the following statement: "Note that if you use rm to remove a file, it is usually possible to recover the contents of that file." While this may be true, it requires expertise beyond the scope of this course, so, for all practical purposes, you should treat this command as non-reversible.

**Examples:**

1. Delete mysong.midi:

   `rm mysong.midi`
2. Remove both `songs.tar` and `novels.tgz`:

   ```bash
   rm songs.tar novels.tgz
   ```

3. Remove `photos.html` from your home directory:

   ```bash
   rm ~/photos.html
   ```

## Clobbering files (oops!)

The warnings contained in the previous section hint at some of the dangerous potential commands like `cp`, `mv` and `rm` can have. Unfortunately, while we usually see the rapids, we often miss the rocks beneath the surface of the water, and Linux has its share of those. This is the price you pay for the power of an operating system like Linux. In this section we highlight one such danger: clobbering files.

Command redirection with `>`, and the commands `cp` and `mv` all can name target files. Normally, these are new filenames, and the commands create the files. But if an existing file is named as a target for redirection, `cp` or `mv`, the existing file will be destroyed without warning. This is known as **clobbering** a file. Because the problem can be so surprisingly subtle to a new user, we'll work through several examples.

**Example 3:**

```bash
[julius@station julius]$ pwd > file1
[julius@station julius]$ cat file1
/home/julius
[julius@station julius]$ ls > file1
[julius@station julius]$ cat file1
mycwd
```

No real surprises here if you've been paying attention -- simply note how the second redirection replaced the previous contents of `file1` without any warning messages. The file `file1` has been clobbered.

**Example 4:**

```bash
[julius@station julius]$ touch file1
[julius@station julius]$ ls
file1
[julius@station julius]$ pwd > file2
[julius@station julius]$ ls
file1 file2
[julius@station julius]$ cat file2
/home/julius
[julius@station julius]$ mv file2 file1
[julius@station julius]$ ls
file1
[julius@station julius]$ cat file1
/home/julius
```

No surprises here, either -- note how `mv` replaced the previous contents of `file1` without any warning messages. Again, `file1` has been clobbered. `cp` would have replaced `file1` in the same way.

In order to understand our final example fully, we need to learn just a little bit more about redirection. In particular, we need to learn **when** redirection happens.

**Example 5:**

```bash
[julius@station julius]$ touch file1
[julius@station julius]$ ls
file1
[julius@station julius]$ ls > file2
```
Managing Files

[julius@station julius]$ ls
file1  file2
[julius@station julius]$ cat file2
file1  file2

Why does file2 list both file1 and file2, when the first ls only shows file1? The answer: because of the sequence in which the critical events took place. Event 1: bash saw the redirection to the file file2, and created it as an empty file (see the earlier discussion of redirection). Event 2: ls then generated a file listing, showing both the previously-created file1 file and the newly-created file2 file. Event 3: the output from ls was sent to file2.

Example 6:

[julius@station julius]$ pwd > mycwd
[julius@station julius]$ cat mycwd
/home/julius
[julius@station julius]$ cp mycwd mycwd
cp: `mycwd' and `mycwd' are the same file
[julius@station julius]$ cat mycwd
/home/julius
[julius@station julius]$ cat mycwd > mycwd
cat: mycwd: input file is output file
[julius@station julius]$ cat mycwd
[julius@station julius]$

This takes some explanation. First Julius creates mycwd using pwd, and displays its contents with cat. Then he uses cp to try and copy the file onto itself, but cp detects the "error," displays a message, and does nothing (mv would have behaved the same way). The cat on the next line verifies that mycwd is unchanged. OK so far. The next sequence, however, produces a real surprise. Julius uses cat with redirection to try and copy a file onto itself. cat also complains about the source (input) and target (output) files being the same, and would seem to do nothing. But the final cat of mycwd produces no response -- the file's contents are lost! Why? Because of the sequence in which the critical events took place. Event 1: bash sees the redirection to an existing file, deletes the file and re-creates it as empty (see the earlier discussion of redirection) -- mycwd is clobbered. Event 2: it no longer matters what cat does, because the source file mycwd has already been destroyed and re-created. It is too late for the error to be prevented.

There are a few command options and special command shell features that can be used to mitigate this behavior somewhat, but the objective in this chapter is simply to begin to learn how commands such as these work, and in this section, how they work on existing files. This is a feature, not a bug, that needs to be understood to be used correctly.

Examples

Moving/Archiving a Simple Website

Madonna attempted to set up a simple personal website, but after she was done her ISP told her she had placed all of the website's files in the wrong directory. She placed the files in ~/html, and they should have been in ~/public_html. She needs to move the files to the correct directory. While she's at it, she also wants to make a backup copy of the files in a directory called archive, just in case she accidentally damages or deletes one from ~/public_html. The archive directory already exists.

[madonna@station madonna]$ ls
html  archive
[madonna@station madonna]$ mv html public_html
[madonna@station madonna]$ cd public_html
[madonna@station public_html]$ ls
index.html  photo.jpeg
[madonna@station public_html]$ cp index.html photo.jpeg ../archive

[madonna@station madonna]$ ls
html  archive
[madonna@station madonna]$ mv html public_html
[madonna@station madonna]$ cd public_html
[madonna@station public_html]$ ls
index.html  photo.jpeg
[madonna@station public_html]$ cp index.html photo.jpeg ../archive
Managing Files

To move the files, all Madonna had to do was rename the directory. To make backup copies, she used the ability of the `cp` command to copy more than one file at a time.

**Moving a Simple Website**

Nero attempted to set up a simple personal website, but after he was done his ISP told him he had placed all of the website’s files in the wrong directory. He placed the files in his home directory, and they should have been in `~/public_html`. He needs to move the files to the correct directory. His problem is harder than Madonna's because he cannot simply rename a subdirectory. The `public_html` directory was created for him by his ISP.

```
[nero@station nero]$ ls
index.html  photo1.jpeg  photo2.jpeg  public_html
[nero@station nero]$ mv index.html photo1.jpeg photo2.jpeg public_html
[nero@station nero]$ ls
index.html  public_html
[nero@station nero]$ cd public_html
[nero@station public_html]$ ls
index.html  photo1.jpeg  photo2.jpeg
```

To move the files, Nero used the ability of the `mv` command to move more than one file at a time.

**Removing robots.txt**

Elvis notices a file in his `~/public_html` directory that he doesn’t recognize, a file named `robots.txt`. He decides to delete the file.

```
[elvis@station elvis]$ ls
public_html
[elvis@station elvis]$ ls public_html
index.html  robots.txt
[elvis@station elvis]$ rm public_html/robots.txt
[elvis@station elvis]$ ls public_html
index.html
```

**Online Exercises**

**Create a Website**

**Lab Exercise**

Objective: Use redirection, `cp` and `mv` to create a simple website, and archive the website.

Estimated Time: 15 mins.

**Setup**

In this exercise you will need to work with the directories `~/html` and `~/archive`. Login to your account, and use the following commands to create these two directories:

```
@student@station student]$ mkdir ~/html
@student@station student]$ mkdir ~/archive
```

**Specification**

You wish to create a simple personal website using the techniques demonstrated in the Examples section of this chapter.
Managing Files


2. Move this file to ~/html.

3. Rename the html directory as public_html.

4. Make a copy of your index.html file in ~/archive.

   **Note**

   The resulting webpage may or may not be visible with a browser, depending on additional features of your system's configuration. Your instructor will tell you if you can view the results of your work in this fashion.

**Deliverables**

1. Two files, ~/public_html/index.html, and ~/archive/index.html, each of which contain identical content (The text "Under Construction", possibly with HTML markup.)

**Possible Solution**

The following sequence of commands provides one possible solution to this exercise.

```
[student@station student]$ mkdir html archive
[student@station student]$ echo '<h1>' > index.html
[student@station student]$ echo 'Under construction' >> index.html
[student@station student]$ echo '</h1>' >> index.html
[student@station student]$ mv index.html html
[student@station student]$ mv html public_html
[student@station student]$ cp public_html/index.html archive
[student@station student]$ ls public_html
index.html
[student@station student]$ ls archive
index.html
```

**Investigate the "noclobber" option**

**Lab Exercise**

**Objective:** Investigate the consequences of the **bash** "noclobber" option.

**Estimated Time:** 10 mins.

**Specification**

You wish to investigate the consequences of clobbering files, and the **bash** "noclobber" option.

1. Open a new terminal session or virtual console (this is necessary for the online grading mechanism to evaluate your work).

2. Create a file called source containing the word "source" and a file called target containing the word "target."

3. Use **cp** to clobber target with source. Verify the effect with **cat**.

4. Re-create target.
5. Clobber `target` again by `catting source` and redirecting the output. Verify the results.

6. Turn on the bash "noclobber" option with the following magic command:

   ```
   [student@station student]$ set -o noclobber
   ```

7. Test both types of clobbering again, and note the results.

8. Turn off the bash "noclobber" option with the following magic command:

   ```
   [student@station student]$ set +o noclobber
   ```

9. Exit the terminal session or virtual console.

Deliverables

1. A `~/.bash_history` file that contains a record of your enabling and disabling of the noclobber options.

Possible Solution

The following sequence of commands provides one possible solution to this exercise.

```
[student@station student]$ echo source > source
[student@station student]$ echo target > target

[student@station student]$ cp source target
[student@station student]$ cat target
source

[student@station student]$ echo target > target
[student@station student]$ cat source > target
source
[student@station student]$ cat target
source

[student@station student]$ echo target > target
[student@station student]$ set -o noclobber
[student@station student]$ cp source target
[student@station student]$ cat target
source

[student@station student]$ echo target > target
[student@station student]$ cat source > target
-bash: target: cannot overwrite existing file

[student@station student]$ set +o noclobber
[student@station student]$ exit
```

Questions

1. The command to make a duplicate of a file is
   a. dupe
   b. rep
   c. copy
   d. cp
2. To copy file1.txt to file2.txt, you could use
   a. `cp file1.txt > file2.txt`
   b. `cp file1.txt file2.txt`
   c. `dupe file1.txt file2.txt`
   d. `mv file1.txt file2.txt`

3. The successful command `mv f1 f2 f3 f4`
   a. moves four items from the current directory
   b. moves four items to the current directory
   c. moves four items to the default directory
   d. moves three items from the current directory

4. To erase the file file1 from the current directory, you could use
   a. `rm file1`
   b. `erase file1`
   c. `del file1`
   d. `cut file1`

5. The command `cp file.a file.b` will clobber file.b if
   a. file.a is an ordinary file
   b. file.a is non-empty
   c. file.b is an ordinary file
   d. file.b is a directory

6. If /home/student is a directory, then the command `rm /home/student` will
   a. remove all files in the directory
   b. delete the directory from the filesystem
   c. delete the student user account
   d. produce an error message

7. The command `echo file.1 file.2` will
   a. copy file.1 to file.2
   b. display the contents of file.1 and file.2 on STDOUT
   c. produce an error message because there is no redirection symbol
   d. display "file.1 file.2" on STDOUT
8. The command `ls > file.1` will
   a. produce an error message because > is an unprotected shell meta-character
   b. append the normal output of `ls` to `file.1`
   c. replace `file.1` if it already exists
   d. fail if `file.1` already exists

9. Assuming both exist, the command `mv item.1 item.2` will move an entire subdirectory tree
   a. only if both `item.1` and `item.2` are existing subdirectories
   b. only if special command options are used
   c. only if `item.2` is empty
   d. only if `item.1` is empty

10. The command `cp item.1 item.2` will copy an entire subdirectory tree
    a. only if both `item.1` and `item.2` are existing subdirectories
    b. only if special command options are used
    c. only if `item.2` is empty
    d. only if `item.1` is empty
Chapter 4. Managing Directories

Key Concepts

- The contents of entire directory trees can be examined with the `ls -R` command.
- The `mkdir` command creates directories.
- The `rmdir` command removes (empty) directories.
- `cp -r` recursively copies directories
- `rm -r` recursively removes directories

Discussion

Creating (new, empty) directories: `mkdir`

Organizing files into directories requires the ability to create the directories you need. In Linux, the command for making a new directory is the `mkdir` (make directory) command.

Usage:

```
mkdir [OPTIONS] [DIRECTORY...]
```

Creates the DIRECTORY(ies), if they do not already exist. Fails with an error message if one or more of the DIRECTORY(ies) do exist, but remaining new directories are still created.

To create a `~/public_html` subdirectory, Elvis could proceed as shown:

```
[elvis@station elvis]$ ls
[elvis@station elvis]$ mkdir public_html
[elvis@station elvis]$ ls
public_html
```

Elvis can then create a subdirectory of `public_html`:

```
[elvis@station elvis] ls public_html
[elvis@station elvis] mkdir public_html/images
[elvis@station elvis] ls public_html
images
```

Note, however, that by default `mkdir` will not create a subdirectory if the parent directory does not already exist:

```
[elvis@station elvis] ls work
ls: work: No such file or directory
[elvis@station elvis] mkdir work/spreadsheets
mkdir: cannot create directory 'work/spreadsheets': No such file or directory
```

Only if Elvis uses `mkdir -p` (as in `parent`) can he create an entire directory tree at once:

```
[elvis@station elvis] ls work
ls: work: No such file or directory
[elvis@station elvis] mkdir -p work/spreadsheets
[elvis@station elvis] mkdir -p work/images/logos
[elvis@station elvis] mkdir -p work/images/advertising
[elvis@station elvis] ls work
images  spreadsheets
```
Listing Directory Trees

By now, you should be aware that the `ls` command, when given a directory as an argument, lists the contents of the directory. But what if the directory contains other directories? By default, the `ls` command will only show the contents of the top level directory. If you would like the `ls` command to list subdirectories as well, you can add the `-R` command line switch.

Whenever a command iterates through a structure, operating on each element of the structure in the same manner as it operated on the topmost element, the command is said to act recursively. The `-R` command line switch specifies that the `ls` command should list with recursion.

For example, hogan is exploring his machine's networking configuration, and suspects that the `/etc/NetworkManager` directory is an important directory. In order discover what files it and its various subdirectories contain, hogan lists the contents of the directory recursively.

```
[hogan@station hogan]$ ls -R /etc/NetworkManager/
/etc/NetworkManager/:
dispatcher.d  NetworkManager.conf  system-connections  VPN

/etc/NetworkManager/dispatcher.d:
00-netreport  05-netfs  10-dhclient

/etc/NetworkManager/system-connections:

/etc/NetworkManager/VPN:
```

The `ls -R` can be combined with any of the previously learned command line switches, as well, such as `ls -sR` to include file sizes, or `ls -lR` to include file attributes.

Removing (empty) directories: `rmdir`

The command for removing a directory is `rmdir` (remove directory).

Usage:

```
rmdir [OPTIONS] {DIRECTORY...}
```

Removes the DIRECTORY(ies), if they are empty. Fails with an error message if one or more of the DIRECTORY(ies) are not empty, but remaining (empty) directories are still removed. Note that a directory containing only a subdirectory is not empty.

To remove his `~/public_html/images` subdirectory, Elvis could proceed as shown:

```
[elvis@station elvis]$ ls
public_html
[elvis@station elvis]$ rmdir public_html/images
[elvis@station elvis]$ ls
[elvis@station elvis]$
```

As we have observed before, note again that commands such as `mkdir` and `rmdir` succeed silently. Silence is golden, as they say.

Listing Directory Trees

By now, you should be aware that the `ls` command, when given a directory as an argument, lists the contents of the directory. But what if the directory contains other directories? By default, the `ls` command will only
show the contents of the top level directory. If you would like the `ls` command to list subdirectories as well, you can add the `-R` command line switch.

Whenever a command iterates through a structure, operating on each element of the structure in the same manner as it operated on the topmost element, the command is said to act *recursively*. The `-R` command line switch specifies that the `ls` command should list with recursion.

For example, elvis would like a more comprehensive list of files and directories in his `~/work` directory. Remember that previously elvis had to perform a separate `ls` command for each subdirectory of `~/work` to see the subdirectory's contents. Now he can use `ls -R` instead.

```
[elvis@station elvis]$ ls -R work
work/:
    images  spreadsheets
work/images:
    advertising  logos
work/images/advertising:
work/images/logos:
work/spreadsheets:
```

Note that the above command will display files as well as directories. In this example there just happen to be no files to display.

The `ls -R` can be combined with any of the previously learned command line switches, as well, such as `ls -sR` to include file sizes, or `ls -lR` to include file attributes.

**Copying directory trees: `cp -r`**

In the previous chapter when `cp` was introduced, we noted that, with the correct options, the `cp` command could be used for copying entire directory trees. Here we discuss one such option: `-r` (for recursive). (Many commands have an option for acting recursively, that is, traversing all of the branches of a directory sub-tree, acting on each subdirectory in turn.)

Consider the following directory structure in Hogan’s home directory:

**Figure 4.1. Directory structure before copying**

```
-- html
 hogan -- website -- -- forms
      |       |       |            -- song1.midi
      |       |       |                       -- song2.midi
      |       | media --        -- picture1.png
      |       |           |                      -- photos--
      |       |           |                                -- picture2.png
```

Even though this directory tree contains many branches and files, it can be copied with a single instruction:
Figure 4.2. Directory structure after copying

```
[hogan@station hogan]$ cp -r website archive
```

```
- html
  - website -- -- forms
    - -- song1.midi
    - -- songs --
      - -- song2.midi
    - -- media --
      - -- picture1.png
  hogan --
    - -- photos--
      - -- picture2.png
- html
  - archive -- -- forms
    - -- song1.midi
    - -- songs --
      - -- song2.midi
    - -- media --
      - -- picture1.png
    - -- photos--
      - -- picture2.png
```

Remember the difference in the behavior of `cp` when the last argument is an existing directory: a copy of the source file(s) is made inside of the target directory. In the previous demonstration, the `archive` directory did not already exist and so was created. But if the `archive` directory had already existed, the effect would have been different.

Figure 4.3. Directory structure before copying

```
- html
  - website -- -- forms
    - -- song1.midi
    - -- songs --
      - -- song2.midi
    - -- media --
      - -- picture1.png
  hogan --
    - -- photos--
      - -- picture2.png
- html
  - archive -- -- forms
    - -- song1.midi
    - -- songs --
      - -- song2.midi
    - -- media --
      - -- picture1.png
    - -- photos--
      - -- picture2.png
```

Hogan uses exactly the same command as before to perform the copy:

```
[hogan@station hogan]$ cp -r website archive
```
Figure 4.4. Directory structure after copying with \texttt{cp -r website archive}

```
-- html
    -- website -- -- forms
        -- song1.midi
        -- songs
            -- song2.midi
        -- media
            -- picture1.png
    hogan
        -- photos
            -- picture2.png
-- archive -- website -- -- forms
        -- song1.midi
        -- songs
            -- song2.midi
        -- media
            -- picture1.png
        -- photos
            -- picture2.png
```

In this second demonstration, the \texttt{website} directory is re-created \textit{inside} the \texttt{archive} directory.

**Removing directory trees: \texttt{rm -r}**

In the previous chapter when \texttt{rm} was introduced, we noted that, with the correct options, the \texttt{rm} command could remove entire directory trees. Again, as for \texttt{cp} the option is \texttt{-r} (for \texttt{recursive}).

Consider the following directory structure in Hogan's home directory:

Figure 4.5. Directory structure before removing

```
-- html
    hogan -- website -- -- forms
        -- song1.midi
        -- songs
            -- song2.midi
        -- media
            -- picture1.png
        -- photos
            -- picture2.png
```

If Hogan wished to remove the \texttt{media} subtree, he could use a single command:
Managing Directories

[hogan@station hogan]$ rm -r website/media

Figure 4.6. Directory structure after removing

```
   -- html

hogan -- website --

   -- forms
```

It's worth observing one more time that, in general, removing files is non-reversible. Used carelessly, **rm -r** is a very dangerous command.

Exercising Directories with tree

The **tree** command will recursively list the contents of the directory, much like the **ls -R** command. The output, however, is displayed more intuitively as an ASCII tree.

```
$ tree /etc/NetworkManager/
/etc/NetworkManager/
   ### dispatcher.d
   #   ### 00-netreport
   #   ### 05-netfs
   #   ### 10-dhclient
   ### NetworkManager.conf
   ### system-connections
   ### VPN

3 directories, 4 files
```

Similarly, **tree -d** just shows directories.

```
$ tree -d /etc/sysconfig/
/etc/sysconfig/
   ### cbq
   ### console
   ### modules
   ### networking
   #   ### devices
   #   ### profiles
   #   ### default
   ### network-scripts
   ### zhn
   ### allowed-actions
   #   ### configfiles
   #   ### script
   ### clientCaps.d

13 directories
```

The output of **tree --help** should entice you to explore more features of **tree**.

Examples

Making a Backup of a Directory Tree

The user hogan has been working hard on a report, which several files which he has organized into subdirectories under a directory called **report**.

```
report/
   |-- chapter01/
```
Managing Directories

He is about to use an automated spell checker to check and replace misspelled words. Because this is the first time he has used the spell checker, he wants to make a backup of his work before continuing. He first tries to make a backup with the `cp` command.

```
[hogan@station hogan]$ cp report report.bak
```

Realizing his mistake, he adds the `-r` command line switch, to tell the `cp` command to copy recursively.

```
[hogan@station hogan]$ cp -r report report.bak
```

Now, the contents of the directory `report` are recursively copied to the directory `report.bak`.

```
   |  `-- report/
   |      |   |-- chapter01/
   |      |   |   |-- figures/
   |      |   |   |   |   |-- image01.png
   |      |   |   |   |   `-- image02.png
   |      |   |   |-- section01.html
   |      |   |   `-- section02.html
   |      `-- chapter02/
   |          |-- figures/
   |          |   |-- image01.png
   |          |   `-- image02.png
   |          |-- section01.html
   |          `-- section02.html
   `-- report.bak/
      |  `-- chapter01/
      |      |-- figures/
      |      |   |-- image01.png
      |      |   `-- image02.png
      |      |-- section01.html
      |      `-- section02.html
      `-- chapter02/
          |-- figures/
          |   |-- image01.png
          |   `-- image02.png
          |-- section01.html
          `-- section02.html
```

10 directories, 16 files

Once hogan has run the spell checker, and is happy with the results, he wants to remove the backup directory `report.bak`. He first uses the `rmdir` command.

```
[hogan@station hogan]$ rmdir report.bak/
```

```
rmdir: `report.bak/': Directory not empty
```

Remembering that `rmdir` only works on empty directories, he uses `rm -r` instead.

```
[hogan@station hogan]$ rm -r report.bak/
```
Creating a Local Copy of Example Files

The user ventura has taken an interest in programming with the python programming language. He has discovered sample scripts included in the /usr/share/doc/pygtk2-2.16.0/examples/ directory. He would like to be able to fiddle with the scripts, in order to try out new ideas, but doesn’t have writable permissions to the files. He decides to make a local copy of the scripts.

```
[ventura@station ventura]$ cp -r /usr/share/doc/pygtk2-2.16.0/examples/ .
[ventura@station ventura]$ ls -R examples/
examples/:
examples/atk:
examples/glade:
examples/gobject:
examples/gtk:
examples/ide:
examples/simple:
examples/Makefile.am:
examples/Makefile.in:
examples/pygtk-demo:
examples/atk-demo.py:
examples/autoconnect.py:
examples/glade-demo.py:
examples/README:
examples/test2.glade:
examples/test.glade:
examples/editable-interface.py:
examples/properties.py:
examples/signal.py:
examples/application.py:
examples/customtreemodel.py:
examples/scrollable.py:
examples/testprint.py:
examples/widget.py:
examples/bin.py:
examples/filechooser.py:
examples/sizegroup.py:
examples/uimanager.py:
...```

1. Note the use of the directory name . to mean "the local directory". When cp receives a directory as its destination (last argument), it copies into that directory, preserving original filenames. In this case, the directory examples was copied into the directory .

2. The ls -R command generates a recursive listing of every file in the newly created examples directory.

Now ventura has a local copy of the python scripts, which he can modify at will.

Online Exercises

Managing Directories

Lab Exercise

Objective: List, copy, move, and remove directories efficiently.

Estimated Time: 20 mins.

Specification

You have become interested in gedit, a simple editor, and want to explore its design.

1. Use the ls command (with the appropriate command line switches) to obtain a recursive listing, including sizes (-s), of the contents of the /usr/share/gedit-2 directory. Redirect the command's output to a file in your home directory called lsgedit.txt. (For grading purposes, it is important that you specify the directory as an absolute reference, i.e., not share/gedit-2).
2. Make a (recursive) copy of the directory /usr/share/gedit-2 in your home directory. The copied directory should also be called gedit-2.

3. Make a second copy of the newly created directory gedit-2, called gedit-2.bak, again in your home directory.

4. From your backup copy, remove (recursively) the gedit-2.bak/plugins subdirectory.

5. Again from your backup copy, move the subdirectory gedit-2.bak/icons to your home directory. (The directory should still be named icons).

**Deliverables**

1. A file in your home directory called lsgedit.txt which contains the redirected output of a recursive listing, including sizes, of the /usr/share/gedit-2 directory.

2. A directory in your home directory called gedit-2, which is an exact copy of the directory /usr/share/gedit-2.

3. A directory in your home directory called gedit-2.bak, which starts as an exact copy of the directory gedit-2, with the subdirectory gedit-2/plugins removed, and the subdirectory gedit-2.bak/icons moved to your home directory.

4. A directory in your home directory called icons, which is a result of the previous deliverable.

**Cleaning Up**

The directories you have created are fairly significant in size, so you may want to conserve disk space by removing them when you are done.

**Questions**

1. Which command will list the contents of the directory report recursively?
   a. `lsdir report`
   b. `ls --recur report`
   c. `ls -r report`
   d. `ls -R report`
   e. `ls report \r`

The user hogan is executing the following command.

```
[hogan@station hogan]$ cp rep1.txt rep2.txt rep3.txt reps
```

2. For the command to succeed, which of the following statements must be true?
   a. The file rep1.txt must be a text file.
   b. The file rep1.txt must be an existing directory.
   c. The file reps must be an existing directory.
d. The files rep1.txt, rep2.txt, and rep3.txt must be existing directories.

e. The command cannot succeed, because the -r command line switch must be specified when using multiple arguments.

3. Which command creates a new, empty directory?
   a. md
   b. nd
   c. mkdir
   d. newdir
   e. touch -d

In his home directory, hogan has a directory called website, which contains various files and subdirectories, and a currently empty directory called archive, as illustrated in the following graph.

|-- archive/
   |-- website/
     |-- forms/
     |-- html/
     |   |-- index.html
     |   |-- page1.html
     |   |-- page2.html
     |   `-- page3.html
     `-- media/
       |-- photos/
         `-- picture1.png
       `-- songs/
         |-- song1.midi
         `-- song2.midi

For the following questions, assume that the current working directory is hogan’s home directory. /home/hogan.

4. What would be the effect of the following command? **cp -r website archive**
   a. The directory archive would be clobbered, and replaced with a copy of the website directory.
   b. The directory archive would contain a new subdirectory called website.
   c. An error, because the cp command does not work on directories.
   d. An error, because -r is not a legitimate switch for the cp command.
   e. None of the above

5. What would be the effect of the following command? **cp website archive/website.bak**
   a. The directory archive would contain a new subdirectory called website.bak.
   b. An error, because the directory archive/website.bak does not exist.
   c. An error, because the -r command line switch must be included when copying directories.
d. An error, because the correct command to use is `cpdir`.

e. None of the above

6. What would be the effect of the following command? `mv -r website archive`

a. The directory `archive` would be clobbered, and replaced with the `website` directory.

b. The directory `archive` would contain a new subdirectory called `website`.

c. An error, because the `mv` command does not work on directories.

d. An error, because the `mv` command does not recognize the `-r` command line switch.

e. None of the above

7. Which of the following commands would move the `website/media` directory and its subdirectories to `archive/media`?

a. `mv -r website/media archive/media`

b. `mv website/media archive/media`

c. `mv website/media archive`

d. Both B and C

e. All of the above

8. Which of the following commands would remove the `website/media` directory and its subdirectories?

a. `rmdir website/media`

b. `rm website/media`

c. `rm -r website/media`

d. Both A and C

e. All of the above

9. Which of the following commands would copy the `website` directory, and all of its subdirectories except `website/media/photos`?

a. `cp -x photos website archive`

b. `cp -d2 website archive`

c. `cp -x photos website archive/website`

d. `cp -d2 website archive/website`

e. None of the above

10. Why would someone be tempted to use `rmdir` instead of `rm -r`?

a. To ensure that overlooked files in the directory are not unintentionally erased.
b. Because the rmdir command works recursively, without needing to specify a command line switch.

c. The rmdir command creates an automatic backup of everything it removes.

d. Both B and C
Chapter 5. File Names and File Globbing

Key Concepts

• Filenames can contain almost any character except /.

• Although filenames can contain almost any character, that doesn't mean they should.

• Files that start . are "hidden" files.

• The *, ?, [...], and[^...] characters can be used to match filenames, through a process commonly called "file globbing".

Discussion

Filenames

Many operating systems restrict the choice and number of characters that can be used in naming files. In Linux, virtually any printable character can be used in a filename, and filenames can be of almost any length.

Linux filenames can be up to 255 characters long, excluding any directory components. When used in a command, an absolute or relative filename, including directory components, may be up to 4095 characters long! This allows names of files and directories to be very descriptive.

Linux filenames can contain any printable character (and some unprintable ones!) except for / (slash). Slash can't be part of a filename since it's the character used to separate directory name components in a relative or fully-qualified name. Since many of the more "unusual" characters are shell metacharacters, they must be protected in quotes to be used in a filename:

```
[julius@station julius]$ touch 'a and b'
[julius@station julius]$ touch '''
[julius@station julius]$ touch '!@#$%^&*()_+-|}{][>:;?.<,~`'
[julius@station julius]$ ls
!@#$%^&*()_+-|}{][>:;?.<,~`  "  a and b
[julius@station julius]$
```

As you can see, four files were created with rather strange names.

Warning

The preceding example is for illustrative purposes only. Many of the characters used in the example are shell meta-characters whose significance we have not discussed. You should avoid using punctuation in filenames since the uninformed use of shell meta-characters can produce disastrous results.

The warning given above should not be taken lightly. In general, filenames should be composed of alphabetic and numeric characters (A-Z, a-z, 0-9) and the punctuation symbols _,+- (dot, underscore, dash, plus, tilde). Normally, filenames should start with an alphanumeric character or a dot (see the section
below on Hidden Files). While you will encounter them with some frequency, filenames with embedded spaces are discouraged.

**Figure 5.1. "Safe" Filename Characters**

A–Z  a–z  0–9 .  _  ~

Here are two examples to show what the ill-advised use of otherwise-meaningful characters can do. We'll abuse two shell meta-characters: the space and greater-than characters.

**Example 1: Watch those spaces!**

```
[hogan@station hogan]$ mkdir bad dir ①
[hogan@station hogan]$ touch this and that ②
[hogan@station hogan]$ mv this and that bad dir ③
[hogan@station hogan]$ ls ④

dir
	hogan

① Hogan attempts to create a directory called 'bad dir' but unwittingly creates two directories, one named 'bad' and the other 'dir'.
② Next, Hogan tries to create a file named 'this and that' with `touch`, but creates three files instead, named 'this', 'and', and 'that'.
③ Now Hogan attempts to move the file he thinks is called 'this and that' to the directory he thinks is called 'bad dir' but instead moves all four of 'this', 'and', 'that', and 'bad' (three ordinary files and one directory) to the directory `dir`.
④ Because all of the above commands "succeeded," only now does Hogan realize something has gone wrong.
```

Of course, if Hogan had used quotes around the names as was done in the explanatory notes, things would have worked as he intended. This is worth remembering since you will encounter filenames with embedded spaces. On a related note, a filename with leading or trailing spaces is legal, but very tricky to identify in a file listing.

**Example 2: A little misdirection.**

```
[hogan@station hogan]$ mkdir bad->dir ①
[hogan@station hogan]$ touch this ②
[hogan@station hogan]$ mv this bad->dir ③
[hogan@station hogan]$ ls ④

bad- dir

① Hogan agrees to avoid spaces. This time he attempts to create a directory called 'bad->dir' but unwittingly creates a directory named 'bad-' and a file named 'dir' (why?).
② Next, Hogan creates a file named 'this' with `touch`. This step he gets right!
③ Now Hogan attempts to move the file called 'this' to the directory he thinks is called 'bad->dir' but instead moves 'this' to the directory 'bad-', redirecting the output (there is none) to the file 'dir'.
④ Again only now does Hogan realize something has gone wrong.
```

As before, if Hogan had used quotes around the names things would have worked as he intended. Quotes mask a multitude of sins, but don't use this as an excuse to create poor filenames. Many of the tasks we have yet to learn are made easier if the files involved have "normal" names.

**Hidden Files**

File and directory names (remember, a directory is a type of file) that start with a dot are "hidden" files. Hidden files do not appear in directory listings produced by `ls` unless the special command option `-a` (also
is used, or unless you specify the leading dot as part of the name. This makes it possible to reduce clutter and confusion by keeping certain files "out of sight, out of mind."

In the next section we will learn how to name multiple files through a technique known as "globbing." In general, hidden files also do not appear in a "glob" unless the leading dot was specifically listed.

Except for hiding the file, the leading dot has no other significance, and hidden files and directories can be used just like any others.

```
[alice@station alice]$ mkdir .secret
[alice@station alice]$ touch sample
[alice@station alice]$ ls
sample
[alice@station alice]$ ls -a
. .. sample .secret
[alice@station alice]$ mv sample .secret
[alice@station alice]$ ls
sample
[alice@station alice]$ ls .secret
sample
[alice@station .secret]$ ls
sample
[alice@station .secret]$ 
```

"Globbing"

Frequently it is necessary to issue a command that acts on more than one file. Commands like `cp -r` and `rm -r` work on entire directory sub-trees, but Linux has a more flexible way of identifying sets of files. The bash command shell treats some of its special meta-characters as wildcard characters. When bash reads the command line, the shell performs a process called meta-character expansion or wildcard expansion, generates a list of filenames that match the pattern described by the wildcard expression, and then passes the generated list on to the command. This is commonly referred to as "filename globbing."

**Table 5.1. Wildcard Characters**

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches zero or more characters (except leading dot)</td>
</tr>
<tr>
<td>?</td>
<td>matches exactly one character (except leading dot)</td>
</tr>
<tr>
<td>[...]</td>
<td>matches exactly one character from the list or range</td>
</tr>
<tr>
<td>[^...]</td>
<td>matches exactly one character not from the list or range</td>
</tr>
</tbody>
</table>

The use of * and ? is fairly straightforward. The list wildcards are a bit more complicated, but also quite powerful and useful.

The bracket wildcards represent a list of single characters. Thus, `[aeiou]` matches any one vowel. A contiguous range of characters may be represented using a dash, as in `[a-z]` for the lowercase alphabet. A leading carat (^) negates the list, so that `[^aeiou]` is any character except a vowel. A "real" dash or carat can be represented by escaping the characters using a backslash (\) (more on this later). Thus, the expression `[a\-z]` matches a, z or a dash only. Ranges and single characters can be mixed together. The pattern `[A-Za-z0-9._\-+~]` matches any one of the "safe" filename characters mentioned above in Figure 5-1.

Suppose the current directory contains the following files:

```
image1.jpg image10.jpeg pagel.html page3.htm script1.pl
image2.jpg image11.jpeg page2.html page40.htm
```
The following table illustrates the use of these wildcard characters.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Generates the list</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>all the files in the list</td>
</tr>
<tr>
<td>*.html</td>
<td>page1.html page2.html</td>
</tr>
<tr>
<td>page*.htm*</td>
<td>page1.html page2.html page3.htm page40.htm</td>
</tr>
<tr>
<td>image?.*</td>
<td>image1.jpeg image2.jpeg</td>
</tr>
<tr>
<td>[ps]*</td>
<td>page1.html page2.html page3.htm page40.htm script1.pl</td>
</tr>
<tr>
<td>[^ps]*</td>
<td>image1.jpeg image2.jpeg image10.jpeg image11.jpeg</td>
</tr>
</tbody>
</table>

Wildcards can be used with any command that expects filenames. For example, the command

```
[bob@station bob] mv *.htm public_html
```

is the same as the command

```
[bob@station bob] mv page3.htm page40.htm public_html
```

assuming Bob's home directory contains the files listed above.

[Technical note: Many commands behave differently when there are no files listed as opposed to when one or more files are given as part of the command. So what happens in such a command if a wildcard pattern is used, and the pattern does not match anything so that the list of files is empty? In the special case where the bash shell attempts to expand a wildcard expression and there is no match, it leaves the original expression as part of the command. This is illustrated in the following:]

```
[bob@station bob]$ rm
rm: too few arguments
Try `rm --help' for more information.
[bob@station bob]$ rm z*
rm: cannot lstat `z*': No such file or directory
```

In the first rm, the command saw zero filenames and failed with an error message. In the second version, the shell could not generate a list of names from the pattern z*, and so the string z* was sent to rm as the filename itself. Since Bob doesn't have a file named z* in his directory, the command failed with a different message.]

Examples

Finding That Configuration File

Alice was recently examining the DNS (Domain Name Service) configuration file, but has since forgotten its name. She remembers that the file ended .conf. In order to reduce the number of files she has to consider, she used file globbing to list all files in the /etc directory that end .conf.

```
[alice@station images]$ ls /etc/*.conf
/etc/aep.conf /etc/iftp.conf /etc/pnm2ppa.conf
/etc/aeplog.conf /etc/libuser.conf /etc/pwdb.conf
/etc/cdrecord.conf /etc/logrotate.conf /etc/resolv.conf
/etc/esd.conf /etc/lpd.conf /etc/rndc.conf
/etc/fam.conf /etc/ltrace.conf /etc/scrollkeeper.conf
/etc/gpm-root.conf /etc/modprobe.conf /etc/sysctl.conf
```

48
File Names and File Globbing

With a little more looking, she recognizes the `resolv.conf` configuration file.

Listing Static Libraries

Bob has been learning the difference between statically and dynamically linked libraries, that are used by programs. He knows that statically linked libraries conventionally have the name `libsomething.a`, where `something` is the name of the library. He also knows that libraries are generally found in the `/usr/lib` directory.

When he goes to see what static libraries are installed on his system, he finds nearly 1000 files in the `/usr/lib` directory. In order to pare down the number of files to examine, he tries listing all files that end in `a`.

```
[bob@station bob]$ ls /usr/lib/*a
/usr/lib/liba2ps.a  /usr/lib/libimlib-ps.a
/usr/lib/libacl.a  /usr/lib/libimlib-tiff.a
/usr/lib/libanl.a  /usr/lib/libimlib-xpm.a
...
/usr/lib/libimlib-jpeg.a  /usr/lib/libz.a
/usr/lib/libimlib-png.a  /usr/lib/libzvt.a
/usr/lib/libimlib-ppm.a
```

Many lines in this long listing have been deleted, but enough have been left to illustrate the problem, bob's "glob" was not specific enough, and caught some directories that ended in `a` as well. With a little more refinement, he produces the list he wants.

```
[bob@station bob]$ ls /usr/lib/lib*.a
/usr/lib/liba2ps.a  /usr/lib/libimlib-ps.a
/usr/lib/libacl.a  /usr/lib/libimlib-tiff.a
/usr/lib/libanl.a  /usr/lib/libimlib-xpm.a
...
/usr/lib/libimlib-jpeg.a  /usr/lib/libz.a
/usr/lib/libimlib-png.a  /usr/lib/libzvt.a
/usr/lib/libimlib-ppm.a
```

Next, bob suspects that the most commonly used libraries would have short, efficient (if not descriptive) names. He proceeds to list all of the libraries that have 1, 2, and 3 letter names.

```
[bob@station bob]$ ls /usr/lib/lib?.a
/usr/lib/libb.a  /usr/lib/libd.a  /usr/lib/libz.a
/usr/lib/libg.a  /usr/lib/libm.a
...
/usr/lib/libimlib-jpeg.a  /usr/lib/libz.a
/usr/lib/libimlib-png.a  /usr/lib/libzvt.a
/usr/lib/libimlib-ppm.a
```

```
[bob@station bob]$ ls /usr/lib/lib???.a
/usr/lib/libacl.a  /usr/lib/libgdk.a  /usr/lib/libosp.a  /usr/lib/librle.a
/usr/lib/libanl.a  /usr/lib/libgif.a  /usr/lib/libpam.a  /usr/lib/librpm.a
/usr/lib/libapm.a  /usr/lib/libgmp.a  /usr/lib/libpbm.a  /usr/lib/libSDL.a
```

```
[bob@station bob]$ ls /usr/lib/lib????.a
/usr/lib/libbacl.a  /usr/lib/libbgdk.a  /usr/lib/libbsp.a  /usr/lib/libbfa
/usr/lib/libbapm.a  /usr/lib/libbgmp.a  /usr/lib/libbpm.a  /usr/lib/libSDL.a
```

49
Is there any easy way, using file globbing, that Bob could have listed libraries with 1, 2, or 3 character names with one expression? No.

Listing Man Pages

The user Alice has discovered that the files that contain the man pages are installed into the directory `/usr/share/man`, in subdirectories that relate to the man page chapters.

```
[alice@station man]$ ls /usr/share/man/
cs de_DE fr hu it man1 man3 man5 man7 man9 pl ru sv
de es fr_FR id ja man2 man4 man6 man8 mann pt_BR sk
```

As a standard user of the system, she knows that she is most interested in chapters 1, 5, and 7. She uses the file globbing match expression to just list the contents of those directories.

```
[alice@station man]$ ls /usr/share/man/man[157]
```

```
/usr/share/man/man1:
  script.1.gz             pbmtolj.1.gz
  [i.1.gz                pbmtoln03.1.gz
  411toppm.1.gz           pbmtolps.1.gz
  a2p.1.gz               pbmtomacp.1.gz

  ...                  wireless.7.gz
  groff_char.7.gz        x25.7.gz
  groff_diff.7.gz
```

If Alice had been interested in all of the man pages in numbered chapters, but not the oddly named chapter "n", she could have used the file globbing range expression instead.

```
[alice@station man]$ ls /usr/share/man/man[1-9]
```

```
/usr/share/man/man1:
  script.1.gz             pbmtolj.1.gz
  [i.1.gz                pbmtoln03.1.gz
  411toppm.1.gz           pbmtolps.1.gz
  a2p.1.gz               pbmtomacp.1.gz

  ...                  wireless.7.gz
  named-checkconf.8.gz    zic.8.gz
  named-checkzone.8.gz
```

Online Exercises

Managing Files with File Globbing

Lab Exercise

**Objective:** Manage files efficiently with file globbing.

**Estimated Time:** 20 mins.
Specification

This lab will use file globbing to manage a large number of files that have loosely structured names. All of the files are found in the /usr/share/tcl8.5/encoding directory. To make your life easier, try to perform each of the following steps with a single command.

1. Create the following directories in your home directory: cp_even, cp_mid, cp_thousand, iso_mid, mac, and xxxn.

2. Examine the files in the /usr/share/tcl8.5/encoding directory. Copy all files that begin mac into the mac directory.

3. Copy all files which start cp and contain an even number into the directory cp_even.

4. Copy all files which start cp and contain a number greater than 1000 into the cp_thousand directory.

5. Copy all files which start cp and contain numbers greater than 800 but less than 1000 into the cp_mid directory.

6. Copy all files which start iso8859- with a number following the hyphen between 3 and 8 (inclusive) into the iso_mid directory.

7. Copy all files whose first four characters are composed of exactly three letters followed by a number into the directory xxxn.

Deliverables

1.

If the exercise has been performed correctly, you should have the following six directories, with the following files.

[bob@station bob]$ ls *

cp_even:
cp1250.enc cp1256.enc cp852.enc cp864.enc cp932.enc
cp1252.enc cp1258.enc cp860.enc cp866.enc cp936.enc
cp1254.enc cp850.enc cp862.enc cp874.enc cp950.enc

cp_mid:
cp850.enc cp857.enc cp862.enc cp865.enc cp874.enc cp949.enc
cp852.enc cp860.enc cp863.enc cp866.enc cp932.enc cp950.enc
cp855.enc cp861.enc cp864.enc cp869.enc cp936.enc

cp_thousand:
cp1250.enc cp1252.enc cp1254.enc cp1256.enc cp1258.enc

cp1251.enc cp1253.enc cp1255.enc cp1257.enc

iso_mid:
iso8859-3.enc iso8859-5.enc iso8859-7.enc
iso8859-4.enc iso8859-6.enc iso8859-8.enc

mac:
macCentEuro.enc macDingbats.enc macJapan.enc macThai.enc
macCroatian.enc macGreek.enc macRoman.enc macTurkish.enc
macCyrillic.enc macIceland.enc macRomania.enc macUkraine.enc

xxxn:
big5.enc iso8859-13.enc iso8859-2.enc iso8859-7.enc jis0212.enc
iso2022.enc iso8859-14.enc iso8859-3.enc iso8859-8.enc ko18-r.enc
iso2022-kr.enc iso8859-16.enc iso8859-5.enc jis0201.enc ksc5601.enc
iso8859-10.enc iso8859-1.enc iso8859-6.enc jis0208.enc
Suggestions

The following command would copy the files that meet the first criteria (assuming the shell's current working directory is bob's home directory).

[bob@station bob]$ cp /usr/share/tcl8.5/encoding/mac* mac/

Questions

Note

The following questions all end in an appropriate question mark, which should not be considered part of any specified commands.

Use the output from the following command to answer the next 5 questions.

[elvis@station files]$ ls
ebbs eras lakes lit loop olden rank renew robe run whirr
echo erect lamed lives lorry one rapid reply robed wagon whorl
elope ergo lash loads lost ounce rasp retry rock wares wile
enact erupt lead loath loves oust rays rho rocky ways windy
end evens leaks lobby lows ovary razor rigid rooms weak witty
enjoy evoke leaps locus loyal overt read rigs roses weep wools
entry ewes learn logic lucks racks reaps rime rouge wench wound
envoy lace leech login oaks radon recta riots rove west wrens
epic laced lick loin obese raged recur risks row wet writ

1. Which of the following files would be included in the output of the command **ls r*s**?
   a. eras
   b. raged
   c. rigs
   d. retry
   e. racks
   f. wrens

2. Which of the following files would be included in the output of the command **ls r??**?
   a. eras
   b. row
   c. riots
   d. rho
   e. wet
   f. run

3. Which of the following files would be included in the output of the command **ls [aeiou]??**?
File Names and File Globbing

a. ebbs
b. lead
c. oft
d. whip
e. oaks
f. oust

4. Which of the following files would be included in the output of the command `ls *[^aeiou]`?
   a. ebbs
   b. rove
   c. one
   d. rasp
   e. ways
   f. era

5. Which of the following files would be included in the output of the command `ls [aeiou]?e*`?
   a. ebbs
   b. rove
   c. one
   d. rasp
   e. ways
   f. rouge

Use the following recursive listing of the `/etc/X11/gdm` directory to answer the following questions.

```
[student@station student]$ ls -R /etc/X11/gdm/
/etc/X11/gdm/:
   factory-gdm.conf  gnomerc locale.alias PreSession  XKeepsCrashing
   gdm.conf          Init   PostSession  Sessions
/etc/X11/gdm/Init:
   Default
/etc/X11/gdm/PostSession:
   Default
/etc/X11/gdm/PreSession:
   Default
/etc/X11/gdm/Sessions:
   default Default Failsafe GNOME KDE
```

6. Which of the following files would be included in the output of the command `ls /etc/X11/gdm/S*/a*`?
a. /etc/X11/gdm/Init/Default
b. /etc/X11/gdm/locale.alias
c. /etc/X11/gdm/factory-gdm.conf
d. /etc/X11/gdm/Sessions/Default
e. /etc/X11/gdm/Sessions/Failsafe
f. /etc/X11/gdm/Sessions/default

7. Which of the following files would be included in the output of the command `ls /etc/X11/gdm/*/*[Ef]`?
   a. /etc/X11/gdm/gdm.conf
   b. /etc/X11/gdm/factory-gdm.conf
   c. /etc/X11/gdm/Init/Default
   d. /etc/X11/gdm/Sessions/GNOME
   e. /etc/X11/gdm/Sessions/KDE
   f. /etc/X11/gdm/Sessions/Failsafe

8. Which of the following files could be included in the output of the command `ls /etc/*.*??`?
   a. /etc/issue.net
   b. /etc/mail.rc
   c. /etc/php.ini
   d. /etc/huh.??
   e. /etc/xpdfrc.ja
   f. /etc/gss-1.0

9. Which of the following files could be included in the output of the command `ls /etc/[akp][0-9]*`?
   a. /etc/krb5.conf
   b. /etc/as5-0
   c. /etc/pbm2ppa.conf
   d. /etc/a2ps.cfg
   e. /etc/php.ini
   f. /etc/krc8

10. Which of the following files would be included in the output of the command `ls /etc/[aeiou][^aeiou][aeiou]`?
a. /etc/adjtime
b. /etc/issue
c. /etc/aeplog.conf
d. /etc/at.deny
e. /etc/at.allow
f. /etc/auto.tab
Chapter 6. Examining Files

Key Concepts

• The file command displays a file’s type
• The cat command displays a file’s contents
• The more and less commands "page" through a file, one screen full at a time.
• The head and tail commands display the first or last few lines of a file.
• In the X graphical environment, nautilus can be used to examine text files.

Discussion

Viewing Files

Linux offers several commands for examining files and file contents. This Lesson looks at some of the most commonly used of these tools.

The file Command

The contents of any given file might be ASCII (plain text, HTML, shell script, program source code, etc.) or binary (compiled executable, compressed archive, audio, etc.).

The commands discussed in this chapter are intended for use on ASCII files only, and attempting to use them with binary files can lead to problems ranging from the mildly irritating (screens full of strange character flashing by) to the more significant (locking the terminal display). This is because binary files can contain arbitrary binary codes, and some of these codes have special meaning when being (mis)interpreted as ASCII text. Therefore, it is a good idea to check the file type of any file you do not recognize using file before attempting to view the file with one of the other tools shown here.

file [OPTIONS] [FILE...]

Tests FILE(s) to determine the file type, and displays results to standard output.

Example of use with ASCII files:

[nero@station nero]$ file index.html novel.ps mailer
index.html:         HTML document text
novel.ps:           PostScript document text conforming at level 3.0
mailer:             perl script text executable

Example of use with binary files:

[nero@station nero]$ file logo.jpeg symphony.mp3 archive.tgz
logo.png:           PNG image data, 387 x 69, 8-bit grayscale, non-interlaced
symphony.mp3:       MP3, 128 bits, 44.1 kHz, JStereo

The cat Command

This command was first introduced in Chapter 3 of this workbook. Here we show some of its options.

cat [OPTIONS] [FILE...]
Examining Files

Concatenate FILE(s) to standard output.

<table>
<thead>
<tr>
<th>Switch</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A</td>
<td>Show all characters, including control characters and non-printing characters</td>
</tr>
<tr>
<td>-s</td>
<td>&quot;Squeeze&quot; multiple adjacent blank lines into a single blank line</td>
</tr>
<tr>
<td>-n</td>
<td>Number lines of output</td>
</tr>
</tbody>
</table>

The cat command, when used for viewing files, simply displays the contents all at once. Large files scroll by too quickly to read, making cat most suitable for files with less than a screen of text.

The more and less Pagers

Both more and less are designed for viewing text files on screen. Their use is quite similar, except that the more modern less command has a few extra features, such as responding correctly to [PgUp], [PgDn] and navigation arrow keystrokes correctly. After all, less is more.

It is especially important to be familiar with less because other tools (such as man) use it behind the scenes to provide paging capabilities.

more [OPTIONS] [FILE...] Displays FILE(s) to standard output one screen full at a time under keyboard control.

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Clear screen and redraw, instead of scrolling</td>
</tr>
<tr>
<td>-s</td>
<td>&quot;Squeeze&quot; multiple adjacent blank lines into a single blank line</td>
</tr>
</tbody>
</table>

less [OPTIONS] [FILE...] Displays FILE(s) to standard output one screen full at a time under keyboard control.

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Clear screen and redraw, instead of scrolling</td>
</tr>
<tr>
<td>-r</td>
<td>Display raw control characters</td>
</tr>
<tr>
<td>-s</td>
<td>&quot;Squeeze&quot; multiple adjacent blank lines into a single blank line</td>
</tr>
</tbody>
</table>

Table 6.1. less (more) navigation commands

<table>
<thead>
<tr>
<th>Command</th>
<th>more?</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;space&gt;</td>
<td>Yes</td>
<td>move ahead one full screen</td>
</tr>
<tr>
<td>[PgDn]</td>
<td>No</td>
<td>move ahead one full screen</td>
</tr>
<tr>
<td>b</td>
<td>Yes</td>
<td>move back one full screen</td>
</tr>
<tr>
<td>[PgUp]</td>
<td>No</td>
<td>move back one full screen</td>
</tr>
<tr>
<td>[DnArrow]</td>
<td>No</td>
<td>move ahead one line</td>
</tr>
<tr>
<td>[UpArrow]</td>
<td>No</td>
<td>move back one line</td>
</tr>
<tr>
<td>/text</td>
<td>Yes</td>
<td>search forward for text</td>
</tr>
<tr>
<td>?text</td>
<td>No</td>
<td>search backward for text</td>
</tr>
</tbody>
</table>
Examining Files

One feature of the less pager relies on a standard Unix concept called pipes. While pipes are officially discussed later, this feature of less is too useful and simple enough to use that it merits introducing now. A pipe acts like redirection, in that the output of a command is redirected somewhere other than a terminal. With redirection (using >), the output gets redirected to a specified file. When piping, the output of one command gets redirected into the input of another command. The bash shell uses the | character (usually found above the RETURN key) to construct pipes.

When running a command that produces a lot of output, the output of the command can be piped into less. Rather than getting dumped to a terminal, the output can be browsed like a file, paging down, paging up, and searching. When finished, quit less, and the output disappears.

The user elvis is using the ps aux command to view a table of all processes running on the machine. Because he expects the output of the command to be lengthy, he chooses to pipe the command into the less browser.

```
[elvis@station elvis]$ ps aux
USER    PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root    1  0.0  0.0  1392   92 ?        S    17:27   0:04 init [   
root    2  0.0  0.0     0    0 ?        SW   17:27   0:00 [keventd]
root    3  0.0  0.0     0    0 ?        SW   17:27   0:00 [kapmd]
root    4  0.0  0.0     0    0 ?        SWN  17:27   0:00 [ksoftirqd_CPU0]
root    9  0.0  0.0     0    0 ?        SWN  17:27   0:00 [bdflush]
...  
[elvis@station elvis]$ ps aux | less
```

Now he can use the same key sequences he uses when browsing a file (or man page) to browse the ps commands output: SPACE to page forward, b to page back, /text RETURN to search for the text text, and q to quit.

### The head Command

Sometimes all that is needed in examining files is to see the first few lines of the file. The head command provides this capability.

```
head [OPTIONS] [FILE...]
```

Print the first 10 lines of each FILE to standard output.

```
Switch         Effect
              
-<num>, -n <num>  Print first <num> lines (default is 10).
              
-q             Suppress headers giving file names
```

In the following example, elvis will examine the first 10, and then the first 5, lines of the /etc/group file.

```
[elvis@station elvis]$ head /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
...  
```
adm:x:4:root,adm,daemon
tty:x:5:
disk:x:6:root
lp:x:7:daemon,lp
mem:x:8:
kmem:x:9:

[elvis@station elvis]$ head -n 5 /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon

If given more than one filename as arguments, head displays the first few lines of each file individually, separated by a header displaying the file name. In the following, elvis lists the first few line of all of the Red Hat service scripts, found in the `/etc/rc.d/init.d` directory.

[elvis@station elvis]$ head -5 /etc/rc.d/init.d/*
==>/etc/rc.d/init.d/aep1000 ==>
#!/bin/sh
#
# Tags
# chkconfig: - 30 70
# description: load and unload AEP1000/AEP2000 coprocessor driver

==>/etc/rc.d/init.d/anacron ==>
#!/bin/sh
# Startup script for anacron
#
# chkconfig: 2345 95 05
# description: Run cron jobs that were left out due to downtime

==>/etc/rc.d/init.d/apmd ==>
#!/bin/sh

# chkconfig: 2345 26 74
# description: apmd is used for monitoring battery status and logging it via \ 
#       syslog(8). It can also be used for shutting down the machine when \

==>/etc/rc.d/init.d/atd ==>
#!/bin/bash
#
# /etc/rc.d/init.d/atd
#
# Starts the at daemon

...

As with the less pager, above, the output from long commands can be truncated by piping it into the head command.

[student@station student]$ ps aux | head -5
USER   PID  %CPU  %MEM  VSZ  RSS  TTY STAT  START   TIME COMMAND
root   1  0.0  0.1  2500  416  ?   S  10:51  0:03  init [5]
root   2  0.0  0.0   0   0   ?   SW  10:51  0:00  [keventd]
root   3  0.0  0.0   0   0   ?   SW  10:51  0:00  [kapmd]
root   4  0.0  0.0   0   0   ?   SWN 10:51  0:00  [ksoftirqd/0]

The tail Command

    tail [OPTIONS] [FILE...]

Complementing head, the tail command prints the last 10 lines of each FILE to standard output.
Examining Files

<table>
<thead>
<tr>
<th>Switch</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-num, -n num</td>
<td>Print last num lines (default is 10).</td>
</tr>
<tr>
<td>-q</td>
<td>Suppress headers giving file names</td>
</tr>
<tr>
<td>-f</td>
<td>Keep file open, and print new lines as appended</td>
</tr>
</tbody>
</table>

In the following, elvis prints the last 10, and then the last 5, lines of the /etc/group file.

```
[elvis@station elvis]$ tail /etc/group
named:x:25:
gdm:x:42:
postgres:x:26:
apache:x:48:
postdrop:x:90:
prefix:x:89:
squid:x:23:
webalizer:x:67:
ldap:x:55:
elvis:x:500:
[elvis@station elvis]$ tail -n 5 /etc/group
postfix:x:89:
squid:x:23:
webalizer:x:67:
ldap:x:55:
elvis:x:500:
```

The tail command has another, very useful, option: the -f (follow) option. With this option, tail will display the last lines of the file, and then "waits" and continues to display any new lines as they are added to the file. This is often used to monitor system log files in real time. Once started with this option, the command prompt does not return. Instead the tail -f stays active and will continue to display new lines as they occur until CTRL+C is pressed.

**Examples**

**Identifying Documents with file**

While exploring the cups printing system’s configuration, prince came across the following files in the /usr/share/cups/banners directory.

```
[prince@station banners]$ ls /usr/share/cups/banners/
classified  confidential  secret  standard  topsecret  unclassified
```

Unsure what types of files these were, he tries to identify them with the file command.

```
[prince@station banners]$ cd /usr/share/cups/banners/
[prince@station banners]$ file *
classified: PostScript document text conforming at level 3.0
confidential: PostScript document text conforming at level 3.0
secret: PostScript document text conforming at level 3.0
standard: PostScript document text conforming at level 3.0
topsecret: PostScript document text conforming at level 3.0
unclassified: PostScript document text conforming at level 3.0
```

As the file command reveals, these are PostScript documents, intended to be sent to the printer as banner pages.

**Determining the Correct File Type with file**

The user elvis has just downloaded some images from his friend’s photo album website, and stored them in a directory called pics.
Examining Files

He notices from the file extensions that 4 of the files seem to be in PNG format, while 2 of them seem to be JPEG format. Thinking this odd, he confirms the file types by running the file command.

```
[elvis@station images]$ file pics/*
pics/dscn0936.png: PNG image data, 360 x 480, 8-bit/color RGB, non-interlaced
pics/dscn0938.png: PNG image data, 640 x 480, 8-bit/color RGB, non-interlaced
pics/dscn0942.png: PNG image data, 360 x 480, 8-bit/color RGB, non-interlaced
pics/dscn0952.png: PNG image data, 360 x 480, 8-bit/color RGB, non-interlaced
pics/dscn0953.jpg: PNG image data, 640 x 480, 8-bit/color RGB, non-interlaced
pics/dscn0959.jpg: PNG image data, 640 x 480, 8-bit/color RGB, non-interlaced
```

The last two files are in the PNG format, like the other four. The last two files were apparently misnamed with the wrong extension.

Skimming Files with head

The user blondie has recently learned that all of the files in the /etc/profile.d directory that end .sh are included in her bash shell’s configuration every time she logs on. Curious about what the files contain, she skims the first few lines of each file with the following single command.

```
[blondie@station blondie]$ head /etc/profile.d/*.sh

==>/etc/profile.d/glib2.sh==>
## This caused GLib2 applications to convert filenames from
## locale encoding to UTF-8. If the locale encoding is already
## UTF-8 then it makes no difference.
export G_BROKEN_FILENAMES=1
==>/etc/profile.d/gnome-ssh-askpass.sh==>
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
export SSH_ASKPASS
==>/etc/profile.d/jdk.sh==>
export PATH=/usr/local/j2sdk1.3.1/bin:$PATH
==>/etc/profile.d/lang.sh==>
# /etc/profile.d/lang.sh - set i18n stuff
sourced=0
for langfile in /etc/sysconfig/i18n $HOME/.i18n ; do
  [ -f $langfile ] && . $langfile && sourced=1
done
if [ -n "$GDM_LANG" ]; then
  sourced=1
  LANG="$GDM_LANG"
...
```

Combining Multiple Files into a Single File with head

The user madonna is learning more about PAM, Linux’s Pluggable Authentication Modules. She has discovered the /usr/share/doc/pam-0.75/txts directory, which contains a large number of README files.

```
[madonna@station madonna]$ cd /usr/share/doc/pam-0.75/txts/
[madonna@station txts]$ ls
pam_appl.txt README.pam_ftp README.pam_shells
```
Examining Files

Rather than deal with each file individually, she would like to combine them into a single file which she can browse easily. She knows she could use the `cat` command, and redirect the output to a file, but the she would no longer know the filenames associated with each of the files. Instead, she decides to use the `head` command, giving a ludicrously large number of lines to select. That way, she will be able to browse the file contents, with an identifying banner giving the file name.

```
[madonna@station txts]$ head -99999 README* > /tmp/pam_READMEs.txt

[madonna@station txts]$ less /tmp/pam_READMEs.txt

==> README <==

$Id: 010_text.dbk,v 1.1.1.1 2003/08/11 13:18:31 bowe Exp $

This is a directory for text versions of the pam documentation

==> README.pam_access <==
# Description of its configuration file
#
# (The default config file is "/etc/security/access.conf". This
# default can be overridden with a module config argument
#

Monitoring Multiple Files with tail

In Red Hat Enterprise Linux, system events get logged into text files found in the `/var/log` directory, such as `/var/log/messages` (for general system events) and `/var/log/secure` (for events which entail sensitive information). As events occur, new lines get appended to these files in real time.

Suppose a system administrator would like to monitor both of these log files, and see new information as soon as it arrives. She uses the `tail -f` command to “follow” the ends of both files. The `tail` command immediately displays the last 10 lines of both files, decorated by filename.

```
[root@station root]# cd /var/log
[root@station log]# tail -f messages secure

==> messages <==

Jun 25 11:20:59 station kernel: 11: @cf13e4c0 length 800000de status 800100de
Jun 25 11:20:59 station kernel: 12: @cf13e500 length 800000de status 000100de
Jun 25 11:20:59 station kernel: 13: @cf13e540 length 80000042 status 00010042
Jun 25 11:20:59 station kernel: 14: @cf13e580 length 800005ea status 000105ea
Jun 25 11:20:59 station kernel: 15: @cf13e5c0 length 8000031e status 0001031e
Jun 25 11:22:06 station su[pam_unix][2229]: session closed for user madonna
Jun 25 11:22:10 station su[pam_unix][2352]: session opened for user root by elvis(uid=2291)
Jun 25 11:23:06 station su[pam_unix][2399]: session opened for user root by elvis(uid=2291)
Jun 25 11:23:55 station su[pam_unix][2399]: session closed for user root
Jun 25 11:24:00 station su[pam_unix][2441]: session opened for user elvis by elvis(uid=2291)

==> secure <==

Jun 24 15:32:15 station sshd[11535]: Accepted publickey for root from 127.0.0.1 port 36253 ssh2
Jun 24 15:32:15 station sshd[11545]: Accepted publickey for root from 127.0.0.1 port 36254 ssh2
Jun 24 15:32:16 station sshd[11555]: Accepted publickey for root from 127.0.0.1 port 36256 ssh2
```
At this point, the **`tail`** command is still running, but the output pauses until a new system event occurs which gets logged to one of the files. The **`tail`** command then shows the newly appended lines, again decorated with the filename of the relevant file.

--- messages ---
Jun 25 11:24:54 station su(pam_unix)[2481]: session opened for user root by elvis (uid=501)
Jun 25 11:25:02 station su(pam_unix)[2481]: session closed for user root

Finally, when the administrator is finished monitoring the files, she cancels the **`tail`** command with the **CTRL+C** control sequence.

## Online Exercises

### Browsing Text Files

**Lab Exercise**

**Objective:** Examine files using several browsing utilities.

**Estimated Time:** 15 mins.

**Specification**

This Lab Exercise will require three terminals (either virtual consoles, or terminal windows within the X graphical environment), each using your primary account.

1. In the first terminal, **`cat`** the contents of the file `/etc/services`, including the correct switch to number all lines. After you have viewed the results, repeat the command (with the same switch), redirecting the output into the file `services.cat.txt` in your home directory.

2. While still in the first terminal, take a recursive listing of all files with the **`ls -R`** command, and "catch" the output (using a pipe) with the **`less`** pager. Use the basic pager commands, such as **`SPACE`** and **`b`**, to browse the output. Explore other pager commands by using the **`h`** command to open a help screen.

When you are finished, **DO NOT** exit the **`less`** pager, but move to a new terminal for the next step.

3. In a second terminal, Use the **`head`** command to collect the first 5 lines of all files in the `/etc/sysconfig` directory whose filename starts `system-` into the file `confheaders.txt` in your home directory. Use the appropriate switch to suppress the filename headers that **`head`** prints by default.

4. While still in the second terminal, run the command **`vmstat 1`**. The command tracks memory and CPU statistics about your machine, generating a new line once per second. When you have seen enough, use the **CTRL+C** control sequence to kill the command.

Run the command again, this time redirecting output to the file `vmstat.out` in your home directory, as in the following.

```
[elvis@station elvis]$ vmstat 1 > vmstat.out
```
Examining Files

DO NOT stop this command, but leave it running. Move to a new terminal for the next step.

5. In a third terminal, follow the growing `vmstat.out` file with the `tail -f` command. DO NOT stop the `tail` command until you have confirmed your exercise below.

Deliverables

1. A file in your home directory called `services.cat.txt`, which contains the contents of the `/etc/services` file, with line numbers as produced by the `cat` command.

2. A running `less` pager which is using a pipe to browse the output of the `ls -R /` command.

3. The file `confheaders.txt` in your home directory, that contains first five lines of every file in the `/etc/sysconfig` that begins with `redhat-`, as produced by the `head` command, suppressing filenames (with the appropriate command line switch(es)).

4. A running `tail` command, which is following the growing file `vmstat.out` in your home directory.

Cleaning Up

After completing this exercise, quit the `less` pager with the `q` command, and terminate the `vmstat` and `tail` commands with a `CTRL+C` control sequence.

Questions

Use the `file` command to help answer the following 2 questions.

1. What type of file is `/usr/bin/run-parts`?
   a. A compiled ELF Executable
   b. An Awk script
   c. A Bash ( Bourne-Again) shell script
   d. A Symbolic Link
   e. A `/usr/bin/perl` script

2. What type of file is `/dev/log`?
   a. A character special file
   b. A block special file
   c. A socket
   d. A named pipe
   e. A compiled ELF executable

3. Which of the following commands would act the same as `cp a.txt b.txt`, assuming the file `b.txt` does not exist?
Examining Files

a. `cat -5 a.txt b.txt`
b. `cat a.txt b.txt`
c. `cat -A a.txt > b.txt`
d. `cat a.txt > b.txt`
e. None of the above.

4. Which of the following key sequences can be used to quit the less pager?

a. `q`
b. `x`
c. `CTRL+C`
d. Both A and C
e. All of the above

5. While reading the file `/etc/services`, which of the following is an advantage of the less pager over the more pager?

a. less allows the user to browse output one page at a time, while more does not.
b. less recognizes familiar navigation keys, such as UpArrow and PgDown.
c. less allows users to page backwards through the file, while more only allows users to page forwards.
d. less allows the user to quit with a simple q, while more requires a CTRL+C
e. All of the above

6. Which of the following commands would display the first 5 lines of the file `/etc/passwd`?

a. `head -5 /etc/passwd`
b. `head -n /etc/passwd`
c. `head --five /etc/passwd`
d. `head /etc/passwd > 5`
e. `head /5 /etc/passwd`

7. Why would you expect trouble from the following command?

```
[bob@station bob]$ head /bin/ls
```

a. The user bob does not have permissions to read the file `/bin/ls`.
b. The head command must be called with the `-n` command line switch, to specify how many lines to show.
c. The file `/bin/ls` is too small for the head command
d. The file /bin/ls is a binary file, and the head command works primarily on text files.

e. None of the above.

8. How can a user stop the tail -f command?

a. There is no need to, as the command times out after 10 seconds.

b. By typing the q key.

c. With the CTRL+Q control sequence.

d. By switching virtual consoles.

e. None of the above.

Use the following output to help in answering the next two questions.

```
[alice@station alice]$ cat -An /etc/fstab
1  LABEL=/           /                       ext3    defaults 1 1$
2  LABEL=/boot       /boot                   ext3    defaults 1 2$
3  none              /dev/pts                devpts  gid=5,mode=620 0 0$
4  none              /proc                   proc    defaults 0 0$
5  none              /dev/shm                tmpfs   defaults 0 0$
6  /dev/hda2         swap                    swap    defaults 0 0$
```

9. Which of the following is NOT true?

a. The /etc/fstab file is 6 lines long.

b. The columns in the /etc/fstab are separated by tabs instead of spaces.

c. The /etc/fstab file is a text, instead of binary, file.

d. The commands head -8 /etc/fstab and tail -6 /etc/fstab would produce identical output.

e. None of the Above.

10. How would the output of the commands cat /etc/fstab /etc/fstab and head /etc/ fstab /etc/fstab differ?

a. The first command would only list the contents of the /etc/fstab file once.

b. The first command would display tabs as visible characters.

c. The second command would only show the first 10 lines of the duplicated /etc/fstab file.

d. The second command would include banners identifying the filename in the output.

e. None of the Above.
Chapter 7. Editing Files

Key Concepts

- A text editor is not a word processor.
- Linux ships with many text editors
- **nano** is a commonly used command line text editor.
- **gedit** is a commonly used command graphical text editor.

Discussion

Text Editor vs. Word Processor

One of the ways Linux achieves its high degree of flexibility is through the robustness of its configuration. Configuration information typically is stored in plain ASCII text files. System administration often involves updating or correcting these configuration files. The tool for doing this work is a text editor.

A text editor is not the same as a word processor. In addition to the words of a document, word processors allow the user to specify fonts, tab settings, margins, justification, and many other features designed to affect the way a document appears on paper. Documents generated by word processors contain embedded binary codes that specify the settings for these extra features. A plain text file such as a configuration file does not contain such codes, and is not designed for printing on paper. A text editor is designed just for modifying the text content of a plain ASCII file, and does not contain any embedded binary codes.

A word processor would be an appropriate choice for writing a letter, a memo, a bulletin, an advertising flier, or any document where the visual presentation is controlled by the editing software.

A text editor is the appropriate choice for creating or modifying configuration files, program source code, system documentation, or any document (such as an HTML document) where the visual presentation is controlled by the viewing software (such as a web browser) and not the editing software. Since this course focuses on Linux system administration, text editors will be our tool of choice for most situations.

Text Editors in Linux

Linux comes complete with a full suite of text editors, each with specific strengths and weaknesses. Linux also includes word processors for document creation as part of products such as OpenOffice.

Since all text editors work on simple text files, you may choose any one you like - there are no problems of compatibility between one product and another. Normally, you make your choice on the basis of availability, ease of use, familiarity, and special features. Some editors, for example, make it easy to generate correctly-formatted program code, making programming easier and faster. This chapter presents two easy-to-use text editors, **nano** and **gedit**. The most popular text editor on Linux systems, **vi**, is also the most robust and the most complicated. The complexity of this editor makes it difficult to address in an introductory course.

**nano**

**nano** is a simple, display-oriented (full screen) text editor. Commands are displayed at the bottom of the screen, and context-sensitive help is provided. As characters are typed they are immediately inserted into the text.
Usage:

```
nano [OPTIONS] [FILE]
```

Opens FILE in insert mode for editing.

As mentioned above, nano commands are displayed at the bottom of the screen throughout the edit session, making recalling editor commands very easy. The primary nano commands are summarized below. In this summary, the carat symbol followed by a letter is used to represent Ctrl-key combinations, so that ^K represents the CTRL+K key combination.

### Table 7.1. nano commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>^G</td>
<td>Get Help - opens a small help screen</td>
</tr>
<tr>
<td>^X</td>
<td>Exit - exits (and optionally saves file)</td>
</tr>
<tr>
<td>^O</td>
<td>Write Out - saves file</td>
</tr>
<tr>
<td>^J</td>
<td>Justify</td>
</tr>
<tr>
<td>^R</td>
<td>Read File - and insert at cursor position</td>
</tr>
<tr>
<td>^W</td>
<td>Where is - search for text</td>
</tr>
<tr>
<td>^Y</td>
<td>Previous Page - move up one page</td>
</tr>
<tr>
<td>^V</td>
<td>Next Page - move down one page</td>
</tr>
<tr>
<td>^K</td>
<td>Cut Text - cuts one line at a time</td>
</tr>
<tr>
<td>^U</td>
<td>UnCut Text - inserts last cut block of text</td>
</tr>
<tr>
<td>^C</td>
<td>Cursor Position - identifies line and column number</td>
</tr>
<tr>
<td>^T</td>
<td>To Spell - starts spell check</td>
</tr>
</tbody>
</table>

### Table 7.2. Selected nano Command Line Switches

<table>
<thead>
<tr>
<th>Switch</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-w</td>
<td>disables word wrap</td>
</tr>
<tr>
<td>-v</td>
<td>&quot;view&quot; file in read-only mode</td>
</tr>
</tbody>
</table>

System configuration files can be damaged by inappropriate line breaks, so nano is frequently started with the -w option when used to edit these files.

**gedit**

gedit is a graphical text editor available for the X-window environment. It can be accessed by entering gedit at a command prompt in a terminal window, or by selecting Accessories -> Text Editor from the GNOME Menu. It provides a familiar document window interface for creating and editing text files, and toolbar icons for accessing such features as search and replace, cut-copy-paste, and printing. Its tabbed window format allows for more than one document to be open at a time. It also supports a standard click-to-browse interface for interacting with the filesystem.
Online Exercises

Text Editors

Lab Exercise

Objective: Text Editors

Estimated Time: 10 mins.

Specification

The /etc/services file is an example of a Linux ASCII text configuration file. This lab will have you make two copies of this file, and edit one using the nano text editor, and edit the other using the gedit graphical text editor. If your system does not have the X server installed, use the nano editor for both files.

The nano Editor

1. In your home directory, create a copy of the file /etc/services, called services.nano

2. Open the file services.nano using the nano text editor.

3. Using the control sequences specified at the bottom of the screen, search for, and remove, all occurrences of the + character. Preserve the spacing of the columns as you do so. Save your file as services.noplus.

4. Continuing in the same editing session, move your cursor to line number 145. The CTRL+C key combination will report your current line number. (Note that CTRL+C, which usually terminates a process, has been overridden by the nano editor, and has new behavior). Delete that line, and the following line. Save your file as services.nonext.

5. Close the nano editor.

The gedit Editor
1. In your home directory, create a copy of the file `/etc/services`, called `services.gedit`

2. Open the file `services.gedit` using the `gedit` text editor.

3. Move your cursor to line number 104. (A little exploration through `gedit`'s menus may simplify this task). Delete that line, and the following 2. Save your file as `services.nosmux`.

4. Replace every occurrence of the letter `tcp` with `udp`. Make your search and replace case sensitive (don't replace `TCP`, for instance). Save your file as `services.notcp`.

5. Close the `gedit` editor.

**Deliverables**

1. All files are to be created in your home directory.

   1. A file named `services.noplus`, which is a duplicate of the file `/etc/services`, with + characters removed.

   2. A file named `services.nonext`, which, in addition to the previous edit, has lines number 145 and 146 removed.

   3. A file named `services.nosmux`, which is a duplicate of the file `/etc/services`, with lines numbered 104, 105, and 106 removed.

   4. A file named `services.notcp`, which in addition to the previous edit, has all occurrences of the characters `tcp` replaced with `udp`. 