LPI exam 101 prep: GNU and UNIX commands

Junior Level Administration (LPIC-1) topic 103

Skill Level: Introductory

Ian Shields (ishields@us.ibm.com)
Senior Programmer
IBM

15 Nov 2005

In this tutorial, Ian Shields continues preparing you to take the Linux Professional Institute® Junior Level Administration (LPIC-1) Exam 101. In this third of five tutorials, Ian introduces you to the Linux® command line and several GNU and UNIX commands. By the end of this tutorial, you will be comfortable using commands on a Linux system.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The Linux Professional Institute (LPI) certifies Linux system administrators at two levels: *junior level* (also called "certification level 1") and *intermediate level* (also called "certification level 2"). To attain certification level 1, you must pass exams 101 and 102; to attain certification level 2, you must pass exams 201 and 202.

developerWorks offers tutorials to help you prepare for each of the four exams. Each exam covers several topics, and each topic has a corresponding self-study tutorial on developerWorks. For LPI exam 101, the five topics and corresponding developerWorks tutorials are:

Table 1. LPI exam 101: Tutorials and topics

LPI exam 101 topic	developerWorks tutorial	Tutorial summary
Topic 101	LPI exam 101 prep (topic	Learn to configure your

	101): Hardware and architecture	system hardware with Linux. By the end of this tutorial, you will know how Linux configures the hardware found on a modern PC and where to look if you have problems.
Topic 102	LPI exam 101 prep: Linux installation and package management	Get an introduction to Linux installation and package management. By the end of this tutorial, you will know how Linux uses disk partitions, how Linux boots, and how to install and manage software packages.
Topic 103	LPI exam 101 prep: GNU and UNIX commands	(This tutorial). Get an introduction to common GNU and UNIX commands. By the end of this tutorial, you will know how to use commands in the bash shell, including how to use text processing commands and filters, how to search files and directories, and how to manage processes.
Topic 104	LPI exam 104 prep: Linux, filesystems, and FHS	Coming soon
Topic 110	LPI exam 110 prep: The X Window system	Coming soon

To pass exams 101 and 102 (and attain certification level 1), you should be able to:

- Work at the Linux command line
- Perform easy maintenance tasks: help out users, add users to a larger system, back up and restore, and shut down and reboot
- Install and configure a workstation (including X) and connect it to a LAN, or connect a stand-alone PC via modem to the Internet

To continue preparing for certification level 1, see the developerWorks tutorials for LPI exam 101. Read more about the entire set of developerWorks LPI tutorials.

The Linux Professional Institute does not endorse any third-party exam preparation material or techniques in particular. For details, please contact info@lpi.org.

About this tutorial

Welcome to "GNU and UNIX commands", the third of five tutorials designed to prepare you for LPI exam 101. In this tutorial, you learn how to use GNU and UNIX

commands.

is organized according to the LPI objectives for this topic. Very roughly, expect more questions on the exam for objectives with higher weight.

Table 2. GNU and UNIX commands: Exam objectives covered in this tutorial

LPI exam objective	Objective weight	Objective summary
1.103.1 Work on the command line	Weight 5	Interact with shells and commands using the command line. This objective includes typing valid commands and command sequences, defining, referencing and exporting environment variables, using command history and editing facilities, invoking commands in the path and outside the path, using command substitution, applying commands recursively through a directory tree and using man to find out about commands.
1.103.2 Process text streams using filters	Weight 6	Apply filters to text streams. This objective includes sending text files and output streams through text utility filters to modify the output, using standard UNIX commands found in the GNU textutils package.
1.103.3 Perform basic file management	Weight 3	Use the basic UNIX commands to copy, move, and remove files and directories. Tasks include advanced file management operations such as copying multiple files recursively, removing directories recursively, and moving files that meet a wildcard pattern. This objective includes using simple and advanced wildcard specifications to refer to files, as well as using find to locate and act on files based on type, size, or time.
1.103.4 Use streams, pipes, and redirects	Weight 5	Redirect streams and connect them in order to efficiently process textual data. Tasks include redirecting standard input, standard output, and standard error, piping the output of one command to the input of another command, using the output of one

		command as arguments to another command, and sending output to both stdout and a file.
1.103.5 Create, monitor, and kill processes	Weight 5	Manage processes. This objective includes knowing how to run jobs in the foreground and background, bringing a job from the background to the foreground and vice versa, starting a process that will run without being connected to a terminal, and signaling a program to continue running after logout. Tasks also include monitoring active processes, selecting and sorting processes for display, sending signals to processes, killing processes, and identifying and killing X applications that did not terminate after the X session closed.
1.103.6 Modify process execution priorities	Weight 3	Manage process execution priorities. Tasks include running a program with higher or lower priority, determining the priority of a process, and changing the priority of a running process.
1.103.7 Search text files using regular expressions	Weight 3	Manipulate files and text data using regular expressions. This objective includes creating simple regular expressions containing several notational elements. It also includes using regular expression tools to perform searches through a filesystem or file content.
1.103.8 Perform basic file editing operations using vi	Weight 1	Edit text files using vi. This objective includes vi navigation, basic vi nodes, inserting, editing, deleting, copying, and finding text.

Prerequisites

To get the most from this tutorial, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this tutorial. Sometimes different versions of a program will format output differently, so your results may not always look exactly like the listings and figures in this tutorial.

Section 2. Using the command line

This section covers material for topic 1.103.1 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 5.

In this section, you learn about the following topics:

- Interacting with shells and commands using the command line
- Valid commands and command sequences
- Defining, referencing, and exporting environment variables
- Command history and editing facilities
- Invoking commands in the path and outside the path
- Using command substitution
- Applying commands recursively through a directory tree
- Using man (manual) pages to find out about commands

This section gives you a brief introduction to some of the major features of the bash shell, with an emphasis on the features that are important for certification. But the shell is a very rich environment, and we encourage you to explore it further. Many excellent books are devoted to UNIX and Linux shells and the bash shell in particular.

The bash shell

The *bash* shell is one of several shells available for Linux. It is also called the *B*ourne-*a*gain *sh*ell, after Stephen Bourne, the creator of an earlier shell (*/bin/sh*). Bash is substantially compatible with sh, but provides many improvements in both function and programming capability. It incorporates features from the Korn shell (ksh) and C shell (csh), and is intended to be a POSIX-compliant shell.

Before we delve deeper into bash, recall that a *shell* is a program that accepts and executes commands. It also supports programming constructs allowing complex commands to be built from smaller parts. These complex commands, or *scripts*, can be saved as files to become new commands in their own right. Indeed, many commands on a typical Linux system **are** scripts.

Shells have some *builtin* commands, such as `cd`, `break`, and `exec`. Other commands are *external*.

Shells also use three standard I/O *streams*:

- *stdin* is the *standard input stream*, which provides input to commands.
- *stdout* is the *standard output stream*, which displays output from commands.
- *stderr* is the *standard error stream*, which displays error output from commands.

Input streams provide input to programs, usually from terminal keystrokes. Output streams print text characters, usually to the terminal. The terminal was originally an ASCII typewriter or display terminal, but it is now more often a window on a graphical desktop. More detail on how to redirect these standard I/O streams is covered in a later section of this tutorial, Streams, pipes, and redirects. The rest of this section focuses on redirection at a high level.

For the rest of this tutorial, we will assume you know how to get a shell prompt. If you don't, the developerWorks article "Basic tasks for new Linux developers shows you how to do this and more.

If you are using a Linux system without a graphical desktop, or if you open a terminal window on a graphical desktop, you will be greeted by a prompt, perhaps like one shown in Listing 1.

Listing 1. Some typical user prompts

```
[db2inst1@echidna db2inst1]$
ian@lyrebird:~>
$
```

If you log in as the root user (or superuser), your prompt may look like one shown in Listing 2.

Listing 2. Superuser, or root, prompt examples

```
[root@echidna root]#
lyrebird:~ #
#
```

The root user has considerable power, so use it with caution. When you have root privileges, most prompts include a trailing pound sign (#). Ordinary user privileges are usually delineated by a different character, commonly a dollar sign ($). Your actual prompt may look different than the examples in this tutorial. Your prompt may include your user name, hostname, current directory, date or time that the prompt was printed, and so on.

Conventions in this tutorial

These developerWorks tutorials for the LPI 101 and 102 exams include code examples that are cut and pasted from real Linux systems using the default prompts for those systems. Our root prompts have a trailing #, so you can distinguish them from ordinary user prompts, which have a trailing $. This convention is consistent

with many books on the subject. Note the prompt carefully in any examples.

Commands and sequences

So now that you have a prompt, let's look at what you can do with it. The shell's main function is to interpret your commands so you can interact with your Linux system. On Linux (and UNIX) systems, commands have a *command name*, and then *options* and *parameters*. Some commands have neither options nor parameters, and some have options but no parameters, while others have no options but do have parameters.

If a line contains a # character, then all remaining characters on the line are ignored. So a # character may indicate a comment as well as a root prompt. Which it is should be evident from the context.

Echo

The `echo` command prints (or echos) its arguments to the terminal as shown in Listing 3.

Listing 3. Echo examples

```
[ian@echidna ian]$ echo Word
Word
[ian@echidna ian]$ echo A phrase
A phrase
[ian@echidna ian]$ echo Where     are   my   spaces?
Where are my spaces?
[ian@echidna ian]$ echo "Here     are   my   spaces." # plus comment
Here     are   my   spaces.
```

In third example of Listing 3, all the extra spaces were compressed down to single spaces in the output. To avoid this, you need to *quote* strings, using either double quotes (") or single quotes ('). Bash uses *white space*, such as blanks, tabs, and new line characters, to separate your input line into *tokens*, which are then passed to your command. Quoting strings preserves additional white space and makes the whole string a single token. In the example above, each token after the command name is a parameter, so we have respectively 1, 2, 4, and 1 parameters.

The echo command has a couple of options. Normally echo will append a trailing new line character to the output. Use the `-n` option to suppress this. Use the `-e` option to enable certain backslash escaped characters to have special meaning. Some of these are shown in Table 3.

Table 3. Echo and escaped characters

Escape sequence	Function
\a	Alert (bell)
\b	Backspace
\c	Suppress trailing newline (same function as -n option)

\f	Form feed (clear the screen on a video display)
\n	New line
\r	Carriage return
\t	Horizontal tab

Escapes and line continuation

There is a small problem with using backslashes in bash. When the backslash character (\) is not quoted, it serves as an escape to signal bash itself to preserve the literal meaning of the following character. This is necessary for special shell metacharacters, which we'll cover in a moment. There is one exception to this rule: a backslash followed by a newline causes bash to swallow both characters and treat the sequence as a line continuation request. This can be handy to break long lines, particularly in shell scripts.

For the sequences described above to be properly handled by the `echo` command or one of the many other commands that use similar escaped control characters, you must include the escape sequences in quotes, or as part of a quoted string, unless you use a second backslash to have the shell preserve one for the command. Listing 4 shows some examples of the various uses of \.

Listing 4. More echo examples

```
[ian@echidna ian]$ echo -n No new line
No new line[ian@echidna ian]$ echo -e "No new line\c"
No new line[ian@echidna ian]$ echo "A line with a typed
> return"
A line with a typed
return
[ian@echidna ian]$ echo -e "A line with an escaped\nreturn"
A line with an escaped
return
[ian@echidna ian]$ echo "A line with an escaped\nreturn but no -e option"
A line with an escaped\nreturn but no -e option
[ian@echidna ian]$ echo -e Doubly escaped\\n\\tmetacharacters
Doubly escaped
        metacharacters
[ian@echidna ian]$ echo Backslash \
> followed by newline \
> serves as line continuation.
Backslash followed by newline serves as line continuation.
```

Note that bash displays a special prompt (>) when you type a line with unmatched quotes. Your input string continues onto a second line and includes the new line character.

Bash shell metacharacters and control operators

Bash has several *metacharacters*, which, when not quoted, also serve to separate input into words. Besides a blank, these are '|', '&', ';', '(', ')', '<', and '>'. We will discuss some of these in more detail in other sections of this tutorial. For now, note that if you want to include a metacharacter as part of your text, it must be either quoted or escaped using a backslash (\) as shown in Listing 4.

The new line and certain metacharacters or pairs of metacharacters also serve as *control operators*. These are '||', '&&', '&', ';', ';;', '|" '(', and ')'. Some of these control operators allow you to create *sequences* or *lists* of commands.

The simplest command sequence is just two commands separated by a semicolon (;). Each command is executed in sequence. In any programmable environment, commands return an indication of success or failure; Linux commands usually return a zero value for success and a non-zero in the event of failure. You can introduce some conditional processing into your list using the && and || control operators. If you separate two commands with the control operator && then the second is executed if and only if the first returns an exit value of zero. If you separate the commands with ||, then the second one is executed only if the first one returns a non-zero exit code. Listing 5 shows some command sequences using the echo command. These aren't very exciting since echo returns 0, but you will see more examples later when we have a few more commands to use.

Listing 5. Command sequences

```
[ian@echidna ian]$ echo line 1;echo line 2; echo line 3
line 1
line 2
line 3
[ian@echidna ian]$ echo line 1&&echo line 2&&echo line 3
line 1
line 2
line 3
[ian@echidna ian]$ echo line 1||echo line 2; echo line 3
line 1
line 3
```

Exit

You can terminate a shell using the `exit` command. You may optionally give an exit code as a parameter. If you are running your shell in a terminal window on a graphical desktop, your window will close. Similarly, if you have connected to a remote system using ssh or telnet (for example), your connection will end. In the bash shell, you can also hold the **Ctrl** key and press the **d** key to exit.

Let's look at another control operator. If you enclose a command or a command list in parentheses, then the command or sequence is executed in a sub shell, so the exit command exits the sub shell rather than exiting the shell you are working in. Listing 6 shows a simple example in conjunction with && and ||.

Listing 6. Subshells and sequences

```
[ian@echidna ian]$ (echo In subshell; exit 0) && echo OK || echo Bad exit
In subshell
OK
[ian@echidna ian]$ (echo In subshell; exit 4) && echo OK || echo Bad exit
In subshell
Bad exit
```

Stay tuned for more command sequences to come in this tutorial.

Environment variables

When you are running in a bash shell, many things constitute your *environment*, such as the form of your prompt, your home directory, your working directory, the name of your shell, files that you have opened, functions that you have defined, and so on. Your environment includes many *variables* that may have been set by bash or by you. The bash shell also allows you to have *shell variables*, which you may *export* to your environment for use by other processes running in the shell or by other shells that you may spawn from the current shell.

Both environment variables and shell variables have a *name*. You reference the value of a variable by prefixing its name with '$'. Some of the common bash environment variables that you will encounter are shown in Table 4.

Table 4.Some common bash environment variables

Name	Function
USER	The name of the logged-in user
UID	The numeric user id of the logged-in user
HOME	The user's home directory
PWD	The current working directory
SHELL	The name of the shell
$	The process id (or *PID*of the running bash shell (or other) process
PPID	The process id of the process that started this process (that is, the id of the parent process)
?	The exit code of the last command

Listing 7 shows what you might see in some of these common bash variables.

Listing 7. Environment and shell variables

```
[ian@echidna ian]$ echo $USER $UID
ian 500
[ian@echidna ian]$ echo $SHELL $HOME $PWD
/bin/bash /home/ian /home/ian
[ian@echidna ian]$ (exit 0);echo $?;(exit 4);echo $?
0
4
[ian@echidna ian]$ echo $$ $PPID
30576 30575
```

Not using bash?

The bash shell is the default shell on most Linux distributions. If you are not running under the bash shell, you may want to consider one of the following ways to practice with the bash shell.

- Use the
  `chsh -s /bin/bash`
  command to change your default shell. The default will take over next time you log in.

- Use the
  `su - $USER -s /bin/bash`
  command to create another process as a child of your current shell. The new process will be a login shell using bash.
- Create an id with a default of a bash shell to use for LPI exam preparation.

You may create or *set* a shell variable by typing a name followed immediately by an equal sign (=). Variables are case sensitive, so var1 and VAR1 are different variables. By convention, variables, particularly exported variables, are upper case, but this is not a requirement. Technically, $$ and $? are shell *parameters* rather than variables. They may only be referenced; you cannot assign a value to them.

When you create a shell variable, you will often want to *export* it to the environment so it will be available to other processes that you start from this shell. Variables that you export are **not** available to a parent shell. You use the `export` command to export a variable name. As a shortcut in bash, you can assign a value and export a variable in one step.

To illustrate assignment and exporting, let's run the bash command while in the bash shell and then run the Korn shell (ksh) from the new bash shell. We will use the `ps` command to display information about the command that is running. We'll learn more about `ps` when we learn about process status later in this tutorial.

Listing 8. More environment and shell variables

```
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
30576 30575 -bash
[ian@echidna ian]$ bash
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
16353 30576 bash
[ian@echidna ian]$ VAR1=var1
[ian@echidna ian]$ VAR2=var2
[ian@echidna ian]$ export VAR2
[ian@echidna ian]$ export VAR3=var3
[ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3
var1 var2 var3
[ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3 $SHELL
var1 var2 var3 /bin/bash
[ian@echidna ian]$ ksh
$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
16448 16353 ksh
$ export VAR4=var4
$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
var2 var3 var4 /bin/bash
$ exit
$ [ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
var1 var2 var3 /bin/bash
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
16353 30576 bash
[ian@echidna ian]$ exit
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
30576 30575 -bash
[ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
/bin/bash
```

Notes:

1. At the start of this sequence, the bash shell had PID 30576 .
2. The second bash shell has PID 16353, and its parent is PID 30576, the original bash shell.
3. We created VAR1, VAR2, and VAR3 in the second bash shell, but only exported VAR2 and VAR3.
4. In the Korn shell, we created VAR4. The `echo` command displayed values only for VAR2, VAR3, and VAR4, confirming that VAR1 was not exported. Were you surprised to see that the value of the SHELL variable had not changed, even though the prompt had changed? You cannot always rely on SHELL to tell you what shell you are running under, but the `ps` command does tell you the actual command. Note that `ps` puts a hyphen (-) in front of the first bash shell to indicate that this is the *login shell*.
5. Back in the second bash shell, we can see VAR1, VAR2, and VAR3.
6. And finally, when we return to the original shell, none of our new variables still exist.

The earlier discussion of quoting mentioned that you could use either single or double quotes. There is an important difference between them. The shell expands shell variables that are between double quotes ($quot;), but expansion is not done when single quotes (') are used. In the previous example, we started another shell within our shell and we got a new process id. Using the `-c` option, you can pass a command to the other shell, which will execute the command and return. If you pass a quoted string as a command, your outer shell will strip the quotes and pass the string. If double quotes are used, variable expansion occurs **before** the string is passed, so the results may not be as you expect. The shell and command will run in another process so they will have another PID. Listing 9 illustrates these concepts. The PID of the top-level bash shell is highlighted.

Listing 9. Quoting and shell variables

```
[ian@echidna ian]$ echo "$SHELL" '$SHELL' "$$" '$$'
/bin/bash $SHELL 19244 $$
[ian@echidna ian]$ bash -c "echo Expand in parent $$ $PPID"
Expand in parent 19244 19243
[ian@echidna ian]$ bash -c 'echo Expand in child $$ $PPID'
Expand in child 19297 19244
```

So far, all our variable references have terminated with white space, so it has been clear where the variable name ends. In fact, variable names may be composed only of letters, numbers or the underscore character. The shell knows that a variable name ends where another character is found. Sometimes you need to use variables in expressions where the meaning is ambiguous. In such cases, you can use curly braces to delineate a variable name as shown in Listing 10.

Listing 10. Using curly braces with variable names

```
[ian@echidna ian]$ echo "-$HOME/abc-"
-/home/ian/abc-
[ian@echidna ian]$ echo "-$HOME_abc-"
--
[ian@echidna ian]$ echo "-${HOME}_abc-"
-/home/ian_abc-
```

Env

The `env` command without any options or parameters displays the current environment variables. You can also use it to execute a command in a custom environment. The `-i` (or just `-`) option clears the current environment before running the command, while the `-u` option unsets environment variables that you do not wish to pass.

Listing 11 shows partial output of the `env` command without any parameters and then three examples of invoking different shells without the parent environment. Look carefully at these before we discuss them.

Listing 11. The env command

```
[ian@echidna ian]$ env
HOSTNAME=echidna
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=9.27.89.137 4339 22
SSH_TTY=/dev/pts/2
USER=ian
...
_=/bin/env
OLDPWD=/usr/src
[ian@echidna ian]$ env -i bash -c 'echo $SHELL; env'
/bin/bash
PWD=/home/ian
SHLVL=1
_=/bin/env
[ian@echidna ian]$ env -i ksh -c 'echo $SHELL; env'

_=/bin/env
PATH=/bin:/usr/bin
[ian@echidna ian]$ env -i tcsh -c 'echo $SHELL; env'
SHELL: Undefined variable.
```

Notice that bash has set the SHELL variable, but not exported it to the environment, although there are three other variables that bash has created in the environment. In the ksh example, we have two environment variables, but our attempt to echo the value of the SHELL variable gives a blank line. Finally, tcsh has not created any environment variables and produces an error at our attempt to reference the value of SHELL.

Unset and set

Listing 11 showed different behavior in how shells handle variables and environments. While this tutorial focuses on bash, it is good to know that not all shells behave the same way. Furthermore, shells behave differently according to whether they are a *login shell* or not. For now, we will just say that a login shell is the shell you get when you log in to a system; you can start other shells to behave as

login shells if you wish. The three shells started above using `env -i` were not login shells. Try passing the `-l` option to the shell command itself to see what differences you would get with a login shell.

So let's consider our attempt to display the SHELL variable's value in these non-login shells:

1. When bash started, it set the SHELL variable, but it did not automatically export it to the environment.

2. When ksh started, it did not set the SHELL variable. However, referencing an undefined environment variable is equivalent to referencing one with an empty value.

3. When tcsh started, it did not set the SHELL variable. In this case, the default behavior is different than ksh (and bash) in that an error is reported when we attempt to use a variable.

You can use the `unset` command to unset a variable and remove it from the shell variable list. If the variable was exported to the environment, this will also remove it from the environment. You can use the `set` command to control many facets of the way bash (or other shells) work. Set is a shell builtin, so the various options are shell specific. In bash, the `-u` option causes bash to report an error with undefined variables rather than treat them as defined but empty. You can turn on the various options to `set` with a `-` and turn them off with a `+`. You can display currently set options using `echo $-`.

Listing 12. Unset and set

```
[ian@echidna ian]$ echo $-
himBH
[ian@echidna ian]$ echo $VAR1

[ian@echidna ian]$ set -u;echo $-
himuBH
[ian@echidna ian]$ echo $VAR1
bash: VAR1: unbound variable
[ian@echidna ian]$ VAR1=v1
[ian@echidna ian]$ VAR1=v1;echo $VAR1
v1
[ian@echidna ian]$ unset VAR1;echo $VAR1
bash: VAR1: unbound variable
[ian@echidna ian]$ set +u;echo $VAR1;echo $-

himBH
```

If you use the `set` command without any options, it displays all your shell variables and their values (if any). There is also another command, `declare`, which you can use to create, export, and display values of shell variables. You can explore the many remaining `set` options and the `declare` command using the man pages. We will discuss man pages later in thsi section.

Exec

One final command to cover in this section is `exec`. You can use the `exec`

command to run another program that **replaces** the current shell. Listing 13 starts a child bash shell and then uses `exec` to replace it with a Korn shell. Upon exit from the Korn shell, you are back at the original bash shell (PID 22985, in this example).

Listing 13. Using exec

```
[ian@echidna ian]$ echo $$
22985
[ian@echidna ian]$ bash
[ian@echidna ian]$ echo $$
25063
[ian@echidna ian]$ exec ksh
$ echo $$
25063
$ exit
[ian@echidna ian]$ echo $$
22985
```

Command history

If you are typing in commands as you read, you may notice that you often use a command many times, either exactly the same, or with slight changes. The good news is that the bash shell can maintain a *history* of your commands. By default, history is on. You can turn it off using the command `set +o history` and turn it back on using `set -o history`. An environment variable called HISTSIZE tells bash how many history lines to keep. A number of other settings control how history works and is managed. See the bash man pages for full details.

Some of the commands that you can use with the history facility are:

history
: Displays the entire history

history *N*
: Displays the last *N* lines of your history

history -d *N*
: Deletes line *N* form your history; you might do this if the line contains a password, for example

!!
: Your most recent command

!*N*
: The *N*th history command

!-*N*
: The command that is *N* commands back in the history (!-1 is equivalent to !!)

!#
: The current command you are typing

!*string*
: The most recent command that starts with *string*

!?*string*?
: The most recent command that contains*string*

You can also use a colon (:) followed by certain values to access or modify part or a command from your history. Listing 14 illustrates some of the history capabilities.

Listing 14. Manipulating history

```
[ian@echidna ian]$ echo $$
22985
[ian@echidna ian]$ env -i bash -c 'echo $$'
1542
[ian@echidna ian]$ !!
env -i bash -c 'echo $$'
1555
[ian@echidna ian]$ !ec
echo $$
22985
[ian@echidna ian]$ !en:s/$$/$PPID/
env -i bash -c 'echo $PPID'
22985
[ian@echidna ian]$ history 6
 1097  echo $$
 1098  env -i bash -c 'echo $$'
 1099  env -i bash -c 'echo $$'
 1100  echo $$
 1101  env -i bash -c 'echo $PPID'
 1102  history 6
[ian@echidna ian]$ history -d1100
```

The commands in Listing 14 do the following:

1. Echo the current shell's PID
2. Run an echo command in a new shell and echo that shell's PID
3. Rerun the last command
4. Rerun the last command starting with 'ec'; this reruns the first command in this example
5. Rerun the last command starting with 'en', but substitute '$PPID' for '$$', so the parent PID is displayed instead
6. Display the last 6 commands of the history
7. Delete history entry 1100, the last echo command

You can also edit the history interactively. The bash shell uses the readline library to manage command editing and history. By default, the keys and key combinations used to move through the history or edit lines are similar to those used in the GNU Emacs editor. Emacs keystroke combinations are usually expressed as **C-x** or **M-x**, where **x** is a regular key and **C** and **M** are the *Control* and *Meta* keys, respectively. On a typical PC system, the **Ctrl** key serves as the Emacs Control key, and the **Alt** key serves as the Meta key. Table 5 summarizes some of the history editing functions available. Besides the key combinations shown in Table 5, cursor movement keys such as the right, left, up, and down arrows, and the Home and End

keys are usually set to work in a logical way. Additional functions as well as how to customize these options using a readline init file (usually inputrc in your home directory) can be found in the man pages.

Table 5. History editing with emacs commands

Command	Common PC key	Description
C-f	Right arrow	Move one space to the right
C-b	Left arrow	Move one space to the left
M-f	Alt-f	Move to beginning of next word; GUI environments usually take this key combination to open the **File** menu of the window
M-b	Alt-b	Move to beginning of previous word
C-a	Home	Move to beginning of line
C-e	End	Move to end of line
Backspace	Backspace	Delete the character preceding the cursor
C-d	Del	Delete the character under the cursor (Del and Backspace functions may be configured with opposite meanings)
C-k	Ctrl-k	Delete (kill) to end of line and save removed text for later use
M-d	Alt-d	Delete (kill) to end of word and save removed text for later use
C-y	Ctrl-y	Yank back text removed with a kill command

If you prefer to manipulate the history using a vi-like editing mode, use the command `set -o vi` to switch to vi mode. Switch back to emacs mode using `set -o emacs`. When you retrieve a command in vi mode, you are initially in vi's insert mode. More details on the vi editor are in the section File editing with vi.

Paths

Some bash commands are builtin, while others are external. Let's now look at external commands and how to run them, and how to tell if a command is internal.

Where does the shell find commands?

External commands are just files in your file system. The later section Basic file management of this tutorial and the tutorial for Topic 104 cover more details. On Linux and UNIX systems, all files are accessed as part of a single large tree that is

rooted at /. In our examples so far, our current directory has been the user's home directory. Non-root users usually have a home directory within the /home directory, such as /home/ian, in my case. Root's home is usually /root. If you type a command name, then bash looks for that command on your *path*, which is a colon-separated list of directories in the PATH environment variable.

If you want to know what command will be executed if you type a particular string, use the `which` or `type` command. Listing 15 shows my default path along with the locations of several commands.

Listing 15. Finding command locations

```
[ian@echidna ian]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/ian/bin
[ian@echidna ian]$ which bash env zip xclock echo set ls
alias ls='ls --color=tty'
        /bin/ls
/bin/bash
/bin/env
/usr/bin/zip
/usr/X11R6/bin/xclock
/bin/echo
/usr/bin/which: no set in (/usr/local/bin:/bin:/usr/bin:/usr/X11R6/b
in:/home/ian/bin)
[ian@echidna ian]$ type bash env zip xclock echo set ls
bash is /bin/bash
env is /bin/env
zip is /usr/bin/zip
xclock is /usr/X11R6/bin/xclock
echo is a shell builtin
set is a shell builtin
ls is aliased to `ls --color=tty'
```

Note that the directories in the path all end in /bin. This is a common convention, but not a requirement. The `which` command reported that the `ls` command is an *alias* and that the `set` command could not be found. In this case, we interpret that to mean that it does not exist or that it is a builtin. The `type` command reports that the `ls` command is an *alias*, but it identifies the `set` command as a shell builtin. It also reports that there is a builtin `echo` command as well as the one in /bin that was found by `which`. The two commands also produce output in different orders.

We saw that the `ls` command, used for listing directory contents, is an *alias*. Aliases are a handy way to configure some commands to use different sets of defaults or to provide an alternate name for a command. In our example, the `--color=tty` option causes directory listings to be color coded according to the type of file or directory. Try running `dircolors --print-database` to see how the color codings are controlled and which colors are used for what kind of file.

Each of these commands has additional options. Depending on your need, you may use either command. I tend to use `which` when I am reasonably sure I'll find an executable and I just need its full path specification. I find that `type` gives me more precise information, which I sometimes need in a shell script.

Running other commands

We saw in Listing 15 that executable files have a full path starting with /, the root directory. For example, the xclock program is really /usr/X11R6/bin/xclock, a file

located in the /usr/X11R6/bin directory. If a command is **not** in your PATH specification, you may still run it by specifying a path as well as a command name. There are two types of paths that you can use:

- *Absolute* paths are those starting with /, such as those we saw in Listing 15 (/bin/bash, /bin/env, etc).
- *Relative* paths are relative to your *current working directory*, as reported by the `pwd` command. These commands do not start with /, but do contain at least one /.

You may use absolute paths regardless of your current working directory, but you will probably use relative paths only when a command is close to your current directory. Suppose you are developing a new version of the classic "Hello World!" program in a subdirectory of your home directory called mytestbin. You might use the relative path to run your command as `mytestbin/hello`. There are two special names you can use in a path; a single dot (.) refers to the current directory, and a pair of dots (..) refers to the parent of the current directory. Because your home directory is usually not on your PATH (and generally should not be), you will need to explicitly provide a path for any executable that you want to run from your home directory. For example, if you had a copy of your hello program in your home directory, you could run it using the command `./hello`. You can use both . and .. as part of an absolute path, although a single . is not very useful in such a case. You can also use a tilde (~) to mean your own home directory and ~*username* to mean the home directory of the user named *username*. Some examples are shown in Listing 16.

Listing 16. Absolute and relative paths

```
[ian@echidna ian]$ /bin/echo Use echo command rather than builtin
Use echo command rather than builtin
[ian@echidna ian]$ /usr/../bin/echo Include parent dir in path
Include parent dir in path
[ian@echidna ian]$ /bin/./../echo Add a couple of useless path components
Add a couple of useless path components
[ian@echidna ian]$ pwd # See where we are
/home/ian
[ian@echidna ian]$ ../../bin/echo Use a relative path to echo
Use a relative path to echo
[ian@echidna ian]$ myprogs/hello # Use a relative path with no dots
-bash: myprogs/hello: No such file or directory
[ian@echidna ian]$ mytestbin/hello # Use a relative path with no dots
Hello World!
[ian@echidna ian]$ ./hello # Run program in current directory
Hello World!
[ian@echidna mytestbin]$ ~/mytestbin/hello # run hello using ~
Hello World!
[ian@echidna ian]$ ../hello # Try running hello from parent
-bash: ../hello: No such file or directory
```

Changing your working directory

Just as you can execute programs from various directories in the system, so too can you change your current working directory using the `cd` command. The argument to `cd` must be the absolute or relative path to a directory. As for commands, you can use ., .., ~, and ~*username* in paths. If you use `cd` with no parameters, the change will be to your home directory. A single hyphen (-) as a parameter means to change

to the previous working directory. Your home directory is stored in the HOME environment variable, and the previous directory is stored in the OLDPWD variable, so `cd` alone is equivalent to `cd $HOME` and `cd -` is equivalent to `cd $OLDPWD`. Usually we say *change directory* instead of the full *change current working directory*.

As for commands, there is also an environment variable, CDPATH, which contains a colon-separated set of directories that should be searched (in addition to the current working directory) when resolving relative paths. If resolution used a path from CDPATH, then `cd` will print the full path of the resulting directory as output. Normally, a successful directory change results in no output other than a new, and possibly changed, prompt. Some examples are shown in Listing 17.

Listing 17. Changing directories

```
[ian@echidna home]$ cd /;pwd
/
[ian@echidna /]$ cd /usr/X11R6;pwd
/usr/X11R6
[ian@echidna X11R6]$ cd ;pwd
/home/ian
[ian@echidna ian]$ cd -;pwd
/usr/X11R6
/usr/X11R6
[ian@echidna X11R6]$ cd ~ian/..;pwd
/home
[ian@echidna home]$ cd ~;pwd
/home/ian
[ian@echidna ian]$ export CDPATH=~
[ian@echidna mytestbin]$ cd /;pwd
/
[ian@echidna /]$ cd mytestbin
/home/ian/mytestbin
```

Applying commands recursively

Many Linux commands can be applied recursively to all files in a directory tree. For example, the `ls` command has a `-R` option to list directories recursively, and the `cp`, `mv`, `rm`, and `diff` commands all have a `-r` option to apply them recursively. The section Basic file management covers recursive application of commands in detail.

Command substitution

The bash shell has an extremely powerful capability that allows the results of one command to be used as input to another; this is called *command substitution*. This can be done by enclosing the command whose results you wish to use in backticks (`). This is still common, but a way that is easier to use with multiply nested commands is to enclose the command between $(and).

In the previous tutorial, "LPI exam 101 prep (topic 102): Linux installation and package management," we saw that the `rpm` command can tell you which package a command comes from; we used the command substitution capability as a handy technique. Now you know that's what we were really doing.

Command substitution is an invaluable tool in shell scripts and is also useful on the

command line. Listing 18 shows an example of how to get the absolute path of a directory from a relative path, how to find which RPM provides the /bin/echo command, and how (as root) to list the labels of three partitions on a hard drive. The last one uses the `seq` command to generate a sequence of integers.

Listing 18. Command substitution

```
[ian@echidna ian]$ echo '../../usr/bin' dir is $(cd ../../usr/bin;pwd)
../../usr/bin dir is /usr/bin
[ian@echidna ian]$ which echo
/bin/echo
[ian@echidna ian]$ rpm -qf `which echo`
sh-utils-2.0.12-3
[ian@echidna ian]$ su -
Password:
[root@echidna root]# for n in $(seq 7 9); do echo p$n `e2label /dev/hda$n`;done
p7 RH73
p8 SUSE81
p9 IMAGES
```

Man pages

Our final topic in this section of the tutorial tells you how to get documentation for Linux commands through manual pages and other sources of documentation.

Manual pages and sections

The primary (and traditional) source of documentation is the *manual pages*, which you can access using the `man` command. Figure 1 illustrates the manual page for the `man` command itself. Use the command `man man` to display this information.

Figure 1. Man page for the man command

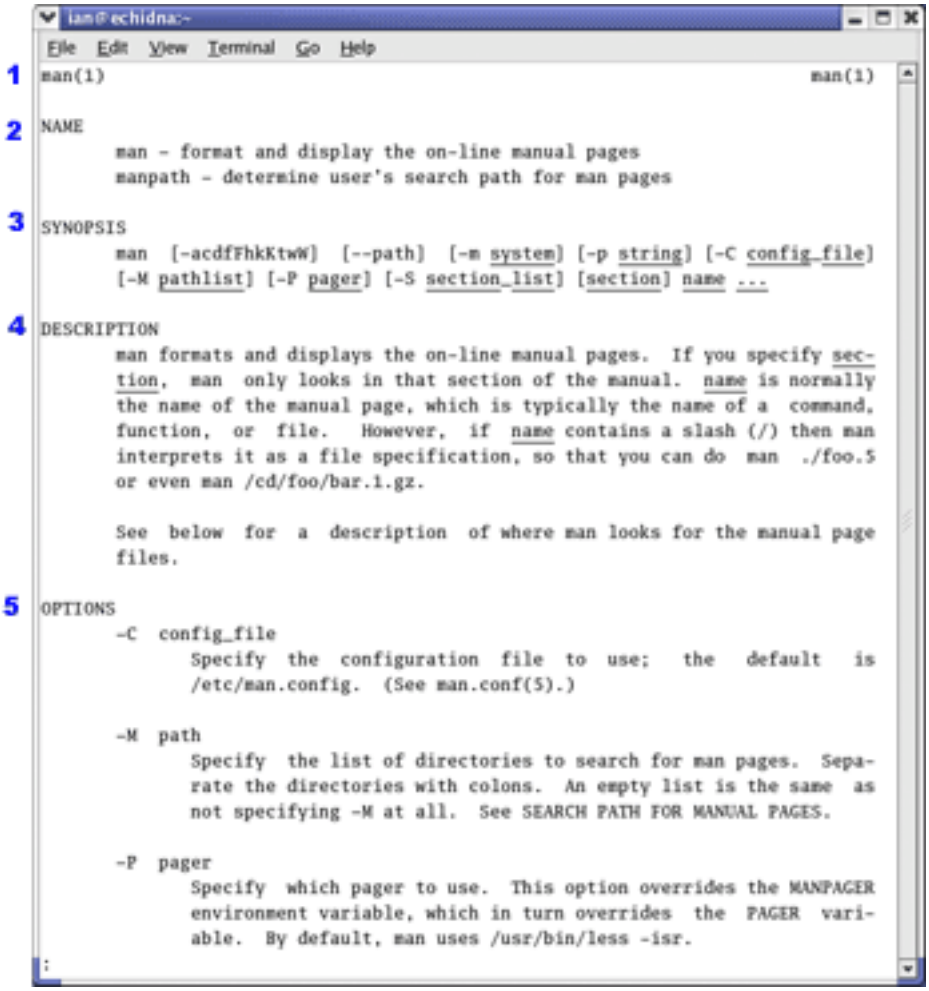


Figure 1 shows some typical items in man pages:

- A heading with the name of the command followed by its section number in parentheses
- The name of the command and any related commands that are described on the same man page
- A synopsis of the options and parameters applicable to the command
- A short description of the command
- Detailed information on each of the options

You might find other sections on usage, how to report bugs, author information, and a list of any related commands. For example, the man page for `man` tells us that related commands (and their manual sections) are:
apropos(1), whatis(1), less(1), groff(1), and man.conf(5).

There are eight common manual page sections. Manual pages are usually installed when you install a package, so if you do not have a package installed, you probably won't have a manual page for it. Similarly, some of your manual sections may be empty or nearly empty. The common manual sections, with some example contents

are:

1. User commands (env, ls, echo, mkdir, tty)
2. System calls or kernel functions (link, sethostname, mkdir)
3. Library routines (acosh, asctime, btree, locale, XML::Parser)
4. Device related information (isdn_audio, mouse, tty, zero)
5. File format descriptions (keymaps, motd, wvdial.conf)
6. Games (note that many games are now graphical and have graphical help outside the man page system)
7. Miscellaneous (arp, boot, regex, unix utf8)
8. System administration (debugfs, fdisk, fsck, mount, renice, rpm)

Other sections that you might find include *9* for Linux kernel documentation, *n* for new documentation, *o* for old documentation, and *l* for local documentation.

Some entries appear in multiple sections. Our examples show mkdir in sections 1 and 2, and tty in sections 1 and 4. You can specify a particular section, for example, `man 4 tty` or `man 2 mkdir`, or you can specify the `-a` option to list all applicable manual sections.

You may have noticed in the figure that `man` has many options for you to explore on your own. For now, let's take a quick look at some of the "See also" commands related to `man`.

See also

Two important commands related to `man` are `whatis` and `apropos`. The `whatis` command searches man pages for the name you give and displays the name information from the appropriate manual pages. The `apropos` command does a keyword search of manual pages and lists ones containing your keyword. Listing 19 illustrates these commands.

Listing 19. Whatis and apropos examples

```
[ian@lyrebird ian]$ whatis man
man                  (1)  - format and display the on-line manual pages
man                  (7)  - macros to format man pages
man [manpath]        (1)  - format and display the on-line manual pages
man.conf [man]       (5)  - configuration data for man
[ian@lyrebird ian]$ whatis mkdir
mkdir                (1)  - make directories
mkdir                (2)  - create a directory
[ian@lyrebird ian]$ apropos mkdir
mkdir                (1)  - make directories
mkdir                (2)  - create a directory
mkdirhier            (1x)  - makes a directory hierarchy
```

By the way, if you cannot find the manual page for man.conf, try running `man man.conf`**`ig`** instead.

The `man` command pages output onto your display using a paging program. On most Linux systems, this is likely to be the `less` program. Another choice might be the older `more` program. If you wish to print the page, specify the `-t` option to format the page for printing using the `groff` or `troff` program.

The less pager has several commands that help you search for strings within the displayed output. Use `man less` to find out more about / (search forwards), *?* (search backwards), and *n* (repeat last search), among many other commands.

Other documentation sources

In addition to manual pages accessible from a command line, the Free Software Foundation has created a number of *info* files that are processed with the *info* program. These provide extensive navigation facilities including the ability to jump to other sections. Try `man info` or `info info` for more information. Not all commands are documented with info, so you will find yourself using both man and info if you become an info user.

There are also some graphical interfaces to man pages, such as `xman` (from the XFree86 Project) and `yelp` (the Gnome 2.0 help browser).

If you can't find help for a command, try running the command with the `--help` option. This may provide the command's help, or it may tell you how to get the help you need.

The next section looks at processing text streams with filters.

Section 3. Text streams and filters

This section covers material for topic 1.103.2 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 6.

In this section, you learn about the following topics:

- Sending text files and output streams through text utility filters to modify the output
- Using standard UNIX commands found in the GNU textutils package

Text filtering

Text *filtering* is the process of taking an input stream of text and performing some

conversion on the text before sending it to an output stream. Although either the input or the output can come from a file, in the Linux and UNIX environments, filtering is most often done by constructing a *pipeline* of commands where the output from one command is *piped* or *redirected* to be used as input to the next. Pipes and redirection are covered more fully in the section on Streams, pipes, and redirects, but for now, let's look at pipes and basic output redirection using the | and > operators.

Piping with |

Recall from the previous section that shells use three standard I/O *streams*:

- *stdin* is the *standard input stream*, which provides input to commands.
- *stdout* is the *standard output stream*, which displays output from commands.
- *stderr* is the *standard error stream*, which displays error output from commands.

So far in this tutorial, input has come from parameters we supply to commands, and output has been displayed on our terminal. Many text processing commands (filters) can take input either from the standard input stream or from a file. In order to use the output of a command, command1, as input to a filter, command2, you connect the commands using the pipe operator (|) as shown in Listing 20.

Listing 20. Piping output from command 1 to input of command2

```
command1 | command2
```

Either command may have options or arguments, as you will see later in this section. You can also use | to redirect the output of command2 in this pipeline to yet another command, command3. Constructing long pipelines of commands that each have limited capability is a common Linux and UNIX way of accomplishing tasks. You will also sometimes see a hyphen (-) used in place of a filename as an argument to a command, meaning the input should come from stdin rather than a file.

Output redirection with >

While it is nice to be able to create a pipeline of several commands and see the output on your terminal, there are times when you want to save the output in a file. You do this with the output redirection operator (>).

For the rest of this section, we will be using some small files, so let's create a directory called lpi103 and then cd into that directory. We will then use > to redirect the output of the `echo` command into a file called text1. This is all shown in Listing 21. Notice that the output does not display on the terminal because it has been redirected to the file.

Listing 21. Piping output from command 1 to a file

```
[ian@echidna ian]$ mkdir lpi103
[ian@echidna ian]$ cd lpi103
```

```
[ian@echidna lpi103]$ echo -e "1 apple\n2 pear\n3 banana">text1
```

Now that we have a couple of basic tools for pipelining and redirection, let's look at some of the common UNIX and Linux text processing commands and filters. This section shows you some of the basic capabilities; check the appropriate man pages to find out more about these commands.

Cat, tac, od, and split

Now that you have created the test1 file, you might want to check what is in it. Use the `cat` (short for *catenate)* command to display the contents of a file on stdout. Listing 22 verifies the contents of the file created above.

Listing 22. Displaying file contents with cat

```
[ian@echidna lpi103]$ cat text1
1 apple
2 pear
3 banana
```

The `cat` command takes input from stdin if you do not specify a file name (or if you specify - as the filename). Let's use this along with output redirection to create another text file as shown in Listing 23.

Listing 23. Creating a text file with cat

```
[ian@echidna lpi103]$ cat>text2
9       plum
3       banana
10      apple
```

In Listing 23, `cat` keeps reading from stdin until end of file. Use the **Ctrl-d** (hold **Ctrl** and press **d**) combination to signal end of file. This is the same key combination to exit from the bash shell. Note also that the tab key helps line up the fruit names in a column.

Occasionally, you might want to display a file in reverse order. Naturally, there is a text filter for that too, called `tac` (reverse of `cat`). Listing 24 shows the new text2 file as well as the old text1 file in reverse order. Notice how the display simply concatenates the two files.

Listing 24. Reverse display with tac

```
[ian@echidna lpi103]$ tac text2 text1
10      apple
3       banana
9       plum
3 banana
2 pear
1 apple
```

Now, suppose you display the two text files using cat or tac and notice alignment differences. To learn what causes this, you need to look at the control characters that are in the file. Since these are acted upon in text display output rather than having some representation of the control character itself displayed, we need to *dump* the file in a format that allows you to find and interpret these special

characters. The GNU text utilities include an `od` (or *O*ctal *D*ump) command for this purpose.

There are several options to `od`, such as the `-A` option to control the radix of the file offsets and `-t` to control the form of the displayed file contents. The radix may be specified as o, (octal - the default), d (decimal), x (hexadecimal), or n (no offsets displayed). You can display output as octal, hex, decimal, floating point, ASCII with backslash escapes or named characters (nl for newline, ht for horizontal tab, etc.). Listing 25 shows some of the formats available for dumping the text2 example file.

Listing 25. Dumping files with od

```
[ian@echidna lpi103]$ od text2
0000000 004471 066160 066565 031412 061011 067141 067141 005141
0000020 030061 060411 070160 062554 000012
0000031
[ian@echidna lpi103]$ od -A d -t c text2
0000000   9  \t   p   l   u   m  \n   3  \t   b   a   n   a   n   a  \n
0000016   1   0  \t   a   p   p   l   e  \n
0000025
[ian@echidna lpi103]$ od -A n -t a text2
   9  ht   p   l   u   m  nl   3  ht   b   a   n   a   n   a  nl
   1   0  ht   a   p   p   l   e  nl
```

Notes:

1. The `-A` option of `cat` provides an alternate way of seeing where your tabs and line endings are. See the man page for more information.

2. If you have a mainframe background, you may be interested in the hexdump utility, which is part of a different utility set. It's not covered here, so check the man pages.

Our sample files are very small, but sometimes you will have large files that you need to split into smaller pieces. For example, you might want to break a large file into CD-sized chunks so you can write it to CD for sending through the mail to someone who could create a DVD for you. The `split` command will do this in such a way that the `cat` command can be used to recreate the file easily. By default, the files resulting from the `split` command have a prefix in their name of 'x' followed by a suffix of 'aa', 'ab', 'ac', ..., 'ba', 'bb', etc. Options permit you to change these defaults. You can also control the size of the output files and whether the resulting files contain whole lines or just byte counts. Listing 26 illustrates splitting our two text files with different prefixes for the output files. We split text1 into files containing at most two lines, and text2 into files containing at most 18 bytes. We then use `cat` to display some of the pieces individually as well as to display a complete file using *globbing*, which is covered in the section on wildcards and globbing later in this tutorial.

Listing 26. Splitting and recombining with split and cat

```
[ian@echidna lpi103]$ split -l 2 text1
[ian@echidna lpi103]$ split -b 18 text2 y
[ian@echidna lpi103]$ cat yaa
9       plum
3       banana
10[ian@echidna lpi103]$ cat yab
```

```
        apple
[ian@echidna lpi103]$ cat y*
9       plum
3       banana
10      apple
```

Note that the split file named yab did not finish with a newline character, so our prompt was offset after we used `cat` to display it.

Wc, head, and tail

`Cat` and `tac` display the whole file. That's fine for small files like our examples, but suppose you have a large file. Well, first you might want to use the `wc` (*W*ord *C*ount) command to see how big the file is. The `wc` command displays the number of lines, words, and bytes in a file. You can also find the number of bytes by using `ls -l`. Listing 27 shows the long format directory listing for our two text files, as well as the output from `wc`.

Listing 27. Using wc with text files

```
[ian@echidna lpi103]$ ls -l text*
-rw-rw-r--    1 ian      ian            24 Sep 23 12:27 text1
-rw-rw-r--    1 ian      ian            25 Sep 23 13:39 text2
[ian@echidna lpi103]$ wc text*
      3       6      24 text1
      3       6      25 text2
      6      12      49 total
```

Options allow you to control the output from `wc` or to display other information such as maximum line length. See the man page for details.

Two commands allow you to display either the first part (*head*) or last part (*tail*). These commands are the `head` and `tail` commands. They can be used as filters,or they can take a file name as an augument. By default they display the first (or last) 10 lines of the file or stream. Listing 28 uses the `dmesg` command to display bootup messages, in conjunction with `wc`, `tail`, and `head` to discover that there are 177 messages, then to display the last 10 of these, and finally to display the six messages starting 15 from the end. Some lines have been truncated in this output (indicated by ...).

Listing 28. Using wc, head, and tail to display boot messages

```
[ian@echidna lpi103]$ dmesg | wc
    177    1164    8366
[ian@echidna lpi103]$ dmesg | tail
i810: Intel ICH2 found at IO 0x1880 and 0x1c00, MEM 0x0000 and ...
i810_audio: Audio Controller supports 6 channels.
i810_audio: Defaulting to base 2 channel mode.
i810_audio: Resetting connection 0
ac97_codec: AC97 Audio codec, id: ADS98 (Unknown)
i810_audio: AC'97 codec 0 Unable to map surround DAC's (or ...
i810_audio: setting clocking to 41319
Attached scsi CD-ROM sr0 at scsi0, channel 0, id 0, lun 0
sr0: scsi3-mmc drive: 0x/32x writer cd/rw xa/form2 cdda tray
Uniform CD-ROM driver Revision: 3.12
[ian@echidna lpi103]$ dmesg | tail -n15 | head -n 6
agpgart: Maximum main memory to use for agp memory: 941M
agpgart: Detected Intel i845 chipset
agpgart: AGP aperture is 64M @ 0xf4000000
Intel 810 + AC97 Audio, version 0.24, 13:01:43 Dec 18 2003
PCI: Setting latency timer of device 00:1f.5 to 64
```

```
i810: Intel ICH2 found at IO 0x1880 and 0x1c00, MEM 0x0000 and ...
```

Another common use of `tail` is to *follow* a file using the `-f` option, usually with a line count of 1. You might use this when you have a background process that is generating output in a file and you want to check in and see how it is doing. In this mode, `tail` will run until you cancel it (using **Ctrl-c**), displaying lines as they are written to the file.

Expand, unexpand, and tr

When we created our text1 and text2 files, we created text2 with tab characters. Sometimes you may want to swap tabs for spaces or vice versa. The `expand` and `unexpand` commands do this. The `-t` option to both commands allows you to set the tab stops. A single value sets repeated tabs at that interval. Listing 29 shows how to expand the tabs in text2 to single spaces and another whimsical sequence of `expand` and `unexpand` that unaligns the text in text2.

Listing 29. Using expand and unexpand

```
[ian@echidna lpi103]$ expand -t 1 text2
9 plum
3 banana
10 apple
[ian@echidna lpi103]$ expand -t8 text2|unexpand -a -t2|expand -t3
9           plum
3           banana
10       apple
```

Unfortunately, you cannot use `unexpand` to replace the spaces in text1 with tabs as `unexpand` requires at least two spaces to convert to tabs. However, you can use the `tr` command, which translates characters in one set (*set1*) to corresponding characters in another set (*set2*). Listing 30 shows how to use `tr` to translate spaces to tabs. Since `tr` is purely a filter, you generate input for it using the `cat` command. This example also illustrates the use of - to signify standard input to `cat`.

Listing 30. Using expand and unexpand

```
[ian@echidna lpi103]$ cat text1 |tr ' ' '\t'|cat - text2
1       apple
2       pear
3       banana
9       plum
3       banana
10      apple
```

If you are not sure what is happening in the last two examples, try using `od` to terminate each stage of the pipeline in turn; for example,
`cat text1 |tr ' ' '\t' | od -tc`

Pr, nl, and fmt

The `pr` command is used to format files for printing. The default header includes the file name and file creation date and time, along with a page number and two lines of blank footer. When output is created from multiple files or the standard input stream,

the current date and time are used instead of the file name and creation date. You can print files side-by-side in columns and control many aspects of formatting through options. As usual, refer to the man page for details.

The `nl` command numbers lines, which can be convenient when printing files. You can also number lines with the `-n` option of the `cat` command. Listing 31 shows how to print our text 1 file, and then how to number text2 and print it side-by-side with text1.

Listing 31. Numbering and formatting for print

```
[ian@echidna lpi103]$ pr text1 | head


2005-09-23 12:27                      text1                      Page 1


1 apple
2 pear
3 banana


[ian@echidna lpi103]$ nl text2 | pr -m - text1 | head


2005-09-26 11:48                                                 Page 1


     1  9       plum                1 apple
     2  3       banana              2 pear
     3  10      apple               3 banana
```

Another useful command for formatting text is the `fmt` command, which formats text so it fits within margins. You can join several short lines as well as split long ones. In Listing 32, we create text3 with a single long line of text using variants of the `!#:*` history feature to save typing our sentence four times. We also create text4 with one word per line. Then we use `cat` to display them unformatted including a displayed '$' character to show line endings. Finally, we use `fmt` to format them to a maximum width of 60 characters. Again, consult the man page for details on additional options.

Listing 32. Formatting to a maximum line length

```
[ian@echidna lpi103]$ echo "This is a sentence. " !#:* !#:1-$>text3
echo "This is a sentence. " "This is a sentence. " "This is a sentenc
e. " "This is a sentence. ">text3
[ian@echidna lpi103]$ echo -e "This\nis\nanother\nsentence.">text4
[ian@echidna lpi103]$ cat -et text3 text4
This is a sentence.  This is a sentence.  This is a sentence.  This i
s a sentence. $
This$
is$
another$
sentence.$
[ian@echidna lpi103]$ fmt -w 60 text3 text4
This is a sentence.  This is a sentence.  This is a
sentence.  This is a sentence.
This is another sentence.
```

Sort and uniq

The `sort` command sorts the input using the collating sequence for the locale (LC_COLLATE) of the system The `sort` command can also merge already sorted files and check whether a file is sorted or not.

Listing 33 illustrates using the `sort`command to sort our two text files after translating blanks to tabs in text1. Since the sort order is by character, you might be surprised at the results. Fortunately, the `sort` command can sort by numeric values or by character values. You can specify this choice for the whole record or for each *field*. Unless you specify a different field separator, fields are delimited by blanks or tabs. The second example in Listing 33 shows sorting the first field numerically and the second by collating sequence (alphabetically). It also illustrates the use of the `-u` option to eliminate any duplicate lines and keep only lines that are unique.

Listing 33. Character and numeric sorting

```
[ian@echidna lpi103]$ cat text1 | tr ' ' '\t' | sort - text2
10      apple
1       apple
2       pear
3       banana
3       banana
9       plum
[ian@echidna lpi103]$ cat text1|tr ' ' '\t'|sort -u -k1n -k2 - text2
1       apple
2       pear
3       banana
9       plum
10      apple
```

Notice that we still have two lines containing the fruit "apple". Another command called `uniq` gives us additional control over the elimination of duplicate lines. The `uniq` command normally operates on sorted files, but remove **consecutive** identical lines from any file, whether sorted or not. The `uniq` command can also ignore some fields. Listing 34 sorts our two text files using the second field (fruit name) and then eliminates lines that are identical, starting at the second field (that is, we skip the first field when testing with `uniq`).

Listing 34. Using uniq

```
[ian@echidna lpi103]$ cat text1|tr ' ' '\t'|sort -k2 - text2|uniq -f1
10      apple
3       banana
2       pear
9       plum
```

Our sort was by collating sequence, so `uniq` gives us the "10 apple" line instead of the "1 apple". Try adding a numeric sort on key field 1 to see how to change this.

Cut, paste, and join

Now let's look at three more commands that deal with fields in textual data. These commands are particularly useful for dealing with tabular data. The first is the `cut` command, which extracts fields from text files. The default field delimiter is the tab character. Listing 35 uses `cut` to separate the two columns of text2 and then uses a space as an output delimiter, which is an exotic way of converting the tab in each line to a space.

Listing 35. Using cut

```
[ian@echidna lpi103]$ cut -f1-2 --output-delimiter=' ' text2
9 plum
3 banana
10 apple
```

The `paste` command pastes lines from two or more files side-by-side, similar to the way that the `pr` command merges files using its `-m` option. Listing 36 shows the result of pasting our two text files.

Listing 36. Pasting files

```
[ian@echidna lpi103]$ paste text1 text2
1 apple 9       plum
2 pear  3       banana
3 banana        10      apple
```

These examples show simple pasting, but `paste` can paste data from one or more files in several other ways. Consult the man page for details.

Our final field-manipulating command is `join`, which joins files based on a matching field. The files should be sorted on the join field. Since text2 is not sorted in numeric order, we could sort it and then join would join the two lines that have a matching join field (3 in this case). Let's also create a new file, text5, by sorting text 1 on the second field (the fruit name) and then replacing spaces with tabs. If we then sort text2 and join that with text 5 using the second field, we should have two matches (apple and banana). Listing 37 illustrates both these joins.

Listing 37. Joining files with join fields

```
[ian@echidna lpi103]$ sort -n text2|join -1 1 -2 1 text1 -
3 banana banana
[ian@echidna lpi103]$ sort -k2 text1|tr ' ' '\t'>text5
[ian@echidna lpi103]$ sort -k2 text2 | join -1 2 -2 2 text5 -
apple 1 10
banana 3 3
```

The field to use for the join is specified separately for each file. You could, for example, join based on field 3 from one file and field 10 from another.

Sed

Sed is the *s*tream *ed*itor. Several developerWorks articles, as well as many books and book chapters, are available on sed (see Resources). Sed is extremely powerful, and the tasks it can accomplish are limited only by your imagination. This small introduction should whet your appetite for sed, but is not intended to be complete or extensive.

As with many of the text commands we have looked at so far, sed can work as a filter or take its input from a file. Output is to the standard output stream. Sed loads lines from the input into the *pattern space*, applies sed editing commands to the contents of the pattern space, and then writes the pattern space to standard output. Sed may combine several lines in the pattern space, and it may write to a file, write only selected output, or not write at all.

Sed uses regular expression syntax (see Searching with regular expressions later in this tutorial) to search for and replace text selectively in the pattern space as well as to control which lines of text should be operated on by sets of editing commands. A *hold buffer* provides temporary storage for text. The hold buffer may replace the pattern space, be added to the pattern space, or be exchanged with the pattern space. Sed has a limited set of commands, but these combined with regular expression syntax and the hold buffer make for some amazing capabilities. A set of sed commands is usually called a *sed script*.

Listing 38 shows three simple sed scripts. In the first one, we use the `s` (substitute) command to substitute an upper case for a lower case 'a' on each line. This example replaces only the first 'a', so in the second example, we add the 'g' (for global) flag to cause sed to change all occurrences. In the third script, we introduce the `d` (delete) command to delete a line. In our example, we use an address of 2 to indicate that only line 2 should be deleted. We separate commands using a semi-colon (;) and use the same global substitution that we used in the second script to replace 'a' with 'A'.

Listing 38. Beginning sed scripts

```
[ian@echidna lpi103]$ sed 's/a/A/' text1
1 Apple
2 peAr
3 bAnana
[ian@echidna lpi103]$ sed 's/a/A/g' text1
1 Apple
2 peAr
3 bAnAnA
[ian@echidna lpi103]$ sed '2d;$s/a/A/g' text1
1 apple
3 bAnAnA
```

In addition to operating on individual lines, sed can operate on a range of lines. The beginning and end of the range are separated by a comma (,) and may be specified as a line number, a caret (^) for beginning of file, or a dollar sign ($) for end of file. Given an address or a range of addresses, you can group several commands between curly braces ({ and }) to have these commands operate only on lines selected by the range. Listing 39 illustrates two ways of having our global substitution applied to only the last two lines of our file. It also illustrates the use of the `-e` option to add commands to the pattern space. When using braces, we must separate commands in this way.

Listing 39. Sed addresses

```
[ian@echidna lpi103]$ sed -e '2,${' -e 's/a/A/g' -e '}' text1
1 apple
2 peAr
3 bAnAnA
[ian@echidna lpi103]$ sed -e '/pear/,/bana/{' -e 's/a/A/g' -e '}' text1
1 apple
2 peAr
3 bAnAnA
```

Sed scripts may also be stored in files. In fact, you will probably want to do this for frequently used scripts. Remember earlier we used the `tr` command to change blanks in text1 to tabs. Let's now do that with a sed script stored in a file. We will use the `echo` command to create the file. The results are shown in Listing 40.

Listing 40. A sed one-liner

```
[ian@echidna lpi103]$ echo -e "s/ /\t/g">sedtab
[ian@echidna lpi103]$ cat sedtab
s/ /    /g
[ian@echidna lpi103]$ sed -f sedtab text1
1       apple
2       pear
3       banana
```

There are many handy sed one-liners such as Listing 40. See Resources for links to some.

Our final sed example uses the = command to print line numbers and then filter the resulting output through sed again to mimic the effect of the `nl` command to number lines. Listing 41 uses = to print line numbers, then uses the `N` command to read a second input line into the pattern space, and finally removes the newline character (\n) between the two lines in the pattern space.

Listing 41. Numbering lines with sed

```
[ian@echidna lpi103]$ sed '=' text2
1
9       plum
2
3       banana
3
10      apple
[ian@echidna lpi103]$ sed '=' text2|sed 'N;s/\n//'
19      plum
23      banana
310     apple
```

Not quite what we wanted! What we would really like is to have our numbers aligned in a column with some space before the lines from the file. In Listing 42, we enter several lines of command (note the > secondary prompt). Study the example and refer to the explanation below.

Listing 42. Numbering lines with sed - round two

```
[ian@echidna lpi103]$ cat text1 text2 text1 text2>text6
[ian@echidna lpi103]$ ht=$(echo -en "\t")
[ian@echidna lpi103]$ sed '=' text6|sed "N
> s/^/      /
> s/^.*\(......\)\n/\1$ht/"
     1  1 apple
     2  2 pear
     3  3 banana
     4  9   plum
     5  3   banana
     6  10  apple
     7  1 apple
     8  2 pear
     9  3 banana
    10  9   plum
    11  3   banana
    12  10  apple
```

Here are the steps we took:

1. We first used `cat` to create a twelve-line file from two copies each of our text1 and text2 files. There's no fun in formatting numbers in columns if we don't have differing numbers of digits.

2. The bash shell uses the tab key for command completion, so it can be handy to have a captive tab character that you can use when you want a real tab. We use the `echo` command to accomplish this and save the character in the shell variable 'ht'.

3. We create a stream containing line numbers followed by data lines as we did before and filter it through a second copy of sed.

4. We read a second line into the pattern space

5. We prefix our line number at the start of the pattern space (denoted by ^) with six blanks.

6. We then substitute all of the line up to the newline with the last six characters before the newline plus a tab character. Note that the left part of the 's' command uses '\(' and '\)' to mark the characters that we want to use in the right part. In the right part, we reference the first such marked set (and only such set in this example) as \1. Note that our command is contained between double quotes (") so that substitution will occur for $ht.

Recent (version 4) versions of sed contain documentation in `info` format and include many excellent examples. These are not included in the older version 3.02. GNU sed will accept `sed --version` to display the version.

Section 4. Basic file management

This section covers material for topic 1.103.3 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 3.

In this section, you learn about the following topics:

- Listing directory contents
- Copying, moving, and removing files and directories
- Manipulating multiple files and directories recursively
- Using wildcard patterns for manipulating files
- Using the `find` command to locate and act on files based on type, size, or time

Listing directories

As we said earlier in our discussion of paths in the section on using the command line, all files on Linux and UNIX® systems are accessed as part of a single large tree-structured filesystem that is rooted at /.

Listing directory entries

If you worked through the previous section with us, you will have created a directory, lpi103, in your home directory. File and directory names are either *absolute* which means they begin with a / or they are *relative* to the *current working directory*, which means they do not begin with a /. The absolute path to a file or directory consists of a / followed by series of 0 or more directory names, each followed by another / and then a final name. Given a file or directory name that is relative to the current working directory, simply concatenate the absolute name of the working directory, a /, and the relative name. for example, the directory, lpi103, that we created in the last section was created in my home directory, /home/ian, so its full, or absolute, path is /home/ian/lpi103. Listing 43 shows three different ways to use the `ls` command to list the files in this directory.

Listing 43. Listing directory entries

```
[ian@echidna lpi103]$ pwd
/home/ian/lpi103
[ian@echidna lpi103]$ echo $PWD
/home/ian/lpi103
[ian@echidna lpi103]$ ls
sedtab  text2  text4  text6  xab  yab
text1   text3  text5  xaa    yaa
[ian@echidna lpi103]$ ls "$PWD"
sedtab  text2  text4  text6  xab  yab
text1   text3  text5  xaa    yaa
[ian@echidna lpi103]$ ls /home/ian/lpi103
sedtab  text2  text4  text6  xab  yab
text1   text3  text5  xaa    yaa
```

As you can see, you can give a directory name as a parameter to the `ls` command and it will list the contents of that directory.

Listing details

On a storage device, a file or directory is contained in a collection of *blocks*. Information about a file is contained in an *inode* which records information such as the owner, when the file was last accessed, how large it is, whether it is a directory or not, and who can read or write it. The inode number is also known as the *file serial number* and is unique within a particular filesystem. We can use the `-l` (or `--format=long`) option to display some of the information stored in the inode.

By default, the `ls` command does not list special files, those whose names start with a dot (.). Every directory other than the root directory has two special entries at least, the directory itself (.) and the parent directory (..). The root directory does not have a parent directory.

Listing 44 uses the `-l` and `-a` options to display a long format listing of all files including the . and .. directory entries.

Listing 44. Long directory listing

```
[ian@echidna lpi103]$ ls -al
total 56
drwxrwxr-x    2 ian      ian          4096 Sep 30 15:01 .
drwxr-xr-x   94 ian      ian          8192 Sep 27 12:57 ..
-rw-rw-r--    1 ian      ian             8 Sep 26 15:24 sedtab
-rw-rw-r--    1 ian      ian            24 Sep 23 12:27 text1
-rw-rw-r--    1 ian      ian            25 Sep 23 13:39 text2
-rw-rw-r--    1 ian      ian            84 Sep 25 17:47 text3
-rw-rw-r--    1 ian      ian            26 Sep 25 22:28 text4
-rw-rw-r--    1 ian      ian            24 Sep 26 12:46 text5
-rw-rw-r--    1 ian      ian            98 Sep 26 16:09 text6
-rw-rw-r--    1 ian      ian            15 Sep 23 14:11 xaa
-rw-rw-r--    1 ian      ian             9 Sep 23 14:11 xab
-rw-rw-r--    1 ian      ian            18 Sep 23 14:11 yaa
-rw-rw-r--    1 ian      ian             7 Sep 23 14:11 yab
```

In Listing 44, the first line shows the total number of disk blocks (56) used by the listed files. The remaining fields tell you about the file.

- The first field (drwxrwxr-x or -rw-rw-r-- in this case) tells us whether the file is a directory (d) or a regular file (-). You may also see symbolic links (l) which we shall learn about later or other values for some special files (such as files in the /dev filesystem). The type is followed by three sets of permissions (such as rwx or r--) for the owner, the members of the owner's group and everyone. The three values respectively indicate whether the user, group or everyone has read (r), write (w) or execute (x) permission. Other uses such as setuid are covered later.
- The next field is a number which tells us the number of *hard links* to the file. We said that the inode contains information about the file. The file's directory entry contains a hard link (or pointer) to the inode for the file, so every entry listed should have at least one hard link. Directory entries have an additional one for the . entry and one for each subdirectory's .. entry. So we can see from the above listing that my home directory has quite a few subdirectories
- The next two fields are the file's owner and the owner's primary group. Some systems, such as the Red Hat system, default to providing a separate group for each user. On other systems, all users may be in one or perhaps a few groups.
- The next field contains the length of the file.
- The penultimate field contains the timestamp of the last modification.
- And the final field contains the name of the file or directory.

The `-i` option of the `ls` command will display the inode numbers for you. We will use this later in this tutorial and also when we discuss hard and symbolic links in the tutorial for Topic 104.

Multiple files

You can also specify multiple parameters to the `ls` command, where each name is either that of a file or of a directory. If a name is that of a directory, then the `ls` command lists the contents of the directory rather than information about the entry itself. In our example, suppose we wanted information about the lpi103 directory

entry itself as it is listed in the parent directory. the command `ls -l ../lpi103` would give us a listing like the previous example. Listing 45 shows how to use `ls -ld` and also how to list entries for multiple files or directories.

Listing 45. Using ls -d

```
[ian@echidna lpi103]$ ls -ld ../lpi103 sedtab xaa
drwxrwxr-x    2 ian      ian          4096 Oct  2 18:49 ../lpi103
-rw-rw-r--    1 ian      ian             8 Sep 26 15:24 sedtab
-rw-rw-r--    1 ian      ian            15 Sep 23 14:11 xaa
```

Note that the modification time for lpi103 is different to that in the previous listing. Also, as in the previous listing, it is different to the timestamps of any of the files in the directory. Is this what you would expect? Not normally. However, in developing this tutorial, I created some extra examples and then deleted them, so the directory time stamps reflect that fact. We will talk more about file times later in this section when we discuss finding files.

Sorting the output

By default, `ls` lists files alphabetically. There are a number of options for sorting the output. For example, `ls -t` will sort by modification time (newest to oldest) while `ls -lS` will produce a long listing sorted by size (largest to smallest). Adding `-r` will reverse the sort order. for example, use `ls -lrt` to produce a long listing sorted from oldest to newest. Consult the man page for other ways you can list files and directories.

Copy, move and delete

We have now learned some ways to create files, but suppose we want to make copies of files, rename files, move them around the filesystem hierarchy, or even delete them. We use three short commands for these purposes.

cp

is used to make a copy of one or more files. You **must** give at least two names, one (or more *source* names and one *target* name. If you specify two file names, then the first is copied to the second. Either source or target file name may include a path specification. If you specify a directory as the last name, then you may specify multiple files to be copied into it. All files will be copied from their existing locations and the copies will have the same file names as the as the original files. Note that that there is no default assumption of the target being the current directory as in DOS and Windows operating systems.

mv

is used to *move* or *rename* one or more files or directories. In general, the names you may use follow the same rules as for copying with `cp`; you can rename a single file or move a set of files into a new directory. Since the name is only a directory entry that links to an inode, it should be no surprise that the inode number does not change **unless** the file is moved to another filesystem, in which case moving it behaves more like a copy followed by deleting the original.

rm
: is used to *remove* one or more files. We will see how to remove directories shortly.

Listing 46 illustrates the use of `cp` and `mv` to make some backup copies of our text files. We also use `ls -i` to show inodes for some of our files.

1. We first make a copy of our text1 file as text1.bkp.
2. We then decide to create a backup subdirectory using the `mkdir` command
3. We make a second backup copy of text 1, this time in the backup directory and show that all three files have different inodes.
4. We then move our text1.bkp to the backup directory and after that rename it to be more consistent with the second backup. While we could have done this with a single command we use two here for illustration.
5. We check the inodes again and confirm that text1.bkp with inode 2129019 is no longer in our lpi103 directory, but that the inode is that of text1.bkp.1 in the backup directory.

Listing 46. Copying and moving files

```
[ian@echidna lpi103]$ cp text1 text1.bkp
[ian@echidna lpi103]$ mkdir backup
[ian@echidna lpi103]$ cp text1 backup/text1.bkp.2
[ian@echidna lpi103]$ ls -i text1 text1.bkp backup
2128984 text1  2129019 text1.bkp

backup:
1564497 text1.bkp.2
[ian@echidna lpi103]$ mv text1.bkp backup
[ian@echidna lpi103]$ mv backup/text1.bkp backup/text1.bkp.1
[ian@echidna lpi103]$ ls -i text1 text1.bkp backup
ls: text1.bkp: No such file or directory
2128984 text1

backup:
2129019 text1.bkp.1  1564497 text1.bkp.2
```

Normally, the `cp` command will copy a file over an existing copy, if the existing file is writable. On the other hand, the `mv` will not move or rename a file if the target exists. There are several useful options relevant to this behavior of `cp` and `mv`.

-f or --force
: will cause cp to attempt to remove an existing target file even if it is not writable.

-i or --interactive
: will ask for confirmation before attempting to replace and existing file

-b or --backup
: will make a backup of any files that would be replaced.

As usual, consult the man pages for full details on these and other options for copying and moving.

In Listing 47, we illustrate copying with backup and then file deletion.

Listing 47. Backup copies and file deletion

```
[ian@echidna lpi103]$ cp text2 backup
[ian@echidna lpi103]$ cp --backup=t text2 backup
[ian@echidna lpi103]$ ls backup
text1.bkp.1  text1.bkp.2  text2  text2.~1~
[ian@echidna lpi103]$ rm backup/text2 backup/text2.~1~
[ian@echidna lpi103]$ ls backup
text1.bkp.1  text1.bkp.2
```

Note that the `rm` command also accepts the `-i` (interactive) and `-f` (force options. Once you remove a file using `rm`, the filesystem no longer has access to it. Some systems default to setting an alias `alias rm='rm -i'` for the root user to help prevent inadvertent file deletion. This is also a good idea if you are nervous about what you might accidentally delete.

Before we leave this discussion, it should be noted that the `cp` command defaults to creating a new timestamp for the new file or files. The owner and group are also set to the owner and group of the user doing the copying. The `-p` option may be used to preserve selected attributes. Note that the root user may be the only user who can preserve ownership. See the man page for details.

Mkdir and rmdir

We have already seen how to create a directory with `mkdir`. Now we will look further at mkdir and introduce its analog for removing directories, `rmdir`.

Mkdir

Suppose we are in our lpi103 directory and we wish to create subdirectories dir1 and dir2. The `mkdir`, like the commands we have just been reviewing, will handle multiple directory creation requests in one pass as shown in Listing 48.

Listing 48. Creating multiple directories

```
[ian@echidna lpi103]$ mkdir dir1 dir2
```

Note that there is no output on successful completion, although you could use `echo $?` to confirm that the exit code is really 0.

If, instead, you wanted to create a nested subdirectory, such as d1/d2/d3, this would fail because the d1 and d2 directories do not exist. Fortunately, `mkdir` has a `-p` option which allows it to create any required parent directories. Listing 49 illustrates this.

Listing 49. Creating parent directories

```
[ian@echidna lpi103]$ mkdir d1/d2/d3
mkdir: cannot create directory `d1/d2/d3': No such file or directory
```

```
[ian@echidna lpi103]$ echo $?
1
[ian@echidna lpi103]$ mkdir -p d1/d2/d3
[ian@echidna lpi103]$ echo $?
0
```

Rmdir

Removing directories using the `rmdir` command is the opposite of creating them. Again, there is a `-p` option to remove parents as well. You can only remove a directory with `rmdir` if it empty as there is no option to force removal. We'll see another way to accomplish that particular trick when we look at recursive manipulation. Once you learn that you will probably seldom use `rmdir` on the command line, but it is good to know about it nevertheless.

To illustrate directory removal, we copied our text1 file into the directory d1/d2 so that it is no longer empty. We then used `rmdir` to remove all the directories we just created with `mkdir`. As you can see, d1 and d2 were not removed because d2 was not empty. The other directories were removed. Once we remove the copy of text1 form d2, we can remove d1 and d2 with a single invocation of `rmdir -p`.

Listing 50. Removing directories

```
[ian@echidna lpi103]$ cp text1 d1/d2
[ian@echidna lpi103]$ rmdir -p d1/d2/d3 dir1 dir2
rmdir: `d1/d2': Directory not empty
[ian@echidna lpi103]$ ls . d1/d2
.:
backup  sedtab  text2  text4  text6  xab  yab
d1      text1   text3  text5  xaa    yaa

d1/d2:
text1
[ian@echidna lpi103]$ rm d1/d2/text1
[ian@echidna lpi103]$ rmdir -p d1/d2
```

Recursive manipulation

In the remaining few parts of this section we will look at various operations for handling multiple files and for recursively manipulating part of a directory tree.

Recursive listing

The `ls` command has a `-R` (note upper case 'R') option for listing a directory and all its subdirectories. The recursive option applies only to directory names; it will not find all the files called, say 'text1' in a directory tree. You may use other options that we have seen already along with `-R`. A recursive listing of our lpi103 directory, including inode numbers, is shown in Listing 51.

Listing 51. Recursive directory listing

```
[ian@echidna lpi103]$ ls -iR ~/lpi103
/home/ian/lpi103:
1564496 backup  2128985 text2  2128982 text5  2128987 xab
2128991 sedtab  2128990 text3  2128995 text6  2128988 yaa
2128984 text1   2128992 text4  2128986 xaa    2128989 yab

/home/ian/lpi103/backup:
2129019 text1.bkp.1  1564497 text1.bkp.2
```

Recursive copy

You can use the `-r` (or `-R` or `--recursive`) option to cause the `cp` command to descend into source directories and copy the contents recursively. To prevent an infinite recursion, the source directory itself may not be copied. Listing 52 shows how to copy everything in our lpi103 directory to a copy1 subdirectory. We use `ls -R` to show the resulting directory tree.

Listing 52. Recursive copy

```
[ian@echidna lpi103]$ cp -pR . copy1
cp: cannot copy a directory, `.', into itself, `copy1'
[ian@echidna lpi103]$ ls -R
.:
backup  sedtab  text2  text4  text6  xab  yab
copy1   text1   text3  text5  xaa    yaa

./backup:
text1.bkp.1  text1.bkp.2

./copy1:
backup  text1  text3  text5  xaa  yaa
sedtab  text2  text4  text6  xab  yab

./copy1/backup:
text1.bkp.1  text1.bkp.2
```

Recursive deletion

We mentioned earlier that `rmdir` only removes empty directories. We can use the `-r` (or `-R` or `--recursive`) option to cause the `rm` command to remove both files **and** directories as shown in Listing 53 where we remove the copy1 directory that we just created, along with its contents, including the backup subdirectory and its contents.

Listing 53. Recursive deletion

```
[ian@echidna lpi103]$ rm -r copy1
[ian@echidna lpi103]$ ls -R
.:
backup  text1  text3  text5  xaa  yaa
sedtab  text2  text4  text6  xab  yab

./backup:
text1.bkp.1  text1.bkp.2
```

If you have files that are not writable by you, you may need to add the `-f` option to force removal. This is often done by the root user when cleaning up, but be warned that you can lose valuable data if you are not careful.

Wildcards and globbing

Often, you may need to perform a single operation on many filesystem objects, without operating on the entire tree as we have just done with recursive operations. For example, you might want to find the modification times of all the text files we created in lpi103, without listing the split files. Although this is easy with our small directory, it is much harder in a large filesystem.

To solve this problem, use the wildcard support that is built in to the bash shell. This support, also called "globbing" (because it was originally implemented as a program called /etc/glob), lets you specify multiple files using wildcard pattern.

A string containing any of the characters '?', '*' or '[', is a *wildcard pattern*. Globbing is the process by which the shell (or possibly another program) expands these patterns into a list of pathnames matching the pattern. The matching is done as follows.

?

matches any single character.

*****

matches any string, including an empty string.

[

introduces a *character class*. A character class is a non-empty string, terminated by a ']'. A match means matching any single character enclosed by the brackets. There are a few special considerations.

- The '*' and '?' characters match themselves. If you use these in filenames, you will need to be really careful about appropriate quoting or escaping.
- Since the string must be non-empty and terminated by ']', you must put ']' **first** in the string if you want to match it.
- The '-' character between two others represents a range which includes the two other characters and all between in the collating sequence. For example, [0-9a-fA-F] represents any upper or lower case hexadecimal digit. You can match a '-' by putting it either first or last within a range.
- The '!' character specified as the first character of a range complements the range so that it matches any character except the remaining characters. For example [!0-9] means any character except the digits 0 through 9. A '!' in any position other than the first matches itself. Remember that '!' is also used with the shell history function, so you need to be careful to properly escape it.

Globbing is applied separately to each component of a path name. You cannot match a '/', nor include one in a range. You can use it anywhere that you might specify multiple file or directory names, for example in the `ls`, `cp`, `mv` or `rm` commands. In Listing 54, we first create a couple of oddly named files and then use the `ls` and `rm` commands with wildcard patterns.

Listing 54. Wildcard pattern examples

```
[ian@echidna lpi103]$ echo odd1>'text[*?!1]'
[ian@echidna lpi103]$ echo odd2>'text[2*?!]'
[ian@echidna lpi103]$ ls
backup  text1       text2       text3  text5  xaa  yaa
sedtab  text[*?!1]  text[2*?!]  text4  text6  xab  yab
[ian@echidna lpi103]$ ls text[2-4]
text2  text3  text4
[ian@echidna lpi103]$ ls text[!2-4]
text1  text5  text6
[ian@echidna lpi103]$ ls text*[2-4]*
```

```
text2  text[2*?!]  text3  text4
[ian@echidna lpi103]$ ls text*[!2-4]* # Surprise!
text1  text[*?!1]  text[2*?!]  text5  text6
[ian@echidna lpi103]$ ls text*[!2-4] # More surprise!
text1  text[*?!1]  text[2*?!]  text5  text6
[ian@echidna lpi103]$ echo text*>text10
[ian@echidna lpi103]$ ls *\!*
text[*?!1]  text[2*?!]
[ian@echidna lpi103]$ ls *[x\!]*
text1       text2       text3  text5  xaa
text[*?!1]  text[2*?!]  text4  text6  xab
[ian@echidna lpi103]$ ls *[y\!]*
text[*?!1]  text[2*?!]  yaa  yab
[ian@echidna lpi103]$ ls tex?[[]*
text[*?!1]  text[2*?!]
[ian@echidna lpi103]$ rm tex?[[]*
[ian@echidna lpi103]$ ls *b*
sedtab  xab  yab

backup:
text1.bkp.1  text1.bkp.2
[ian@echidna lpi103]$ ls backup/*2
backup/text1.bkp.2
[ian@echidna lpi103]$ ls -d .*
.  ..
```

Notes:

1. Complementation in conjunction with '*' can lead to some surprises. The pattern '*[!2-4]' matches the longest part of a name that does not have 2, 3, or 4 following it, which is matched by **both** text[*?!1] and text[2*?!]. So now both surprises should be clear.

2. As with earlier examples of `ls`, if pattern expansion results in a name that is a directory name and the `-d` option is not specified, then the contents of that directory will be listed (as in our example above for the pattern '*b*').

3. If a filename starts with a period (.) then that character must be matched explicitly. Notice that only the last `ls` command listed the two special directory entries (. and ..).

Remember that any wildcard characters in a command are liable to be expanded by the shell which may lead to unexpected results. Furthermore, If you specify a pattern that does not match any filesystem objects then POSIX requires that the original pattern string be passed to the command. We illustrate this in Listing 55. Some earlier implementations passed a null list to the command, so you may run into old scripts that give unusual behavior. We illustrate these points in Listing 55.

Listing 55. Wildcard pattern surprises

```
[ian@echidna lpi103]$ echo text*
text1 text2 text3 text4 text5 text6
[ian[ian@echidna lpi103]$ echo "text*"
text*
@echidna lpi103]$ echo text[[\!?]z??
text[[!?]z??
```

For more information on globbing, look at `man 7 glob`. You will need the section number as there is also glob information in section 3. The best way to understand all the various shell interactions is by practice, so try these wildcards out whenever you have a chance. Remember to try `ls` to check your wildcard pattern before

committing it to whatever `cp`, `mv` or worse, `rm` might unexpectedly do for you.

Touching files

We will now look at the `touch` command which can update file access and modification times or create empty files. In the next part we will see how to use this information to find files and directories. We will use the lpi103 directory that we created earlier in this tutorial. We will also look

touch

The `touch` command with no options takes one or more filenames as parameters and updates the **modification** time of the files. This is the same timestamp normally displayed with a long directory listing. In Listing 56, we use our old friend `echo` to create a small file called f1, and then use a long directory listing to display the modification time (or *mtime*). In this case, it happens also to be the time the file was created. We then use the `sleep` command to wait for 60 seconds and run `ls` again. Note that the timestamp for the file has changed by a minute.

Listing 56. Updating modification time with touch

```
[ian@echidna lpi103]$ echo xxx>f1; ls -l f1; sleep 60; touch f1; ls -l f1
-rw-rw-r--    1 ian      ian             4 Nov  4 15:57 f1
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
```

If you specify a filename for a file that does not exist, then `touch` will normally create an empty file for you, unless you specify the `-c` or `--no-create` option. Listing 57 illustrates both these commands. Note that only f2 is created.

Listing 57. Creating empty files with touch

```
[ian@echidna lpi103]$ touch f2; touch -c f3; ls -l f*
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 16:12 f2
```

The `touch` command can also set a file's mtime to a specific date and time using either the `-d` or `-t` options. The `-d` is very flexible in the date and time formats that it will accept, while the `-t` option needs at least an MMDDhhmm time with optional year and seconds values. Listing 58 shows some examples.

Listing 58. Setting mtime with touch

```
[ian@echidna lpi103]$ touch -t 200511051510.59 f3
[ian@echidna lpi103]$ touch -d 11am f4
[ian@echidna lpi103]$ touch -d "last fortnight" f5
[ian@echidna lpi103]$ touch -d "yesterday 6am" f6
[ian@echidna lpi103]$ touch -d "2 days ago 12:00" f7
[ian@echidna lpi103]$ touch -d "tomorrow 02:00" f8
[ian@echidna lpi103]$ touch -d "5 Nov" f9
[ian@echidna lpi103]$ ls -lrt f*
-rw-rw-r--    1 ian      ian             0 Oct 24 12:32 f5
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 16:12 f2
-rw-rw-r--    1 ian      ian             0 Nov  5 00:00 f9
-rw-rw-r--    1 ian      ian             0 Nov  5 12:00 f7
-rw-rw-r--    1 ian      ian             0 Nov  5 15:10 f3
-rw-rw-r--    1 ian      ian             0 Nov  6 06:00 f6
-rw-rw-r--    1 ian      ian             0 Nov  7 11:00 f4
-rw-rw-r--    1 ian      ian             0 Nov  8  2005 f8
```

If you're not sure what date a date expression might resolve to, you can use the `date` command to find out. It also accepts the `-d` option and can resolve the same kind of date formats that `touch` can.

You can use the `-r` (or `--reference`) option along with a *reference filename* to indicate that `touch` (or `date`) should use the timestamp of an existing file. Listing 59 shows some examples.

Listing 59. Timestamps from reference files

```
[ian@echidna lpi103]$ date
Mon Nov  7 12:40:11 EST 2005
[ian@echidna lpi103]$ date -r f1
Fri Nov  4 15:58:27 EST 2005
[ian@echidna lpi103]$ touch -r f1 f1a
[ian@echidna lpi103]$ ls -l f1*
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 15:58 f1a
```

A Linux system records both a file *modification* time and a file *access* time. Both timestamps are set to the same value when a file is created, and both are reset when it is modified. If a file is accessed at all, then the access time is updated, even if the file is not modified. For our last example with `touch`, we will look at file *access* times. The `-a` (or `--time=atime`, `--time=access` or `--time=use`) option specify that the access time should be updated. Listing 60 uses the `cat` command to access the f1 file and display its contents. We then use `ls -l` and `ls -lu` to display the modification and access times respectively for f1 and f1a, which we created using f1 as a reference file. We then reset the access time of f1 to that of f1a using `touch -a`.

Listing 60. Access time and modification time

```
[ian@echidna lpi103]$ cat f1
xxx
[ian@echidna lpi103]$ ls -lu f1*
-rw-rw-r--    1 ian      ian             4 Nov  7 14:13 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 15:58 f1a
[ian@echidna lpi103]$ ls -l f1*
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 15:58 f1a
[ian@echidna lpi103]$ cat f1
xxx
[ian@echidna lpi103]$ ls -l f1*
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 15:58 f1a
[ian@echidna lpi103]$ ls -lu f1*
-rw-rw-r--    1 ian      ian             4 Nov  7 14:13 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 15:58 f1a
[ian@echidna lpi103]$ touch -a -r f1a f1
[ian@echidna lpi103]$ ls -lu f1*
-rw-rw-r--    1 ian      ian             4 Nov  4 15:58 f1
-rw-rw-r--    1 ian      ian             0 Nov  4 15:58 f1a
```

For more complete information on the many allowable time and date specifications see the man or info pages for the touch and date commands.

Finding files

In the final topic for this part of the tutorial, we will look at the `find` command which is used to find files in one or more directory trees, based on criteria such as name, timestamp, or size. Again, we will use the lpi103 directory that we created earlier in this tutorial.

find

The `find` command will search for files or directories using all or part of the name, or by other search criteria, such as size, type, file owner, creation date, or last access date. The most basic find is a search by name or part of a name. Listing 61 shows an example from our lpi103 directory where we first search for all files that have either a '1' or a 'k' in their name, then perform some path searches that we will explain in the notes below.

Listing 61. Finding files by name

```
[ian@echidna lpi103]$ find . -name "*[1k]*"
./text1
./f1
./backup
./backup/text1.bkp.2
./backup/text1.bkp.1
./f1a
[ian@echidna lpi103]$ find . -ipath "*ACK*1"
./backup/text1.bkp.1
[ian@echidna lpi103]$ find . -ipath "*ACK*/*1"
./backup/text1.bkp.1
[
```

Notes:

1. The patterns that you may use are shell wildcard patterns like those we saw earlier when we discussed under Wildcards and globbing.
2. You can use `-path` instead of `-name` to match full paths instead of just base file names. In this case, the pattern **may** span path components.
3. If you want case-insensitive search as shown in the use of `ipath` above, precede the `find` options that search on a string or pattern with an 'i'
4. If you want to find a file or directory whose name begins with a dot, such as .bashrc or the current directory (.), then you **must** specify a leading dot as part of the pattern. Otherwise, name searches will ignore these files or directories.

In the first example above, we found both files and a directory (./backup). Use the `-type` parameter along with one-letter type to restrict the search. Use 'f' for regular files, 'd' for directories, and 'l' for symbolic links. See the man page for `find` for other possible types. Listing 62 shows the result of searching for directories (`-type d`).

Listing 62. Find files by type

```
[ian@echidna lpi103]$ find . -type d
.
./backup
[ian@echidna lpi103]$ find . -type d -name "*"
```

```
./backup
```

Note that the `-type d`.specification without any form of name specification displays directories that have a leading dot in their names (only the current directory in this case).

We can also search by file size, either for a specific size (n) or for files that are either larger (+n) or smaller than a given value (-n). By using both upper and lower size bounds, we can find files whose size is within a given range. By default the `-size` option of `find` assumes a unit of 'b' for 512-byte blocks. Among other choices, specify 'c' for bytes, or 'k' for kilobytes. In Listing 63 we first find all files with size 0, and then all with size of either 24 or 25 bytes. Note that specifying `-empty` instead of `-size 0` also finds empty files.

Listing 63. Finding files by size

```
[ian@echidna lpi103]$ find . -size 0
./f2
./f3
./f4
./f5
./f6
./f7
./f8
./f9
./f1a
[ian@echidna lpi103]$ find . -size -26c -size +23c -print
./text1
./text2
./text5
./backup/text1.bkp.2
./backup/text1.bkp.1
```

Listing 63 introduces the `-print` option which is an example of an *action* that may be taken on the results returned by the search. In the bash shell, this is the default action if no action is specified. On some systems and some shells, an action is required, otherwise there is no output.

Other actions include `-ls` which prints file information equivalent to that from the `ls -lids` command, or `-exec` which executes a command for each file. The `-exec` must be terminated by a semi-colon which must be escaped to avoid the shell interpreting it first. Also specify {} wherever you want the returned file used in the command. as we saw above, the curly braces also have meaning to the shell and should be escaped (or quoted). Listing 64 shows how the `-ls` and the `-exec` options can be used to list file information.

Listing 64. Finding and acting on files

```
[ian@echidna lpi103]$ find . -size -26c -size +23c -ls
2128984    4 -rw-rw-r--   1 ian    ian      24 Sep 23 12:27 ./text1
2128985    4 -rw-rw-r--   1 ian    ian      25 Sep 23 13:39 ./text2
2128982    4 -rw-rw-r--   1 ian    ian      24 Sep 26 12:46 ./text5
1564497    4 -rw-rw-r--   1 ian    ian      24 Oct  4 09:45 ./backup/text1.bkp.2
2129019    4 -rw-rw-r--   1 ian    ian      24 Oct  4 09:43 ./backup/text1.bkp.1
[ian@echidna lpi103]$ find . -size -26c -size +23c -exec ls -l '{}' \;
-rw-rw-r--    1 ian    ian      24 Sep 23 12:27 ./text1
-rw-rw-r--    1 ian    ian      25 Sep 23 13:39 ./text2
-rw-rw-r--    1 ian    ian      24 Sep 26 12:46 ./text5
-rw-rw-r--    1 ian    ian      24 Oct  4 09:45 ./backup/text1.bkp.2
-rw-rw-r--    1 ian    ian      24 Oct  4 09:43 ./backup/text1.bkp.1
```

The `-exec` option can be used for as many purposes as your imagination can dream up. For example:
`find . -empty -exec rm '{}' \;`
removes all the empty files in a directory tree, while
`find . -name "*.htm" -exec mv '{}' '{}l' \;`
renames all .htm file to .html files.

For our final examples, we use the timestamps described with the `touch` command to locate files having particular timestamps. Listing 65 shows three examples:

1. When used with `-mtime -2`, the `find` command finds all files modified within the last two days. A day in this case is a 24-hour period relative to the current date and time. Note that you would use `-atime` if you wanted to find files based on access time rather than modification time.

2. Adding the `-daystart` option means that we want to consider days as calendar days, starting at midnight. Now the f3 file is excluded from the list.

3. Finally, we show how to use a time range in minutes rather than days to find files modified between one hour (60 minutes) and 10 hours (600 minutes) ago.

Listing 65. Finding files by timestamp

```
[ian@echidna lpi103]$ date
Mon Nov  7 14:59:02 EST 2005
[ian@echidna lpi103]$ find . -mtime -2 -type f -exec ls -l '{}' \;
-rw-rw-r--    1 ian      ian             0 Nov  5 15:10 ./f3
-rw-rw-r--    1 ian      ian             0 Nov  7 11:00 ./f4
-rw-rw-r--    1 ian      ian             0 Nov  6 06:00 ./f6
-rw-rw-r--    1 ian      ian             0 Nov  8  2005 ./f8
[ian@echidna lpi103]$ find . -daystart -mtime -2 -type f -exec ls -l '{}' \;
-rw-rw-r--    1 ian      ian             0 Nov  7 11:00 ./f4
-rw-rw-r--    1 ian      ian             0 Nov  6 06:00 ./f6
-rw-rw-r--    1 ian      ian             0 Nov  8  2005 ./f8
[ian@echidna lpi103]$ find . -mmin -600 -mmin +60 -type f -exec ls -l '{}' \;
-rw-rw-r--    1 ian      ian             0 Nov  7 11:00 ./f4
```

The man pages for the `find` command can help you learn the extensive range of options that we cannot cover in this brief introduction.

Section 5. Streams, pipes, and redirects

This section covers material for topic 1.103.4 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 5.

In this section, you learn about the following topics:

- Redirecting the standard IO streams: standard input, standard output, and

standard error

- Piping output from one command to the input of another
- Sending output to both stdout and a file
- Using command output as arguments to another command

Redirecting standard IO

Recall that shells use three standard I/O *streams*.

1. *stdout* is the *standard output stream* which displays output from commands. It has file descriptor 1.
2. *stderr* is the *standard error stream* which displays error output from commands. It has file descriptor 2.
3. *stdin* is the *standard input stream* which provides input to commands. It has file descriptor 0.

Input streams provide input to programs, usually from terminal keystrokes. Output streams print text characters, usually to the terminal. The terminal was originally an ASCII typewriter or display terminal, but is now more often a window on a graphical desktop.

As we saw in Text streams and filters we can redirect standard output to a file or to the standard input of another command and we can redirect standard input from a file or from the output of another command.

Redirecting output

There are two ways to redirect output:

n>
: redirects output from file descriptor *n* to a file. You must have write authority to the file. If the file does not exist, it is created. If it does exist, the existing contents are usually lost without any warning.

n>>
: also redirects output from file descriptor *n* to a file. Again, you must have write authority to the file. If the file does not exist, it is created. If it does exist, the output is appended to the existing file.

The *n* in n> or n>> refers to the *file descriptor*. If it omitted, then standard output is assumed. Listing 66 illustrates using redirection to separate the standard output and standard error from the `ls` command using files we created earlier in our lpi103 directory. We also illustrate appending output to existing files.

Listing 66. Output redirection

```
[ian@echidna lpi103]$  ls x* z*
```

```
ls: z*: No such file or directory
xaa  xab
[ian@echidna lpi103]$ ls x* z* >stdout.txt 2>stderr.txt
[ian@echidna lpi103]$  ls w* y*
ls: w*: No such file or directory
yaa  yab
[ian@echidna lpi103]$ ls w* y* >>stdout.txt 2>>stderr.txt
[ian@echidna lpi103]$ cat stdout.txt
xaa
xab
yaa
yab
[ian@echidna lpi103]$ cat stderr.txt
ls: z*: No such file or directory
ls: w*: No such file or directory
```

We said that output redirection using n> usually overwrites existing files. You can control this with the `noclobber` option of the `set` builtin. If it has been set, you can override it using n>| as shown in Listing 67.

Listing 67. Output redirection with noclobber

```
[ian@echidna lpi103]$ set -o noclobber
[ian@echidna lpi103]$ ls x* z* >stdout.txt 2>stderr.txt
-bash: stdout.txt: cannot overwrite existing file
[ian@echidna lpi103]$ ls x* z* >|stdout.txt 2>|stderr.txt
[ian@echidna lpi103]$ cat stdout.txt
xaa
xab
[ian@echidna lpi103]$ cat stderr.txt
ls: z*: No such file or directory
[ian@echidna lpi103]$ set +o noclobber #restore original noclobber setting
```

Sometimes you may want to redirect both standard output and standard error into a file. Thi is often done for automated processes or background jobs so that you can review the output later. Use &> or &>> to redirect both standard output and standard error to the same place. Another way of doing this is to redirect file descriptor *n* and then redirect file descriptor *m* to the same place using the construct m>&n or m>>&n. The order in which outputs are redirected is important. For example,
`command 2>&1 >output.txt`
is not the same as
`command >output.txt 2>&1`
We illustrate these redirections in Listing 68. Notice in the last command that standard output was redirected after standard error, so the standard error output still goes to the terminal window.

Listing 68. Redirecting two streams to one file

```
[ian@echidna lpi103]$ ls x* z* &>output.txt
[ian@echidna lpi103]$ cat output.txt
ls: z*: No such file or directory
xaa
xab
[ian@echidna lpi103]$ ls x* z* >output.txt 2>&1
[ian@echidna lpi103]$ cat output.txt
ls: z*: No such file or directory
xaa
xab
[ian@echidna lpi103]$ ls x* z* 2>&1 >output.txt
ls: z*: No such file or directory
[ian@echidna lpi103]$ cat output.txt
xaa
xab
```

At other times you may want to ignore either standard output or standard error

entirely. To do this, redirect the appropriate stream to /dev/null. In Listing 69 we show how to ignore error output from the `ls` command.

Listing 69. Ignoring output using /dev/null

```
[ian@echidna lpi103]$ ls x* z* 2>/dev/null
xaa  xab
[ian@echidna lpi103]$ cat /dev/null
```

Redirecting input

Just as we can redirect the stdout and stderr streams, so too we can redirect stdin from a file, using the < operator. If you recall, in our discussion of sort and uniq that we used the `tr` command to replace the spaces in our text1 file with tabs. In that example we used the output from the `cat` command to create standard input for the `tr` command. Instead of needlessly calling `cat`, we can now use input redirection to translate the spaces to tabs, as shown in Listing 70.

Listing 70. Input redirection

```
[ian@echidna lpi103]$ tr ' ' '\t'<text1
1       apple
2       pear
3       banana
```

Shells, including bash, also have the concept of a *here-document*, which is another form of input redirection. This uses the << along with a word, such as END, for a marker or sentinel to indicate the end of the input. We illustrate this in Listing 71.

Listing 71. Input redirection with a here-document

```
[ian@echidna lpi103]$ sort -k2 <<END
> 1 apple
> 2 pear
> 3 banana
> END
1 apple
3 banana
2 pear
```

Remember how you created the text2 file back in Listing 23? You may wonder why you couldn't have typed just `sort -k2`, entered your data, and then pressed **Ctrl-d** to signal end of input. And the short answer is that you could, but you would not have learned about here-documents. The real answer is that here-documents are more often used in shell scripts (which are covered in the tutorial for topic 109 on shells, scripting, programming, and compiling). A script doesn't have any other way of signaling which lines of the script should be treated as input. Because shell scripts make extensive use of tabbing to provide indenting for readability, there is another twist to here-documents. if you use <<- instead of just <<, then leading tabs are stripped. In Listing 72 we use the same technique for creating a captive tab character that we used in Listing 42. We then create a very small shell script containing two `cat` commands which each read from a here-document. Finally, we use the `.` (dot) command to *source* the script, which means to run it in the current shell context.

Listing 72. Input redirection with a here-document

```
[ian@echidna lpi103]$ ht=$(echo -en "\t")
[ian@echidna lpi103]$ cat<<END>ex-here.sh
> cat <<-EOF
> apple
> EOF
> ${ht}cat <<-EOF
> ${ht}pear
> ${ht}EOF
> END
[ian@echidna lpi103]$ cat ex-here.sh
cat <<-EOF
apple
EOF
        cat <<-EOF
        pear
        EOF
[ian@echidna lpi103]$ . ex-here.sh
apple
pear
```

Pipelines

In the section on Text streams and filters we described text *filtering* as the process of taking an input stream of text and performing some conversion on the text before sending it to an output stream. We also said that filtering is most often done by constructing a *pipeline* of commands where the output from one command is *piped* or *redirected* to be used as input to the next. Using pipelines in this way is not restricted to text streams., although that is often where they are used.

Piping stdout to stdin

As we have already seen, we use the | (pipe) operator between two commands to direct the stdout of the first to the stdin of the second. We construct longer pipelines by adding more commands and more pipe operators as shown in Listing 73.

Listing 73. Piping output through several commands

```
command1 | command2 | command3
```

One thing to note is that pipelines **only** pipe stdout to stdin. You cannot use 2| to pipe stderr alone, at least, not with the tools we have learned so far. If stderr has been redirected to stdout, then both streams will be piped. We illustrate this in Listing 74 where we use a pipe to sort both the error and normal output messages from an unlikely `ls` command with four wildcard arguments that are not in alphabetical order.

Listing 74. Piping two output streams

```
[ian@echidna lpi103]$ ls y* x* z* u* q*  2>&1 |sort
ls: q*: No such file or directory
ls: u*: No such file or directory
ls: z*: No such file or directory
xaa
xab
yaa
yab
```

Any of the commands may have options or arguments. Many commands use a hyphen (-) used in place of a filename as an argument to indicate when the input should come from stdin rather than a file. Check the man pages for the command to

be sure. Constructing long pipelines of commands that each have limited capability is a common Linux and UNIX way of accomplishing tasks.

One advantage of pipes on Linux and UNIX systems is that, unlike some other popular operating systems, there is no intermediate file involved with a pipe. The stdout of the first command is **not** written to a file and then read by the second command. If your particular version of `tar` doesn't happen to support unzipping files compressed using `bzip2`, that's no problem. As we saw in the tutorial for Topic 102, you can just use a pipeline like
`bunzip2 -c drgeo-1.1.0.tar.bz2 | tar -xvf -`
to do the task.

Output as arguments

In the section Using the command line we learned about command substitution and how to use the output from a command as part of another. In the previous section on Basic file management we learned how to use the `-i` option of the `find` command to use the output of the `find` command as input for another command. Listing 75 shows three ways of displaying the contents of our text1 and text2 files using these techniques.

Listing 75. Using output as arguments with command substitution and find -exec

```
[ian@echidna lpi103]$ cat `ls text[12]`
1 apple
2 pear
3 banana
9       plum
3       banana
10      apple
[ian@echidna lpi103]$ cat $(find . -name "text[12]")
1 apple
2 pear
3 banana
9       plum
3       banana
10      apple
[ian@echidna lpi103]$ find . -name "text[12]" -exec cat '{}' \;
1 apple
2 pear
3 banana
9       plum
3       banana
10      apple
```

The above methods are fine as far as they go, but they have some limitations. Let's consider the case of a filename containing whitespace (a blank in this case). Look at Listing 76 and see if you can understand what is happening with each of the commands before reading the explanations below.

Listing 76. Using output as arguments with command substitution and find -exec

```
[ian@echidna lpi103]$ echo grapes>"text sample2"
[ian@echidna lpi103]$ cat `ls text*le2`
cat: text: No such file or directory
cat: sample2: No such file or directory
[ian@echidna lpi103]$ cat "`ls text*le2`"
```

```
grapes
[ian@echidna lpi103]$ cat "`ls text*2`"
cat: text2
text sample2: No such file or directory
```

Here is what we did.

- We created a file called "text sample2" containing one line with the word "grapes"
- We attempted to use command substitution to display the contents of "text sample2". This failed because the shell passed **two** parameters to cat, namely text and sample2.
- Being smarter than the shell, we decided to put quotes around the command substitution values. This works
- Finally, we changed the wildcard expression and the output is a very strange looking error. What is happening here is that the shell is giving the `cat` command a **single** parameter which is equivalent to the string that would result from
  echo -e "text2\ntext sample2"
  If this seems strange, try it yourself!

What we need here is some way of delineating the individual file names regardless of whether they consist of a single word or multiple words. We haven't mentioned it previously, but when the output of commands such as `ls` is used in a pipe or command substitution, the output is usually delivered one item per line. One method to handle this is with a `read` builtin in a loop with the `while` builtin. Although beyond the scope of this objective, we illustrate it to whet your appetite and as a lead-in to the solution we present next.

Listing 77. Using while and read in a loop

```
[ian@echidna lpi103]$ ls text*2 | while read l; do cat "$l";done
9       plum
3       banana
10      apple
grapes
```

xargs

Most of the time we want to process lists of files, so what we really need is some way of finding them and then processing them. Fortunately, the `find` command has an option, `-print0`, which separates output filenames with a null character instead of a newline. Commands such as `tar` and `xargs` have a `-0` (or `--null` option that allows them to understand this form of parameter. We have already seen `tar`. The `xargs` command works a little like the `-exec` option of `find`, but there are some important differences as we shall see. First let's look at an example.

Listing 78. Using xargs with -0

```
[ian@echidna lpi103]$ find . -name "text*2" -print0 |xargs -0 cat
9       plum
3       banana
10      apple
1 apple
2 pear
```

```
3 banana
grapes
```

Notice that we now pipe the output from `find` to `xargs`. You don't need the delimited semi-colon on the end of the command and, by default, xargs appends the arguments to the command string. However, we seem to have seven lines of output rather than the expected four. What went wrong?

find again

We can use the `wc` command to check that there are only four lines of output in the two files we thought we were printing. The answer to our problem lies in the fact that `find` is searching our backup directory where it also finds backup/text1.bkp.2 which matches our wildcard pattern. To solve this, we use the `-maxdepth` option of `find` to restrict the search to a depth of one directory, the current one. There's also a corresponding `-mindepth` option which allows you to be quite specific in where you search. Listing 79 illustrates our final solution.

Listing 79. Restricting find to our four desired lines

```
[ian@echidna lpi103]$ ls text*2
text2  text sample2
[ian@echidna lpi103]$ wc text*2
      3       6      25 text2
      1       1       7 text sample2
      4       7      32 total
[ian@echidna lpi103]$ find . -name "text*2" -maxdepth 1 -print0 |xargs -0 cat
9       plum
3       banana
10      apple
grapes
```

More on xargs

There are some other differences between `xargs` and `find -exec`.

- The `xargs` command defaults to passing as many arguments as possible to the command. You may limit the number of input lines used with `-l` or `--max-lines` and a number. Alternatively, you may use `-n` or `--max-args` to limit the number of arguments passed, or `-s` or `--max-chars` to limit the maximum number of characters used in the argument string. If your command can process multiple arguments, it is generally more efficient to process as many as possible at a time.
- You may use '{}' as you did for `find -exec` if you specify the `-i` or `--replace` option. You may change the default of '{}' for the string that indicates where to substitute the input parameter by specifying a value for `-i`. This implies `-l 1`.

Our final examples for `xargs` are shown in Listing 80.

Listing 80. Additional xargs examples

```
[ian@echidna lpi103]$ # pass all arguments at once
[ian@echidna lpi103]$ find . -name "text*2"  |xargs echo
./text2 ./backup/text1.bkp.2 ./text sample2
[ian@echidna lpi103]$ # show the files we created earlier with the touch command
[ian@echidna lpi103]$ ls f[0-n]*|xargs echo
f1 f1a f2 f3 f4 f5 f6 f7 f8 f9
[ian@echidna lpi103]$ # remove them in one stroke
```

```
[ian@echidna lpi103]$ ls f[0-n]*|xargs rm
[ian@echidna lpi103]$ # Use a replace string
[ian@echidna lpi103]$ find . -name "text*2"  |xargs -i echo - '{}' -
- ./text2 -
- ./backup/text1.bkp.2 -
- ./text sample2 -
[ian@echidna lpi103]$ # Limit of one input line per invocation
[ian@echidna lpi103]$ find . -name "text*2"  |xargs -l1 echo
./text2
./backup/text1.bkp.2
./text sample2
[ian@echidna lpi103]$ # Limit of one argument per invocation
[ian@echidna lpi103]$ find . -name "text*2"  |xargs -n1 echo
./text2
./backup/text1.bkp.2
./text
sample2
```

Note that we did not use `-print0` here. Does that explain the final example in Listing 80?

Splitting output

This section wraps up with a brief discussion of one more command. Sometimes you may want to see output on your screen while saving a copy for later. While you **could** do this by redirecting the command output to a file in one window and then using `tail -fn1` to follow the output in another screen, using the `tee` command is easier.

You use `tee` with a pipeline. The arguments are a file (or multiple files) for standard output. The `-a` option appends rather than overwriting files. As we saw earlier in our discussion of pipelines, you need to redirect stderr to stdout before piping to `tee` if you want to save both. Listing 81 shows `tee` being used to save output in two files, f1 and f2.

Listing 81. Splitting stdout with tee

```
[ian@echidna lpi103]$ ls text[1-3]|tee f1 f2
text1
text2
text3
[ian@echidna lpi103]$ cat f1
text1
text2
text3
[ian@echidna lpi103]$ cat f2
text1
text2
text3
```

Section 6. Create, monitor, and kill processes

This section covers material for topic 1.103.5 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 5.

In this section, you learn about the following topics:

- Foreground and background jobs
- Starting processes without a terminal for I/O
- Monitoring and displaying processes
- Sending signals to processes
- Identifying and killing processes

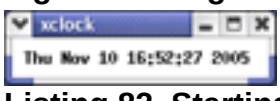
If you stop an reflect for a moment, it is pretty obvious that lots of things are running on your computer, other than the terminal programs we have been running. Indeed, if you are using a graphical desktop, you may have opened more than one terminal window at one time, or perhaps have opened a file browser, internet browser, game, spreadsheet, or other application. So far, our examples have entered commands at a terminal window. The command runs and we wait for it to complete before we do anything else. In the section Using the command line, we encountered the `ps` command which displayed process status and we saw that processes have a Process ID (PID) and a Parent Process id (PPID). In this section, you learn how to do more than one thing at a time using your terminal window.

Foreground and background jobs

When you run a command in your terminal window, such as we have done to this point, you are running it in the *foreground*. Our commands to date have run quickly, but suppose we are running a graphical desktop and would like a digital clock displayed on the desktop. For now, let's ignore the fact that most desktops already have one; we're just using this as an example.

If you have the X Window System installed, you probably also have some utilities such as `xclock` or `xeyes`. Either works for this exercise, but we'll use xclock. The man page explains that you can launch a digital clock on your graphical desktop using the command
`xclock -d -update 1`
The `-update 1` part requests updates every second, otherwise the clock updates only every minute. So let's run this in a terminal window. We should see a clock like Figure 2, and our terminal window should look like Listing 82. If you don't have xclock or the X Window System, we'll show you shortly how to create a poor man's digital clock with your terminal, so you might want to follow along for now and then retry these exercises with that clock.

Figure 2. A digital clock with xclock



Listing 82. Starting xclock

```
[ian@echidna ian]$ xclock -d -update 1
```

Unfortunately, your terminal window no longer has a prompt, so we really need to

get control back. Fortunately, the Bash shell has a *suspend* key, Ctrl-z. Pressing this key combination gets you a terminal prompt again as shown in Listing 83

Listing 83. Suspending xclock with Ctrl-z

```
[ian@echidna ian]$ xclock -d -update 1

[1]+  Stopped                 xclock -d -update 1
[ian@echidna ian]$
```

The clock is still on your desktop, but it has stopped running. Suspending it did exactly that. In fact, if you drag another window over part of it, that part won't even redraw. You also see a terminal output message indicating "[1]+ Stopped". The 1 in this message is a *job number*. You can restart the clock by typing `fg %1`. You could also use the command name or part of it by using `fg %xclock` or `fg %?clo`. Finally, if you just use `fg` with no parameters, you can restart the most recently stopped job, job 1 in this case. Restarting it with `fg` also brings the job right back to the foreground, and you no longer have a shell prompt. What you need to do is place the job in the *background*; a `bg` command takes the same type of job specification as the `fg` command and does exactly that. Listing 84 shows how to bring the xclock job back to the foreground and suspend it using two forms of the `fg command`.. You can suspend it again and place it in the background; the clock continues to run while you do other work at your terminal.

Listing 84. Placing xclock in the background

```
[ian@echidna ian]$ fg %1
xclock -d -update 1

[1]+  Stopped                 xclock -d -update 1
[ian@echidna ian]$ fg %?clo
xclock -d -update 1

[1]+  Stopped                 xclock -d -update 1
[ian@echidna ian]$ bg
[1]+ xclock -d -update 1 &
[ian@echidna ian]$
```

Using "&"

You may have noticed that when we placed the xclock job in the background, the message no longer said "Stopped" and that it was terminated with an ampersand (&). In fact, you don't need to suspend the process to place it in the background at all. You can simply append an ampersand to the command and the shell will start the command (or command list) in the background. Let' start an analog clock with a wheat background and red hands using this method. You should see a clock like that in Figure 3 and terminal output like Listing 85.

Figure 3. An analog clock with xclock

Listing 85. Starting xclock in background with &

```
[ian@echidna ian]$ xclock -bg wheat -hd red -update 1&
[2] 5659
```

Notice that the message is slightly different this time. It represents a job number and a process id (PID). We will cover PIDs and more about status in a moment. For now, let's use the `jobs` command to find out what jobs are running. Add the `-l` option to list PIDs, and you see that job 2 indeed has PID 5659 as shown in Listing 86. Note also that job 2 has a plus sign (+) beside the job number, indicating that it is the *current job*. This job will come to the foreground if no job specification is given with the `fg` command.

Listing 86. Displaying job and process information

```
[ian@echidna ian]$ jobs -l
[1]-  4234 Running                 xclock -d -update 1 &
[2]+  5659 Running                 xclock -bg wheat -hd red -update 1 &
```

Before we address some other issues related to background jobs, let's create a poor man's digital clock. We use the `sleep` command to cause a delay for two seconds and we use the `date` command to print the current date and time. We wrap these commands in a `while` loop with a `do/done` block to crate an infinite loop. Finally we put the whole lot in parentheses to make a command list and put the entire list in the background using an ampersand.

Listing 87. Poor man's digital clock

```
[ian@echidna ian]$ (while sleep 2; do date;done) &
[1] 16291
[ian@echidna ian]$ Thu Nov 10 22:58:02 EST 2005
Thu Nov 10 22:58:04 EST 2005
Thu Nov 10 22:58:06 EST 2005
Thu Nov 10 22:58:08 EST 2005
fThu Nov 10 22:58:10 EST 2005
Thu Nov 10 22:58:12 EST 2005
gThu Nov 10 22:58:14 EST 2005

( while sleep 2; do
    date;
done )
Thu Nov 10 22:58:16 EST 2005
Thu Nov 10 22:58:18 EST 2005
```

As expected, our list is running as job 1 with PID 16291. Every two seconds, the date command runs and a date and time are printed on the terminal. The input that you type is highlighted. A slow typist will have characters interspersed with several lines of output before a full command can be typed. In fact, notice how the 'f' 'g' that we type in to bring the command list to foreground are a couple of lines apart. When we finally get the `fg` command entered, bash displays the command that is now running in our shell, namely, the command list, which is still happily printing the time every two seconds.

Once we succeed in getting the job into the foreground, we can either terminate (or *kill*), or take some other action, In this case, we user Ctrl-c to terminate our 'clock'.

Standard IO and background processes

The output from the `date` command in our previous example is interspersed with echoed characters for the `fg` command that we are trying to type. This raises an interesting issue. What happens to a process if it needs input from stdin?

The terminal process under which we start a background application is called the *controlling terminal*. Unless redirected elsewhere, the stdout and stderr streams from the background process are directed to the controlling terminal. Similarly, the background task expects input from the controlling terminal, but the controlling terminal has no way of directing characters you type to the stdin of a background process. In such a case, the Bash shell suspends the process, so that it is no longer executing. You may bring it to the foreground and supply the necessary input. Listing 88 illustrates a simple case where you can put a command list in the background. After a moment, press **Enter** and the process stops. Bring it to the foreground and provide a line of input followed by Ctrl-d to signal end of input file. The command list completes and we display the file we created.

Listing 88. Waiting for stdin

```
[ian@echidna ian]$ (date; cat - >bginput.txt; date)&
[1] 18648
[ian@echidna ian]$ Fri Nov 11 00:03:28 EST 2005


[1]+  Stopped                 ( date; cat - >bginput.txt; date )
[ian@echidna ian]$ fg
( date; cat - >ginput.txt; date )
input data
Fri Nov 11 00:03:53 EST 2005
[ian@echidna ian]$ cat bginput.txt
input data
```

Jobs without terminals

In practice, we probably want to have standard IO streams for background processes redirected to or from a file. There is another related question; what happens to the process if the controlling terminal closes or the user logs off? The answer depends on the shell in use. If the shell sends a SIGHUP (or hangup) signal, then the application is likely to close. We cover signals shortly, but for now we'll consider another way around this problem.

nohup

The `nohup` command is used to start a command that will ignore hangup signals and will append stdout and stderr to a file. The default file is either nohup.out or $HOME/nohup.out. If the file cannot be written, then the command will not run. If you want output to go somewhere else, redirect stdout, or stderr as we learned in the previous section of this tutorial.

One other aspect of `nohup` is that it will not execute a pipeline or a command list. In the topic Redirecting standard IO we showed how to save a set of commands in a shell script and source it. You can save a pipeline or list in a file and then run it using the `sh` (default shell) or the `bash` command, although you can't use the . or `source` command as we did in the earlier example. The next tutorial in this series (on topic

104, covering Devices, Linux Filesystems, and Filesystem Hierarchy Standard) shows how to make the script file executable, but for now we'll stick to running scripts by sourcing them or by using the `sh` or the `bash` command, Listing 89 shows how we might do this for our poor man's digital clock. Needless to say, having the time written to a file isn't particularly useful, and the file will keep growing, so we'll set the clock to update every 30 seconds instead of every second.

Listing 89. Using nohup with a command list in a script

```
[ian@echidna ian]$ echo "while sleep 30; do date;done">pmc.sh
[ian@echidna ian]$ nohup . pmc.sh&
[1] 21700
[ian@echidna ian]$ nohup: appending output to `nohup.out'

[1]+  Exit 126                nohup . pmc.sh
[ian@echidna ian]$ nohup sh pmc.sh&
[1] 21709
[ian@echidna ian]$ nohup: appending output to `nohup.out'

[ian@echidna ian]$ nohup bash pmc.sh&
[2] 21719
[ian@echidna ian]$ nohup: appending output to `nohup.out'
```

If we display the contents of nohup.out, we see that the first line indicates why we got an exit code of 126 on our first attempt above. Subsequent lines will be output from the two versions of pmc.sh that are now running in background. This is illustrated in Listing 90.

Listing 90. Output from nohup processes

```
[ian@echidna ian]$ cat nohup.out
/bin/nice: .: Permission denied
Fri Nov 11 15:30:03 EST 2005
Fri Nov 11 15:30:15 EST 2005
Fri Nov 11 15:30:33 EST 2005
Fri Nov 11 15:30:45 EST 2005
Fri Nov 11 15:31:03 EST 2005
```

Now let's turn our attention to the status of our processes. If you are are following along and planning to take a break at this point, please stay around as you now have two jobs that are creating ever larger files in your file system. You can use the `fg` command to bring each to foreground, and then use Ctrl-c to terminate it, but if you let them run for a little longer we'll see other ways to monitor and interact with them.

Process status

In the previous part of this section, we had a brief introduction to the `jobs` command and saw how to use it to list the Process IDs (or PIDs) of our jobs.

ps

There is another command, the `ps` command, which we use to display various pieces of process status information. Remember "ps" as an acronym for "process status". The `ps` command accepts zero or more PIDs as argument and displays the associated process status. If we use the `jobs` command with the `-p` option, the

output is simply the PID of the *process group leader* for each job. We'll use this output as arguments to the `ps` command as shown in Listing 91.

Listing 91. Status of background processes

```
[ian@echidna ian]$ jobs
[1]-  Running                 nohup sh pmc.sh &
[2]+  Running                 nohup bash pmc.sh &
[ian@echidna ian]$ jobs -p
21709
21719
[ian@echidna ian]$ ps `jobs -p`
  PID TTY      STAT   TIME COMMAND
21709 pts/3    SN     0:00 sh pmc.sh
21719 pts/3    SN     0:00 bash pmc.sh
```

If we use `ps` with no options we see a list of processes that have our terminal as their controlling terminal as shown in Listing 92.

Listing 92. Displaying status with ps

```
[ian@echidna ian]$ ps
  PID TTY          TIME CMD
20475 pts/3    00:00:00 bash
21709 pts/3    00:00:00 sh
21719 pts/3    00:00:00 bash
21922 pts/3    00:00:00 sleep
21930 pts/3    00:00:00 sleep
21937 pts/3    00:00:00 ps
```

Several options, including `-f` (full), `-j` (jobs), and `-l` (long) give control of how much information is displayed. If we do not specify any PIDs, then another useful option is the `--forest` option, which displays the commands in a tree hierarchy, showing us which process has which other process as a parent. In particular, we see that the `sleep` commands of the previous listing are children of the scripts we have running in background. If we happened to run the command at a different instant, we might see the `date` command listed in the process status instead, but the odds are very small with this script. We illustrate some of these in Listing 93.

Listing 93. More status information

```
[ian@echidna ian]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
ian      20475 20474  0 15:02 pts/3    00:00:00 -bash
ian      21709 20475  0 15:29 pts/3    00:00:00 sh pmc.sh
ian      21719 20475  0 15:29 pts/3    00:00:00 bash pmc.sh
ian      21945 21709  0 15:34 pts/3    00:00:00 sleep 30
ian      21953 21719  0 15:34 pts/3    00:00:00 sleep 30
ian      21954 20475  0 15:34 pts/3    00:00:00 ps -f
[ian@echidna ian]$ ps -j --forest
  PID  PGID   SID TTY          TIME CMD
20475 20475 20475 pts/3    00:00:00 bash
21709 21709 20475 pts/3    00:00:00 sh
21945 21709 20475 pts/3    00:00:00  \_ sleep
21719 21719 20475 pts/3    00:00:00 bash
21953 21719 20475 pts/3    00:00:00  \_ sleep
21961 21961 20475 pts/3    00:00:00 ps
```

Listing other processes

The `ps` commands we have used so far only list processes that were started from your terminal session session (note the SID column in the the second example of Listing 93). To see all the processes with controlling terminals use the `-a` option.

The `-x` option displays processes without a controlling terminal, and the `-e` option displays information for **every** process. Listing 94 shows full format for all the processes with a controlling terminal.

Listing 94. Displaying other processes

```
[ian@echidna ian]$ ps -af
UID        PID  PPID  C STIME TTY          TIME CMD
ian       4234 32537  0 Nov10 pts/0    00:00:00 xclock -d -update 1
ian       5659 32537  0 Nov10 pts/0    00:00:00 xclock -bg wheat -hd red -update
ian      21709 20475  0 15:29 pts/3    00:00:00 sh pmc.sh
ian      21719 20475  0 15:29 pts/3    00:00:00 bash pmc.sh
ian      21969 21709  0 15:35 pts/3    00:00:00 sleep 30
ian      21977 21719  0 15:35 pts/3    00:00:00 sleep 30
ian      21978 20475  0 15:35 pts/3    00:00:00 ps -af
```

Note that this listing includes the two xclock processes that we started earlier from the main graphical terminal of this system (indicated here by pts/0), while the remaining processes displayed are those associated with an ssh (Secure Shell) connection (pts/3 in this case).

There are many more options for `ps`, including a number which provide significant control over what fields are displayed and how they are displayed. Others provide control over the selection of processes for display, for example, by selecting those processes for a particular user. See the man pages for `ps` for full details, or get a brief summary by using `ps --help`.

top

If you run `ps` several times in a row, to see what is changing, you probably need the `top` command instead. It displays a continuously updated process list, along with useful summary information. See the man pages for `top` for full details on options, including how to sort by memory usage or other criteria. Listing 95 shows the first few lines of a `top` display.

Listing 95. Displaying other processes

```
  3:37pm  up 46 days,  5:11,  2 users,  load average: 0.01, 0.17, 0.19
96 processes: 94 sleeping, 1 running, 0 zombie, 1 stopped
CPU states:  0.1% user,  1.0% system,  0.0% nice,  0.9% idle
Mem:  1030268K av,  933956K used,   96312K free,       0K shrd,  119428K buff
Swap: 1052216K av,    1176K used, 1051040K free                  355156K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
22069 ian       17   0  1104 1104   848 R     0.9  0.1   0:00 top
    1 root       8   0   500  480   444 S     0.0  0.0   0:04 init
    2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
    3 root       9   0     0    0     0 SW    0.0  0.0   0:00 kapmd
    4 root      19  19     0    0     0 SWN   0.0  0.0   0:00 ksoftirqd_CPU0
    5 root       9   0     0    0     0 SW    0.0  0.0   0:00 kswapd
```

Signals

Let's now look at Linux *signals*, which are an asynchronous way to communicating with processes. We have already mentioned the SIGHUP signal and we have used both Ctrl-c and Ctrl-z, which are another way of sending a signal to processes. The general way to send a signal is with the `kill` command.

Sending signals using kill

The `kill` command sends a signal to a specified job or process. Listing 96 shows the use of the SIGTSTP and SIGCONT signals to stop and resume a background job. Use of the SIGTSTP signal is equivalent to using the `fg` command to bring the job to the foreground and then Ctrl-z to suspend it. Using SIGCONT is like using the `bg` command.

Listing 96. Stopping and restarting background jobs

```
[ian@echidna ian]$ kill -s SIGTSTP %1
[ian@echidna ian]$ jobs -l
[1]+ 21709 Stopped                 nohup sh pmc.sh
[2]- 21719 Running                 nohup bash pmc.sh &
[ian@echidna ian]$ kill -s SIGCONT %1
[ian@echidna ian]$ jobs -l
[1]+ 21709 Running                 nohup sh pmc.sh &
[2]- 21719 Running                 nohup bash pmc.sh &
```

We used the job specification (%1) in this example, but you can also send signals to a process id (such as 21709 which is the PID of job %1). If you use the `tail` command while job %1 is stopped, only one process is updating the nohup.out file.

There are a number of other possible signals which you can display on your system using `kill -l`. Some are used to report errors such as illegal operation codes, floating point exceptions, or attempts to access memory that a process does not have access to. Notice that signals have both a number such as 20, and a name, such as SIGTSTP. You may use either the number or the name with the `-s` option. You should always check the signal numbers on your system before assuming which number belongs to which signal.

Signal handlers and process termination

We have seen that Ctrl-c terminates a process. In fact, it sends a SIGINT (or interrupt) signal to the process. If you use kill without any signal name, it sends a SIGTERM signal. For most purposes, these two signals are equivalent.

We said that the `nohup` command makes a process immune to the SIGHUP signal. In general, a process can implement a *signal handler* to *catch* signals. So a process could implement a signal handler to catch either SIGINT or SIGTERM. Since the signal handler knows what signal was sent, it may choose to ignore SIGINT and only terminate when it receives SIGTERM, for example. Listing 97 shows how to send the SIGTERM signal to job %1. Notice that the process status shows as "Terminated" right after we send the signal. This would show as "Interrupt" if we used SIGINT instead. After a few moments, the process cleanup has occurred and the job no longer shows in the job list.

Listing 97. Terminating a process with SIGTERM

```
[ian@echidna ian]$ kill -s SIGTERM %1
[ian@echidna ian]$ jobs -l
[1]  21709 Terminated              nohup sh pmc.sh
[2]- 21719 Running                 nohup bash pmc.sh &
[ian@echidna ian]$ jobs -l
[2]+ 21719 Running                 nohup bash pmc.sh &
```

Signal handlers give a process great flexibility in that a process can do its normal work and be interrupted by a signal for some special purpose. Besides allowing a process to catch termination requests and take possible action such as closing files or checkpointing transactions in progress, signals are often used to tell a daemon process to reread its configuration file and possibly restart operation. You might do this for the inetd process when you change network parameters, or the line printer daemon (lpd) when you add a new printer.

Terminating processes unconditionally

Some signals cannot be caught, such as some hardware exceptions. SIGKILL, the most likely one you will use, cannot be caught by a signal handler and unconditionally terminates a process. In general, you should need this only if all other means of terminating the process have failed.

Logout and nohup

Remember that we said that using `nohup` would allow our processes to keep running after we log out. Well, let's do that and then log back in again. After we log back in, let's check our remaining poor man's clock process using `jobs` and `ps` as we have done above. The output is shown in Listing 98.

Listing 98. Logging back in

```
[ian@echidna ian]$ jobs
[ian@echidna ian]$ ps -a
  PID TTY          TIME CMD
 4234 pts/0    00:00:00 xclock
 5659 pts/0    00:00:00 xclock
27217 pts/4    00:00:00 ps
```

We see that we are running on pts/4 this time, but there is no sign of our jobs, just the `ps` command and the two xclock processes we started from the graphical terminal (pts/0). Not what we were expecting perhaps. However, all is not lost. In Listing 99 we show one way to find our missing job using the `-s` option for session ID, along with the session ID of 20475 that we saw in Listing 93. Think about other ways you might have found them if you didn't happen to have the session ID available.

Listing 99. Logging back in

```
[ian@echidna ian]$ ps -js 20475
  PID  PGID   SID TTY          TIME CMD
21719 21719 20475 ?        00:00:00 bash
27335 21719 20475 ?        00:00:00 sleep
```

Given what we have now learned about killing processes, you should be able to kill these processes using their PID and the `kill` command.

Section 7. Process execution priorities

This section covers material for topic 1.103.6 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 3.

In this section, you learn about the following topics:

- Process execution priorities
- Setting priorities
- Changing priorities

Priorities

As we have seen in the previous section, Linux, like most modern operating systems can run multiple processes. It does this by sharing the CPU and other resources among the processes. If some process can use 100% of the CPU, then other processes may become unresponsive. When we looked at Process status in the previous section, we saw that the `top` command's default output was to list processes in descending order of CPU usage. If we ran the top command with our Poor Man's Clock script, that process probably wouldn't make it onto the list because the process spends most of its time not using the CPU.

Your system may have many commands that are capable of using lots of CPU. Examples include movie editing tools, and programs to convert between different image types or between different sound encoding, such as mp3 to ogg.

We'll create a small script that just uses CPU and does little else. The script takes two inputs, a count and a label. It prints the label and the current date and time, then decrements the count till it reaches 0, then prints the label and the date again. This script has no error checking and is not very robust, but it illustrates our point.

Listing 100. CPU-intensive script

```
[ian@echidna ian]$ echo 'x="$1"'>count1.sh
[ian@echidna ian]$ echo 'echo "$2" $(date)'>>count1.sh
[ian@echidna ian]$ echo 'while [ $x -gt 0 ]; do let x=$x-1;done'>>count1.sh
[ian@echidna ian]$ echo 'echo "$2" $(date)'>>count1.sh
[ian@echidna ian]$ cat count1.sh
x="$1"
echo "$2" $(date)
while [ $x -gt 0 ]; do let x=$x-1;done
echo "$2" $(date)
```

If you run this on your own system, you might see output such as is shown in Listing 101. This script uses lots of CPU, as we'll see in a moment. If you are not using your own workstation, make sure that it is OK to use lots of CPU before you run the script.

Listing 101. Running count1.sh

```
[ian@echidna ian]$ sh count1.sh 10000 A
A Mon Nov 14 07:14:04 EST 2005
A Mon Nov 14 07:14:05 EST 2005
[ian@echidna ian]$ sh count1.sh 99000 A
A Mon Nov 14 07:14:26 EST 2005
A Mon Nov 14 07:14:32 EST 2005
```

So far, so good. Now let's use some of the other things we learned in this tutorial to create a list of commands so we can run the script in background and launch the `top` command to see how much CPU the script is using. The command list is shown in Listing 102 and the output from `top` in Listing 103.

Listing 102. Running count1.sh and top

```
[ian@echidna ian]$ (sh count1.sh 99000 A&);top
```

Listing 103. Using lots of CPU

```
  7:20am  up 48 days, 20:54,  2 users,  load average: 0.05, 0.05, 0.00
91 processes: 88 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  0.1% user,  0.0% system,  0.0% nice,  0.9% idle
Mem:  1030268K av, 1002864K used,   27404K free,       0K shrd,  240336K buff
Swap: 1052216K av,  118500K used,  933716K free                  605152K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 8684 ian       20   0  1044 1044   932 R    98.4  0.1   0:01 sh
```

Not bad. We are using 98.4% of the CPU with a simple script.

Displaying and setting priorities

If we had a long running job such as this, we might find that it interfered with our ability (or the ability of other users) to do other work on our system. Linux and UNIX systems use a priority system with 40 priorities, ranging from -20 (highest priority) to 19 (lowest priority.

nice

Processes started by regular users usually have priority 0. The `nice` command displays our default priority. The `ps` command can also display the priority (nice, or NI, level), for example using the `-l` option. We illustrate this in Listing 104, where we have highlighted the nice value of 0.

Listing 104. Displaying priority information

```
[ian@echidna ian]$ nice
0
[ian@echidna ian]$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   500  7283  7282  0  70   0    -  1103 wait4  pts/2    00:00:00 bash
000 R   500  9578  7283  0  72   0    -   784 -      pts/2    00:00:00 ps
```

The `nice` command can also be used to start a process with a different priority. You use the `-n` or (`--adjustment`) option with a positive value to increase the priority value and a negative value to decrease it. Remember that processes with the lowest priority value run at highest scheduling priority, so think of increasing the priority

value as being *nice* to other processes. Note that you usually need to be the superuser (root) to specify negative priority adjustments. In other words, regular users can usually only make their processes nicer. In Listing 105, we run two copies of the count1.sh script in background with different scheduling priorities. Notice that there is about a 5 second gap between the completion times. Try experimenting with different nice values, or by running the first process with a priority adjustment instead of the second one to demonstrate the different possibilities for yourself.

Listing 105. Using nice to set priorities

```
[ian@echidna ian]$ (sh count1.sh 99000 A&);\
> (nice -n 19 sh count1.sh 99000 B&);\
> sleep 2;ps -l;sleep 20
B Mon Nov 14 08:17:36 EST 2005
A Mon Nov 14 08:17:36 EST 2005
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   500  7283  7282  0  70   0    -  1104 wait4  pts/2    00:00:00 bash
000 R   500 10765     1 84  80   0    -  1033 -      pts/2    00:00:01 sh
000 R   500 10767     1 14  79  19    -  1033 -      pts/2    00:00:00 sh
000 R   500 10771  7283  0  72   0    -   784 -      pts/2    00:00:00 ps
A Mon Nov 14 08:17:43 EST 2005
B Mon Nov 14 08:17:48 EST 2005
```

Note that, as with the `nohup` command, you cannot use a command list or a pipeline as the argument of `nice`.

Changing priorities

renice

If you happen to start a process and realize that it should run at a different priority, there is a way to change it after it has started, using the `renice` command. You specify an absolute priority (and not an adjustment) for the process or processes to be changed as shown in Listing 106.

Listing 106. Using renice to change priorities

```
[ian@echidna ian]$ sh count1.sh 299000 A&
[1] 11322
[ian@echidna ian]$ A Mon Nov 14 08:30:29 EST 2005

[ian@echidna ian]$ renice +1  11322;ps -l
11322: old priority 0, new priority 1
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   500  7283  7282  0  75   0    -  1104 wait4  pts/2    00:00:00 bash
000 R   500 11322  7283 96  77   1    -  1032 -      pts/2    00:00:11 sh
000 R   500 11331  7283  0  76   0    -   786 -      pts/2    00:00:00 ps
[ian@echidna ian]$ renice +3  11322;ps -l
11322: old priority 1, new priority 3
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   500  7283  7282  0  75   0    -  1104 wait4  pts/2    00:00:00 bash
000 R   500 11322  7283 93  76   3    -  1032 -      pts/2    00:00:16 sh
000 R   500 11339  7283  0  76   0    -   785 -      pts/2    00:00:00 ps
```

You can find more information on `nice` and `renice` in the man pages.

Section 8. Searching with regular expressions

This section covers material for topic 1.103.7 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 3.

In this section, you learn about the following topics:

- Regular expressions
- Searching files and filesystems using regular expressions
- Using regular expressions with sed

Regular expressions

Regular expressions have their roots in computer language theory. Most students of computer science learn that the languages that can be denoted by regular expressions are precisely the same as those accepted by finite automata. The regular expressions covered in this section are capable of expressing more complexity and so are **not** the same as those you may have learned about in computer science classes, although the heritage is clear.

A regular expression (also called a "regex" or "regexp") is a way of describing a text string or *pattern* so that a program can *match* the pattern against arbitrary text strings, providing an extremely powerful search capability. The `grep` (for *g*eneralized *r*egular *e*xpression *p*rocessor) is a standard part of any Linux or UNIX programmer's or administrator's toolbox, allowing regular expressions to be used in searching files, or command output. In the section on text streams and filters, we introduced `sed`, the *s*tream *ed*itor, which is another standard tool that uses regular expressions extensively for finding and replacing text in files or text streams. This section helps you better understand the regular expressions used by both `grep` and `sed`. Another program that uses regular expression extensively is `awk`, which is part or the material for exam 201 for LPIC-2 certification.

As with other parts of this tutorial series, whole books have been written on regular expressions and computer language theory. See Resources for some suggestions.

As you learn about regular expressions, you will see similarities between regular expression syntax and the wildcard (or globbing) syntax discussed under Wildcards and globbing. The similarity is only superficial.

Basic building blocks

Two forms of regular expression syntax are used with the GNU grep program found on most Linux systems: *basic* and *extended*. With GNU grep, there is no difference in functionality. Basic syntax is described here, along with the differences between it

and extended syntax.

Regular expressions are built from *characters* and *operators*, augmented by *metacharacters*. Most characters match themselves, and most metacharacters must be escaped using a backslash (\). The fundamental operations are

Concatenation

Concatenating two regular expressions creates a longer expression. For example, the regular expression **a** will match the string **abcdcba** twice (the first and last **a**) and so will the regular expression **b**. However, **ab** will only match **ab**cdcba, while **ba** will only match abcdc**ba**.

Repetition

The Kleene * or repetition operator will match zero or more occurrences of the preceding regular expression. Thus an expression like **a*b** will match any string of **a**'s terminated by a **b**, including just **b** itself. The Kleene * does not have to be escaped, so an expression in which you want to match a literal asterisk (*) must have the asterisk escaped.

Alternation

The alternation operator (|) matches either the preceding or following expression. It must be escaped in basic syntax. For example, the expression **a*\|b*c** will match a string consisting of any number of **a**'s or any number of **b**'s (but not both) terminated by a single **c**. Again, the single character **c** would match.

You often need to quote your regular expressions to avoid shell expansion.

We will use the text files that we created earlier in the lpi103 directory as examples. Study the simple examples of Listing 107. Note that `grep` takes a regular expression as a required parameter and a list of zero or more files to search. If no files are given, grep searches stdin, which makes it a filter that can be used in pipelines. If no lines match, there is no output from `grep`, although its exit code can be tested.

Listing 107. Simple regular expressions

```
[ian@echidna lpi103]$ grep p text1
1 apple
2 pear
[ian@echidna lpi103]$ grep pea text1
2 pear
[ian@echidna lpi103]$ grep "p*" text1
1 apple
2 pear
3 banana
[ian@echidna lpi103]$ grep "pp*" text1
1 apple
2 pear
[ian@echidna lpi103]$ grep "x" text1
[ian@echidna lpi103]$ grep "x*" text1
1 apple
2 pear
3 banana
[ian@echidna lpi103]$ cat text1 | grep "l\|n"
1 apple
3 banana
[ian@echidna lpi103]$ echo -e "find an\n* here" | grep "\*"
* here
```

As you can see from these examples, you may sometimes get surprising results, particularly with repetition. You may have expected **p*** or at least **pp*** to match a couple of **p**'s, but **p***, and **x*** too, match every line in the file because the * operator matches **zero** or more of the preceding regular espression.

First shortcuts

Now that you have the basic building blocks of regular expressions, let's look at a few convenient shortcuts.

+

The + operator is like the * operator, except that it matches **one** or more occurrences of the preceding regular expression. It must be escaped for basic expressions.

?

The ? indicates that the preceding expression is optional, so it represents zero or one occurences of it. This is not the same as the ? used in globbing.

.

The . (dot) is a metacharacter that stands for any character. One of the most commonly used patterns is **.***, which matches an arbitrary length string containing any characters (or no characters at all). Needless to say, you will find this used as part of a longer expression. Compare a single dot with the ? used in globbing and .* with the * used in globbing.

Listing 108. More regular expressions

```
[ian@echidna lpi103]$ grep "pp\+" text1 # at least to p's
1 apple
[ian@echidna lpi103]$ grep "pl\?e" text1
1 apple
2 pear
[ian@echidna lpi103]$ grep "pl\?e" text1 # pe with optional l between
1 apple
2 pear
[ian@echidna lpi103]$ grep "p.*r" text1 # p, some string then r
2 pear
[ian@echidna lpi103]$ grep "a.." text1 # a followed by two other letters
1 apple
3 banana
```

Matching beginning or end of line

The ^ (caret) matches the beginning of a line while the $ (dollar sign) matches the end of line. So **^..b** matches any two characters at the beginning of a line followed by a **b**, while **ar$** matches any line ending in **ar**. The regular expression **^$** matches an empty line.

More complex expressions

So far, we have seen repetition applied to a single character. If you wanted to search for one or more occurrences of a multicharacter string such as the **an** that occurs twice in b**anan**a, use parentheses, which must be escaped in basic syntax. Similarly, you might want to search for a few characters, without using something as general

as the . or as long-winded as alternation. You can do this too, by enclosing the alternatives in square brackets ([]), which do not need to be escaped for regular syntax. Expressions in square brackets constitute a *character class*. With a few exceptions covered later, using square brackets also eliminates the need for escaping special characters such as . and *.

Listing 109. Parentheses and character classes

```
[ian@echidna lpi103]$ grep "\(an\)\+" text1 # find at least 1 an
3 banana
[ian@echidna lpi103]$ grep "an\(an\)\+" text1 # find at least 2 an's
3 banana
[ian@echidna lpi103]$ grep "[3p]" text1 # find p or 3
1 apple
2 pear
3 banana
[ian@echidna lpi103]$ echo -e "find an\n* here\nsomewhere." | grep "[.*]"
* here
somewhere.
```

There are several other interesting possibilities for character classes.

Range expression

A range expression is two characters separated by a - (hyphen), such as 0-9 for digits, or 0-9a-fA-F for hexadecimal digits. Note that ranges are locale-dependent.

Named classes

Several named classes provide a convenient shorthand for commonly used classes. Named classes open with [: and close with :]. Some examples:

[:alnum:]
Alphanumeric characters

[:blank:]
Space and tab characters

[:digit:]
The digits 0 through 9 (equivalent to 0-9)

`[:upper:] and [:lower:]
Upper and lower case letters repectively.

^ (negation)

When used as the first character in a square brackets, the ^ (caret) negates the sense fo the remaining characters so the match occurs only if a character (except the leading ^) is **not in the class.**

Given the special meanings above, if you want to match a literal - (hyphen) within a character class you must put it first or last. If you want to match a literal ^ (caret) it must not be first. And a] (right square bracket) closes the class, unless it is placed first.

Character classes are one area where regular expressions and globbing **are** similar, although the negation differs (^ vs. !). Lisitng 108 shows some examples of character

classes.

Listing 110. More character classes

```
[ian@echidna lpi103]$ # Match on range 3 through 7
[ian@echidna lpi103]$ echo -e "123\n456\n789\n0" | grep "[3-7]"
123
456
789
[ian@echidna lpi103]$ # Find digit followed by no n or r till end of line
[ian@echidna lpi103]$ grep "[[:digit:]][^nr]*$" text1
1 apple
```

Using regular expressions with sed

The brief introduction to Sed mentioned that sed uses regular expressions. Regexps can be used in address expressions as well as in substitution expressions. So the expression `/abc/s/xyz/XYZ/g` means to apply the substitution command that will substitute XYZ for every occurrence of xyz **only** to lines that contain abc. Listing 1111 shows two examples with our text1 file and one where we change the last word before a period (.) to the string LAST WORD. Notice that the string First was not changed as it was not preceded by a blank.

Listing 111. Regular expressions in sed

```
[ian@echidna lpi103]$ sed -e '/\(a.*a\)\|\(p.*p\)/s/a/A/g' text1
1 Apple
2 pear
3 bAnAnA
[ian@echidna lpi103]$ sed -e '/^[^lmnXYZ]*$/s/ear/each/g' text1
1 apple
2 peach
3 banana
[ian@echidna lpi103]$ echo "First. A phrase. This is a sentence." |\
> sed -e 's/ [^ ]*\./ LAST WORD./g'
First. A LAST WORD. This is a LAST WORD.
```

Extended regular expressions

Extended regular expression syntax eliminates the need to escape several characters when used as we have used them in basic syntax, including parentheses, '?', '+', '|', and '{'. This means that they must be escaped if you want them interpreted as characters. You may use the `-E` (or `--extended-regexp` option of grep to indicate that you are usign extended regular expression syntax Alternatively, the `egrep` command does this for you. Some older versions of `sed` do not support extended regular expressions. If your version of `sed` does support extended regexps, use the `-r` option to tell `sed` that you are using extended syntax. Listing 112 shows an example used earlier in this section along with the corresponding extended expression used with `egrep`.

Listing 112. Extended regular expressions

```
[ian@echidna lpi103]$ grep "an\(an\)\+" text1 # find at least 2 an's
3 banana
[ian@echidna lpi103]$ egrep "an(an)+" text1 # find at least 2 an's
3 banana
```

Finding stuff in files

This section wraps up by giving you some taste of the power of `grep` and `find` to hunt down things in your filesystem. Again, the examples are relatively simple; they use the files created in the lpi103 directory and its children.

First, `grep` can search multiple files at once. If you add the `-n` option, it tells you what line numbers matched. If you simply want to know how many lines matched, use the `-c` option, and if you want just a list of files with matches, use the `-l` option. Listing 113 shows some examples.

Listing 113. Grepping multiple files

```
[ian@echidna lpi103]$ grep plum *
text2:9 plum
text6:9 plum
text6:9 plum
yaa:9   plum
[ian@echidna lpi103]$ grep -n banana text[1-4]
text1:3:3 banana
text2:2:3       banana
[ian@echidna lpi103]$ grep -c banana text[1-4]
text1:1
text2:1
text3:0
text4:0
[ian@echidna lpi103]$ grep -l pear *
ex-here.sh
nohup.out
text1
text5
text6
xaa
```

The final example uses `find` to locate all the regular file in the current directory and its children and then `xargs` to pass the file list to `grep` to determine the number of occurrences of **banana** in each file. Finally, this output is filtered through another invocation of `grep`, this time with the `-v` option to find all files that do **not** have zero occurrences of the search string.

Listing 114. Hunting down files with at least one banana

```
[ian@echidna lpi103]$  find . -type f -print0| xargs -0 grep -c banana| grep -v ":0$"
./text1:1
./text2:1
./xab:1
./yaa:1
./text5:1
./text6:4
./backup/text1.bkp.2:1
./backup/text1.bkp.1:1
```

This section has only scratched the surface of what you can do with the Linux command line and regular expressions. Use the man pages to learn more about these invaluable tools.

Section 9. File editing with vi

This section covers material for topic 1.103.8 for the Junior Level Administration (LPIC-1) exam 101. The topic has a weight of 1.

In this section, you learn about the following topic:

- Editing text with vi

Using vi

This section of the tutorial will give you a very brief introduction to a text editor that is almost certainly on every Linux and UNIX system. If a system has no other editor, it probably has vi, so it is a good thing to know you way around in vi. This section will introduce you to some basic vi editing commands, but is not a complete vi tutorial. Try our tutorial "vi intro -- the cheat sheet method" (see Resources) for a tutorial on vi, or consult the man pages or one of the many excellent books that are available.

Starting vi

Most Linux distributions now ship with the vim (for *Vi* *IM*proved) editor rather than classic vi. Vim is upward compatible with vi and has a graphical mode available (gvim) as well as the standard vi text mode interface. You will usually find that the `vi` command is an alias or symbolic link to the vim program. Refer back to the topic "Where does the shell find commands?" if you'd like to know exactly what command is used.

If you recall our section on changing priorities we wanted to change the priority of our running count1.sh shell script. Perhaps you tried this yourself and found that the command ran fast enough that you didn't have a lot of time to accomplish the priority change with `renice`. So let's start by using the vi editor to add a line at the beginning of the file to sleep for 20 seconds so we have some time to change priorities.

To start the vi editor, you use the `vi` with a filename as a parameter. There are many options that you may use, so see the man pages for details. Use the command
`vi count1.sh`
You should see a display something like Listing 115. If you are using vim, then some of the words may be in different colors. Vim has a syntax highlighting mode (which was not part of the original vi editor) and it may be turned on by default on your system.

Listing 115. Editing count1.sh using vi.

```
x="$1"
echo "$2" $(date)
while [ $x -gt 0 ]; do let x=$x-1;done
```

```
echo "$2" $(date)
~
~
~
~
~
~
"count1.sh" 4L, 82C
```

vi modes

The vi editor has two modes of operation:

Command mode

In command mode you move around the file and perform editing operations such as searching for text, deleting text, changing text and so on. You will usually start in command mode.

Insert mode

In insert mode you type new text into the file at the insertion point. To return to command mode, press the **Esc** (escape) key.

It is important to understand these two modes as the way the editor behaves depends on which mode it is in. Vi dates from the time when terminal keyboards may not have had cursor movement keys, so everything you can do in vi can be done with the keys typically found on a standard typewriter plus a couple of keys such as **Esc** or **Insert**. However, vi can be configured to use additional keys if they are available and you will probably find that most of the keys on your keyboard do something useful in vi. Because of this legacy and the slow nature of early terminal connections, vi has a well-deserved reputation for using very brief and quite cryptic commands.

Getting out of vi

One of the first things I like to learn about a new editor is how to get out of it before I do anything I shouldn't have done. It helps to avoid accidental disasters. Here are some ways to get out of vi, either saving or abandoning your changes, or to restart from the beginning. If these commands don't seem to be working for you, you may be in insert mode, so press **Esc** to leave insert mode and return to command mode.

:q!

Quit editing the file and abandon all changes. This is a very common idiom for getting out of trouble.

:w!

Write the file (whether modified or not). Attempt to overwrite existing files or read-only or other unwriteable files. You may give a filename as a parameter and that file will be written instead of the one your started with. It's generally safer to omit the ! unless you know what you're doing here.

ZZ

Write the file if it has been modified. Then exit. This is a very common idiom for normal vi exit.

:e!
Edit the current disk copy of the file. This will reload the file, abandoning changes you have made. You may also use this if the disk copy has changed for some other reason and you want the latest version.

:!
Run a shell command. Type the command and press **Enter**. When the command completes you will see the output and a prompt to return to vi editing.

Notes:

1. When you type the colon (:), your cursor will move to the bottom line of your screen where you type in the command and any parameters.
2. If you omit the exclamation point from the above commands you may receive an error message such as one saying changes have not been saved, or the output file cannot be written (for example, you are editing a read-only file).
3. The : commands have a longer forms (:quit, :write, :edit), but the longer forms are very seldom used.

Moving around

The following commands help you move around in a file.

h
Move left one character on the current line

j
Move down to the next line

k
Move up to the previous line

l
Move right one character on the current line

w
Move to next word on current line

e
Move to next end of word on current line

b
Move to previous beginning of word on current line

Ctrl-f
Scroll forward one page

Ctrl-b
Scroll backward one page

If you type a number before any of these commands, then the command will be executed that many times. This number is called a *repetition count* or simply *count*. For example, 5h will move left five characters. You can use repetition counts with many vi commands.

Moving to lines

The following commands help you move to specific lines in your file

G
Moves to a specific line in your file. For example, 3G moves to line 3. With no parameter, G moves to the last line of the file.

H
Moves relative to the top line on the screen. For example, 3H moves to the line currently 3rd from the top of your screen.

L
Is like H, except that movement is relative to the last line on screen, Thus, 2H moves to the second last line on your screen.

Searching

You can search for text in your file using regular expressions.

/
Use / followed by a regular expression to search forward in your file.

?
Use ? followed by a regular expression to search backward in your file.

n
Use n to repeat the last search in either direction.

You may precede any of the above search commands with a number indicating a repetition count. So 3/x will find the third occurrence of x from the current point, as will /x followed by 2n.

Modifying text

Use the following commands if you need to insert, delete or modify text.

i
Enter insert mode before the character at the current position. Type your text and press **Esc** to return to command mode. Use I to insert at the beginning of the current line

a

Enter insert mode after the character at the current position. Type your text and press **Esc** to return to command mode. Use A to insert at the end of the current line.

c

Use c to change the current character and enter insert mode to type replacement characters.

o

Open a new line for text insertion below the current line. Use O to open a line above the current line.

cw

Delete the remainder of the current word and enter insert mode to replace it. Use a repetition count to replace multiple words. Use c$ to replace to end of line.

dw

As for cw (and c$) above, except that insert mode is not entered.

dd

Delete the current line. Use a repetition count to delete multiple lines.

x

Delete the character at the cursor position. Use a repetition count to delete multiple characters.

p

Put the last deleted text after the current character. Use P to put it before the current character.

xp

Although this is only a combination of x and p, it is such useful idiom that we have put it separately. This swaps the character at the cursor position with the one on its right.

Putting it together.

We set out to add a line to our count1.sh file. If we want to keep the original and saved the modified version as count2.sh here are some vi commands we could use. once we open the file with `vi`. Note that <Esc> means to press the **Esc** key.

Listing 116. Editor commands to add a line to count1.sh

```
1G
O
sleep 20<Esc>
:w! count2.sh
:q
```

Simple when you know how.

The next tutorial in this series covers Topic 104 -- Linux, filesystems, and the

Filesystem Hierarchy Standard (FHS).

Resources

Learn

- Review the entire LPI exam prep tutorial series on developerWorks to learn Linux fundamentals and prepare for system administrator certification.
- At the LPIC Program, find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- In "Basic tasks for new Linux developers" (developerWorks, March 2005), learn how to open a terminal window or shell prompt and much more.
- In the three-part series "Sed by example" (developerWorks, October and November 2000), Daniel Robbins shows you how to use the powerful (but often forgotten) UNIX stream editor, sed. Sed is an ideal tool for batch-editing files or for creating shell scripts to modify existing files in powerful ways.
- sed . . . the stream editor is a useful site maintained by Eric Pement with lots of sed links, a sed FAQ, and a handy set of sed one-liners.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- In the Linux Man Page Howto, learn how man pages work.
- Visit the home of the Filesystem Hierarchy Standard (FHS).
- At the LSB home page, learn about The Linux Standard Base (LSB), a Free Standards Group (FSG) project to develop a standard binary operating environment.
- *Introduction to Automata Theory, Languages, and Computation (2nd Edition)* (Addison-Wesley, 2001) is a good source of information on regular expressions and finite state machines.
- *Mastering Regular Expressions, Second Edition* (O'Reilly Media, Inc., 2002) covers regular expressions as used in grep, sed, and other programming environments.
- *LPI Linux Certification in a Nutshell* (O'Reilly, 2001) and *LPIC I Exam Cram 2: Linux Professional Institute Certification Exams 101 and 102 (Exam Cram 2)* (Que, 2004) are references for those who prefer book format.
- Find more tutorials for Linux developers in the developerWorks Linux zone.

Get products and technologies

- Order the no-charge SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Ian Shields
Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. You can contact Ian at ishields@us.ibm.com.