Previous parts on the DVD

The Mike Saunders SCHOOL of LINUX

Part 6: After mastering the basics of the command line it's time to move on to advanced tricks and techniques.

Mike

Linux Professional Institute

Our expert

Mike Saunders has been writing about Linux for over a decade, and has installed more distros than he's had hot dinners.

As we discovered last issue, the command line isn't a crusty, old-fashioned way to interact with a computer, made obsolete by GUIs, but rather a fantastically flexible and powerful way to perform tasks in seconds that would otherwise take hundreds of mouse clicks.

Additionally, you can't always rely on the X Window System functioning properly – in which case knowledge of the command line is essential – and if you're running Linux as a server OS, you don't want a hulking great GUI sitting on the hard drive anyway.

If you've just started reading the magazine and therefore haven't been following this series, you can find the PDFs in the Magazine section of the coverdisc. You can peruse those at your own pleasure, but to really get the most out of this month's instalment, we recommend reading last issue's tutorial first.

That explains the fundamentals of the command line, including editing commands, using wildcards and manipulating files, and is an important preparation for the advanced topics we're going to handle here.

Section 1: Redirecting output

In the vast majority of cases when you're using the command line, you'll just want the results of your commands to be printed to the screen. However, there's nothing magical about the screen, and in UNIX terms it's equal to any other device. Indeed, because of UNIX's "everything is a file" philosophy, then output from commands can be sent to files rather than to the screen. Consider this command:

```
uname -a > output.txt
```

As we saw last issue, **uname -a** prints information about the operating system you're running. On its own, it displays the results on the screen. With the greater-than **>** character, however, the output is not shown on the screen, but is redirected into the file **output.txt**. You can open the file **output.txt** in your text editor to see this, or display it on the screen using **cat output.txt**.

Now try this:

```
df > output.txt
```

Look at the contents of **output.txt**, and it'll show the results of the disk usage command **df**. An important point here is that the contents are overwritten; there's no trace of the previous **uname -a** command. If you want to append the contents of a command to a file, do it like this:

```
uname -a > output.txt
df >> output.txt
```

In the second line, the double greater-than characters **>>** mean append, rather than overwrite. So you can build up an output file from a series of commands in this way.

This is redirecting. There is, however, another thing you can do with the output of a command, and that's send it directly to another program, a process known as **piping**. For instance, say you want to view the output of a long command such as **ls -la**. With the previous redirect operation, you could do this:

```
ls -la > output.txt
```

Last month We got to grips with the fundamentals of the command line.

What are regular expressions?

At first glance, there's nothing regular about a regular expression. Indeed, when you come across something like this:

```
a\(\(b\)*\2\)*d
```

you might be tempted to run away screaming. Regular expressions are ways of identifying chunks of text, and they're very, very complicated. Whatever you want to do – be it locate all words that begin with three capital letters and end with a number, or pluck out all chunks of text that are surrounded by hyphens – there's a regular expression to do just that.

They usually look like gobbledygook, and vast books have been written about them, so don't worry if you find them painful. Even the mighty beings that produce this magazine don't like to spend much time with them.

Fortunately, for LPIC 1 training you don't need to be a regular expression (**regexp**) guru – just be aware of them. The most you're likely to come across is an expression for replacing text, typically in conjunction with **sed**, the streamed text editor. **sed** operates on input, does edits in place, and then sends the output.

You can use it with the regular expression to replace text like this:

```
cat file.txt | sed s/apple/banana/g > file2.txt
```

Here we send the contents of **file.txt** to **sed**, telling it to use a substitution regular expression, changing all instances of the word **apple** to **banana**. Then we redirect the output to another file. This is by far the most common use of regular expressions for most administrators, and gives you a taste of what it's all about. For more information, enter **man regex**, but don't go mad reading it.

```
less output.txt
```

This sends the list to a file, and then we view it with the **less** tool, scrolling around with the cursor keys and using **q** to quit. But we can simplify this and obviate the need for a separate file using piping:

```
ls -la | less
```

This **|** pipe character doesn't always look well in print; its position varies amongst keyboard layouts, but you'll typically find it broken into two mini lines and accessed by pressing Shift+Backslash. The pipe character tells the shell that we want to send the output of one command to another – in this case, the output of **ls -la** straight to **less**. So instead of reading a file, **less** now reads the output from the program before the pipe.

In certain situations, you might want to use the output of one command as a series of arguments for another. For instance, imagine that you want *Gimp* to open up all JPEG images in the current directory and any subdirectories. The first stage of this operation is to build up a list, which we can do with the **find** command:

```
find . -name "*.jpg"
```

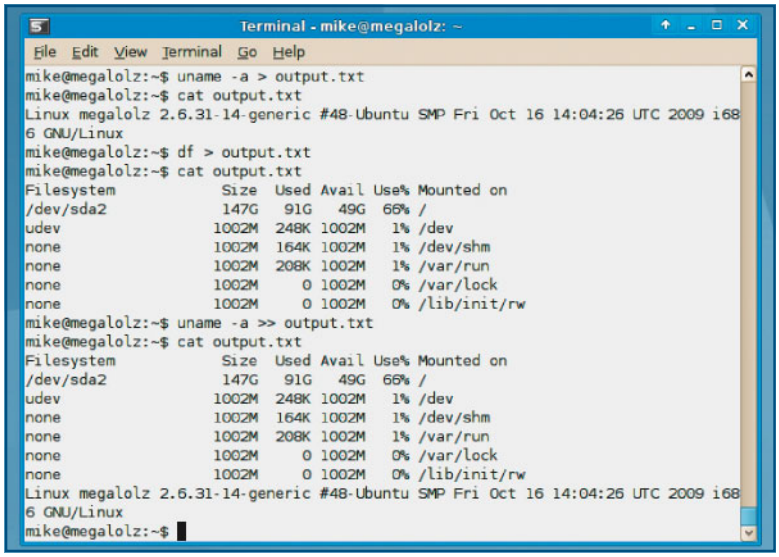
We can't just pipe this information directly to *Gimp*, as it's just raw data when sent through a pipe, whereas *Gimp* expects filenames to be specified as arguments. We do this using **xargs**, a very useful utility that builds up argument lists from sources and passes them onto the program. So the command we need is:

```
find . -name "*.jpg" | xargs gimp
```

Another scenario that occasionally pops up is that you might want to display the output of a command on the screen, but also redirect its output to a file. You can accomplish this with the **tee** utility:

```
free -m | tee output.txt
```

Here, the output of the **free -m** command (which shows memory usage in megabytes) is displayed on the screen, but also sent to the file **output.txt** for later viewing. You can add the **-a** option to the **tee** command to append data to the output file, rather than overwriting it.



› Redirecting output to create new files (or append to existing files) is done with > and >> operators.

Quick tip

Want to store all of your work at the command line in a file? Enter **script**, and you'll start a new shell session inside the current one. Run your commands, type **exit** and you'll see a file called **typescript** has been created with all the output from your work stored.

Section 2: Processing text

UNIX has always been a fantastic operating system for performing operations on text (both in files and being piped around as before), and Linux continues that. Most distributions include a wide range of GNU utilities for manipulating text streams, letting you take a bunch of characters and reorganise them into many different formats. They're often used together with the handy pipe character, and we'll explain the most important tools you need for LPI certification here.

First, let's look at a way to generate a stream of text. If you have a file called **words.txt** containing **Foo bar baz**, then entering:

```
cat words.txt
```

will output it to the screen. **cat** means concatenate, and can be used with redirects or pipe characters as covered earlier. Often you'll only want a certain portion of a command's output, and you can trim it down with the **cut** command, like this:

```
cat words.txt | cut -c 5-7
```

Here, we're sending the contents of **words.txt** to the **cut** command, telling it to cut out characters 5 through to (and including) 7. Note that spaces are characters, so in this case, the result we see is **bar**. This is very specific, however, and you may need to cut out a word that's not guaranteed to be at »

» **If you missed last issue** Call 0870 837 4773 or +44 1858 438795.

» character 5 in the text (and three characters long).

Fortunately, **cut** can use any number of ways to break up text. Look at this command:

```
cat words.txt | cut -d " " -f 2
```

Here, we're telling **cut** to use space characters as the delimiter – ie, the thing it should use to separate fields in the text – and then show the second field of the text. Because our text contains **Foo bar baz**, the result here is **bar**. Try changing the final number to 1 and you'll get **Foo**, or 3 and you'll get **baz**.

So that covers specific locations in an individual line of text, but how about restricting the number of lines of text that a command outputs? We can do this via the **head** and **tail** utilities. For instance, say you want to list the biggest five files in the current directory: you can use **ls -lSh** to show a list view, ordered by size, with those sizes in human-readable formats (ie megabytes and gigabytes rather than just bytes).

However, that will show everything, and in a large directory that can get messy. We can narrow this down with the **head** command:

```
ls -lSh | head -n 6
```

Here, we're telling **head** to just restrict output to the top six lines, one of which is the **total** figure, so we get the five filenames following it. The sworn enemy of this command is **tail**, which does the same job but from the bottom of a text stream:

```
cat /var/log/messages | tail -n 5
```

This shows the final five lines in **/var/log/messages**. **tail** has an especially handy feature, which is the ability to watch a file for updates and show them accordingly. It's called **follow** and is used like this:

```
tail -f /var/log/messages
```

This command won't end until you press Ctrl+C, and will constantly show any updates to the log.

When you're working with large quantities of text, you'll often want to sort it before doing any kind of process on it. Fittingly, then, there's a **sort** command part of every typical Linux installation.

To see it in action, first create a file called **list.txt** with the following contents:

```
ant
bear
dolphin
ant
bear
```

Run **cat list.txt** and you'll get the output, as expected. But run this:

```
cat list.txt | sort
```

And you'll see that the lines are sorted alphabetically, so you have two lines of **ant**, two lines of **bear**, and one of **dolphin**. If you tack the **-r** option onto the end of the **sort** command, the order will be reversed.

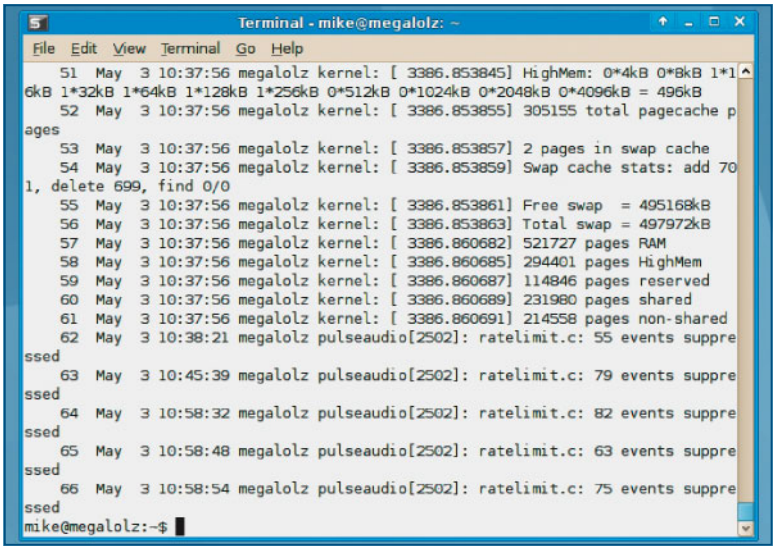
This is all good and well, but there are duplicates here, and if you're not interested in those then it just wastes processing time. Thankfully there's a solution in the form of the **uniq** command, and a bit of double-piping magic. Try this:

```
cat list.txt | sort | uniq
```

Here, **uniq** filters out repeated consecutive lines in a text stream, leaving just the original intact. So when it sees two or more lines containing **ant**, it removes all of them except for the first. **uniq** is tremendously powerful and has a bag of options for modifying the output further: for instance, try **uniq -u** to only show lines that are never repeated, or **uniq -c** to show a line count number next to each line. You'll find **uniq** very useful when you're processing log files and trying to filter out a lot of extraneous output.

Quick tip

If you need help on any of the commands used here, go to the manual page. For instance, to read the manual for the **cut** command, enter **man cut**. Use the cursor keys to scroll, hit forward slash and type text to search, and press Q to quit.



Tally up the number of *PulseAudio* fails in your log files by piping output to the nl command.

Finding text with the mighty grep

If you've been reading **Linux Format** for a while, you might've come across the term **grep** as a generic verb, meaning to search through things. While **find** and **locate** are the standard Linux tools for locating files, **grep** looks inside them, letting you locate certain words or phrases. Here's its most simple use:

```
cat /var/log/messages | grep CPU
```

This prints all lines in the file **/var/log/messages** that contain the word **CPU**. Note that by default this is case-sensitive; if you want to make it insensitive, use the **-i** flag after the **grep** command. Occasionally you might want to perform a search that filters out lines, rather than showing them, in which case you can use the **-v** flag – that omits all lines containing the word.

grep works well with regular expressions (see the previous box). There are a couple of characters we use in **regexps** to identify the start and end of a line. To demonstrate this, create a plain text file containing three lines: **bird**, **badger**, **hamster**. Then run this:

```
cat file.txt | grep -e ^b
```

Here, we tell **grep** to use a regular expression search, and the ^ character refers to the start of the line. So here, we just get the lines that begin with **b** – **bird** and **badger**. If we want to do our searches around the end of lines, we use the **$** character like this:

```
cat file.txt | grep -e r$
```

In this instance, we're searching for lines that end in the **r** character – so the result is **badger** and **hamster**. You can use multiple **grep** operations in sequence, separated by pipes, in order to build up very advanced searches. Occasionally, especially in older materials, you'll see references to **egrep** and **fgrep** commands – they used to be variants of the **grep** tool, but now they're just shortcuts to specify certain options to the **grep** command. See the manual page (**man grep**) for more information.

» **Never miss another issue** Subscribe to the #1 source for Linux on page 66.

Let's move on to reformatting text. Open the previously used file, **list.txt**, and copy and paste its contents several times so that it's about 100 lines long. Save it and then enter this command:

```
cat list.txt | fmt
```

Here, the **fmt** utility formats text into different shapes and styles. By default, it takes our list – separated by newline characters – and writes out the result like a regular block of text, wrapping it to the width of the terminal window. We can control where it wraps the text using the **-w** flag, eg **cat list.txt | fmt -w 30**. Now the lines will be, at most, 30 characters wide.

If you love gathering statistics, then you'll need a way to count lines in an output stream. There are two ways to do this, using **nl** and **wc**. The first is a very immediate method which simply adds line numbers to the start of a stream, for instance:

```
cat /var/log/messages | nl
```

This outputs the textual content of **/var/log/messages**, but with line numbers inserted at the start of each line. If you don't want to see the output, but rather just the number itself, then use the **wc** utility like so:

```
cat /var/log/messages | wc -l
```

(That's dash-lowercase-L at the end.) **wc** actually comes from **word count**, so if you run it without the **-l** flag to show lines, you get more detailed results for words, lines and characters in the text stream.

Formatting fun

One of the tasks you'll do a lot as a trained Linux administrator is comparing the contents of configuration and log files.

If you're an experienced coder then you'll know your way around the **diff** utility, but a simpler tool to show which lines match in two files is **join**. Create a text file called **file1** with the lines bird, cat and dog. Then create **file2** with adder, cat and horse. Then run:

```
join file1 file2
```

You'll see that the word **cat** is output to the screen, as it's the only word that matches in the files. If you want to make the matches case-insensitive, use the **-i** flag.

For splitting up files, there's the appropriately named **split** command, which is useful for both textual content and binary files. For the former, you can specify how many lines you want to split a file into using the **-l** flag, like this:

```
split -l 10 file.txt
```

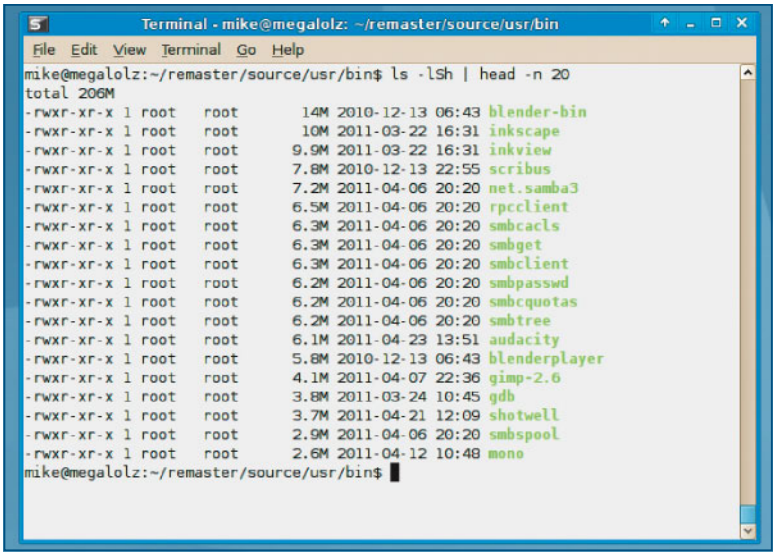
This will take **file.txt** and split it into separate 10-line files, starting with **xaa**, then **xab**, **xac** and so forth – how many files are produced will depend on the size of the original file. You can also do this with non-text files, which is useful if you need to transfer a file across a medium that can't handle its size. For instance, FAT32 USB keys have a 4GB file size limit, so if you have a 6GB file then you'll want to split it into two parts:

```
split -b 4096m largefile
```

This splits it into two parts: the first, **xaa**, is 4GB (4096MB) and the second, **xab**, contains the remainder. Once you've transferred these chunks to the target machine, you can reassemble them by appending the second file onto the first like this:

```
cat xab >> xaa
```

Now **xaa** will contain the original data, and you can rename it.



› Want to limit the output of a command to the first or last few lines? The head and tail commands are your friends.

And some more...

Finally, a mention of a few other utilities that may pop up if you take an LPI exam. If you want to see the raw byte data in a file, you can use the **hd** and **od** tools to generate hexadecimal and octal dumps respectively. Their manual pages list the plethora of flags and settings available.

Then there's **paste**, which takes multiple files and puts their lines side-by-side, separated by tabs, along with **pr** which can format text for printing. Lastly we have **tr**, a utility for modifying or deleting individual characters in a text stream. LXF

Quick tip

Juggling text files that contain tabs can be tricky, but there's a solution in the form of the **expand** command. This changes tabs into blank spaces, making text easier to work with. There's also an **unexpand** command which does the reverse.

Test yourself!

Read this tutorial in full? Tried out the commands at your shell prompt? Think you've fully internalised all the concepts covered here? Then it's time to put your knowledge to the test! Read the following questions, come up with an answer, and then check with the solutions printed upside-down underneath.

1. You have a file called **data.txt**, and you want to append the output of the **uname** command to it. How?
2. How would you display the output of **df** and simultaneously write it to **myfile.txt**?
3. You have **file.txt** containing this line: **bird,badger,hamster**. How would you chop out the second word?
4. You have a 500-line file that you want to split into two 250-line chunks. How?
5. And how do you reassemble the two parts?
6. You have **file1.txt**, and you want to change all instances of the word **Windows** to **MikeOS**. How?
7. And finally, take **myfile.txt**, sort it, remove duplicates, and output it with prefixed line numbers.

1 - uname >> data.txt. 2 - df | tee myfile.txt. 3 - cat file.txt | cut -d "," -f 2. 4 - split -l 250 file.txt. 5 - cat xab >> xaa. 6 - cat file.txt | sed s/Windows/MikeOS/g > output.txt. 7 - cat myfile.txt | sort | uniq | nl

» **Next month** Managing filesystem integrity, and the ultra-terse Vi editor.