Previous parts on the DVD

# The Mike Saunders SCHOOL of LINUX

**Part 5:** After much delving around in hardware, the filesystem and packages, it's time to fully master the command line.

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.

Some naysayers would have you believe that the command line is a crusty old relic of the 1970s, a pointless propellerhead playground which real human beings don't touch. But when it comes to the world of a system administrator, nothing could be further from the truth. The command line, aka shell, is more important than ever – and for good reason:

» It's always there. It exists underneath all the layers of GUI goodness that we see on a typical desktop Linux installation, so even if your window manager is playing up, you can hit Ctrl+Alt+F2 to bring up a prompt and fix it.

» It doesn't require graphics. You can log into a machine remotely (using SSH) from the other side of the planet over a dial-up connection, and you'll be able to work just like it was your local machine. No sluggish VNC or remote desktop required. Similarly, on many machines, such as servers, you won't want all the fluff of a GUI installed. The command line is all you need.

» It's direct. It does exactly what you tell it to do. No "click over on that red button kind-of near the top-left, then find the menu that says Foo and check the box beside it" madness. You just type in exactly what you want the computer to do, and it does it. No messing around.

Consequently, all good administrators have a very solid understanding of the command line, and if you're heading for LPI certification then you'll need to grasp the concepts and tools discovered here. If you're new to Linux, it's also a good way to understand just how powerful and versatile the command line is. We've used a few standalone commands in previous instalments of this series, but now we're going to explore Bash – the default shell in 99.9% of Linux distros – in more depth, so let's get started!

## Section 1: Getting orientated

If you're running a graphical Linux installation, you can bring up a command line prompt via your desktop menus – it's typically called Terminal, Shell, *XTerm* or *Konsole*. In this case we're using CentOS 5.5, where the command line is found under Applications > Accessories > Terminal. When it's launched, we see this:

```
[mike@localhost ~]$
```

That's the prompt, and there are four parts to it: first is the username currently logged in, in this case **mike**. Then there's the hostname of the machine we're using – **localhost**. The tilde (**~**) character shows which directory we're currently working in; it could show **bin** if we were in **/usr/bin** for instance. The user's home directory is typically where a terminal session starts its life, and the tilde is a shorthand way of saying **/home/mike** here, so that's why it appears.

Finally, we have the dollar sign, which is our prompt for input. This indicates that we're running as a regular user; if you enter **su** to switch to the superuser (root) account, and then your password, the dollar sign will change into a hash mark (**#**) instead. Let's try entering a command. Many exist as standalone words. For instance, enter:

```
uname
```

» **Last month** We got to grips with package management – RPMs and Debs.

## Man, I need help!

Want to learn more about the options available to a particular command or program? Most commands have associated documentation in the form of manual ('man') pages. These aren't friendly guides to using the program, but quick references that you can bring up when you need to check for a particular option. You can access these using **man** followed by the name of the command in question; for instance, **man ls**. In the viewer, use the cursor keys to scroll up and down, and press Q to quit out. If you want to search for a particular term, hit the forward slash (**/**) key and then type what you're looking for – for instance, **/size** to search for the word 'size' in the man page.

This outputs the name of the operating system, 'Linux'. However, **uname** has more features up its sleeve, and these can be accessed using flags (also known as parameters or switches).

These flags are usually specified with hyphens and letters or words. For instance, try:

```
uname -a
```

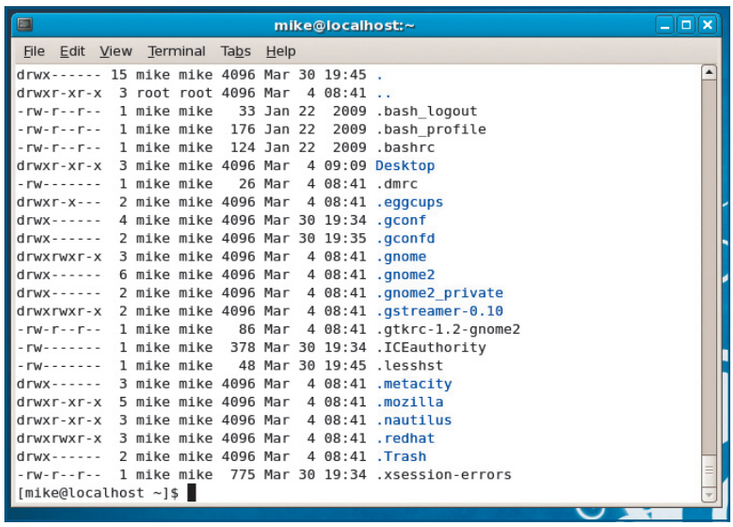
This runs the **uname** program, but passes the **-a** flag to it which means 'show all information' – so you get much more verbose output. For most commands you can see which options are available using the **--help** flag, eg **uname --help**.

So, now you know what you're looking at in the prompt, how to input a command, and how to change its behaviour. That's the essentials covered – let's move on to file management.

First up, enter **ls** – list files. This shows the files and directories in the current directory, and depending on your system, it might use colours to differentiate between items: subdirectories could be blue, for instance.

The **ls** command on its own doesn't show any hidden items – that is, files and directories beginning with full stops. Enter **ls -a** to see everything. (Hidden files are normally used for configuration files that you don't want cluttering up your normal view.) For a detailed list, use **ls -l**. Note that you can combine multiple flags, such as **ls -l -a** or even quicker, **ls -la**. This shows much more information about the items, including the owner, size, modification date and more.

So far we're in our **home** directory, but you'll want to move around in your day-to-day life as an administrator. First of all, let's make a new directory:




**› The ls command for listing files is very flexible, and can display items in a variety of ways, such as the detailed list on show here.**

```
mkdir newdir
```

We can move into this using the **cd** (change directory) command:

```
cd newdir
```

If you enter **ls** in here, you'll see that there's nothing inside. To go back down into the previous directory, enter **cd ..** (cd space dot dot).

If you've used DOS back in the day, you might recognise this – **..** always refers to the directory above the current one. However, unlike DOS, you need the space in the command. And it's also worth noting that, unlike in DOS, all commands and filenames are case-sensitive here.

So you can use **cd** with directories in the current one, but you can also specify complete paths. For instance, you can switch into the **/usr/bin** directory with:

```
cd /usr/bin
```

There's another handy feature of the **cd** command, which is this: enter **cd** on its own and you'll switch back to your home directory. This saves time when you have a particularly long login name, so you no longer have to type something huge like **cd /home/bobthebob1234**.

To display the full path of the directory you're currently in, enter **pwd** (short for 'print working directory'). If you've just changed directory, eg from **/usr/bin** to **/etc**, enter **cd -** (cd space hyphen) to switch back to where you were before. »

## The $PATH to freedom

In true Linux fashion, this box has a dependency: the main text of the article. Please read it first so that you understand environment variables! There's a special variable called **$PATH** which contains a list of locations from which you can run programs. Enter **echo $PATH** and you'll see these directories, separated by colons. For instance, there's **/usr/bin**, **/usr/sbin** and so forth (see **LXF143**'s LPI tutorial for a description of filesystem locations). When you enter a command, like **nano** to run the *Nano* text editor, the shell searches in these locations to find it.

However, it's important to note that the current directory isn't part of the **$PATH**. This is a security measure, to stop trojans (like a malicious **ls** binary) ending up in your home directory, and being executed each time you type **ls**. If you need to run something from the current directory, prefix it with dot-slash, eg **./myprog**. It might seem annoying, but this has proven to be a great aspect of Linux and Unix system security over the years. You might have installed something in **/opt** which needs to be added to your **$PATH** to function correctly. To do this, use the **export** command as described in the main text, but we don't want to wipe out the existing **$PATH** locations so we do it like this:

```
export PATH=$PATH:/opt/newprog
```

Now, when you do **echo $PATH** you'll see the previous locations along with **/opt/newprog** added to the end.

## Section 2: Delving deeper

**Quick tip**

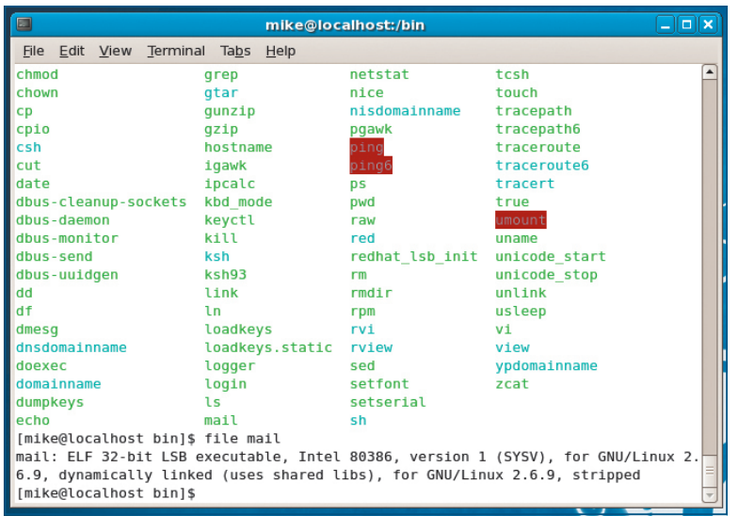
Want to run a command, and close the shell session (eg terminal window) when it has completed? Use the **exec** command: eg **exec nano**. On leaving the *Nano* text editor, the window will close.

Directory and filenames can be long, especially when they're strung together into paths, but *Bash* has a crafty feature: tab completion. Type the first few letters of a file or directory name, hit tab, and *Bash* will try to complete it. For instance, type **cd /usr/lo** and then hit tab – it should expand to **/usr/local**. If you have two or more directories in **/usr** beginning with **lo**, *Bash* will show you which ones are available.

Tab-completion will save you hours of time in your Linux-using life, as will command history. Using the up and down cursor keys, you can navigate through previous commands (these are stored in **.bash_history** in your **home** directory). You can use the left and right cursor keys to move through the command and edit it. If you enter **history**, you'll see a list of the most recent commands entered.

Let's look at some more file manipulation commands. To copy a file, use **cp**:

```
cp file1.txt file2.txt
```




**Not sure what type a particular file is? Find out in an instant with the file command, which pokes into the first few bytes to work it out.**

You can copy multiple files into a directory with **cp file1 file2 file3 dir**. The command to move files works in a similar way, and can be used to rename files: **mv oldname newname**. To remove a file, use **rm filename**. A note of caution though: **rm** doesn't go deep into directories and remove everything inside, including subdirectories. For that you need the recursive switch:

```
rm -r directory
```

This removes the directory, all files inside it and all subdirectories inside it too – a very powerful and destructive command! (An alternative to this is the **rmdir** command.) If you come across a file that you can't identify, eg its filename isn't very descriptive or it doesn't have a sensible extension, you can use the ever-handy **file** command:

```
file /usr/bin/emacs
```

This excellent little tool probes the first few bytes of a file to determine its type (if possible). For instance, if it spots a JPEG header, it'll tell you that it's a JPEG file. The system **file** uses is called 'magic', which is a database of bytes to look out for in files which determine their types. Of course, this isn't always 100% accurate, and you might find a plain text file identified as 'Microsoft FoxPro Database' or something crazy like that, if it just so happens to have a certain sequence of bytes inside.

In some cases you may want to update the timestamp of a file, or create an empty file, and that's where the **touch** command comes in. Similarly, you'll often want to locate files at the command line, and there are two ways of doing this: **locate** and **find**. They sound the same, but there's a fundamental difference: if you do **locate foobar.txt**, it will consult a pre-made database of files on the system and tell you where it is at light speed. This database is typically updated every day by a scheduling program called *Cron*, so it can be out-of-date.

For more to-the-second results, use **find**, for example:

```
find /home/mike -name hamster
```

This will perform a thorough search of **/home/mike** (and all subdirectories) for any items with 'hamster' in the name. But what if you want to search the current directory without

## Creating and expanding archives

Software, patches and other bundles are typically distributed as compressed files, and there are a variety of formats in use. Fortunately, most Linux distributions include the necessary tools to explode and re-compress them, but unfortunately, they don't share the same flags. It's really a historical thing, and a bit annoying at first, but in time you'll remember. Here's a quick reference:

- **.gz** A single compressed file. Expand with **gunzip foo.gz**. To compress a file, use **gzip foo**.
- **.bz2** Like the above, but with stronger (and slower) compression. Expand with **bunzip2**. To compress a file, use **bzip2**. This format used to be heavy going on older machines, but with today's PCs it's the preferred choice for distributing large source code archives such as the Linux kernel.
- **.tar** A tape archive. Few people use tapes today, but it's a system of bundling multiple files together into a single file (without compression). Expand with **tar xfv foo.tar**. Join with **tar cfv foo.tar file1 file2 dir3** (that creates a new archive called **foo.tar** with the files and/or directories inside).
- **.tar.gz** / **.tar.bz2** A combination of the previous formats, and the most common way for distributing source code. Files are gathered together into a single lump with tar, and then compressed with gzip or bzip2. To extract: **tar xfv filename.tar.gz** or **tar xfv filename.tar.bz2**. To compress: **tar cfvz foo.tar.gz file1 file2** (for **.tar.gz**) or **tar cfvj foo.tar.bz2 file1 file2** (for **.tar.bz2**).
- **.cpio** A relatively rare format that bundles files together into a single file (without compression). Extract with **cpio -id <filename**. Join with **ls file1 file2 | cpio -ov > foo.cpio** (that character between **file2** and **cpio** is a pipe – more on that next month). You'll come across CPIO files if you work with initrd images.

Another useful utility is **dd**, which copies data from one source to another. It's particularly useful for extracting disc images from physical media. For instance, if you pop in a CD or DVD and enter **dd if=/dev/cdrom of=myfile.iso**, you end up with an ISO image (which you can then redistribute or burn to another disc).

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

having to type its full path? Well, remember before we said that **..** is the directory above the current one? Well, **.** is the current directory. So you could rewrite the previous command, providing you're already in **/home/mike**, as:

```
find . -name hamster
```

The **find** command can also be used for sizes: **find . -size +100k** locates all files bigger than 100 kilobytes in the current directory (use **M** for megabytes and **G** for gigabytes). Another alternative is to find by type: **find . -type f** will only show files, whereas **-type d** shows only directories. You can mix -name, -size and -type options to create very specific searches.

*Bash* includes comprehensive wildcard functionality for matching multiple filenames without having to specify them. The asterisk character (*), for example, means 'any combination of letters, numbers or other characters'. So consider this command:

```
ls *.jpg
```

This lists all files that end in **.jpg**, whether they're **bunnyrabbit.jpg**, **4357634.jpg** or whatever. This is useful for moving and deleting files: if you have a directory full of images, and you want to get rid of those silly ones ending in **.bmp**, you can do **rm *.bmp**. If you want a wildcard for just a single letter, use a question mark:

```
mv picture?.jpg mypics
```

This command will move **picture1.jpg**, **pictureA.jpg** and so forth into the **mypics** directory.

**Quick tip**

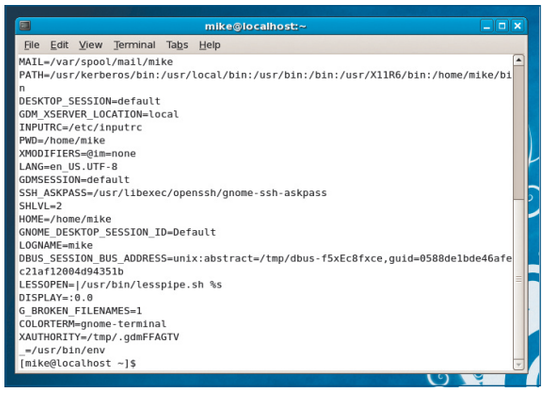
If you've entered a command that looks like it's going to take hours to complete, and want to stop it, hit Ctrl+C. Beware that it interrupts the command immediately, so chances are it won't clean up any files it was working on!

## Section 3: Understanding the environment

While typical use of the command line involves typing in commands one-by-one, these commands are subject to the environment in which they operate. There are environment variables which store bits of information such as options and settings, and programs can take the information from these to determine how they operate. Environment variables are usually in capital letters and begin with a dollar sign. For instance, try this:

```
echo $BASH_VERSION
```

Here, **echo** is a command which simply outputs text to the screen, in this case the contents of the **$BASH_VERSION** environment variable. You'll see a number such as 3.2.25. Programs can probe this variable for their own purposes, such as to determine whether or not a user is running version 3.0 or better and therefore with certain features available. To see a full list of the environment variables in use, along with their contents, enter **env**.




**› Environment variables alter the way that programs are run – get a full list with the env command.**

You can set up your own environment variables in this way:

```
export FOO="bar"
echo $FOO
```

(Note that there's no dollar sign in the first command.) This new **$FOO** variable will only last as long as the terminal session is open; when you end it by closing the window, typing **exit** or hitting Ctrl+D, it will be lost. To fix that, edit the text file **.bashrc** in your **home** directory, which contains variable definitions and other settings that are read when a command line session starts. Save your changes, restart the terminal and they will take effect.

*Bash* has other variables alongside those for the environment in which programs run; it has its own variables too. Enter the **set** command and you'll see a full list of them. If there's a variable, either for *Bash* or the environment, that you want to remove, you can do it as follows:

```
unset FOO
```

These features, combined with the tab completion, wildcard expansion and history facilities, make the Linux command line extremely efficient to work in and miles apart from the clunky old DOS prompts of yore. As you get more and more familiar with the command line, you'll be tempted to leave the file manager behind.

Above all, you feel totally in control. Typos aside, there's no way you can accidentally select the wrong option when working at the command line: you are stating exactly what you want to achieve. This is just half of the story though – next we'll look at tricks to send the output of one command to another, or to a file for later viewing. Don't miss it! LXF

**Quick tip**

Accidentally messed up the display in your terminal? You can enter **clear** (or press Ctrl+L) to clear the screen. If that doesn't work, and strange characters are appearing due to the terminal spewing out random binary data, try **reset**.

## Test yourself!

Think you've internalised the topics and commands we've described here? Want to see if you're ready to use this information in an LPI setting or the real world? See if you can answer these questions, and rotate the mag to see the answers underneath:

1. What does the tilde (~) sign in a command prompt mean?
2. How would you list all files in the current directory, in detailed mode?
3. Which command to find files uses a pre-made database?
4. How would you set the environment variable **$WM** to **icewm**?
5. How would you add **/opt/kde/bin** to your **$PATH**?
6. How would you make a **.tar.bz2** archive of the directory **myfiles**?
7. You want to run a version of *Nano* from your current directory, not in your **$PATH**. How?

**1 - Home directory. 2 - ls -la. 3 - locate. 4 - export WM="icewm". 5 - export PATH=$PATH:/opt/kde/bin. 6 - tar cfj archive.tar.bz2 myfiles. 7 - ./nano.**

**» Next month** Advanced command line techniques with pipes and redirects.