LINUX FORMAT On the DVD » Example CLI files.

Immense power is at your fingertips – you just need to know how to use it. **Jonathan Roberts** gets to grips with the CLI...

$ conquer the\ > command line

The command line is an incredibly powerful way to interact with your computer. As well as giving you access to many low-level administrative tools, it's also an efficient and flexible way to go about your day-to-day business.

Unfortunately, it's more than a little intimidating to the uninitiated. That blinking cursor, and the cryptic text that sits before it, hints at a world of possibilities but gives no indication of how you might begin to use it. For all you know, pressing the wrong key might put the computer into a super-secret overdrive mode, melt your processor and destroy all your data.

But you don't have anything to worry about. Sure, the command line is powerful, and with great power comes great responsibility, but with just the tiniest bit of knowledge, you can safely harness this power to work more efficiently and get a whole lot more out of your machine.

In this article, we're going to teach you everything you need to know. We'll start by explaining what the command line is, how to issue commands and how to interpret the results. We'll move on to look at two mini projects. The first will introduce you to the commands that you'll need in day-to-day computing, as well as show you some of the tricks that will make you more productive; the

"That blinking cursor, and the cryptic text before it, hints at a world of possibilities..."

second will look at the tools and tricks necessary to use the command line when diagnosing and fixing problems.

Keep an eye out for the example boxes, as putting theory into practice is by far the best way to get familiar with this material. Also, as an added bonus, each of the **LXF** staff has revealed their top command-line trick. Lots to see, lots to learn and lots of fun to be had, so what are you waiting for?

First things first

Just to make sure we're all on the same page, let's start at the very beginning (as Julie Andrews once said), before diving in to some actual work. The command line is just another interface to your computer, like Gnome or KDE. What makes it different is that, instead of clicking nicely labelled buttons with your mouse, you control your computer by typing commands on your keyboard.

There's nothing magical about these commands, they're just combinations of letters that the computer interprets according to a well-defined set of rules. Each command starts with its name, so the computer knows which one you're invoking, and can be followed by options and arguments that modify the way it works.

Many of these options and arguments specify which folder or file you want the command to operate on, and in the absence of a graphical file manager, there's a special syntax for referencing these on the command line.

On a Linux system, files and folders are organised in a hierarchy, descending from the root folder. On the command line, this root folder is represented by a single forward-slash, **/**. All the folders and files that come below this are then represented by their names, which are case-sensitive, with the different folders being separated by another forward-slash.

For example, my home folder, which is a sub-folder of the system home folder, which is a sub-folder of root, looks like this: **/home/jon/**. This is a representation we'll be using a lot, and it will soon become second-nature.

Your first command: ls

Now that you know the absolute basics, it's time to issue your first command. The command we're going to begin with is called **ls**, and all it does is list the contents of a directory. A good way to remember this program and what it does is that **ls** looks like 'list'. The first thing to do is launch a terminal (see the boxout below), after which you'll see a new window appear on your desktop. As well as having a plain white or black background, there will also be some obscure-looking text in the window. This text, with the flashing cursor that follows it, is called the prompt, and indicates that your computer is ready to accept commands.

With your computer at the ready, type **ls**, the name of the command, into the terminal window and press Return. You'll immediately see several lines of text appear in the terminal, followed by a new prompt indicating that your computer is again ready to accept commands.

If you look closely at the text that appears in the terminal, you'll notice that it looks familiar... it's a list of all the files in your **home** folder.

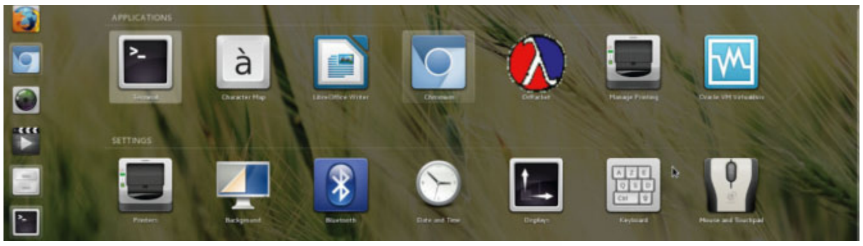
Your first argument

Pretty cool, and pretty simple, right? But why did it list the contents of your home folder and not your Music folder or some other altogether?

Well, while working at the command line, you're always working within one directory or another – no command is ever run outside of this kind of context. Obviously, every command-line session has to start somewhere, and by default it's the current user's home directory.

That's all well and good, but it still doesn't really explain why **ls** returned the contents of your home directory. What happened is that, since we didn't tell **ls**

Terminology and the terminal



› To access the command line, sometimes known as the shell, launch the *Terminal* application in Gnome or *Konsole* in KDE.

If you've ever read about the command line on the internet, you'll have seen there's a lot of terminology floating around, all of which seems to refer to very similar concepts.

Two common terms, the shell and the command line, are quite interchangeable and refer to the text-based interface; one other common term, the terminal, has a slightly different meaning.

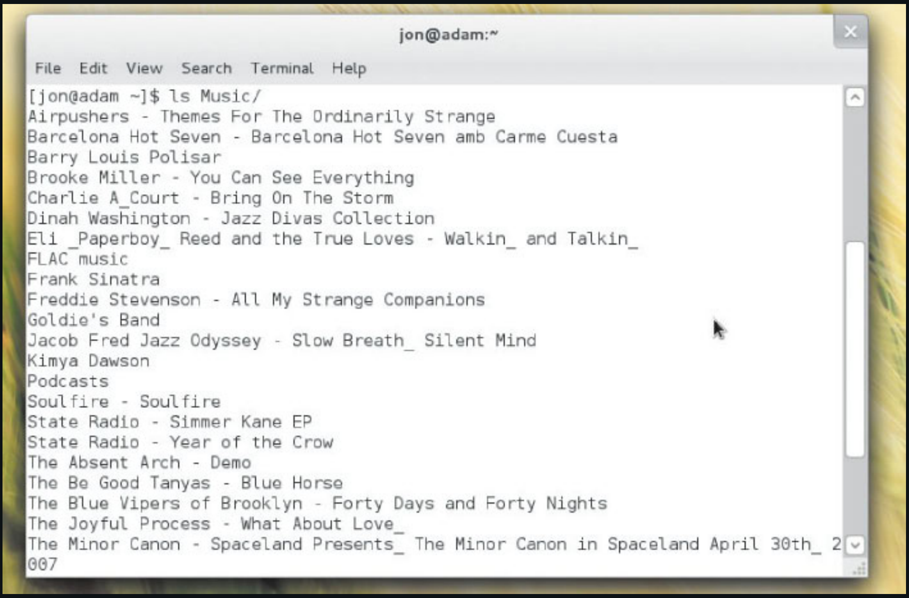
The terminal is a program that you access the command line through. It's quite simple really – it receives the input you type on the keyboard and displays it on the screen; it also receives the output of the various commands that you might run, and displays this on your screen.

In this article, when we say 'launch a terminal', we mean launch this program to get access to the command line. On Gnome or *Xfce*, you'll most likely be looking for a program named *Terminal*, while on KDE the program you'll want to launch will be called *Konsole*.

otherwise, it simply assumed that we wanted to see the contents of the current directory, which at the time was your home folder.

We can, however, tell **ls** that we want to look elsewhere by 'passing an argument' to it. All this really means is that, as well as typing the name of the command, you'll also type the location of the folder that you want to look in. So, if you wanted to inspect the contents of your music folder, the command would be **ls Music**.

Notice that we just put Music, rather than the full path to the folder, which would have been **/home/jon/Music**. This is known as a 'relative path': since we're already in our home folder, if you don't specify the full path, the command line will simply look for a folder with the same name in the current directory.



```
[jon@adam ~]$ ls Music/
Airpushers - Themes For The Ordinarily Strange
Barcelona Hot Seven - Barcelona Hot Seven amb Carme Cuesta
Barry Louis Polisar
Brooke Miller - You Can See Everything
Charlie A_Court - Bring On The Storm
Dinah Washington - Jazz Divas Collection
Eli _Paperboy_ Reed and the True Loves - Walkin_ and Talkin_
FLAC music
Frank Sinatra
Freddie Stevenson - All My Strange Companions
Goldie's Band
Jacob Fred Jazz Odyssey - Slow Breath_ Silent Mind
Kimya Dawson
Podcasts
Soulfire - Soulfire
State Radio - Simmer Kane EP
State Radio - Year of the Crow
The Absent Arch - Demo
The Be Good Tanyas - Blue Horse
The Blue Vipers of Brooklyn - Forty Days and Forty Nights
The Joyful Process - What About Love_
The Minor Canon - Spaceland Presents_ The Minor Canon in Spaceland April 30th_ 2
007
```

›The ls command with my Music folder passed to it as an argument.

MINI PROJECT 1 Manage files and folders

You now know how to use one command, and how to modify the way it works with a single argument. For your new-found command line knowledge and skills to be useful, however, you're going to need to know a few more commands and begin to get a sense of how you might use them together.

In this section we're going to walk you through a simple mini project that will introduce you to the programs that will help you manage files and folders, move about the filesystem and edit text files.

So that we're all working from the same page here, we've included a mock home folder on this month's coverdisc. Copy it to your own home folder and then extract it using whatever tool suits you best, being careful to take note of the folder name (**jons-home**).

If you take a look inside, you'll quickly get a sense for this mini project's premise. My **LXF** files have gotten scattered all over my home folder, and you've got to help me track them down and finish the work. It's more than a little contrived, but it should get the job done.

Our first job on the command line will be to get inside the mock home folder. Launch a new terminal window, and run the **ls** command to check what you have in your home folder. If you extracted the mock folder here, amongst this list you should also see **jons-home**. To get inside it, you're going to need to use the **cd** command.

cd stands for, at least in our minds, change directory, and that's exactly what it does. If you run **cd** without any arguments, it will return you to your own home folder – not that helpful. If you run it with a single argument, specifying which folder you want to change to, it will send you there instead – much more helpful. The command to get inside **jons-home**, then, is **cd jons-home**.

As soon as you issue that command... nothing seems to happen. Well, nothing quite so striking as when you run the **ls** command. If you look closely, however, you'll see that the text inside the prompt has changed. The ~ has disappeared, and in its place is **jons-home**, the name of the current folder.

This part of the prompt will always display the name of the current folder, saving you from getting lost. That ~, which looked strange before, is actually just an abbreviation for the current user's home directory. You can test this if you want, by typing **cd ~** and then running **ls** to see if

Examples 1

In this section, we've mostly been looking at the **ls** command to demonstrate the basics of the command line.

Of course, **ls** is capable of far more than just listing the contents of a directory; here's a few examples of ways you can manipulate its output, although you might want to come back to this after reading about 'options' later in the article:

```
[jon@adam ~]$ ls -a
```

Lists all files in a directory, including those that are 'hidden' by placing a dot at the front of their name.

```
[jon@adam ~]$ ls --color
```

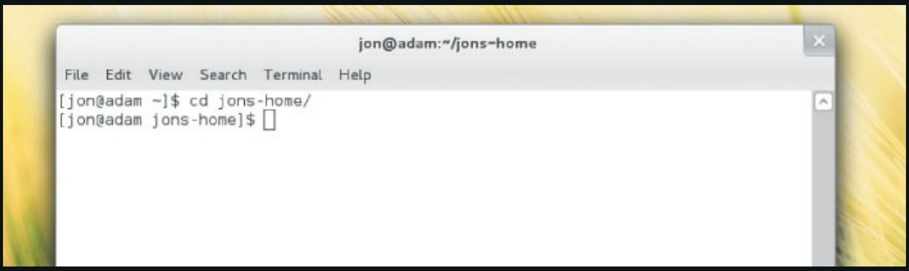
Colourises the output to make it easier to read.

```
[jon@adam ~]$ ls --sort=X
```

Sorts the output by something other than filename. **X** sorts by extension, **S** size, **t** time and **v** version.

```
[jon@adam ~]$ ls -l <filename>
```

Lists details, including permissions, owner and last modification time, about **<filename>**.



```
[jon@adam ~]$ cd jons-home/
[jon@adam jons-home]$ 
```

❯ A close-up of the prompt. Notice how the prompt changes after the **cd** command, so that it always shows the current directory.

you're back in your home folder. Hooray! Now, take a look inside **jons-home** with **ls**. You'll see that there are several files for each of the articles I've written in this issue. How silly, and how messy. Wouldn't it be much neater if there were a folder for each article? I think so, and you're going to do it for me.

The first step will be to create a folder for each article using the **mkdir** command. You'll probably find this easy to remember because it looks a lot like 'make directory'.

For **mkdir** to work, you must pass it one argument which specifies the name of the directory you want to create. So, to create a directory for this command-line feature's files, you'd run **mkdir command-feature**.

Brace yourself

You could just run this command three times, replacing command-feature with first-steps and WoE to create folders for all the articles, but this is slow and the command line provides a much faster way of doing this kind of operation. It's called brace expansion.

The name sounds a bit silly, but it makes perfect sense. Take a look at this command:

```
[jon@adam jons-home]$ mkdir {command-feature,WoE,first-steps}
```

By wrapping the braces around the three folder names, we tell the command line to run the **mkdir** command once for each of the arguments contained within. Instead of having to write **mkdir** three times, we only write it once and the command line does all the hard work for us – how efficient.

This little trick doesn't just work for the **mkdir** command, but any. For instance, now that we have all the folders, we can move the files into them (using **mv**) with a single command for each group of files:

```
[jon@adam jons-home]$ mv {command-notes,command-feature-1,command-feature-2} command-feature/
```

Notice how the **mv** command takes two arguments, whereas before we've only ever used one. The first argument specifies the file that's being moved, and the second the destination. With commands that take multiple arguments, it's important to get the order right, as there's no other indication of each argument's purpose.

Editing text files

Now that we've got the files organised, I also need some help finishing the articles. Fortunately, I work in plain text files, so we don't need to use a big, bulky program such as *LibreOffice*; instead, we can use one of the many command-line text editors.

The text editors are different to the commands we've seen so far, in that they're interactive rather than set-and-run. This is tricky to imagine, but if you follow our step-by-step, it will quickly make sense. We'll be using the *nano* text editor, since it's by far and away the most intuitive available. There are many other choices available, and we'd encourage you to investigate them at your leisure, since *nano* won't always be available (whereas *Vi* will almost certainly be on every Linux system).

Keep it tidy

Now that we've organised all the files and finished the articles, there are just two more things we should do before finishing. The first is to create a copy of the files we've edited, in case we accidentally delete anything, and the second is to tidy up and remove old, unwanted sets of notes.

Both are easily achieved and make use of vital everyday commands. The first, making a copy of a file, uses the **cp** command, which works in the same way as move – specify the file to be copied as the first argument, and the name and location of the copy as the second argument (this

"Instead of writing mkdir three times, we only write it once and the command line does all the hard work for us."

White space and special characters

You might wonder why our mock folder was called **jons-home**; surely a much more natural name would be Jon's Home?

The problem with this second name is that it contains a space and a single quote (although acting as an apostrophe), both of which have a special meaning on the command line.

The space, for instance, is used to separate the command name from its arguments, and arguments from one another. If you were to use a space in the folder name, how would **cd** or **ls** know to interpret the entire name as one, rather than as separate arguments?

To avoid this kind of confusion, it's best to restrict your file and folder names to letters, numbers, and the hyphen, underscore and full-stop symbols. If you come across a command that doesn't seem to work, and the folder or file you're trying to operate on has a strange-looking name, this might be the problem.

You can get around this by adding the escape character (a ****) before the special character, but this can get messy.

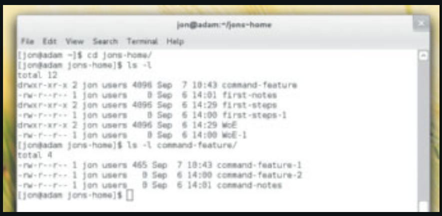
For reference, to make a directory called Jon's Home, you'd have to run the command **mkdir Jon\'s\ Home**.

means that the copy doesn't have to be stored in the same directory as the original file).

The second, using the **rm** command, is a simple job of passing the filename to be removed as an argument. Be careful with this command, however, as there's no recycle bin for it – once a file's deleted, it's gone forever.

Neither of these methods will work if you want to apply **cp** or **rm** to an entire directory, however – they'll only work on individual files. To show you how to apply them to a directory, we need to introduce you to the idea of options.

An option is much like an argument – it modifies the way a command runs – only it's a lot more specific and allows for far more possibilities.



```
[jon@adam ~]$ cd jons-home/
[jon@adam jons-home]$ ls -l
total 12
drwxr-xr-x 2 jon users 4096 Sep  7 10:43 command-feature
-rw-r--r-- 1 jon users    0 Sep  6 14:01 first-notes
drwxr-xr-x 2 jon users 4096 Sep  6 14:29 first-steps
-rw-r--r-- 1 jon users    0 Sep  6 14:00 first-steps-1
drwxr-xr-x 2 jon users 4096 Sep  6 14:29 WoE
-rw-r--r-- 1 jon users    0 Sep  6 14:00 WoE-1
[jon@adam jons-home]$ ls -l command-feature/
total 4
-rw-r--r-- 1 jon users 465 Sep  7 10:43 command-feature-1
-rw-r--r-- 1 jon users   0 Sep  6 14:00 command-feature-2
-rw-r--r-- 1 jon users   0 Sep  6 14:01 command-notes
[jon@adam jons-home]$
```

› Notice how the -l option modified the way ls works.

Know your options

Each option is associated with a single letter or a string of text, and is specified on the command line by stating this letter or text, with a single dash or two dashes, respectively, preceding it. To see what we mean, take a look at this example:

```
[jon@adam ~]$ ls -l jons-home
```

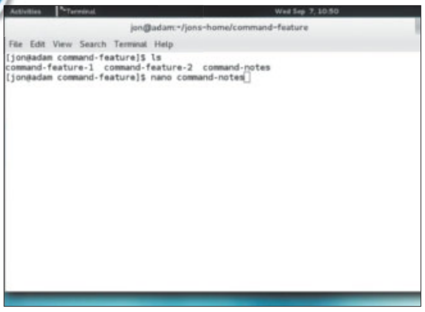
The **-l** option of the **ls** command stands for 'long', and instructs it to provide further details about the contents of the folder being inspected. What's really cool is that options can be combined with arguments, so even while we told **ls** to provide extra detail, we were still able to specify which folder we wanted to inspect.

Applying this to the **cp** and **rm** commands, there's an **-r** option to both that instructs it to work 'recursively'. This means they will copy or delete everything contained within a folder, including sub-folders, and the folder itself.

Jon's top tip

When performing actions on multiple files, all of which share some common characteristic in their filename, you can use a wildcard to act on all of them. For instance, **cp *.txt** will copy every text file in the current directory.

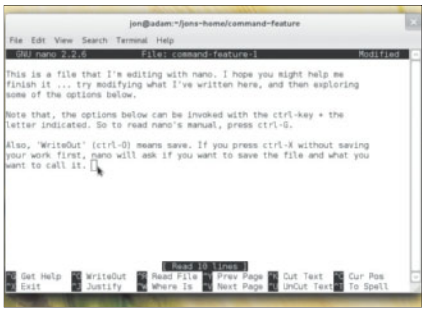
Step-by-step: Nano text editor



```
[jon@adam command-feature]$ ls
command-feature-1  command-feature-2  command-notes
[jon@adam command-feature]$ nano command-notes
```

1 Open a file ›

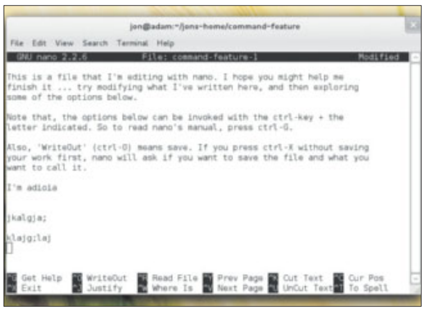
To open a text file in *nano*, change to the directory where the file is stored, type **nano** and pass the filename as an argument, eg **nano command-notes.txt**



```
  GNU nano 2.2.6          File: command-feature-1                    Modified

This is a file that I'm editing with nano. I hope you might help me
finish it ... try modifying what I've written here, and then exploring
some of the options below.

Note that, the options below can be invoked with the ctrl-key + the
letter indicated. So to read nano's manual, press ctrl-G.

Also, 'WriteOut' (ctrl-O) means save. If you press ctrl-X without saving
your work first, nano will ask if you want to save the file and what you
want to call it.

                         [ Read 10 lines ]
   Get Help    WriteOut    Read File   Prev Page   Cut Text    Cur Pos
   Exit        Justify     Where Is    Next Page   UnCut Text  To Spell
```

2 Start typing ›

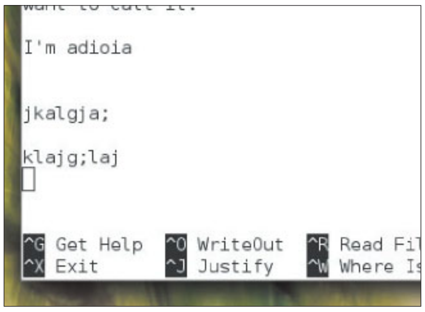
The entire terminal screen will be taken over by the *nano* interface. You can start typing immediately and see your text appear in the main part of the window.



```
  GNU nano 2.2.6          File: command-feature-1                    Modified

This is a file that I'm editing with nano. I hope you might help me
finish it ... try modifying what I've written here, and then exploring
some of the options below.

Note that, the options below can be invoked with the ctrl-key + the
letter indicated. So to read nano's manual, press ctrl-G.

Also, 'WriteOut' (ctrl-O) means save. If you press ctrl-X without saving
your work first, nano will ask if you want to save the file and what you
want to call it.

I'm adioia


jkalgja;

klajg;laj

   Get Help    WriteOut    Read File   Prev Page   Cut Text    Cur Pos
   Exit        Justify     Where Is    Next Page   UnCut Text  To Spell
```

3 Write as normal ›

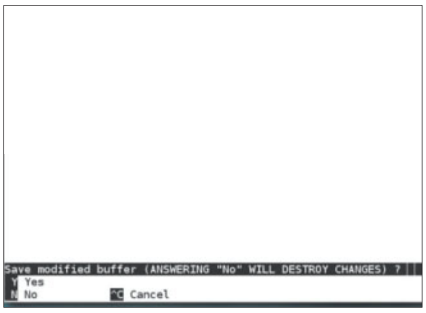
Everything will work as expected, including the Return key for a new line, and the Arrow keys to scroll when the file gets large enough.



```
want to call it.

I'm adioia


jkalgja;

klajg;laj


^G Get Help  ^O WriteOut  ^R Read Fi
^X Exit      ^J Justify   ^W Where Is
```

4 Keyboard shortcuts ›

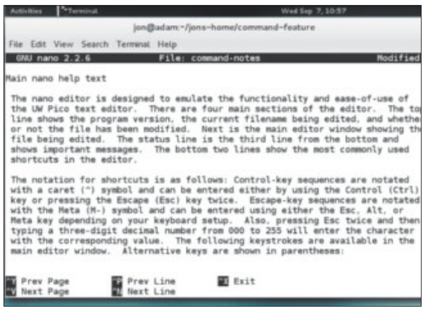
At the bottom are commonly used keyboard shortcuts. The ^ indicates that you need to press Ctrl at the same time as the letter. Note that WriteOut means save.



```
Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?
 Y Yes
 N No                ^C Cancel
```

5 Status messages ›

Some shortcuts will result in a status message and a question being asked of you. Read the message, and respond according to the options presented below.



```
  GNU nano 2.2.6              File: command-notes                    Modified

Main nano help text

 The nano editor is designed to emulate the functionality and ease-of-use of
 the UW Pico text editor.  There are four main sections of the editor.  The to
 line shows the program version, the current filename being edited, and whethe
 or not the file has been modified.  Next is the main editor window showing th
 file being edited.  The status line is the third line from the bottom and
 shows important messages.  The bottom two lines show the most commonly used
 shortcuts in the editor.

 The notation for shortcuts is as follows: Control-key sequences are notated
 with a caret (^) symbol and can be entered either by using the Control (Ctrl)
 key or pressing the Escape (Esc) key twice.  Escape-key sequences are notated
 with the Meta (M-) symbol and can be entered using either the Esc, Alt, or
 Meta key depending on your keyboard setup.  Also, pressing Esc twice and then
 typing a three-digit decimal number from 000 to 255 will enter the character
 with the corresponding value.  The following keystrokes are available in the
 main editor window.  Alternative keys are shown in parentheses:

   Prev Page          Prev Line          Exit
   Next Page          Next Line
```

6 Help screen

Be sure to investigate the Ctrl+G option, which will provide a help screen with all the options available. Happy text-editing!

MINI PROJECT 2 Diagnose and fix problems

Well, at this point my work is all done and we can safely move on to our next, less contrived, mini project. The plan here is to look at the command-line programs and tricks that are useful for diagnosing and fixing problems on your machine – one of the command line's greatest strengths.

We'll begin by introducing you to permissions and the **su -** command, two concepts that are vital if you're going to be able to collect information about the system and modify the way it works.

We'll then look at the commands needed to collect information about hardware and show you the important log files that store information about how the system is running.

We won't show you configuration files to edit or how to edit them, since there are far too many possibilities, and the task of editing them boils down to running a text editor in combination with the **su -** command – all things we'll cover anyway.

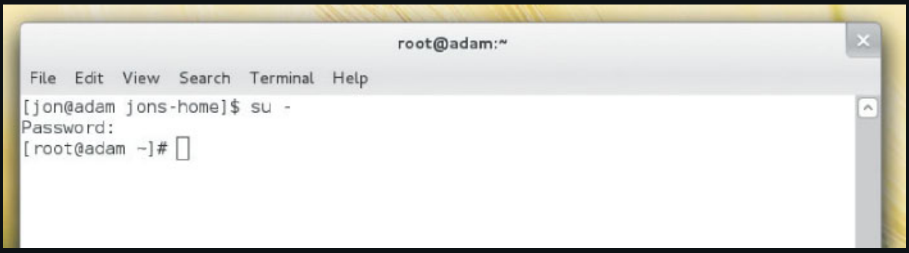
Permissions, su and sudo

The first thing you need to know regarding permissions is that Linux is designed with multiple users in mind.

Every file on the system is then 'owned' by particular users, who can specify which other users are allowed to read or modify their own files – that is, they can set the file's permissions.

This is most apparent in day-to-day computing if you try to look inside the home directory of another user. The system just won't let you do it. Obviously this is a great feature since it brings a lot of added privacy and security to the system.

Where it's even more important, however, is in the separation of user and system files. Most of the time, when you're logged in as jon or fred, you have limited access to the system and can, for the most part, only modify files in your home directory.



```
[jon@adam jons-home]$ su -
Password:
[root@adam ~]# 
```

❯ **Notice how jon has been replaced by root, to indicate that after su - all our commands will now be run by the root user, rather than our usual one.**

There are many other files on your system, however, that are important to the way your computer works, and these are not accessible to your normal user. Instead, they are 'owned' by the root user. So, to modify the way your computer runs, or investigate many problems, you're going to need to become the root user, since that's the only one with permission to do so.

On the command line, the way to do this is with the **su** command (which stands for switch user). If you type **su -** you'll immediately be prompted for the root user's password. Entering this, a new prompt will appear and, if you look closely, you should notice some differences.

The part of the prompt that looks like an email address no longer has your username in it, but instead says **root@**. This part of the prompt will always display the name of the user you're currently working as. As long as you know the user's password, you can use **su** to switch to any user on the system by replacing the **-** with their username.

On Ubuntu and similar systems, it's more common to use a command called **sudo** (super-user do) to execute individual commands, rather than run a full session as root. So, on Ubuntu, you'll need to prepend **sudo** to all of the commands in this section.

Graham's top tip

My most used command-line couplet is **sudo bash**. It's only of use with Ubuntu-like distributions (and OS X!), but it lets you easily create a new session with super-user credentials – very handy if you're playing with config files or services, as you no longer need to precede every command with **sudo**.

The first time you do it, you'll be prompted for *your* password, but subsequent times you won't be (unless you're idle for five minutes or more).

Total control

These simple steps have given you complete control and power over your computer. Bear this in mind going forward because, while you're running as root, the potential damage a mis-typed command can do is far greater (there's actually not too much you can do wrong, just be careful

Examples 2

In this section, we've looked at a range of commands and a few advanced command-line features. To help you solidify your understanding, here are a few more examples:

```
[jon@adam ~]$ cp -r /home/jon/Documents /home/jon/Documents-bk
```

Copies the entire Documents folder and its contents to **Documents-bk** for safekeeping.

```
[jon@adam ~]$ mv {demo.txt,demo-1.txt,demo-3.txt} Documents/Examples
```

Moves the three demo files to the **Examples** folder, which is inside **/home/jon/Documents**.

```
[jon@adam ~]$ mkdir LXF\ \(Version\ 1\)
```

Makes a directory called **LXF (Version 1)** using the escape character (****) to ensure the command line properly interprets the special characters (**,)** and **.**

when using the **rm** command, and never direct it at the **/** folder).

Now that you know about permissions, and how to become the root user, we can begin to look at how to gather information about your system and get help when you have a problem. We'll look at log files first.

While your computer is running, it's constantly recording information about how it's working and problems it encounters. All of this information is stored in log files, which are just plain text and kept in the **/var/log** directory.

Log files

As your normal user, you can use **ls** on this directory and take a look at the files inside. If you try to open any of these files with a text editor or print their output with the **cat** command, you'll get a message telling you that permission was denied.

This is because the **/var/log** files are owned by the root user, as discussed above. To look inside any of the files, you'll have to use the **su -** or **sudo** command to get root permissions, and then run **nano** or **cat** afterwards.

After using **ls**, you might have felt a little overwhelmed by the number of log files and their awkward-looking names; if you were brave enough to look inside one with **cat** or **nano**, you're probably feeling even more overwhelmed as their contents are often completely indecipherable unless you know exactly what you're looking for.

While we can't offer you a full rundown of what every log file does, we can at least offer some pointers on interpreting **errors.log**, perhaps the most useful of the log files, to ease your worries.

As its name suggests, every error on the system is recorded here, along with the time at which it occurred. So, when facing a particular problem, note the time at which it happened and then look for its corresponding entry in this file.

Take careful note of the error message, and then put it into Google or share it in any cries for help you make in forums, IRC or mailing lists. As well as giving more information to make it easier for others to solve your problem, you'll demonstrate that you've already done as much of the leg-work as you can and people will be far friendlier when helping.

Hardware information

As well as the log files, there are also times when it's useful to provide detailed information about your system's hardware.

Rather than looking inside a particular file, there are specific commands that will output information about all of the hardware connected to your system. Two of the most frequently used are **lspci** and **lsusb**, which, respectively, print out a list of all the connected PCI and USB devices.

While **lsusb** might not always be quite so useful, **lspci** is incredibly handy as much of your system's vital hardware is connected via this technology, including most graphics and network cards.

If you try running **lspci**, you'll see that the output it prints out is long and, for the novice, difficult to decipher. Because of this, if you wanted to include information about hardware on a forum or mailing list post,

Examples 3

Our final set of examples for this article.

```
[jon@adam ~]$ su -
```

Become the root user. Replace - with any username to switch to that user, although you'll need to know their password.

```
[root@adam ~]$ cat /var/log/errors.log
```

As the root user, output the contents of **errors.log** to the terminal window.

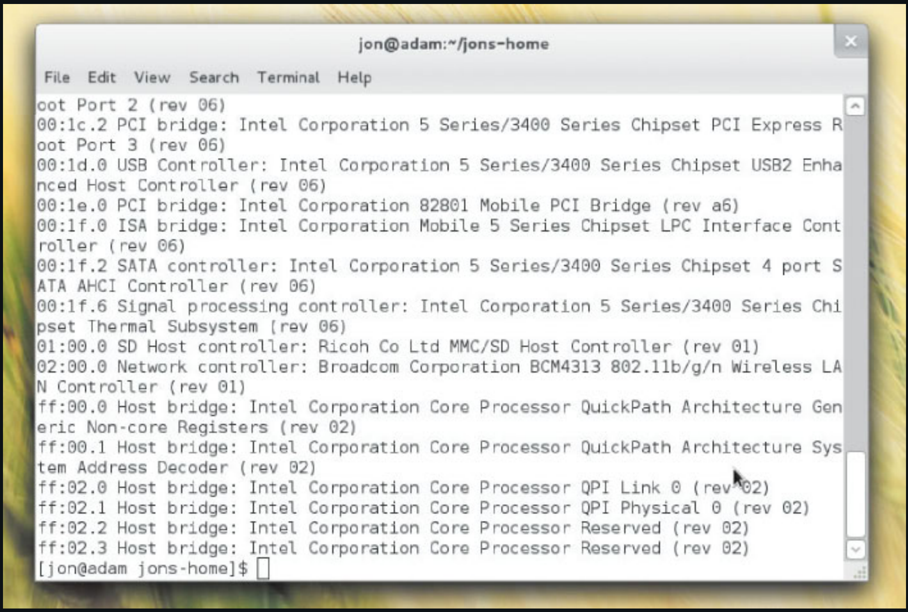
```
[root@adam ~]$ cat /var/log/errors.log >> /home/jon/error-file
```

Output the contents of the **errors.log** to the file **/home/jon/error-file**, appending the new data to any existing information in **error-file**.

```
[root@adam ~]$ dmesg | grep error
```

Reroute the output of **dmesg** to the **grep** command, which will then search through the output and display only lines that contain the word **error**. Note that we use **>** for redirecting output to a file, and **|** for redirecting output to become the input for another command.

"We can begin to look at how to gather information about your system and get help when you have a problem."



```
oot Port 2 (rev 06)
00:1c.2 PCI bridge: Intel Corporation 5 Series/3400 Series Chipset PCI Express R
oot Port 3 (rev 06)
00:1d.0 USB Controller: Intel Corporation 5 Series/3400 Series Chipset USB2 Enha
nced Host Controller (rev 06)
00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev a6)
00:1f.0 ISA bridge: Intel Corporation Mobile 5 Series Chipset LPC Interface Cont
roller (rev 06)
00:1f.2 SATA controller: Intel Corporation 5 Series/3400 Series Chipset 4 port S
ATA AHCI Controller (rev 06)
00:1f.6 Signal processing controller: Intel Corporation 5 Series/3400 Series Chi
pset Thermal Subsystem (rev 06)
01:00.0 SD Host controller: Ricoh Co Ltd MMC/SD Host Controller (rev 01)
02:00.0 Network controller: Broadcom Corporation BCM4313 802.11b/g/n Wireless LA
N Controller (rev 01)
ff:00.0 Host bridge: Intel Corporation Core Processor QuickPath Architecture Gen
eric Non-core Registers (rev 02)
ff:00.1 Host bridge: Intel Corporation Core Processor QuickPath Architecture Sys
tem Address Decoder (rev 02)
ff:02.0 Host bridge: Intel Corporation Core Processor QPI Link 0 (rev 02)
ff:02.1 Host bridge: Intel Corporation Core Processor QPI Physical 0 (rev 02)
ff:02.2 Host bridge: Intel Corporation Core Processor Reserved (rev 02)
ff:02.3 Host bridge: Intel Corporation Core Processor Reserved (rev 02)
[jon@adam jons-home]$
```

› The long and intimidating output of **lspci** is best captured by redirecting the output to a file.

Mike's top tip

Tired of typing **logout** or **exit** every time you're at the command line? There's a handy shortcut. Hit Ctrl+D at the same time and you'll be logged out of the current shell session. If you logged in as a normal user and switched to root, you'll be returned back to the normal user account.

Andrew's top tip

Converting large numbers of audio files (from MP3 to Ogg for example) would take forever using a GUI, so use **pacpl** instead. **pacpl --to ogg -r -p /home/music/mp3s --outdir /home/music/oggs** will convert all your non-free music to Ogg files in a matter of minutes.

you might be better off quoting the entire output and allowing those more experienced than you to find the relevant information.

The question is, after running **lspci**, how are you meant to get this text to the forum? You're really not going to want to copy it by hand, and if you're stuck at a text-only terminal, rather than a graphical emulator, there's no copy and paste support.

Redirection

Instead, the secret is to use a command-line trick called redirection. The basic idea is that you can capture the output of any command, and redirect it away from the terminal, either to a file or to another program for further processing. You can then forward this file to someone via email, save it to a USB stick or anything else you can do with a normal file.

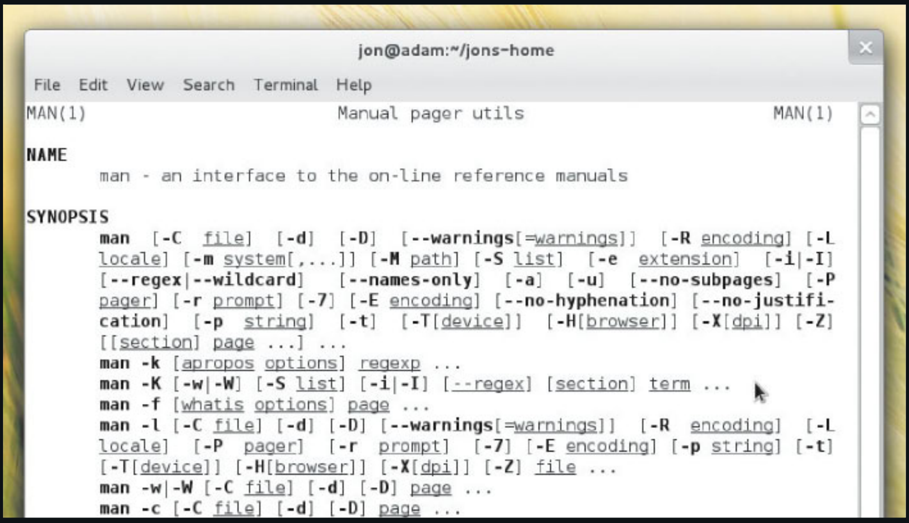
To make this happen, you use either the **>** or **|** symbols, placing them between the command whose output you want to capture and the file or program you want to redirect it to. Look at this example with **lspci**:

```
[jon@adam ~]$ lspci > lspci.txt
```

The result is that the long list that **lspci** usually prints is no longer displayed on the terminal, but instead stored in the file **lspci.txt**. You can check this with **nano**, or you can use the **cat** command which will print out the contents of any text file in the terminal window.

Note that a single **>** will overwrite the contents of any existing files, whereas **>>** will append the new data to the end of an existing file – an important and useful distinction.

Another very useful command that often gets coupled with redirection to get help from other users is **dmesg**. This outputs all of the information the kernel generates, including information about the boot process and whether or not it recognises certain pieces of hardware as you plug it in.



```
MAN(1)                    Manual pager utils                    MAN(1)

NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-C  file]  [-d]  [-D]  [--warnings[=warnings]]  [-R encoding] [-L
       locale] [-m system[,...]] [-M path] [-S list]  [-e  extension]  [-i|-I]
       [--regex|--wildcard]   [--names-only]  [-a]  [-u]  [--no-subpages]  [-P
       pager] [-r prompt] [-7] [-E encoding] [--no-hyphenation] [--no-justifi-
       cation]  [-p  string]  [-t]  [-T[device]]  [-H[browser]] [-X[dpi]] [-Z]
       [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
       man -f [whatis options] page ...
       man -l [-C file] [-d] [-D] [--warnings[=warnings]]  [-R  encoding]  [-L
       locale]  [-P  pager]  [-r  prompt]  [-7] [-E encoding] [-p string] [-t]
       [-T[device]] [-H[browser]] [-X[dpi]] [-Z] file ...
       man -w|-W [-C file] [-d] [-D] page ...
       man -c [-C file] [-d] [-D] page ...
```

› The most useful command ever made – man. Be sure to run man man to find out how to use it effectively, and remember that the keyboard arrows scroll through the display.

Man-to-man advice

We've covered a lot of information but, having taken your first steps, hopefully you now feel more comfortable at the command line. At least, we hope that if you read articles on the internet that refer to it, you'll have enough knowledge to sensibly and safely put their advice into practice.

Before we finish completely, however, and in the finest traditions of TuxRadar, there's one more thing. Many of the commands we've looked at take different arguments or options, but how on earth are you meant to remember which programs take which arguments, in which order, and have which options available?

Well, thanks to a command called **man**, there's no need to remember the ins-and-outs of every command. Run **man**, and pass to it the name of almost any program on your system and it will display the manual page for that command.

These manual pages list all of the available options, specify in which order arguments should appear, and many even provide examples of how you can put them into practice. They're an indispensable tool. What's really cool is that if you want to learn how to use man pages efficiently, you can run the command **man man** and get the manual page for the **man** command. LXF